PSA Certified Firmware Update API 1.0

Document number:	IHI 0093
Release Quality:	Beta
Issue Number:	0
Confidentiality:	Non-confidential
Date of Issue:	17/10/2022

This is a Beta quality release. The content is subject to change. To provide feedback, see *Feedback* on page x.

Abstract

This document defines a standard firmware interface for installing firmware updates.

Contents

About this document

Release information

The change history table lists the changes that have been made to this document.

Table 1 Document revision history

Date	Version	Confidentiality	Change
Feb 2021	0.7 Beta 0	Non-confidential	First release at Beta quality.
October 2022	1.0 Beta 0	Non-confidential	Major update of programming model and API. Relicensed as open source under CC BY-SA 4.0.

For a detailed list of changes, see *Document change history* on page 74.

PSA Certified Firmware Update API

License

Text and illustrations

About the license

The language in the additional patent license is largely identical to that in section 3 of the Apache License, Version 2.0 (Apache 2.0), with two exceptions:

1. Changes are made related to the defined terms, to align those defined terms with the terminology in CC BY-SA 4.0 rather than Apache 2.0 (for example, changing "Work" to "Licensed Material").
2. The scope of the defensive termination clause is changed from "any patent licenses granted to You" to "any licenses granted to You". This change is intended to help maintain a healthy ecosystem by providing additional protection to the community against patent litigation claims.

To view the full text of the Apache 2.0 license, visit apache.org/licenses/LICENSE-2.0.

Source code

References

This document refers to the following documents.

Table 2 Documents referenced by this document

Ref	Document Number	Title
[C99]		ISO/IEC, *ISO/IEC 9899:1999 — Programming Languages — C*, December 1999. www.iso.org/standard/29237.html
[EBBR]		Arm Limited and Contributors, *Embedded Base Boot Requirements (EBBR) Specification*. arm-software.github.io/ebbr
[EN303645]	EN 303 645	ETSI, *Cyber Security for Consumer Internet of Things: Baseline Requirements*, June 2020. www.etsi.org/standards/get-standards#search=303%20645
[IR8259]	IR 8259	NIST, *Foundational Cybersecurity Activities for IoT Device Manufacturers*, May 2020. doi.org/10.6028/NIST.IR.8259
[LWM2M]	LwM2M v1.2	OMA, *Lightweight M2M*, November 2020. openmobilealliance.org/release/LightweightM2M
[PSA-CERT]	JSA DEN 002	*PSA Certified™ Level 2 Lightweight Protection Profile*. psacertified.org/development-resources/certification-resources/#leveltwo
[PSA-FFM]	ARM DEN 0063	*Arm® Platform Security Architecture Firmware Framework*. pages.arm.com/psa-apis
[PSA-SS]	ARM IHI 0087	*PSA Certified Secure Storage API*. arm-software.github.io/psa-api/storage
[PSA-STAT]	ARM IHI 0097	*PSA Certified Status code API*. arm-software.github.io/psa-api/status-code
[PSM]	ARM DEN 0128	*Platform Security Model*. developer.arm.com/documentation/den0128
[RFC4122]		IETF, *A Universally Unique IDentifier (UUID) URN Namespace*. tools.ietf.org/html/rfc4122
[RFC8240]		IAB, *Report from the Internet of Things Software Update (IoTSU) Workshop 2016*, September 2017. tools.ietf.org/html/rfc8240
[RFC9019]		IETF, *A Firmware Update Architecture for Internet of Things*, April 2021. tools.ietf.org/html/rfc9019
[RFC9124]		IETF, *A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices*, January 2022. tools.ietf.org/html/rfc9124

Table 2 (continued)

Ref	Document Number	Title
[SUIT-MFST]		IETF, *A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest*, October 2022. datatracker.ietf.org/doc/draft-ietf-suit-manifest
[UEFI]	UEFI v2.10	UEFI Forum, Inc., *Unified Extensible Firmware Interface (UEFI) Specification*, August 2022. uefi.org/specifications

Terms and abbreviations

This document uses the following terms and abbreviations.

Table 3 Terms and abbreviations

Term	Meaning
Application firmware	The main application firmware for the platform, typically comprising an Operating System (OS) and application tasks. On a platform with isolation, the application firmware runs in the *NSPE*.
Application Root of Trust	This is the security domain in which additional security services are implemented. See *Platform Security Model* [PSM].
Immutable Platform Root of Trust	Part of the *Platform Root of Trust*, which is inherently trusted. This refers to the hardware and firmware that cannot be updated on a production device. See *Platform Security Model* [PSM].
IMPLEMENTATION DEFINED	Behavior that is not defined by the this specification, but is defined and documented by individual implementations. Firmware developers can choose to depend on IMPLEMENTATION DEFINED behavior, but must be aware that their code might not be portable to another implementation.
Manifest	Firmware image metadata that is signed with a cryptographic key. The manifest can be bundled within the firmware image, or detached from it. See *Manifest* on page 19.
MPU	Memory protection unit
Non-secure Processing Environment (NSPE)	This is the security domain outside of the *Secure Processing Environment*. It is the Application domain, typically containing the *application firmware* and hardware.
NSPE	See *Non-secure Processing Environment*.
OEM	Original equipment manufacturer
OTA	See *Over-the-Air*.

Table 3 (continued)

Term	Meaning
Over-the-Air (OTA)	The procedure where a device downloads an update from a remote location ("over the air").
PKI	Public key infrastructure
Platform Root of Trust (PRoT)	The overall trust anchor for the system. This ensures the platform is securely booted and configured, and establishes the secure environments required to protect security services. See *Platform Security Model* [PSM].
PRoT	See *Platform Root of Trust*.
PSA	Platform Security Architecture
Root of Trust (RoT)	This is the minimal set of software, hardware and data that is implicitly trusted in the platform — there is no software or hardware at a deeper level that can verify that the Root of Trust is authentic and unmodified.
RoT	See *Root of Trust*.
Secure boot	Secure boot is technology to provide a chain of trust for all the components during boot.
Secure Processing Environment (SPE)	This is the security domain that includes the *Platform Root of Trust* and the *Application Root of Trust* domains.
SPE	See *Secure Processing Environment*.
Updatable Platform Root of Trust	Part of the *Platform Root of Trust* firmware that can be updated following manufacturing. See *Platform Security Model* [PSM].
Update client	Software component that is responsible for downloading firmware updates to the device. The Update client is part of the *application firmware*.

Potential for change

The following may change in updates to the version 1.0 specification:

- Important API changes required before issuing a final v1.0.0 specification.
- Optional feature additions.
- Corrections and additions to the informative chapters, use cases and examples.
- Clarifications.

Conventions

Typographical conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

`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 *Terms and abbreviations*.

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 example.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.

Current status and anticipated changes

This document is at Beta quality status which has a particular meaning to Arm of which the recipient must be aware. A Beta quality specification will be sufficiently stable & committed for initial product development, however all aspects of the architecture described herein remain SUBJECT TO CHANGE. Please ensure that you have the latest revision.

Feedback

We welcome feedback on the PSA Certified API documentation.

If you have comments on the content of this book, visit github.com/arm-software/psa-api/issues to create a new issue at the PSA Certified API GitHub project. Give:

- The title (Firmware Update API).
- The number and issue (IHI 0093 1.0 Beta (Issue 0)).
- The location in the document to which your comments apply.

- A concise explanation of your comments.

We also welcome general suggestions for additions and improvements.

1 Introduction

1.1 About Platform Security Architecture

This document is one of a set of resources provided by Arm that can help organizations 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 at www.psacertified.org and find more Arm resources here at developer.arm.com/platform-security-resources.

1.2 About the Firmware Update API

The interface described in this document is a PSA Certified API, that provides a portable programming interface to firmware update and installation operations on a wide range of hardware.

The interface enables the software and systems that manage and deliver a firmware update to a device, to be developed independently from the hardware-specific mechanisms required to apply the update to the device. Reusing the deployment and delivery system for firmware updates reduces the complexity of providing firmware updates across a diverse set of managed devices.

You can find additional resources relating to the Firmware Update API here at arm-software.github.io/psa-api/fwu, and find other PSA Certified APIs here at arm-software.github.io/psa-api.

1.3 Firmware update

Connected devices need a reliable and secure firmware update mechanism. Incorporating such an update mechanism is a fundamental requirement for fixing vulnerabilities, but it also enables other important capabilities such as updating configuration settings and adding new functionality. This can be particularly challenging for devices with resource constraints, as highlighted in *Report from the Internet of Things Software Update (IoTSU) Workshop 2016* [RFC8240].

Figure 1 on page 13 depicts the actors and agents involved in a typical firmware update scenario.

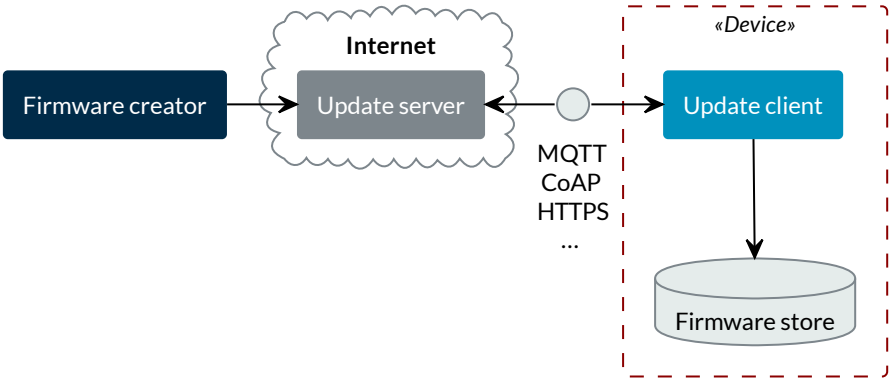


Figure 1 A typical over-the-air firmware update scenario

In this example, the new firmware is uploaded by the Firmware creator to an Update server. The Update server communicates with an Update client application on the device, announcing the availability of new firmware. The Client downloads the new firmware, and installs it into the device firmware storage.

In Figure 1, the Update client has to combine the following capabilities:

- The specific protocols used by the network operator in which the device is deployed
- The specific mechanism used by the hardware platform to install firmware for execution

Devices developed for the Internet of Things (IoT) have a very diverse ecosystem of hardware and software developers, and utilize a broad set of communication protocols and technologies. This will lead to a large, fragmented set of Update Clients, that are each tightly coupled to one hardware platform and one network protocol.

The Firmware Update API separates the software responsible for delivering the new firmware in the device, from the software that is responsible for storing and installing it in the device memory. Figure 2 shows how the Firmware Update API separates an Update client, which obtains the new firmware from the Firmware Server, from an Update service, which stores the firmware in the device memory.

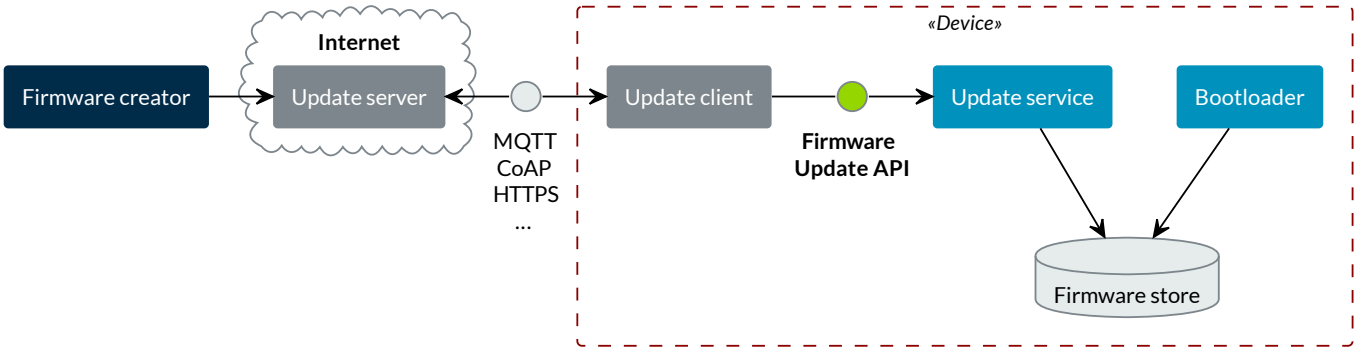


Figure 2 The Firmware Update API

In practice, this enables an Update client to be written independently of the firmware storage design, and the Update service to be written independently of the delivery mechanism.

The remainder of this document includes:

- The design goals for the Firmware Update API. See *Design goals* on page 15.

- A definition of the concepts and terminology used in this document. See *Architecture* on page 19.
- A description of the interface design. See *Programming model* on page 26.
- A detailed definition of the API. See *API reference* on page 39.

The appendixes provide additional information:

- A sample header file containing all of the API elements. See *Example header file* on page 62.
- Some example code demonstrating various use cases. See *Example usage* on page 64.

2 Design goals

This section describes the main goals and use cases for the Firmware Update API.

2.1 Suitable for constrained devices

The interface is suitable for a range of embedded devices: from those with resource-limited microcontrollers with one or two simple firmware images, to richer devices that have firmware images for multiple subsystems and separated applications.

For example, the following resource constraints can affect the Firmware Update API:

Resource	Impact on interface requirements
Volatile memory capacity	Firmware images must be transferred to the device in blocks small enough to fit in device RAM.
Non-volatile memory capacity	Firmware updates must be small enough to be stored in memory prior to installation.
Delivery bandwidth	Firmware download can take an extended period of time. The device might restart during this process.
Energy and power	Downloading and installing updates must be reliable to wasting energy on failed or repeated update attempts.
Performance of cryptographic primitives	The use of cryptographic protection for firmware updates must match the security requirements for the device.

For devices with sufficient resources, it is recommended to follow the *Embedded Base Boot Requirements (EBBR) Specification* [EBBR] specification, which prescribes the *Unified Extensible Firmware Interface (UEFI) Specification* [UEFI] capsule update interface.

2.2 Updating the Platform Root of Trust

The Firmware Update API is suitable for updating the device's *Platform Root of Trust* (PRoT) firmware.

The *Platform Security Model* [PSM] requires all of the *Updatable Platform Root of Trust* firmware to be updatable. This can include bootloaders, Secure Partition Manager, Trusted OS, and runtime services. In some implementations, the PRoT can include a trusted subsystem with its own isolated and updatable firmware.

The [PSM] requirements for firmware update are also reflected in certifications like *Foundational Cybersecurity Activities for IoT Device Manufacturers* [IR8259], *Cyber Security for Consumer Internet of Things: Baseline Requirements* [EN303645], and *PSA Certified™ Level 2 Lightweight Protection Profile* [PSA-CERT]. [PSA-CERT] provides the following definition of the F.FIRMWARE_UPDATE security function, where the Target of Evaluation (TOE) refers to the PRoT:

The TOE verifies the integrity and authenticity of the TOE update prior to performing the update.

The TOE also rejects attempts of firmware downgrade.

2.3 Updating the Application Root of Trust

In addition to the PRoT firmware, other services that run in the *Secure processing environment* (SPE), but outside of the PRoT, can require update via the Firmware Update API. These services may be combined with the updatable PRoT in a single firmware image, or provided in a separate firmware image.

2.4 Flexibility for different trust models

There are a number of factors that impact the trust model that is used to authorize device updates and firmware execution. For example:

- A device can require firmware updates from multiple, mutually distrustful, firmware vendors.
- Regulation can require implementations to use specified Certificate Authorities and PKI.
- The entity that signs a firmware image can be distinct from the device owner or operator. An operator of a device can have a security policy that requires additional authorization to the firmware author's policy.

The Firmware Update API must be flexible enough to support the trust model required for particular products, without imposing unnecessary overheads on constrained devices.

2.5 Protocol independence

Different protocols are used to communicate with a device depending on the industry and application context. This includes open protocols, such as *Lightweight M2M* [LWM2M], and proprietary protocols from cloud service providers. These protocols serve the specific needs of their respective markets.

Some of the protocols have *manifest* data that is separate from the firmware image.

The Firmware Update API must be independent of the protocol used by the update client to receive an update.

2.6 Transport independence

Embedded devices can receive over-the-air (OTA) firmware updates over different transport technologies, depending on the industry and the application. For example, this includes Wi-Fi, LTE, LoRa, and commercial low-power wide-area networks.

Some devices might not be directly connected to a network but may receive updates through a physical interface from an adjacent device, such as UART, CAN bus, or USB.

The Firmware Update API must be independent of the transport used by the update client to receive an update.

Note:

The Firmware Update API does not cover reprogramming of a device using a debug interface, for example, JTAG or SWD.

2.7 Firmware format independence

Many device manufacturers and cloud service providers have established formats for firmware images and manifests, tailored to the specific needs of their systems and markets.

The Firmware Update API must be independent of the format and encoding of firmware images and manifests, to enable adoption of the interface by systems with existing formats.

Note:

New standards for firmware update within IoT are being developed, such as *A Firmware Update Architecture for Internet of Things* [RFC9019].

This version of the Firmware Update API is suitable for some of the use cases that are defined by *A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices* [RFC9124] and *A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest* [SUIT-MFST]. For example, where the payloads are integrated in the manifest envelope, or there is just one external payload to the envelope.

Support for the more complex use cases from [RFC9124], with multiple external payloads, is directly considered in version 1.0 of the Firmware Update API, but might be in scope for future versions of the interface.

2.8 Flexibility for different hardware designs

The Firmware Update API is designed to be reasonably efficient to implement on different system-on-chip (SoC) architectures, while providing a consistent interface for update clients to target.

For example, the Firmware Update API should be effective in the following types of system:

- SoCs that use bus filters, or equivalent security IP, to protect the *SPE*.
- SoCs that use multiple CPUs, providing an isolated CPU and memories for the SPE and another for the *NSPE*.
- Simple SoCs that use an *MPU* or equivalent to protect the SPE.
- Systems that have unified on-chip non-volatile memory used for firmware storage.
- Systems that have isolated on-chip non-volatile memory used for firmware storage.
- Systems that have a mixture of on-chip and external non-volatile memory used for firmware storage.

2.9 Suitable for composite devices

Some platforms have independent subsystems that are isolated from the main microprocessor. These subsystems can have their own firmware, which can also require updates. For example, radios, secure elements, secure enclaves, or other kinds of microcontroller.

The Firmware Update API must support an implementation updates these types of subsystem.

2.10 Robust and reliable update

Devices that are remotely deployed, or are deployed in large numbers, must use an update process that does not have routine failure modes that result in devices that cannot be remotely recovered.

The Firmware Update API must support an update process that reduces the risk of in-field update failure, without compromising the requirements for *secure boot*.

> **Note:**
>
> A device can also have an additional recovery capability, for example, a separate recovery firmware image that the bootloader can execute if the installed firmware cannot be verified.
>
> The Firmware Update API might be useful for implementation of recovery firmware, but the requirements of recovery firmware are not considered in the interface design.

2.11 Flexibility in implementation design

The Firmware Update API is architectural and does not define a single implementation. An implementation can make trade-offs to target specific device needs. For example:

- An implementation can provide a more robust solution, while others optimize for device cost.
- An implementation can optimize for bandwidth efficiency, while others optimize for simplicity
- An implementation can provide fine-grained update of personalization data, while others perform monolithic updates of all code and data.
- An implementation can provide enhanced security for stricter markets, such as those which require encrypted firmware images, while others only use the Firmware Update API to provide a common interface across all products.

The Firmware Update API permits the omission of optional features that are not used by the implementation.

3 Architecture

3.1 Concepts and terminology

This section describes important concepts and terminology used in the Firmware Update API specification.

Figure 3 identifies the main actors and agents involved in a typical firmware update scenario.

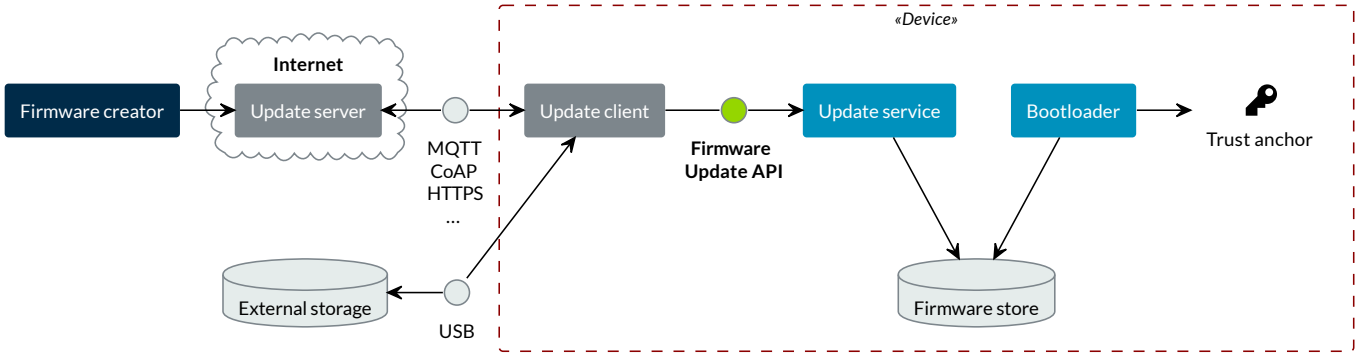


Figure 3 The Firmware Update API in context

3.1.1 Firmware image

A firmware image, or simply the “image”, is a binary that can contain the complete software of a device or a subset of it. A firmware image can consist of multiple images if the device contains more than one microcontroller. It can also be a compressed archive that contains code, configuration data, and even the entire file system. An image may consist of a differential update for performance reasons.

The terms “firmware image”, “firmware”, and “image” are used in this document and are interchangeable.

3.1.2 Manifest

A manifest contains metadata about the firmware image. The manifest is typically protected against modification using a signed hash of its contents, see *Manifest verification* on page 32.

Metadata that can be in a manifest includes the following:

- The intended device, which might be a specific instance or class.
- The intended device component.
- The version or serial-number of the firmware image.
- A digest of the image.
- Information relating to rollback prevention, or other security policies.
- Dependencies on other firmware images.
- Hints or explicit instructions on how to decrypt, decompress or install an image.

- Information on additional steps required to apply the update.

A manifest can be bundled within the firmware image, or detached from it.

3.1.3 Component

A component is a logical part of the device which needs a firmware image. Each firmware image is designed for exactly one component.

A component can have a one to one correspondence with a physical processor in the system, other mappings are possible:

- A single physical processor might have multiple components. For example:
  - If the *SPE* and *NSPE* have separate firmware images, these are separate components.
  - If configuration data for the system can be updated independently, this is a separate component.
- Multiple processors, or even the whole system, can have the firmware packaged together in a single firmware image. As a whole, this forms a single component in the context of the Firmware Update API.

3.1.4 Component identifier

The component identifier is a small numerical value, that precisely identifies the component within this device.

The identifier values are typically allocated by the device developer or integrator. A component identifier can be used within the manifest during the update process, or can be translated from another identification scheme via a mapping configured in the update client.

3.1.5 Firmware creator

A developer or integrator of the firmware for the device being updated.

The firmware creator is responsible for constructing firmware images and manifests for the device. For devices that implement a *secure boot* protocol, the firmware creator signs the manifest using a signing key associated with a trust anchor on the device. See *Trust anchor* on page 21.

In systems with multiple components, each component can have a different firmware creator.

3.1.6 Update server

A system within the operational network of the device that hosts firmware images and manages the rollout of updates to devices within that network.

3.1.7 Update client

The update client is a software component that obtains firmware images. For example, this can be downloaded from an update server, or accessed from an attached storages device. When it obtains an image, it transfers it to the update service using the interface described in this document.

The update client runs as part of the *application firmware*.

It can report device identity and installation state to a remote party, such as the update server. For example, the reported installation state can include the versions of installed images and error information of images that did not install successfully.

3.1.8 Update service

The update service is a software component that stores a firmware image in device memory, ready for installation. The update service implements the interface described in this document.

Depending on the system design, the installation process can be implemented within the update service, or it can be implemented within a bootloader or other system component.

3.1.9 Firmware store

The firmware store is the location where firmware images are stored. Conceptually the Firmware store is shared between the update service and the bootloader. Both components share access to the firmware store to manage the firmware update process.

The Firmware Update API presents a separate firmware store for each component. Each component's firmware store can have one or more images present. The state of the firmware store determines how those images are used, and what is required to proceed with a firmware update.

The "staging area" is a region within a firmware store used for a firmware image that is being transferred to the device. Once transfer is complete, the image in the staging area can be verified during installation.

3.1.10 Bootloader

A bootloader selects a firmware image to execute when a device boots. The bootloader can also implement the verification and installation process for a firmware update.

In a system that implements *secure boot*, the bootloader will always verify the authenticity of the firmware image prior to execution.

3.1.11 Trust anchor

A device contains one or more trust anchors. A trust anchor is used to check if an image, or its manifest, are signed by a signing authority that the device trusts.

Each trust anchor is pre-provisioned on the device. A trust anchor can be implemented in many ways, but typically takes the form of a public key or a certificate chain, depending on the complexity of the trust model.

The management and provisioning of trust anchors is not within the scope of this document.

3.2 Firmware image format

The Firmware Update API does not define the format for the firmware image and manifest. This is defined and documented by the implementation, so that a firmware creator can construct valid firmware images and manifests for the device.

The Firmware Update API assumes that manifests and firmware images passed to the update service conform to the format expected by the implementation. The implementation is responsible for verifying that data provided by the client represents a valid manifest or firmware image.

Examples of the firmware image and manifest design details that need to be provided by the implementation, include the following:

- Whether the manifest is detached from, or bundled with, the firmware image.
- The format and encoding of the manifest and firmware image.
- The attributes provided by the manifest, and their impact on processing of the firmware image.
- Support for encrypted, compressed, or delta firmware image.
- Firmware image integrity and authentication data.

If firmware images must be signed — for example, for devices implementing *secure boot* — the device creator must enable the firmware creator to sign new firmware images in accordance with the device policy.

For some deployments, the firmware and manifest formats used by a device can be affected by the protocols used by the update server and update client to notify and transfer firmware updates. In other deployments, the update server and update client can have independent formats for describing firmware updates, to those used by the firmware creator and update service.

3.3 Deployment scenarios

There are different ways in which the Firmware Update API can be implemented, that apply to different system designs. The primary differences relate to the presence and location of trust boundaries within the system, in particular trust boundaries that protect a device *Root of Trust*.

The implementation architecture can affect the behavior of the Firmware Update API, particularly in regard to if, and when, a firmware update is verified.

These implementation architectures provide use cases for the design of the Firmware Update API.

3.3.1 Untrusted client

Figure 4 on page 23 shows an implementation architecture for a system where the firmware store is fully protected by the *Platform Root of Trust* (PRoT).

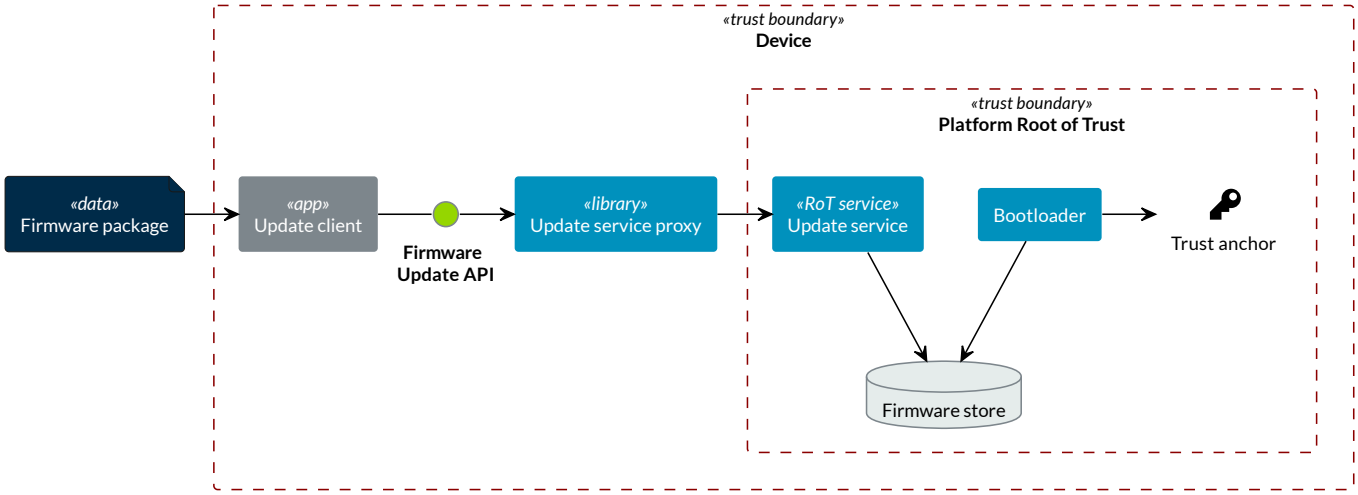


Figure 4 Implementation architecture with an untrusted update client

In this architecture, part of the update service must run as a service within the PRoT, to query and update the firmware store. The update client accesses this service via an update service proxy library, which implements the Firmware Update API.

The Firmware Update API is designed for implementation across a security boundary, as used in this architecture.

This architecture enables all of the firmware verification requirements to be fulfilled by the update service within the PRoT.

As the PRoT trusts the update service, but not the update client, this architecture is referred to as an "untrusted client" implementation.

3.3.2 Untrusted service

Figure 5 on page 24 shows an implementation architecture for a system where the *active* image is protected by the *Platform Root of Trust* (PRoT), but the staging area for a new firmware image is not protected.

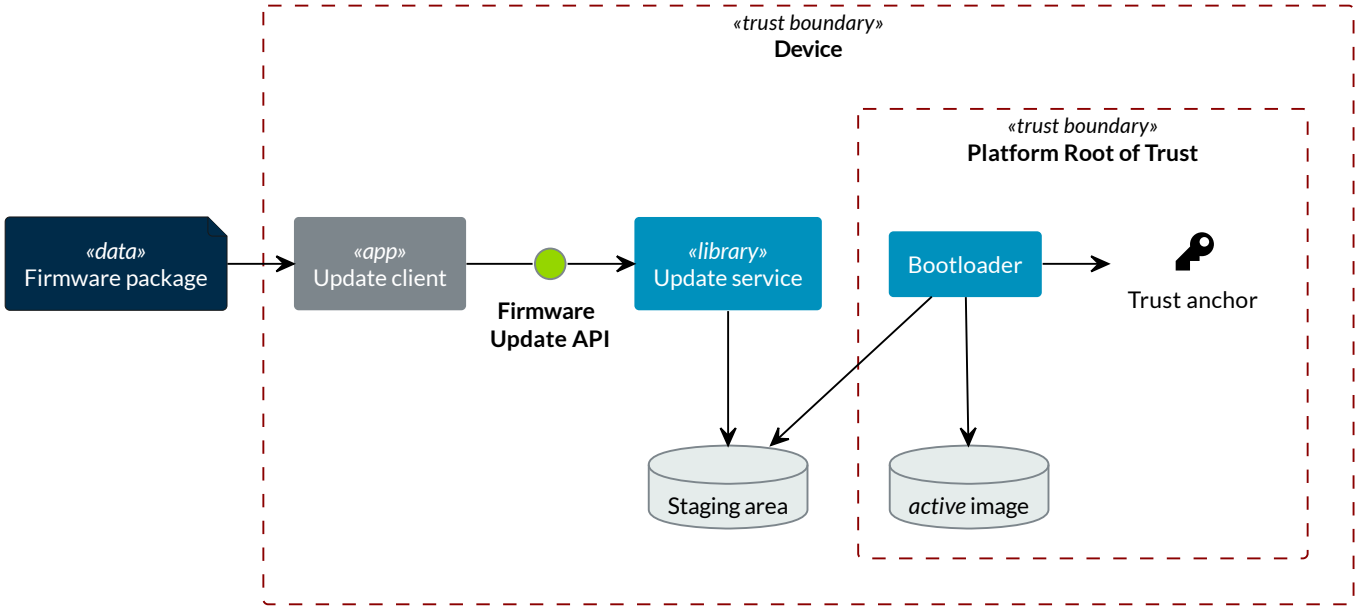


Figure 5 Implementation architecture with an untrusted update service

The staging area is accessible to untrusted components, so the bootloader cannot rely on any verification done by the update service prior to system restart. The bootloader must do all firmware verification prior to completing installation of the firmware.

In this type of implementation, it is still beneficial for the update service to perform some verification of firmware updates: this can reduce the system impact of a malicious or accidental invalid update.

As the PRoT does not trust the update service, this architecture is referred to as an “untrusted service” implementation.

3.3.3 Trusted client

Figure 6 shows an implementation architecture for a system where the update client application is within the system’s Root of Trust.

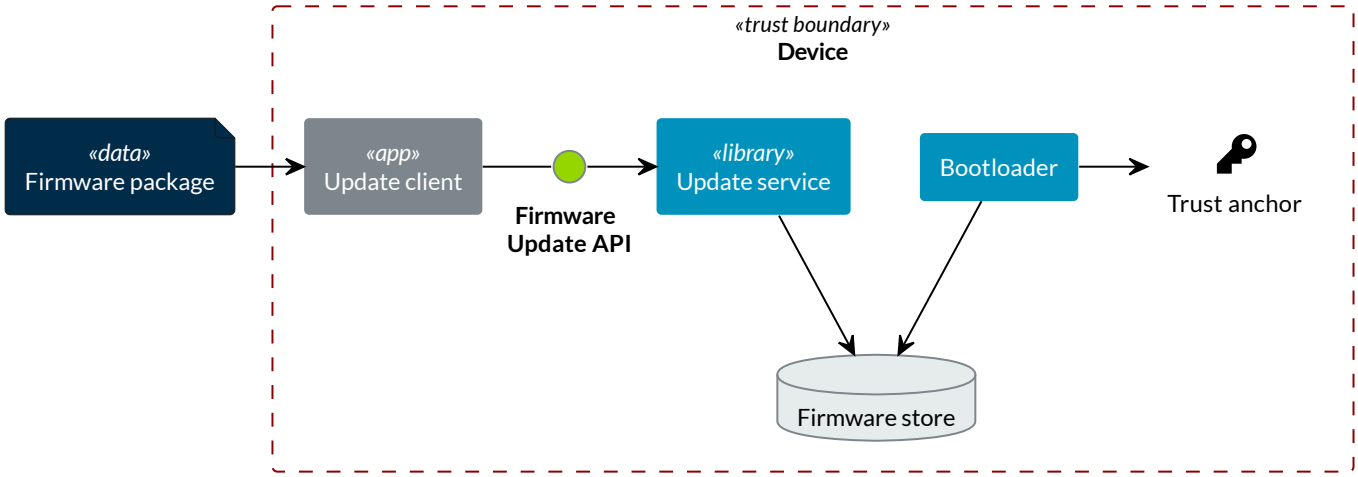


Figure 6 Implementation architecture with a trusted update client

In this architecture, it is possible for verification of an update to happen in any component, including the update client itself. This approach can be suitable for highly constrained devices, and relies on the security provided by the protocol used between the update server and update client.

Warning: If the implementation assumes that manifests and firmware images provided by the client are valid, and carries out the preparation and installation without further verification, then the Firmware Update API is being used purely as a hardware abstraction layer (HAL) for the firmware store.

An implementation like this must clearly document this assumption to ensure update clients carry out sufficient verification of firmware images before calling the Firmware Update API.

This implementation architecture can also be used in a device that does not enforce a *secure boot* policy. For example, this can enable code reuse by using a single API for firmware update across devices that have different security requirements and policies. Although permitted by the Firmware Update API, this usage is not a focus for the specification.

4 Programming model

4.1 The firmware store

For each component, depending on the state or progress of a firmware update, there can be one or more firmware images currently in the component's firmware store:

- An *active* image that is actively in use by the system.
- A *staged* image that is being prepared for installation.
- A *backup* of a previously installed image, used to recover if an attempted update fails.
- A *dirty* image that can be erased.

For a component that is essential for system operation, there will always be exactly one *active* image. Other images might, or might not, be present in the firmware store.

The Firmware Update API uses a state model for the firmware store that requires storage for a minimum of two images. This is possible because the store does not need to hold more than one *staged*, *backup*, or *dirty* image concurrently. An implementation of the Firmware Update API can have storage for more than two images, and selects the appropriate storage area for a requested operation. For example, providing additional image storage locations can reduce the need to carry out expensive erase operations on the storage during normal device operation.

This document uses the following names to identify the two required locations:

Location	Present	Description
Active	Always	The image that is actively in use by the system
Second	Some-times	An image that is being prepared, or is kept for recovery, or needs to be erased

Depending on the system and memory design, the *active* and *second* locations can be fixed physical storage locations, or can refer to different physical storage locations over time as an update progresses. The implementation of the Firmware Update API is responsible for mapping the logical storage locations to the stored firmware images.

During the course of an update, a specific firmware image can change from being *active* to *second*, or from *second* to *active*. For example:

- An image will switch from being *second* — while being prepared — to *active* following installation.
- An image will switch from being *active* to *second* when it becomes the backup image during installation of new firmware.

4.2 State model

4.2.1 Component state

Table 6 shows the possible update states for a component. The states have corresponding elements in the API, see *Component states* on page 46.

Table 6 Component states

State	Description
READY	This is the normal state for the component. There is just one image, it is *active*, and is currently in use by the system. The component is ready for a new firmware update to be started.
WRITING	The update client is writing a new firmware image to the *second*, in preparation for installation. When writing is complete, it can be prepared for installation.
CANDI-DATE	The update client has completed transfer of the new firmware image to the *second* image. When all components for update are prepared, they can be installed.
STAGED	Installation of the *second* has been requested, but the system must be restarted as the final update operation runs within the botloader. This state is transient.
TRIAL	Installation of the *second* has succeeded, and is now the *active* running in 'trial mode'. This state is transient, and requires the update client to explicitly accept the trial to make the update permanent. In this state, the previously installed *active* image is preserved as the *second*. If the trial is explicitly rejected, or the system restarts without accepting the trial, the previously installed image is re-installed and the trial image is rejected.
REJECTED	The *active* trial image has been rejected, but the system must be restarted so the bootloader can revert to the previous image, which was previously saved as the *second*. This state is transient.
FAILED	An installation of the *second* has been attempted, but has been cancelled or failed for some reason. The failure reason is recorded in the firmware store. The *second* needs to be cleaned before another update can be attempted.
UPDATED	The *active* trial image has been accepted. The *second* contains the now-expired previous firmware image, which needs to be cleaned before another update can be started.

The full set of states is necessary for components that require both of the following:

1. A reboot is required to complete installation of a new image.
2. The image must be tested prior to acceptance.

The description of the state model in *State transitions* assumes this type of component.

For components that do not require testing of new firmware before acceptance, or components that do not require a reboot to complete installation, only a subset of these states are visible to the update client. Some common variations are describe in *Variation in system design parameters* on page 70, including the impact on the state model for such components.

Implementation note

An implementation might support additional internal states, provided that implementation-specific states are not visible to the caller of the Firmware Update API.

4.2.2 State transitions

The state transitions occur either as a result of an function call from the update client, or when the bootloader carries out an installation operation. The installation operations that occur within the bootloader are determined by the state of the component, and do not depend on the reason for the restart.

Table Table 7 shows the operations that the update client uses to trigger transitions in the state model. The operations have corresponding elements in the API, see *Firmware installation* on page 50.

Table 7 Operations on components

`start`	Begin a firmware update operation
`write`	Write all, or part, of a firmware image
`finish`	Complete preparation of a firmware image
`cancel`	Abandon a firmware image that is being prepared
`install`	Start the installation of new firmware images
`accept`	Accept an installation that is being trialed
`reject`	Abandon an installation
`clean`	Erase firmware storage before starting a new update

The `start`, `write`, and `finish` operations are used to prepare a new firmware image. The `cancel` and `clean` operations are used to clean up a component after a successful, failed, or abandoned update. It is an error to invoke these operations on a component that is not in a valid starting state for the operation.

The `install`, `accept`, and `reject` operations apply to all components in the system, affecting any component in the required starting state for the transition. This allows an update client to update multiple components atomically, if directed by the firmware image manifests. Components that are not in a valid starting state for these operations are not affected by the operation.

Figure 7 on page 29 shows the typical flow through the component states.

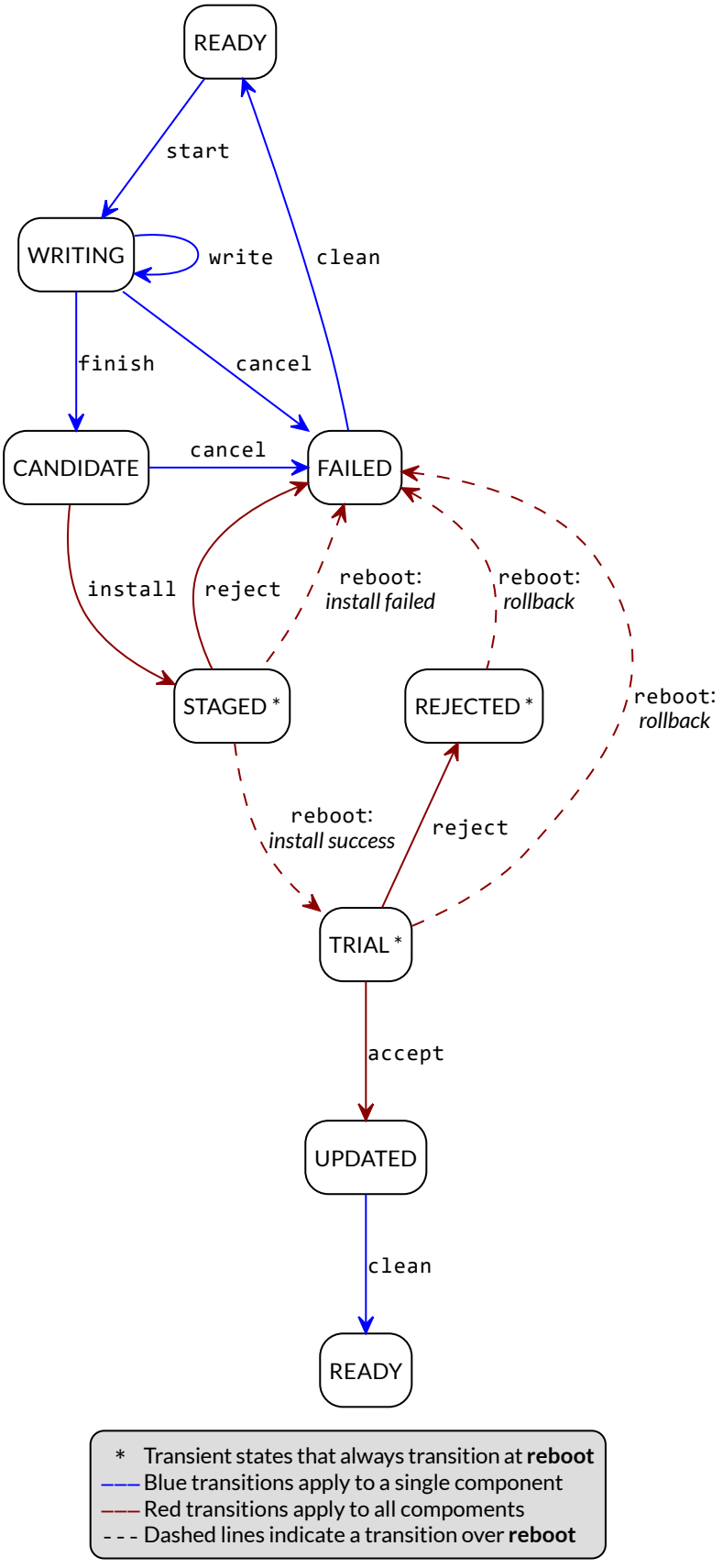


Figure 7 The component state model transitions

Note, that the READY state at the end is distinct from the starting READY state — at the end the *active* firmware image is the updated version. The component is ready to start the process again from the beginning for the next update.

The depicted flow does not show the behavior in error scenarios, except for the transitions over reboot where a failure can only be reported to the update client by changing the state of the component.

4.2.3 Behavior on error

Many of the operations in the Firmware Update API modify the firmware store. These operations are not required to have atomic operation with respect to the firmware store — when a failure occurs during one of these operations, the firmware store can be left in a different state after the operation reports an error status.

The following behavior is required by every implementation:

- When an operation returns the status `PSA_SUCCESS`, the requested action has been carried out.
- When a operation returns the status `PSA_SUCCESS_RESTART`, or `PSA_SUCCESS_REBOOT`, the requested action has been carried out, and appropriate action must be taken by the caller to continue the installation or rollback process.
- When a operation returns the status `PSA_ERROR_BAD_STATE`, `PSA_ERROR_DOES_NOT_EXIST`, or `PSA_ERROR_NOT_SUPPORTED`, no action has been carried out, and the affected components' states are unchanged.
- If firmware image dependencies are verified when the component is in CANDIDATE state, a missing dependency leaves the component unchanged, in CANDIDATE state.
- If there is a failure when verifying other manifest or firmware image properties of a component in WRITING, CANDIDATE or STAGED state, the component is transitioned to FAILED state.
- If there is a failure when verifying or installing a new firmware image during a component restart, or system reboot, the component is transitioned to FAILED state.
- A component does not follow a transition that is not shown in the state model, except for transitions to FAILED state as described in these rules.

If an operation fails because of other conditions, it is IMPLEMENTATION DEFINED whether the component state is unchanged, or is transitioned to FAILED state. In this situation, it is recommended that the update client abort the update process with a `cancel` operation.

If an unexpected system restart interrupts an operation, it is IMPLEMENTATION DEFINED whether the component state is unchanged, is transitioned to FAILED state, or is processed to a following state by the bootloader as described by the state model. In this situation, the update client must query the component status when it restarts, to determine the result.

4.2.4 Rationale

The complexity of the state model is a response to the requirements that follow from the use cases for the Firmware Update API. Table 8 provides a rationale for the state model design.

Table 8 Use case implications for the state model

State model feature	Rationale
Persistent WRITING state	Devices with slow download due to bandwidth or energy constraints can take an extended period to obtain the firmware image.
Incremental image transfer in WRITING state	Devices with limited RAM cannot store the entire image in the update client before writing to the firmware store.
CANDIDATE state	Enables the update client to explicitly indicate which components are part of an atomic multi-component `install` operation.
FAILED state	Enables the update client to detect failed installation operations that occur in the bootloader.
TRIAL and REJECTED states	Enables a new firmware image to be tested by application firmware, prior to accepting the update, without compromising a firmware rollback-prevention policy.
UPDATED state and `cancel` operation	Erasing non-volatile storage can be a high-latency operation. In some systems, this activity might block other memory i/o operations, including code execution. Isolating the erase activity within the `clean` operation enables an update client to manage when such disruptive actions take place.

4.3 Verifying an update

A firmware update is essentially authorized remote code execution. Any security weaknesses in the update process expose that remote code execution system. Failure to secure the firmware update process will help attackers take control of devices.

It is not sufficient to rely on a *secure boot* process to prevent execution of unauthorized firmware. This situation can easily result in an unusable device, as the installed firmware cannot be run, and the device can no longer update itself.

It is important for the update process to verify that an update is appropriate for the device, authentic, correctly authorized, and not expected to result in a non-functioning system. This is achieved by verifying various aspects of the firmware and its manifest. The various checks can take place at different points in the update process, depending on the firmware update implementation architecture — as a result, a verification failure can cause an error response in different function calls depending on the implementation.

The following sections provide example of verification checks that can be implemented as part of the update process.

4.3.1 Manifest verification

Before processing the content of the manifest, the implementation must verify that the manifest is valid, and authentic. This is typically achieved using a digital signature on the manifest, that can be verified by a trust anchor that is associated with the component.

The manifest must conform to a format that is expected by the implementation. It is recommended that the implementation treats unexpected manifest content as an error.

The manifest describes the type of device, and component, that the firmware is for. The implementation must check that this information matches the device and component being updated.

The manifest provides the version of the new firmware image. The implementation must only install a later version of firmware than is currently installed.

The manifest can provide information about dependencies on other firmware images. The implementation must only install the new firmware if its dependencies are satisfied. See *Dependencies*.

> **Implementation note**
>
> In a trusted-client implementation of the Firmware Update API, these steps can be carried out by the update client, and no verification is done by the implementation. See *Trusted client* on page 24.

4.3.2 Firmware image verification

Before installation, the firmware integrity must be verified. This can be done by checking that a hash of the firmware image matches the associated value in the manifest, or by checking that a provided image signature matches the firmware image using the trust anchor associated with the component.

In a system that implements *secure boot*, the firmware verification processes that occur during firmware update do not replace the requirement for the bootloader to ensure that only correctly authorized firmware can execute on the device.

The implementation is permitted to defer all of the verification of the manifest and firmware image to the bootloader. However, it is recommended that as much verification as possible is carried out before rebooting the system. This reduces the loss of system availability during a reboot, or the cost of storing the firmware image, when it can be determined ahead of time that the update will fail at least one verification check. This recommendation is also made for systems which repeat the verification in the bootloader, prior to final installation and execution of the new firmware.

4.4 Dependencies

A firmware image can have a dependency on another component's firmware image. When a firmware image has a dependency it cannot be installed until all of its dependencies are satisfied.

A dependency can be satisfied by a firmware image that is already installed, or by a firmware image that is installed at the same time as the dependent image. In the latter case, both images must be prepared, and in CANDIDATE state, before the `install` operation. If new firmware images for multiple components are inter-dependent, then the components must be installed at the same time. The *Multiple components with dependent images* on page 67 example shows how this can be done.

Dependencies are described in the firmware image manifest. It is the responsibility of the update client to update components in an order that ensures that dependencies are met during the installation process. Typically, the firmware creator and update server ensure that firmware image updates are presented to the update client in an appropriate order. In more advanced systems, a manifest might provide the update client with sufficient information to determine dependencies and installation order of multiple components itself.

4.5 Update client operation

A typical sequence of activity relating to a firmware update within a device is as follows:

1. Query the current component status, to determine if an update is required
2. Obtain the required manifests and firmware images for the update
3. Validate the manifest
4. Store the firmware image
5. Verify the firmware image
6. Invoke the updated firmware image
7. Clean up any outdated stored firmware image

The design of the Firmware Update API offers functions for these actions.

The activity does not always follow this sequence in order. For example,

- To support devices with constrained download bandwidth, the interface permits an implementation to retain a partially stored firmware image across a system restart. The transfer of the image to the update service can be resumed after the update client has determined the component status.
- For components where the manifest and image are bundled together, the image will be stored prior to verification of the manifest data.
- Some components require execution of the new image to complete verification of the update functionality, before committing to the update.

4.5.1 Querying installed firmware

Each component has a local component identifier. Component queries are based on the component identifier.

The update client calls `psa_fwu_query()` with each component identifier to retrieve information about the component firmware. This information is reported in a `psa_fwu_component_info_t` object, and includes the state of the component, and version of the current active firmware.

If a component state is not READY, the update client should proceed with the appropriate operations to continue or abandon the update that is in progress.

4.5.2 Preparing a new firmware image

To start this process, the component must be in READY state.

To prepare a new firmware image for a component, the update client calls `psa_fwu_start()`. For components with a detached manifest, the manifest data is passed as part of the call to `psa_fwu_start()`. The implementation can verify the manifest at this point, or can defer verification until later in the process.

The update client can now transfer the firmware image data to the firmware store by calling `psa_fwu_write()` one or more times. In systems with sufficient resources, the firmware image can be transferred in a single call. In systems with limited RAM, the update client can transfer the image incrementally, and specify the location of the provided data within the overall firmware image.

When all of the firmware image has been transferred to the update service, the update client calls `psa_fwu_finish()` to complete the preparation of the firmware image. The implementation can verify the manifest and verify the image at this point, or can defer this until later in the process.

If preparation is successful, the component is now in CANDIDATE state.

To abandon a component update at any stage during the image preparation, the update client calls `psa_fwu_cancel()`, and the `psa_fwu_clean()` to remove the abandoned firmware image.

Multi-component updates

A system with multiple components might sometimes require that more than one component is updated atomically.

To update multiple components atomically, all of the new firmware images must be prepared before proceeding to the installation step.

4.5.3 Installing the new firmware image

Once the images have been prepared, the update client calls `psa_fwu_install()` to begin the installation process. This operation will apply to all components in CANDIDATE state. The implementation will complete the verification of the manifest data at this point, and can also verify the new firmware image.

Invoking the new firmware image can require part, or all, of the system to be restarted. If this is required, the affected components will be in STAGED state, and the call to `psa_fwu_install()` returns a status code that informs the update client of the action required.

If a system restart is required, the update client can call `psa_fwu_request_reboot()`. If a component restart is required, this requires an IMPLEMENTATION DEFINED action by the update client.

When the update requires a system reboot, the bootloader will perform additional manifest and firmware image verification, prior to invoking the new firmware. On restart, the update client must query the component status to determine the result of the installation operation within the bootloader.

If the installation succeeds, the components will be in TRIAL or UPDATED state.

4.5.4 Testing the new firmware image

Some components need to execute the new firmware to verify the updated functionality, before accepting the new firmware. For systems that implement a rollback-prevention policy, the testing is done with the component in TRIAL state. The tests are run immediately after the update, and results used to determine whether to accept or reject the update.

The update client reports a successful test result by calling `psa_fwu_accept()`. In an atomic, multi-component update, this will apply to all of the components in the update. The components will now be in UPDATED state.

The update client reports a test failure by calling `psa_fwu_reject()`. In an atomic, multi-component update, this will apply to all of the components in the update. Rolling back to the previous firmware can require part, or all, of the system to be restarted. If this is required, the affected components will be in REJECTED state, and the call to `psa_fwu_reject()` returns a status code that informs the update client of the action required. If a restart is not required, then following the call to `psa_fwu_reject()`, the components will now be in FAILED state.

The updated firmware is automatically rejected if the system restarts while a component is in TRIAL state.

> **Implementation note**
>
> Where possible, it is recommended that a firmware update can be accepted by the system prior to executing the new firmware. This reduces the complexity of the firmware update process, and reduces risks related to firmware rollback. However, for complex devices that require very reliable, remote update, support for in-field testing of new firmware can be important.

4.5.5 Cleaning up the firmware store

After a successful, failed, or abandoned update, the storage containing the inactive firmware image needs to be reclaimed for reuse. The update client calls to `psa_fwu_clean()` to do this.

> **Rationale**
>
> Erasing non-volatile storage can be a high-latency operation. In some systems, this activity might block other memory i/o operations, including code execution. Isolating the erase activity within the call to `psa_fwu_clean()` enables an update client to manage when such disruptive actions take place.

4.6 Bootloader operation

When the bootloader is involved in the firmware installation process, it does more than select and verify a firmware image to execute. This section describes the responsibilities of the bootloader for the type of component depicted in *State transitions* on page 28.

4.6.1 Determine firmware state

The bootloader checks the state of each component:

- If there are any STAGED components, proceed to install them. See *Install components*.
- If there are any TRIAL or REJECTED components, proceed to roll them back. See *Rollback trial components*.
- Otherwise, proceed to boot the firmware. See *Authenticate and execute active firmware* on page 37.

Note:

The design of the state model prevents the situation in which there is a STAGED component at the same time as a TRIAL or REJECTED component.

4.6.2 Install components

If the implementation defers verification of the updated firmware to the bootloader, or the bootloader does not trust the update service (see *Untrusted service* on page 23), the bootloader must verify all components that are in STAGED state. If verification fails, all STAGED components are set to FAILED state, and the reason for failure stored for retrieval by the update client. The bootloader proceeds to boot the existing firmware. See *Authenticate and execute active firmware* on page 37.

The new firmware images for all STAGED components are installed as the *active* firmware. If the installation fails for any component, the previous images are restored for all components, the components are set to FAILED state, and the reason for failure stored for retrieval by the update client. The bootloader proceeds to boot the existing firmware. See *Authenticate and execute active firmware* on page 37.

If the components require the new firmware to be tested before acceptance, the bootloader stores the previously *active* firmware images as backup, for recovery if the new firmware images fail. The components are set to TRIAL state, and the bootloader proceeds to boot the new firmware. See *Authenticate and execute active firmware* on page 37.

Otherwise, the components are set to UPDATED state, and the bootloader proceeds to boot the new firmware. See *Authenticate and execute active firmware* on page 37.

4.6.3 Rollback trial components

If the system restarts while components are in TRIAL state, or after an update has been explicitly rejected by the update client, the bootloader restores the previous firmware images for the affected components as the *active* image. These images were stored as a backup during the installation of the firmware being tested (see *Install components*).

The components are set to FAILED state, and the reason for failure stored for retrieval by the update client. This will result in the firmware images, that failed the trial, being erased when the update client carries out a `clean` operation.

The bootloader proceeds to boot the previous firmware. See *Authenticate and execute active firmware* on page 37.

4.6.4 Authenticate and execute *active* firmware

In a system that implements a *secure boot* policy, the bootloader verifies the integrity and authenticity of the *active* firmware. If this verification fails, the result is IMPLEMENTATION DEFINED, for example:

- The bootloader can rollback to a previous firmware image, if one is available and policy permits.
- The bootloader can run a special recovery firmware image, if this is provided by the system.
- The device can become non-functional and unrecoverable.

Otherwise, the bootloader will complete initialization and transfer execution to the *active* firmware image.

4.7 Sample sequence during firmware update

Figure 8 on page 38 is a detailed sequence diagram shows how the overall logic could be implemented.

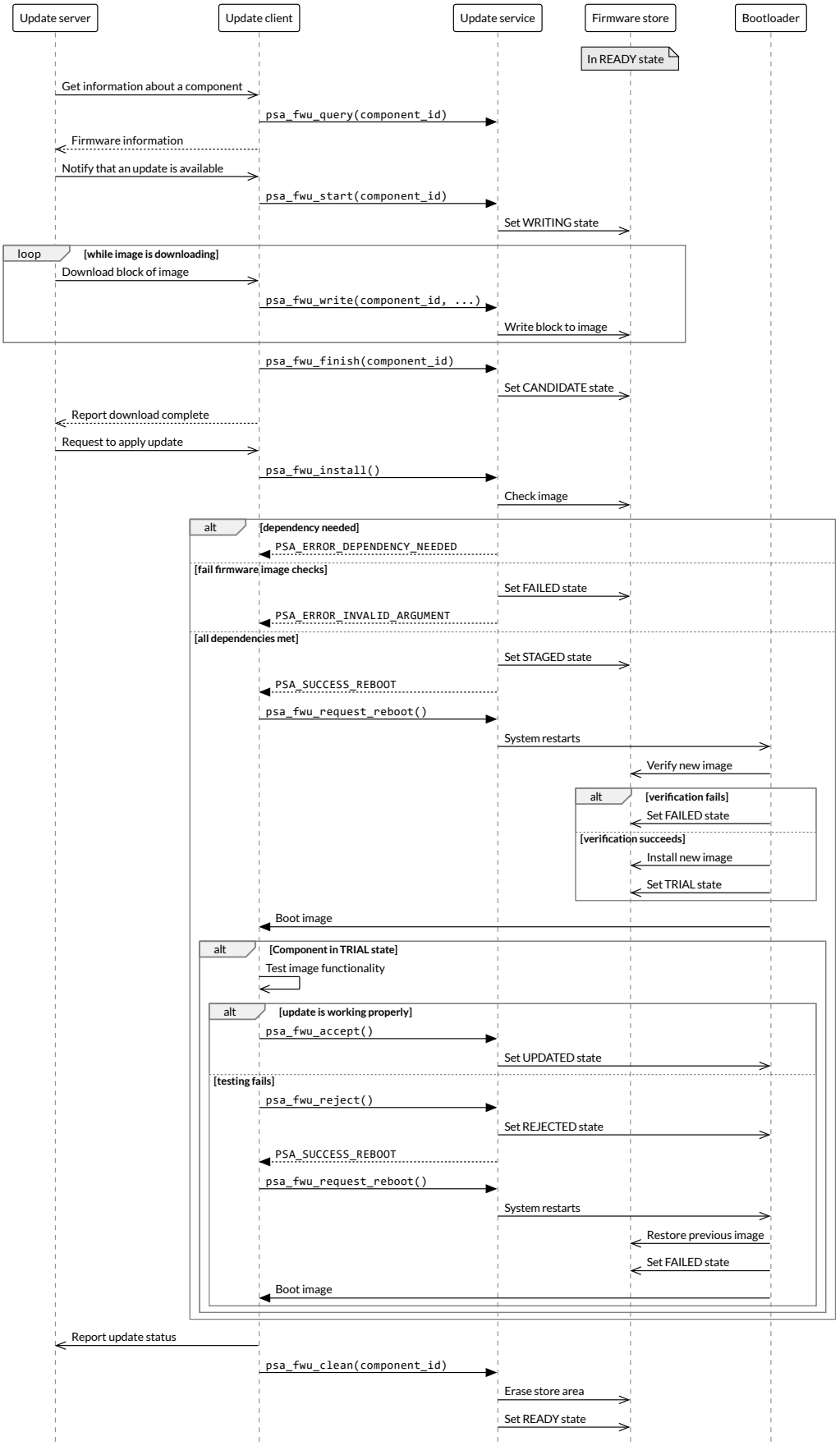


Figure 8 A sequence diagram showing an example flow

5 API reference

To enable implementation optimization for constrained devices, the Firmware Update API does not require binary compatibility between different implementations. The Firmware Update API is defined as a source-level interface, and applications that target this interface will typically need to be recompiled for different implementations.

5.1 API conventions

The interface in this specification is defined in terms of C macros, data types, and functions.

5.1.1 Identifier names

All of the identifiers defined in the Firmware Update API begin with the prefix `psa_`, for types and functions, or `PSA_` for macros.

Future versions of this specification will use the same prefix for additional API elements. It is recommended that applications and implementations do not use this prefix for their own identifiers, to avoid a potential conflict with a future version of the Firmware Update API.

5.1.2 Basic types

This specification makes use of standard C data types, including the fixed-width integer types from the ISO C99 specification update [C99]. The following standard C types are used:

`int32_t`	a 32-bit signed integer
`uint8_t`	an 8-bit unsigned integer
`uint16_t`	a 16-bit unsigned integer
`uint32_t`	a 32-bit unsigned integer
`size_t`	an unsigned integer large enough to hold the size of an object in memory

5.1.3 Data types

Integral types are defined for specific API elements to provide clarity in the interface definition, and to improve code readability. For example, `psa_fwu_component_t` and `psa_status_t`.

Structure types are declared using `typedef` instead of a `struct` tag, also to improve code readability.

Fully-defined types must be declared exactly as defined in this specification. Types that are not fully defined in this specification must be defined by an implementation. See *Implementation-specific types* on page 41.

5.1.4 Constants

Constant values are defined using C macros. Constants defined in this specification have names that are all upper-case.

A constant macro evaluates to a compile-time constant expression.

5.1.5 Functions

Functions defined in this specification have names that are all lower-case.

An implementation is permitted to declare any API function with `static inline` linkage, instead of the default `extern` linkage.

An implementation is permitted to also define a function-like macro with the same name as a function in this specification. If an implementation defines a function-like macro for a function from this specification, then:

- The implementation must also provide a definition of the function. This enables an application to take the address of a function defined in this specification.
- The function-like macro must expand to code that evaluates each of its arguments exactly once, as if the call was made to a C function. This enables an application to safely use arbitrary expressions as arguments to a function defined in this specification.

If a non-pointer argument to a function has an invalid value (for example, a value outside the domain of the function), then the function will normally return an error, as specified in the function definition.

If a pointer argument to a function has an invalid value (for example, a pointer outside the address space of the program, or a null pointer), the result is IMPLEMENTATION DEFINED. See also *Pointer conventions* on page 41.

5.1.6 Return status

All functions return a status indication of type `psa_status_t`. This is an integer value, with `0` (`PSA_SUCCESS`), or a positive value, indicating successful operation, and other values indicating errors.

Unless specified otherwise, if multiple error conditions apply, an implementation is free to return any of the applicable error codes.

If the behavior is undefined — for example, if a function receives an invalid pointer as a parameter — this specification does not require that the function will return an error. Implementations are encouraged to return an error or halt the application in a manner that is appropriate for the platform if the undefined behavior condition can be detected. However, application developers need to be aware that undefined behavior conditions cannot be detected in general.

5.1.7 Pointer conventions

Unless explicitly stated in the documentation of a function, all pointers must be valid pointers to an object of the specified type.

A parameter is considered to be a buffer if it points to an array of bytes. A buffer parameter always has the type `uint8_t *` or `const uint8_t *`, and always has an associated parameter indicating the size of the array. Note that a parameter of type `void *` is never considered a buffer.

All parameters of pointer type must be valid non-null pointers, unless the pointer is to a buffer of length `0` or the function's documentation explicitly describes the behavior when the pointer is null.

Pointers to input parameters can be in read-only memory. Output parameters must be in writable memory.

Unless otherwise documented, the content of output parameters is not defined when a function returns an error status. It is recommended that implementations set output parameters to safe defaults to reduce risk, in case the caller does not properly handle all errors.

5.1.8 Implementation-specific types

This specification defines a number of implementation-specific types, which represent objects whose content depends on the implementation. These are defined as C `typedef` types in this specification, with a comment */* *implementation-defined type* */* in place of the underlying type definition. For some types the specification constrains the type, for example, by requiring that the type is a `struct`, or that it is convertible to and from an unsigned integer. In the implementation's version of the Firmware Update API header file, these types need to be defined as complete C types so that objects of these types can be instantiated by application code.

Applications that rely on the implementation specific definition of any of these types might not be portable to other implementations of this specification.

5.2 Header file

The header file for the Firmware Update API has the name `psa/update.h`. All of the interface elements that are provided by an implementation must be visible to an application program that includes this header file.

```
#include "psa/update.h"
```

Implementations must provide their own version of the `psa/update.h` header file. *Example header file* on page 62 provides an incomplete, example header file which includes all of the Firmware Update API elements.

This Firmware Update API uses some of the common status codes that are defined by *PSA Certified Status code API* [PSA-STAT] as part of the `psa/error.h` header file. Applications are not required to explicitly include the `psa/error.h` header file when using these status codes with the Firmware Update API. See *Status codes* on page 43.

> **Note:**
>
> The common error codes in `psa/error.h` were previously defined in *Arm® Platform Security Architecture Firmware Framework* [PSA-FFM].

5.2.1 Required functions

All of the API elements defined in *API reference* on page 39 must be present for an implementation to claim compliance with this spec.

Mandatory function implementations cannot simply return `PSA_ERROR_NOT_SUPPORTED`. Optional functions must be present, but are permitted to always return `PSA_ERROR_NOT_SUPPORTED`.

The following functions are mandatory for all implementations:

- `psa_fwu_query()`
- `psa_fwu_start()`
- `psa_fwu_write()`
- `psa_fwu_finish()`
- `psa_fwu_install()`
- `psa_fwu_cancel()`
- `psa_fwu_clean()`

If the implementation includes components that use the STAGED state, the following functions are also mandatory:

- `psa_fwu_reject()`

If the implementation includes components that use the TRIAL state, the following functions are also mandatory:

- `psa_fwu_reject()`
- `psa_fwu_accept()`

If the implementation includes components that require a system restart, the following functions are also mandatory:

- `psa_fwu_request_reboot()`

5.3 Library management

5.3.1 Library version

`PSA_FWU_API_VERSION_MAJOR` (macro)

The major version of this implementation of the Firmware Update API.

```
#define PSA_FWU_API_VERSION_MAJOR 1
```

PSA_FWU_API_VERSION_MINOR (macro)

The minor version of this implementation of the Firmware Update API.

```
#define PSA_FWU_API_VERSION_MINOR 0
```

5.4 Status codes

The Firmware Update API uses the status code definitions that are shared with the other PSA Certified APIs. The Firmware Update API also provides some Firmware Update API-specific status codes, see *Error codes specific to the Firmware Update API* and *Success status codes specific to the Firmware Update API* on page 44.

5.4.1 Common status codes

The following elements are defined in `psa/error.h` from [PSA-STAT] (previously defined in [PSA-FFM]):

```
typedef int32_t psa_status_t;

#define PSA_SUCCESS ((psa_status_t)0)

#define PSA_ERROR_NOT_PERMITTED          ((psa_status_t)-133)
#define PSA_ERROR_NOT_SUPPORTED          ((psa_status_t)-134)
#define PSA_ERROR_INVALID_ARGUMENT       ((psa_status_t)-135)
#define PSA_ERROR_BAD_STATE              ((psa_status_t)-137)
#define PSA_ERROR_DOES_NOT_EXIST         ((psa_status_t)-140)
#define PSA_ERROR_INSUFFICIENT_MEMORY    ((psa_status_t)-141)
#define PSA_ERROR_INSUFFICIENT_STORAGE   ((psa_status_t)-142)
#define PSA_ERROR_COMMUNICATION_FAILURE ((psa_status_t)-145)
#define PSA_ERROR_STORAGE_FAILURE        ((psa_status_t)-146)
#define PSA_ERROR_INVALID_SIGNATURE      ((psa_status_t)-149)
```

Implementation note

An implementation is permitted to define these interface elements within the `psa/update.h` header, or to define them via inclusion of a `psa/error.h` header file that is shared with the implementation of other PSA Certified APIs.

5.4.2 Error codes specific to the Firmware Update API

These error codes are defined in `psa/update.h`.

PSA_ERROR_DEPENDENCY_NEEDED (macro)

A status code that indicates that the firmware of another component requires updating.

```
#define PSA_ERROR_DEPENDENCY_NEEDED ((psa_status_t)-156)
```

This error indicates that the firmware image depends on a newer version of the firmware for another component. The firmware of the other component must be updated before this firmware image can be installed, or both components must be updated at the same time.

See *Dependencies* on page 32 and *Multi-component updates* on page 34.

PSA_ERROR_FLASH_ABUSE (macro)

A status code that indicates that the system is limiting i/o operations to avoid rapid flash exhaustion.

```
#define PSA_ERROR_FLASH_ABUSE ((psa_status_t)-160)
```

Excessive i/o operations can cause certain types of flash memories to wear out, resulting in storage device failure. This error code can be used by a system that detects unusually high i/o activity, to reduce the risk of flash exhaustion.

The time-out period is IMPLEMENTATION DEFINED.

PSA_ERROR_INSUFFICIENT_POWER (macro)

A status code that indicates that the system does not have enough power to carry out the request.

```
#define PSA_ERROR_INSUFFICIENT_POWER ((psa_status_t)-161)
```

A function can return this error code if it determines that there is not sufficient power or energy available to reliably complete the operation.

Operations that update the state of the firmware can require significant energy to reprogram the non-volatile memories. It is recommended to wait until sufficient energy is available for the update process, rather than failing to update the firmware and leaving the device temporarily or permanently non-operational.

5.4.3 Success status codes specific to the Firmware Update API

These success codes are defined in `psa/update.h`.

PSA_SUCCESS_REBOOT (macro)

The action was completed successfully and requires a system reboot to complete installation.

```
#define PSA_SUCCESS_REBOOT ((psa_status_t)+1)
```

PSA_SUCCESS_RESTART (macro)

The action was completed successfully and requires a restart of the component to complete installation.

```
#define PSA_SUCCESS_RESTART ((psa_status_t)+2)
```

5.5 Firmware components

5.5.1 Component identifier

psa_fwu_component_t (typedef)

Firmware component type identifier.

```
typedef uint8_t psa_fwu_component_t;
```

A value of type `psa_fwu_component_t` identifies a firmware component on this device. This is used to specify which component a function call applies to.

In systems that only have a single component, it is recommended that the caller uses the value `0` in calls that require a component identifier.

5.5.2 Component version

psa_fwu_image_version_t (struct)

Version information about a firmware image.

```
typedef struct psa_fwu_image_version_t {
    uint8_t major;
    uint8_t minor;
    uint16_t patch;
    uint32_t build;
} psa_fwu_image_version_t;
```

Fields

`major`	The major version of an image.
`minor`	The minor version of an image. If the image has no minor version then this field is set to `0`.
`patch`	The revision or patch version of an image. If the image has no such version then this field is set to `0`.
`build`	The build number of an image. If the image has no such number then this field is set to `0`.

5.5.3 Component states

Each of the component states defined in *State model* on page 27 has a corresponding identifier in the API. These are used to indicate the state of a component, in the `state` field of a `psa_fwu_component_info_t` structure returned by a call to `psa_fwu_query()`.

PSA_FWU_READY (macro)

The READY state: the component is ready to start another update.

```
#define PSA_FWU_READY 0u
```

In this state, the update client can start a new firmware update, by calling `psa_fwu_start()`.

PSA_FWU_WRITING (macro)

The WRITING state: a new firmware image is being written to the firmware store.

```
#define PSA_FWU_WRITING 1u
```

In this state, the update client transfers the firmware image to the firmware store, by calling `psa_fwu_write()`.

When all of the image has been transferred, the update client marks the new firmware image as ready for installation, by calling `psa_fwu_finish()`.

The update client can abort an update that is in this state, by calling `psa_fwu_cancel()`.

PSA_FWU_CANDIDATE (macro)

The CANDIDATE state: a new firmware image is ready for installation.

```
#define PSA_FWU_CANDIDATE 2u
```

In this state, the update client starts the installation process of the component, by calling `psa_fwu_install()`.

The update client can abort an update that is in this state, by calling `psa_fwu_cancel()`.

PSA_FWU_STAGED (macro)

The STAGED state: a new firmware image is queued for installation.

```
#define PSA_FWU_STAGED 3u
```

A system reboot, or component restart, is required to complete the installation process.

The update client can abort an update that is in this state, by calling `psa_fwu_reject()`.

> **Note:**
>
> This state is transient — on a reboot the system will attempt to install the new firmware image.

PSA_FWU_FAILED (macro)

The FAILED state: a firmware update has been cancelled or has failed.

```
#define PSA_FWU_FAILED 4u
```

The `error` field of the `psa_fwu_component_info_t` structure will contain an status code indicating the reason for the failure.

The failed firmware image needs to be erased using a call to `psa_fwu_clean()` before another update can be started.

PSA_FWU_TRIAL (macro)

The TRIAL state: a new firmware image requires testing prior to acceptance of the update.

```
#define PSA_FWU_TRIAL 5u
```

In this state, the update client calls `psa_fwu_accept()` or `psa_fwu_reject()` to either accept or reject the new firmware image.

It is recommended that the new firmware is tested for correct operation, before accepting the update. This is particularly important to for systems that implement an update policy that prevents rollback to old firmware versions.

> **Note:**
>
> This state is transient — on a reboot, a component in this state will be rolled back to the previous firmware image.

PSA_FWU_REJECTED (macro)

The REJECTED state: a new firmware image has been rejected after testing.

```
#define PSA_FWU_REJECTED 6u
```

A system reboot, or component restart, is required to complete the process of reverting to the previous firmware image.

> **Note:**
>
> This state is transient — on a reboot, a component in this state will be rolled back to the previous firmware image.

PSA_FWU_UPDATED (macro)

The UPDATED state: a firmware update has been successful, and the new image is now *active*.

```
#define PSA_FWU_UPDATED 7u
```

The previous firmware image needs to be erased using a call to `psa_fwu_clean()` before another update can be started.

5.5.4 Component flags

PSA_FWU_FLAG_VOLATILE_STAGING (macro)

Flag to indicate whether the image data in the component staging area is discarded at system reset.

```
#define PSA_FWU_FLAG_VOLATILE_STAGING 0x00000001u
```

If set, then image data written to the staging area is discarded after a system reset. If the system restarts while the component in is WRITING or CANDIDATE state, the component will be in the READY state after the restart.

If not set, then image data written to the staging area is guaranteed to exist after a system reset.

PSA_FWU_FLAG_ENCRYPTION (macro)

Flag to indicate whether a firmware component expects encrypted images during an update.

```
#define PSA_FWU_FLAG_ENCRYPTION 0x00000002u
```

If set, then the firmware image for this component must be encrypted when installing.

If not set, then the firmware image for this component must not be encrypted when installing.

5.5.5 Component information

psa_fwu_impl_info_t (typedef)

The implementation-specific data in the component information structure.

```
typedef struct { /* implementation-defined type */ } psa_fwu_impl_info_t;
```

The members of this data structure are IMPLEMENTATION DEFINED. This can be an empty data structure.

psa_fwu_component_info_t (struct)

Information about the firmware store for a firmware component.

```
typedef struct psa_fwu_component_info_t {
    uint8_t state;
    psa_status_t error;
    psa_fwu_image_version_t version;
    uint32_t max_size;
```

(continues on next page)

(continued from previous page)

```
    uint32_t flags;
    uint32_t location;
    psa_fwu_impl_info_t impl;
} psa_fwu_component_info_t;
```

Fields

`state`	State of the component. This is one of the values defined in *Component states* on page 46.
`error`	Error for *second* image when store state is REJECTED or FAILED.
`version`	Version of *active* image.
`max_size`	Maximum image size in bytes.
`flags`	Flags that describe extra information about the firmware component. See *Component flags* on page 48 for defined flag values.
`location`	Implementation-defined image location.
`impl`	Reserved for implementation-specific usage. For example, provide information about image encryption or compression.

Description

The attributes of a component are retrieved using a call to `psa_fwu_query()`.

Rationale

When a component is in a state that is not READY, there is a *second* image, or partial image, present in the firmware store. The Firmware Update API provides no mechanism to report the version of the *second* image, for the following reasons:

- During preparation of a new firmware image, the implementation is not required to extract version information from the firmware image manifest:
  - This information might not be available if the firmware image has not been completely written.
  - The update service might not be capable of extracting the version information. For example, in the untrusted-service deployment model, verification of the manifest can be deferred until the image is installed. See *Untrusted service* on page 23.

  If the version of an image that is being prepared is required by the update client, the update client must maintain this information locally.
- In TRIAL or REJECTED states, the *second* image is the previously installed firmware, which is required in case of rollback. Reporting the version of this, while interesting, is of no value to the update client.
- In UPDATED or FAILED states, the *second* image needs to be erased. The version of the image data in this state has no effect on the behavior of the update client.

`psa_fwu_query` (function)

Retrieve the firmware store information for a specific firmware component.

```
psa_status_t psa_fwu_query(psa_fwu_component_t component,
                           psa_fwu_component_info_t *info);
```

Parameters

`component`	Firmware component for which information is requested.
`info`	Output parameter for component information.

Returns: `psa_status_t`

Result status.

`PSA_SUCCESS`	Component information has been returned in the `psa_fwu_component_t` object at `*info`.
`PSA_ERROR_DOES_NOT_EXIST`	There is no firmware component with the specified Id.
`PSA_ERROR_NOT_PERMITTED`	The caller is not authorized to call this function.

Description

This function is used to query the status of a component.

The caller is expected to know the component identifiers for all of the firmware components. This information might be built into the update client, provided by configuration data, or provided alongside the firmware images from the update server.

5.6 Firmware installation

Each of the component operations defined in *State model* on page 27 has a corresponding function in the API, described in sections §5.6.1 to §5.6.3 on page 60.

5.6.1 Image preparation

The following functions are used to prepare a new firmware image in the component's firmware store. They act on a single component, specified by a component identifier parameter.

`psa_fwu_start` (function)

Begin a firmware update operation for a specific firmware component.

```
psa_status_t psa_fwu_start(psa_fwu_component_t component,
                           const void *manifest,
                           size_t manifest_size);
```

Parameters

`component`	Identifier of the firmware component to be updated.
`manifest`	A pointer to a buffer containing a detached manifest for the update. If the manifest is bundled with the firmware image, `manifest` must be `NULL`.
`manifest_size`	The size of the detached manifest. If the manifest is bundled with the firmware image, `manifest_size` must be `0`.

Returns: `psa_status_t`

Result status.

`PSA_SUCCESS`	Success: the component is now in WRITING state, and ready for the new image to be transferred using `psa_fwu_write()`.
`PSA_ERROR_DOES_NOT_EXIST`	There is no firmware component with the specified Id.
`PSA_ERROR_BAD_STATE`	The component is not in the READY state.
`PSA_ERROR_NOT_PERMITTED`	The following conditions can result in this error: • The caller is not authorized to call this function. • The provided manifest is valid, but fails to comply with the update service's firmware update policy.
`PSA_ERROR_INVALID_SIGNATURE`	A signature or integrity check on the manifest has failed.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • The provided manifest is unexpected, or invalid. • A detached manifest was expected, but none was provided.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_INSUFFICIENT_STORAGE`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_STORAGE_FAILURE`	

Description

This function is used to begin the process of preparing a new firmware image for a component, optionally providing a detached manifest. On success, the component is in WRITING state, and the update client can call `psa_fwu_write()` to transfer the new firmware image.

If the firmware image *manifest* is detached from the firmware image, it must be provided to the update service using the `manifest` and `manifest_size` parameters in `psa_fwu_start()`.

If a detached manifest is expected by the update service for a firmware component, but none is provided, `psa_fwu_start()` returns `PSA_ERROR_INVALID_ARGUMENT`. If a detached manifest is provided for a component which expects the manifest to be bundled with the image, `psa_fwu_start()` returns `PSA_ERROR_INVALID_ARGUMENT`.

To abandon an update that has been started, call `psa_fwu_cancel()`, and then `psa_fwu_clean()`.

PSA_FWU_MAX_WRITE_SIZE (macro)

The maximum permitted size for `block` in `psa_fwu_write()`, in bytes.

```
#define PSA_FWU_MAX_WRITE_SIZE /* implementation-defined value */
```

The specific value is IMPLEMENTATION DEFINED, and is greater than `0`.

> **Implementation note**
>
> This value is the maximum size for transferring data to the update service. The reasons for selecting a particular value can include the following:
>
> - The size of the available RAM buffer within the update service used for storing the data into the firmware store.
> - A value that is optimized for storing the data in the firmware store, for example, a multiple of the block-size of the storage media.

psa_fwu_write (function)

Write a firmware image, or part of a firmware image, to its staging area.

```
psa_status_t psa_fwu_write(psa_fwu_component_t component,
                           size_t image_offset,
                           const void *block,
                           size_t block_size);
```

Parameters

`component`	Identifier of the firmware component being updated.
`image_offset`	The offset of the data block in the whole image. The offset of the first block is `0`.
`block`	A buffer containing a block of image data. This can be a complete image or part of the image.
`block_size`	Size of `block`, in bytes. `block_size` must not be greater than `PSA_FWU_MAX_WRITE_SIZE`.

Returns: `psa_status_t`

Result status.

`PSA_SUCCESS`	Success: the data in `block` has been successfully stored.
`PSA_ERROR_DOES_NOT_EXIST`	There is no firmware component with the specified Id.
`PSA_ERROR_BAD_STATE`	The component is not in the WRITING state.
`PSA_ERROR_NOT_PERMITTED`	The caller is not authorized to call this function.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • The parameter `block_size` is greater than `PSA_FWU_MAX_WRITE_SIZE`.

	• The parameter `block_size` is `0`. • The image region specified by `image_offset` and `block_size` does not lie inside the supported image storage.
`PSA_ERROR_FLASH_ABUSE`	The system has temporarily limited i/o operations to avoid rapid flash exhaustion.
`PSA_ERROR_INVALID_SIGNATURE`	A signature or integrity check on the provided data has failed.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_INSUFFICIENT_STORAGE`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_STORAGE_FAILURE`	

Description

This function is used to transfer all, or part, of a firmware image to the component's firmware store. On success, the component remains in WRITING state. Once all of the firmware image has been written to the store, a call to `psa_fwu_finish()` is required to continue the installation process.

If the image size is less than or equal to `PSA_FWU_MAX_WRITE_SIZE`, the caller can provide the entire image in one call.

If the image size is greater than `PSA_FWU_MAX_WRITE_SIZE`, the caller must provide the image in parts, by calling `psa_fwu_write()` multiple times with different data blocks.

Write operations can take an extended execution time on flash memories. The caller can provide data in blocks smaller than `PSA_FWU_MAX_WRITE_SIZE` to reduce the time for each call to `psa_fwu_write()`.

When data is written in multiple calls to `psa_fwu_write()`, it is the caller's responsibility to account for how much data is written at which offset within the image. If no persistent storage is directly available for the caller to perform accounting, then the caller can use a different storage mechanism, such as the *PSA Certified Secure Storage API* [PSA-SS].

On error, the component will typically remain in WRITING state. In this situation, it is not possible to determine how much of the data in `block` has been written to the staging area. It is IMPLEMENTATION DEFINED whether repeating the write operation again with the same data at the same offset will correctly store the data to the staging area.

If the data fails an integrity check, the implementation is permitted to transition the component to the FAILED state. From this state, the caller is required to use `psa_fwu_clean()` to return the store to READY state before attempting another firmware update.

To abandon an update that has been started, call `psa_fwu_cancel()` and then `psa_fwu_clean()`.

`psa_fwu_finish` (function)

Mark a firmware image in the staging area as ready for installation.

```
psa_status_t psa_fwu_finish(psa_fwu_component_t component);
```

Parameters

`component`	Identifier of the firmware component to install.

Returns: `psa_status_t`

Result status.

`PSA_SUCCESS`	The operation completed successfully: the component is now in CANDIDATE state.
`PSA_ERROR_DOES_NOT_EXIST`	There is no firmware component with the specified Id.
`PSA_ERROR_BAD_STATE`	The component is not in the WRITING state.
`PSA_ERROR_INVALID_SIGNATURE`	A signature or integrity check for the image has failed.
`PSA_ERROR_INVALID_ARGUMENT`	The firmware image is not valid.
`PSA_ERROR_NOT_PERMITTED`	The following conditions can result in this error: • The caller is not authorized to call this function. • The firmware image is valid, but fails to comply with the update service's firmware update policy. For example, the update service can deny the installation of older versions of firmware (rollback prevention).
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_INSUFFICIENT_STORAGE`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_STORAGE_FAILURE`	

Description

This function is used to complete the preparation of a new firmware image for a component. On success, the component is in CANDIDATE state, and the update client calls `psa_fwu_install()` to initiate the installation process.

The validity, authenticity and integrity of the image can be checked during this operation. If this verification fails, the component is transitioned to the FAILED state. From the FAILED state, the caller is required to use `psa_fwu_clean()` to return the component to READY state before attempting another firmware update.

Dependencies on other firmware components are not checked as part of `psa_fwu_finish()`. If the implementation provides dependency verification, this is done as part of `psa_fwu_install()`, or during installation at reboot.

To abandon an update that is in CANDIDATE state, call `psa_fwu_cancel()` and then `psa_fwu_clean()`.

psa_fwu_cancel (function)

Abandon an update that is in WRITING or CANDIDATE state.

```
psa_status_t psa_fwu_cancel(psa_fwu_component_t component);
```

Parameters

component	Identifier of the firmware component to be cancelled.

Returns: psa_status_t

Result status.

PSA_SUCCESS	Success: the new firmware image is rejected. The component is now in FAILED state.
PSA_ERROR_DOES_NOT_EXIST	There is no firmware component with the specified Id.
PSA_ERROR_BAD_STATE	The component is not in the WRITING or CANDIDATE state.
PSA_ERROR_NOT_PERMITTED	The caller is not authorized to call this function.

Description

This function is used when the caller wants to abort an incomplete update process, for a component in WRITING or CANDIDATE state. This will discard the uninstalled image or partial image, and leave the component in FAILED state. To prepare for a new update after this, call psa_fwu_clean().

psa_fwu_clean (function)

Prepare the component for another update.

```
psa_status_t psa_fwu_clean(psa_fwu_component_t component);
```

Parameters

component	Identifier of the firmware component to tidy up.

Returns: psa_status_t

Result status.

PSA_SUCCESS	Success: the staging area is ready for a new update. The component is now in state READY.
PSA_ERROR_DOES_NOT_EXIST	There is no firmware component with the specified Id.
PSA_ERROR_BAD_STATE	The component is not in the FAILED or UPDATED state.
PSA_ERROR_NOT_PERMITTED	The caller is not authorized to call this function.
PSA_ERROR_INSUFFICIENT_POWER	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_STORAGE_FAILURE	

Description

This function is used to ensure that the component is ready to start another update process, after an update has succeeded, failed, or been rejected.

If the implementation needs to perform long-running operations to erase firmware store memories, it is recommended that this is done as part of `psa_fwu_clean()`, rather than during other operations. This enables the update client to schedule this long-running operation at a time when this is less disruptive to the application.

If this function is called when the component state is FAILED, then the staging area is cleaned, leaving the current *active* image installed.

If this function is called when the component state is UPDATED, then the previously installed image is cleaned, leaving the new *active* image installed.

5.6.2 Image installation

The following functions are used to install prepared firmware images. They act concurrently on all components that have been prepared for installation.

`psa_fwu_install` (function)

Start the installation of all firmware images that have been prepared for update.

```
psa_status_t psa_fwu_install(void);
```

Returns: `psa_status_t`

Result status.

`PSA_SUCCESS`	The installation completed successfully: the affected components are now in TRIAL or UPDATED state.
`PSA_SUCCESS_REBOOT`	The installation has been prepared, but a system reboot is needed to complete the installation. The affected components are now in STAGED state. A system reboot can be requested using `psa_fwu_request_reboot()`.
`PSA_SUCCESS_RESTART`	The installation has been prepared, but the components must be restarted to complete the installation. The affected components are now in STAGED state. The component restart mechanism is IMPLEMENTATION DEFINED.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • An existing installation process is in progress: there is at least one component in STAGED, TRIAL, or REJECTED state. • There is no component in the CANDIDATE state.
`PSA_ERROR_INVALID_SIGNATURE`	A signature or integrity check for the image has failed.
`PSA_ERROR_DEPENDENCY_NEEDED`	A different firmware image must be installed first.
`PSA_ERROR_INVALID_ARGUMENT`	The firmware image is not valid.

PSA_ERROR_NOT_PERMITTED	The following conditions can result in this error: • The caller is not authorized to call this function. • The firmware image is valid, but fails to comply with the update service's firmware update policy. For example, the update service can deny the installation of older versions of firmware (rollback prevention).
PSA_ERROR_INSUFFICIENT_POWER	The system does not have enough power to safely install the firmware.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_STORAGE_FAILURE	

Description

This function starts the installation process atomically on all components that are in CANDIDATE state. This function reports an error if there are no components in this state. If an error occurs when installing any of the images, then none of the images will be installed.

Only one installation process can be in progress at a time. After a successful call to psa_fwu_install(), another call is only permitted once the affected components have transitioned to FAILED or UPDATED state.

Support for concurrent installation of multiple components is IMPLEMENTATION DEFINED. Concurrent installation enables new firmware images that are interdependent to be installed. If concurrent installation is not supported, each new firmware image must be compatible with the current version of other firmware components in the system. Device updates that affect multiple components must be carried out in line with the system capabilities. For example:

- An implementation is permitted to require each component to be installed separately.
- An implementation is permitted to support atomic installation of any combination of components.
- An implementation is permitted to support atomic installation of a specific subset of components, but require other components to be installed individually

The validity, authenticity and integrity of the images can be checked during this operation. If this verification fails, the components are transitioned to the FAILED state. From the FAILED state, the caller is required to use psa_fwu_clean() on each component to return them to the READY state before attempting another firmware update.

Dependencies on other firmware components can be checked as part of psa_fwu_install(). The dependency check is carried out against the version of the prepared image for a component that is in CANDIDATE state, and the *active* image for other components. If this verification fails, then PSA_ERROR_DEPENDENCY_NEEDED is returned, and the components will remain in CANDIDATE state. A later call to psa_fwu_install() can be attempted after preparing a new firmware image for the dependency.

On other error conditions, it is IMPLEMENTATION DEFINED whether the components are all transitioned to FAILED state, or all remain in CANDIDATE state. See *Behavior on error* on page 30.

If a component restart, or system reboot, is required to complete installation then the implementation is permitted to defer verification checks to that point. Verification failures during a reboot will result in the

components being transitioned to FAILED state. The failure reason is recorded in the `error` field in the `psa_fwu_component_info_t` object for each firmware component, which can be queried by the update client after restart.

To abandon an update that is STAGED, before restarting the system or component, call `psa_fwu_reject()` and then `psa_fwu_clean()` on each component.

`psa_fwu_request_reboot` (function)

Requests the platform to reboot.

```
psa_status_t psa_fwu_request_reboot(void);
```

Returns: `psa_status_t`

Result status. It is IMPLEMENTATION DEFINED whether this function returns to the caller.

`PSA_SUCCESS`	The platform will reboot soon.
`PSA_ERROR_NOT_PERMITTED`	The caller is not authorized to call this function.
`PSA_ERROR_NOT_SUPPORTED`	This function call is not implemented.

Description

On success, the platform initiates a reboot, and might not return to the caller.

> **Implementation note**
>
> This function is mandatory in an implementation where one or more components require a system reboot to complete installation.
>
> On other implementations, this function is optional.
>
> See *Required functions* on page 42.

`psa_fwu_reject` (function)

Abandon an installation that is in STAGED or TRIAL state.

```
psa_status_t psa_fwu_reject(psa_status_t error);
```

Parameters

`error`	An application-specific error code chosen by the application. If a specific error does not need to be reported, the value should be 0. On success, this error is recorded in the `error` field of the `psa_fwu_component_info_t` structure corresponding to each affected component.

Returns: `psa_status_t`

Result status.

`PSA_SUCCESS`	Success: the new firmware images are rejected, and the previous firmware is now *active*. The affected components are now in FAILED state.
`PSA_SUCCESS_REBOOT`	The new firmware images are rejected, but a system reboot is needed to complete the rollback to the previous firmware. The affected components are now in REJECTED state. A system reboot can be requested using `psa_fwu_request_reboot()`.
`PSA_SUCCESS_RESTART`	The new firmware images are rejected, but the components must be restarted to complete the rollback to the previous firmware. The affected components are now in REJECTED state. The component restart mechanism is IMPLEMENTATION DEFINED.
`PSA_ERROR_BAD_STATE`	There are no components in the STAGED or TRIAL state.
`PSA_ERROR_NOT_PERMITTED`	The caller is not authorized to call this function.
`PSA_ERROR_NOT_SUPPORTED`	This function call is not implemented.
`PSA_ERROR_INSUFFICIENT_POWER`	The system does not have enough power to safely uninstall the firmware.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_INSUFFICIENT_STORAGE`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_STORAGE_FAILURE`	

Description

This function is used in the following situations:

- When the caller wants to abort an incomplete update process, for components in STAGED state. This will discard the uninstalled images.
- When the caller detects an error in new firmware that is in TRIAL state.

If this function is called when the installation state is STAGED, then the state of affected components changes to FAILED. To prepare for a new update after this, call `psa_fwu_clean()` for each component.

If this function is called when the installation state is TRIAL, then the action depends on whether a reboot or component restart is required to complete the rollback process:

- If a reboot is required, the state of affected components changes to REJECTED and `PSA_SUCCESS_REBOOT` is returned. To continue the rollback process, call `psa_fwu_request_reboot()`.
  After reboot, the affected components will be in FAILED state. To prepare for a new update after this, call `psa_fwu_clean()` for each component.
- If a component restart is required, the state of affected components changes to REJECTED and `PSA_SUCCESS_RESTART` is returned. To continue the rollback process, restart the affected components.
  After restart, the affected components will be in FAILED state. To prepare for a new update after this, call `psa_fwu_clean()` for each component.

- If no reboot or component restart is required, the state of affected components changes to FAILED and PSA_SUCCESS is returned. To prepare for a new update after this, call psa_fwu_clean() for each component.

Implementation note

This function is mandatory in an implementation for which any of the following are true:

- One or more components have a TRIAL state
- One or more components require a system reboot to complete installation
- One or more components require a component restart to complete installation

On implementations where none of these hold, this function is optional.

See *Required functions* on page 42.

5.6.3 Image trial

The following function is used to manage a trial of new firmware images. It acts atomically on all components that are in TRIAL state.

psa_fwu_accept (function)

Accept a firmware update that is currently in TRIAL state.

```
psa_status_t psa_fwu_accept(void);
```

Returns: psa_status_t

Result status.

PSA_SUCCESS	Success: the affected components are now in UPDATED state.
PSA_ERROR_BAD_STATE	There are no components in the TRIAL state.
PSA_ERROR_NOT_PERMITTED	The caller is not authorized to call this function.
PSA_ERROR_NOT_SUPPORTED	This function call is not implemented.
PSA_ERROR_INSUFFICIENT_POWER	The system does not have enough power to safely update the firmware.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_STORAGE_FAILURE	

Description

This function is used when new firmware images in TRIAL state have been determined to be functional, to permanently accept the new firmware images. If successful, the state of affected components changes to UPDATED. To prepare for another update after this, call `psa_fwu_clean()` for each component.

For firmware components in TRIAL state, if `psa_fwu_accept()` is not called, then rebooting the system results in the image being automatically rejected. To explicitly reject a firmware update in TRIAL state, call `psa_fwu_reject()`.

Implementation note

This function is mandatory in an implementation where one or more components have a TRIAL state.

On implementations where none of these hold, this function is optional.

See *Required functions* on page 42.

Appendix A: Example header file

Each implementation of the Firmware Update API must provide a header file named `psa/update.h`, in which the API elements in this specification are defined.

This appendix provides a example of the `psa/update.h` header file with all of the API elements. This can be used as a starting point or reference for an implementation.

> **Note:**
>
> Not all of the API elements are fully defined. An implementation must provide the full definition.
>
> The header will not compile without these missing definitions, and might require reordering to satisfy C compilation rules.

A.1 psa/update.h

```
/* This file is a reference template for implementation of the
 * PSA Certified Firmware Update API v1.0.0
 */

#ifndef PSA_UPDATE_H
#define PSA_UPDATE_H

#include <stdint.h>

#include "psa/error.h"

#ifdef __cplusplus
extern "C" {
#endif

#define PSA_FWU_API_VERSION_MAJOR 1
#define PSA_FWU_API_VERSION_MINOR 0
#define PSA_ERROR_DEPENDENCY_NEEDED ((psa_status_t)-156)
#define PSA_ERROR_FLASH_ABUSE ((psa_status_t)-160)
#define PSA_ERROR_INSUFFICIENT_POWER ((psa_status_t)-161)
#define PSA_SUCCESS_REBOOT ((psa_status_t)+1)
#define PSA_SUCCESS_RESTART ((psa_status_t)+2)
typedef uint8_t psa_fwu_component_t;
typedef struct psa_fwu_image_version_t {
    uint8_t major;
    uint8_t minor;
    uint16_t patch;
    uint32_t build;
} psa_fwu_image_version_t;
#define PSA_FWU_READY 0u
```

(continues on next page)

(continued from previous page)

```
#define PSA_FWU_WRITING 1u
#define PSA_FWU_CANDIDATE 2u
#define PSA_FWU_STAGED 3u
#define PSA_FWU_FAILED 4u
#define PSA_FWU_TRIAL 5u
#define PSA_FWU_REJECTED 6u
#define PSA_FWU_UPDATED 7u
#define PSA_FWU_FLAG_VOLATILE_STAGING 0x00000001u
#define PSA_FWU_FLAG_ENCRYPTION 0x00000002u
typedef struct { /* implementation-defined type */ } psa_fwu_impl_info_t;
typedef struct psa_fwu_component_info_t {
    uint8_t state;
    psa_status_t error;
    psa_fwu_image_version_t version;
    uint32_t max_size;
    uint32_t flags;
    uint32_t location;
    psa_fwu_impl_info_t impl;
} psa_fwu_component_info_t;
psa_status_t psa_fwu_query(psa_fwu_component_t component,
                           psa_fwu_component_info_t *info);
psa_status_t psa_fwu_start(psa_fwu_component_t component,
                           const void *manifest,
                           size_t manifest_size);
#define PSA_FWU_MAX_WRITE_SIZE /* implementation-defined value */
psa_status_t psa_fwu_write(psa_fwu_component_t component,
                           size_t image_offset,
                           const void *block,
                           size_t block_size);
psa_status_t psa_fwu_finish(psa_fwu_component_t component);
psa_status_t psa_fwu_cancel(psa_fwu_component_t component);
psa_status_t psa_fwu_clean(psa_fwu_component_t component);
psa_status_t psa_fwu_install(void);
psa_status_t psa_fwu_request_reboot(void);
psa_status_t psa_fwu_reject(psa_status_t error);
psa_status_t psa_fwu_accept(void);

#ifdef __cplusplus
}
#endif

#endif // PSA_UPDATE_H
```

Appendix B: Example usage

Warning: These examples are for illustrative purposes only and are not guaranteed to compile. Many error codes are not handled in order to keep the examples brief. A real implementation will need to initialize variables appropriately and handle failures as they see fit.

B.1 Retrieve versions of installed images

This example shows the retrieval of image versions for all components.

```
#include <psa/update.h>

/* Assume that the components in this system have sequential identifiers
 * starting at zero.
 */
#define NUM_COMPONENTS 3

void example_get_installation_info() {

    psa_status_t rc;
    psa_fwu_component_t id;
    psa_fwu_component_info_t info;

    for (id = 0; id < NUM_COMPONENTS; ++id) {
        rc = psa_fwu_query(id, &info);

        if (rc == PSA_SUCCESS) {
            specific_protocol_report(id, info.version);
        }
    }
}
```

B.2 Individual component update (single part operation)

This example shows the installation of a single component that is smaller than PSA_FWU_MAX_WRITE_SIZE.

```
#include <psa/update.h>

/* Simple, single image update with a bundled manifest.
 * Component requires reboot
 */

void example_install_single_image(psa_fwu_component_t id,
                                  const void *image, size_t image_size) {
    psa_status_t rc;

```

(continues on next page)

(continued from previous page)

```
11      // Assume the component state is READY
12      rc = psa_fwu_start(id, NULL, 0);
13
14      if (rc == PSA_SUCCESS) {
15          rc = psa_fwu_write(id, 0, image, image_size);
16
17          if (rc == PSA_SUCCESS) {
18              rc = psa_fwu_finish(id);
19
20              if (rc == PSA_SUCCESS) {
21                  rc = psa_fwu_install();
22
23                  if (rc == PSA_SUCCESS_REBOOT) {
24                      // do other things and then eventually...
25                      psa_fwu_request_reboot();
26                      return;     // or wait for reboot to happen
27                  }
28              }
29          }
30          // an error occurred during image preparation: clean up
31          psa_fwu_cancel(id);
32          psa_fwu_clean(id);
33      }
34      // report failure...
35  }
```

B.3 Individual component update (multi part operation)

This example shows the installation of a component that can be larger than `PSA_FWU_MAX_WRITE_SIZE`, and requires writing in multiple blocks.

```
1   #include <psa/update.h>
2   #include <stdlib.h>
3   #include <stddef.h>
4
5   /* Single image update with a bundled manifest.
6    * Image data is fetched and written incrementally in blocks
7    */
8
9   void example_install_single_image_multipart(psa_fwu_component_t id,
10                                              size_t total_image_size) {
11      psa_status_t rc;
12      size_t offset;
13      size_t to_send;
14      void *image;
15
16      // Assume the component state is READY
17      rc = psa_fwu_start(id, NULL, 0);
18
19      if (rc == PSA_SUCCESS) {
20          // Using dynamically allocated memory for this example
```

(continues on next page)

(continued from previous page)

```
21
22          image = malloc(PSA_FWU_MAX_WRITE_SIZE);
23          if (image == NULL) {
24              rc == PSA_ERROR_INSUFFICIENT_MEMORY;
25          } else {
26              for (offset = 0;
27                  offset < total_image_size,
28                  offset += PSA_FWU_MAX_WRITE_SIZE) {
29                  to_send = min(PSA_FWU_MAX_WRITE_SIZE, total_image_size - offset);
30                  if (fetch_next_part_of_image(id, image, to_send)) {
31                      // failed to obtain next block of image
32                      rc == PSA_ERROR_GENERIC_ERROR;
33                      break;
34                  } else {
35                      rc = psa_fwu_write(id, offset, image, to_send);
36                      if (rc != PSA_SUCCESS) {
37                          break;
38                      }
39                  }
40              }
41              free(image);
42          }
43
44          if (rc == PSA_SUCCESS) {
45              rc = psa_fwu_finish(id);
46
47              if (rc == PSA_SUCCESS) {
48                  rc = psa_fwu_install();
49
50                  if (rc == PSA_SUCCESS) {
51                      // installation completed, now clean up
52                      psa_fwu_clean(id);
53                      // report success ...
54                      return;
55                  } else if (rc == PSA_SUCCESS_REBOOT) {
56                      // do other things and then eventually...
57                      psa_fwu_request_reboot();
58                      return;     // or wait for reboot to happen
59                  }
60              }
61          }
62          // an error occurred during image preparation: clean up
63          psa_fwu_cancel(id);
64          psa_fwu_clean(id);
65      }
66      // report failure...
67  }
```

B.4 Multiple components with dependent images

This example shows how multiple components can be installed together. This is required if the images are inter-dependent, and it is not possible to install them in sequence because of the dependencies.

> **Note:**
>
> Not all implementations that have multiple components support this type of multi-component update.

```
#include <psa/update.h>

/* Atomic, multiple image update, with bundled manifests.
 * Installation requires reboot
 */

// Prepare a single image for update
static psa_status_t prepare_image(psa_fwu_component_t id,
                            const void *image, size_t image_size) {
    psa_status_t rc;

    // Assume the component state is READY
    rc = psa_fwu_start(id, NULL, 0);

    if (rc == PSA_SUCCESS) {
        rc = psa_fwu_write(id, 0, image, image_size);

        if (rc == PSA_SUCCESS) {
            rc = psa_fwu_finish(id);

        if (rc != PSA_SUCCESS) {
            // an error occurred during image preparation: clean up
            psa_fwu_cancel(id);
            psa_fwu_clean(id);
        }
    }
    return rc;
}

// Fetch and prepare a single image for update
static psa_status_t fetch_and_prepare_image(psa_fwu_component_t id) {
    psa_status_t rc;
    void *image;
    size_t image_size;

    // Get image data.
    // Assume this is dynamically allocated memory in this example
    image = fetch_image_data(id, &image_size);
    if (image == NULL)
        return PSA_ERROR_INSUFFICIENT_MEMORY;

    rc = prepare_image(id, image, image_size);
```

(continues on next page)

(continued from previous page)

```
43      free(image);
44      return rc;
45  }
46
47  // Update a set of components atomically
48  // Prepare all the images before installing
49  // Clean up all preparation on error
50  void example_install_multiple_images(psa_fwu_component_id ids[],
51                                       size_t num_ids) {
52      psa_status_t rc;
53      int ix;
54
55      for (ix = 0, ix < num_ids; ++ix) {
56          rc = fetch_and_prepare_image(ids[ix]);
57          if (rc != PSA_SUCCESS)
58              break;
59      }
60
61      if (rc == PSA_SUCCESS) {
62          // All images are prepared, so now install them
63          rc = psa_fwu_install();
64
65          if (rc == PSA_SUCCESS_REBOOT) {
66              // do other things and then eventually...
67              psa_fwu_request_reboot();
68              return;     // or wait for reboot to happen
69          }
70      }
71      // an error occurred during image preparation: clean up.
72      // All of the components prior to element ix have been prepared
73      // Update of these needs to be aborted and erased.
74      while (--ix >= 0) {
75          psa_fwu_cancel(ids[ix]);
76          psa_fwu_clean(ids[ix]);
77      }
78      // Report the failure ...
79  }
```

B.5 Clean up all component updates

This example removes any prepared and failed update images for all components.

```
1   #include <psa/update.h>
2
3   /* Assume that the components in this system have sequential identifiers
4    * starting at zero.
5    */
6   #define NUM_COMPONENTS 3
7
8   /* Forcibly cancel and clean up all components to return to READY state */
9
```

(continues on next page)

(continued from previous page)

```
10  void example_clean_all_components() {
11
12      psa_status_t rc;
13      psa_fwu_component_t id;
14      psa_fwu_component_info_t info;
15
16      rc = psa_fwu_reject();
17      if (rc == PSA_SUCCESS_REBOOT) {
18          psa_fwu_request_reboot();
19          // After reboot, run this function again to finish clean up
20          return;
21      }
22
23      for (id = 0; id < NUM_COMPONENTS; ++id) {
24          rc = psa_fwu_query(id, &info);
25
26          if (rc == PSA_SUCCESS) {
27              switch (info.state) {
28                 case PSA_FWU_WRITING:
29                 case PSA_FWU_CANDIDATE:
30                    psa_fwu_cancel(id);
31                    psa_fwu_clean(id);
32                    break;
33                 case PSA_FWU_FAILED:
34                 case PSA_FWU_UPDATED:
35                    psa_fwu_clean(id);
36                    break;
37              }
38          }
39      }
40  }
```

Appendix C: Variation in system design parameters

Depending on the system design and product requirements, an implementation can collapse a chain of transitions for a component, where this does not remove information that is required by the Client, or compromise other system requirements. This can result in some states and transitions being eliminated from the state model for that component's firmware store.

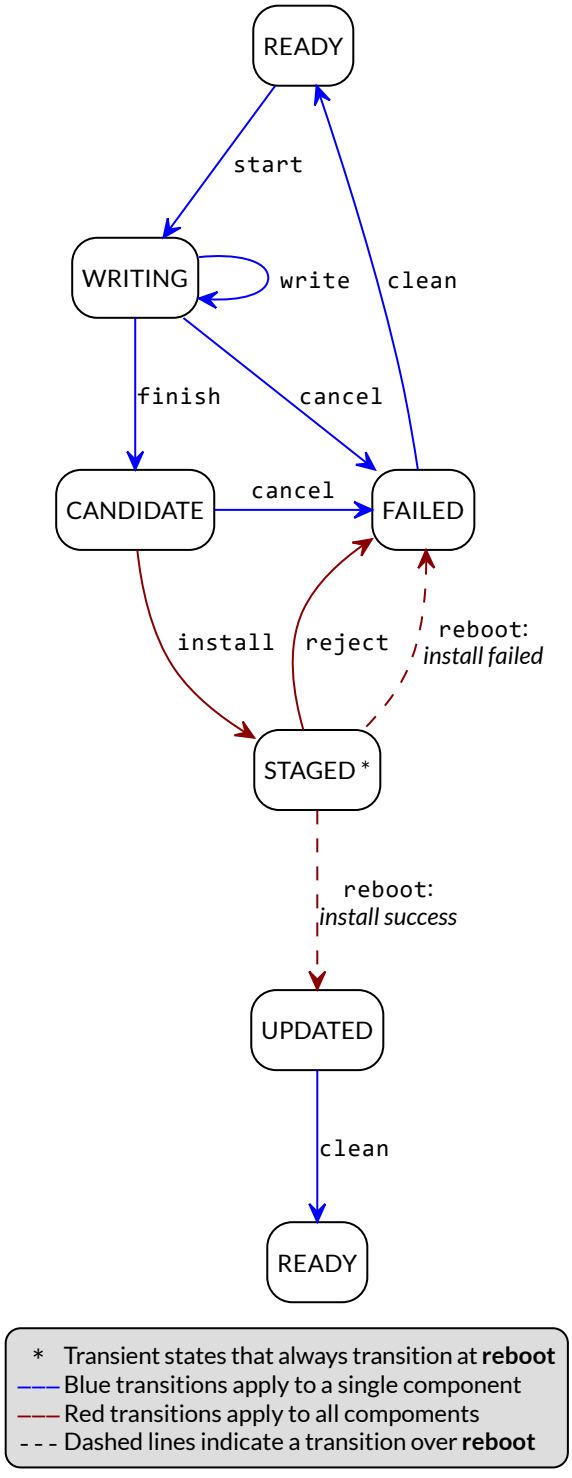
Some possible variations are the following:

Reboot required	Trial required	Description
Yes	Yes	See full state model
Yes	No	See no-trial model
No	Yes	See no-reboot model
No	No	See basic state model

C.1 Components that require a reboot, but no trial

If a component does not require testing before committing the update, the the TRIAL and REJECTED states are not used. The `reboot` operation that installs the firmware will transition to UPDATED on success, or FAILED on failure. The `accept` operation is never used, the `reject` operation is still used to abandon an update that has been STAGED.

The simplified flow is as follows:

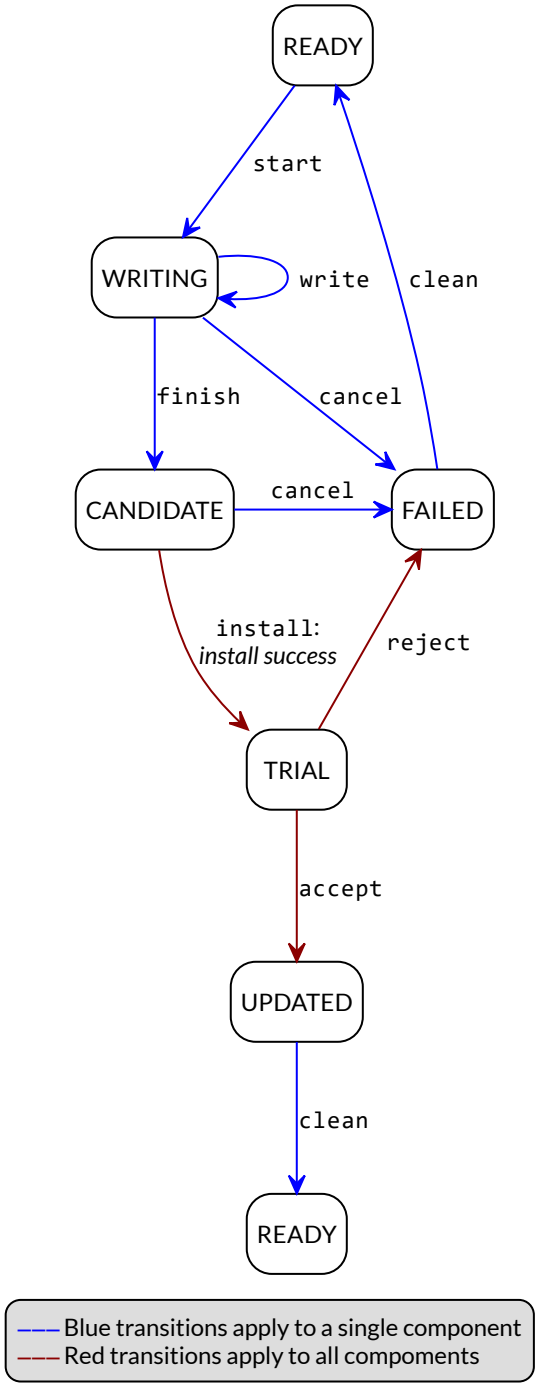


C.2 Components that require a trial, but no reboot

If a component does not require a reboot to complete installation, the STAGED state is not required. The `install` operation will complete the installation immediately, transitioning to TRIAL if successful.

This use case also removes the REJECTED state, because the `reject` operation from TRIAL state does not require a `reboot` to complete. A `reject` operation from TRIAL states transitions directly to FAILED.

The simplified flow is as follows:



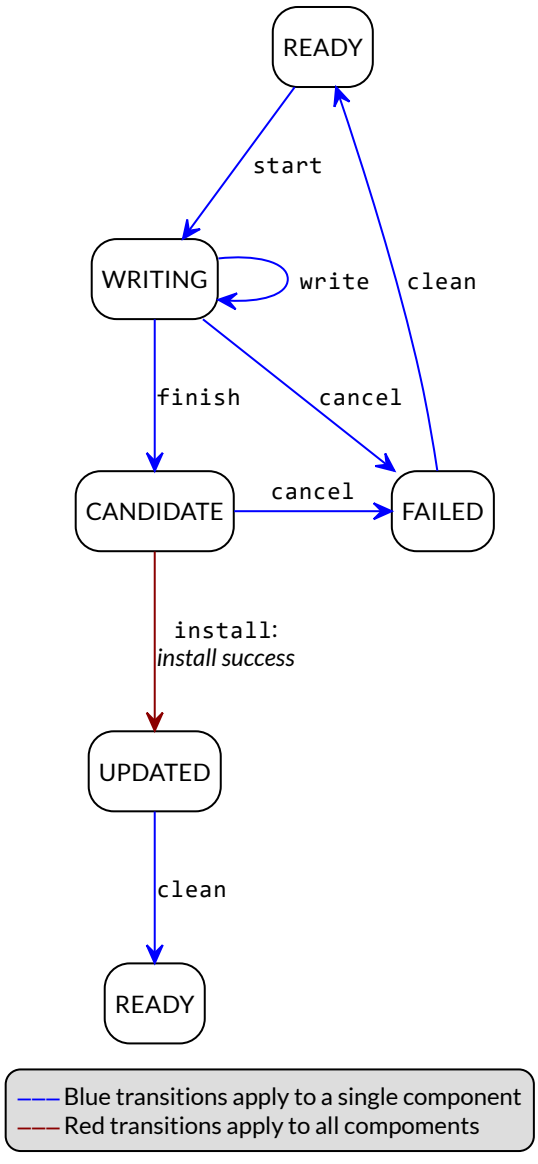
Notes

1. There is no ability for the update service to automatically reject a TRIAL, because the "`reboot` without `accept`" condition used for this purpose in the full state model is not available in this use case.

C.3 Components that require neither a reboot, nor a trial

If a component does not require a reboot to complete installation, and does not require testing before committing the update, then the STAGED, TRIAL, and REJECTED states are not required. The `install` operation will complete the installation immediately, transitioning to UPDATED if successful. The `accept` and `reject` operations are not used.

The simplified flow is as follows:



Appendix D: Document change history

D.1 Changes between version 0.7 and 1.0-bet

General changes

- Relicensed the document under Attribution-ShareAlike 4.0 International with a patent license derived from Apache License 2.0. See *License* on page vi.
- Removed Profile IDs, and discussion of SUIT and manifest formats
- Revised and extended all of the early chapters covering the goals, architecture and design of the API.
- Updated code examples to match the v1.0 API. See *Example usage* on page 64.

API changes

- Renamed `psa_image_id_t` to `psa_fwu_component_t`, and changed the type to `uint8_t`.
- Renamed `psa_image_info_t` to `psa_fwu_component_info_t`.
  - Removed Image ID, Vendor ID and Class ID from `psa_fwu_component_info_t` structure.
  - Removed `psa_fwu_staging_info_t`, adding any important members directly to `psa_fwu_component_info_t`.
- Renamed `psa_image_version_t` to `psa_fwu_image_version_t`.
  - Resized the fields in `psa_fwu_image_version_t` to align with other project structures.
  - Added `build` field to `psa_fwu_image_version_t`.
- Reworked the state model to reflect the overall state of a firmware component, not a specific image.
  - Renamed `PSA_FWU_UNDEFINED` to `PSA_FWU_READY` - the default starting state for the state model.
  - Renamed `CANDIDATE` state to `WRITING` state. The new definition is `PSA_FWU_WRITING`.
  - Renamed `REBOOT_NEEDED` state to `STAGED` state. The new definition is `PSA_FWU_STAGED`.
  - Renamed `PENDING_INSTALL` state to `TRIAL` state. The new definition is `PSA_FWU_TRIAL`.
  - Renamed `INSTALLED` state to `UPDATED` state. The new definition is `PSA_FWU_UPDATED`.
  - Renamed `REJECTED` state to `FAILED` state. The new definition is `PSA_FWU_FAILED`.
  - Reintroduced `REJECTED` as a transient state when rollback has been requested, but reboot has not yet occurred.
- Renamed some of the installation functions:
  - Rename `psa_fwu_set_manifest()` to `psa_fwu_start()`. This call is now mandatory, but the manifest data is optional.
  - Rename `psa_fwu_request_rollback()` to `psa_fwu_reject()`, to mirror `psa_fwu_accept()`.
  - Rename `psa_fwu_abort()` to `psa_fwu_clean()`.
- Explicit support for concurrent installation of multiple components:
  - Reintroduced `CANDIDATE` state for an image that has been prepared for installation, but not installed.

  - Add psa_fwu_finish() to mark a new firmware image as ready for installation.
  - Add psa_fwu_cancel() to abandon an update that is being prepared.
  - Removed the component_id parameter from psa_fwu_install(), psa_fwu_accept(), and psa_fwu_reject(): these now act atomically on all components in the initial state for the operation.
- Reference the standard definition of the status codes, and remove them from this specification. See *Status codes* on page 43.
  - Rationalize the API-specific error codes. This removes the following error codes:
    - PSA_ERROR_WRONG_DEVICE
    - PSA_ERROR_CURRENTLY_INSTALLING
    - PSA_ERROR_ALREADY_INSTALLED
    - PSA_ERROR_INSTALL_INTERRUPTED
    - PSA_ERROR_DECRYPTION_FAILURE
    - PSA_ERROR_MISSING_MANIFEST
  - Standardize the use of error codes, aligning with other PSA Certified APIs:
    - Use PSA_ERROR_BAD_STATE when operations are called in the wrong sequence.
    - Use PSA_ERROR_DOES_NOT_EXIST when operations are called with an unknown component Id.
    - Use PSA_ERROR_NOT_PERMITTED when firmware images do not comply with update policy.
- Removed the discovery API functions and types
  - psa_fwu_get_image_id_iterator()
  - psa_fwu_get_image_id_next()
  - psa_fwu_get_image_id_valid()
  - psa_fwu_get_image_id()
  - psa_fwu_iterator_t
- Removed Profile IDs, and discussion of SUIT and metadata formats

D.2 Changes between version 0.6 and 0.7

This section describes detailed changes between past versions.

- PSA_FWU_API_VERSION_MINOR has increased from 6 to 7
- psa_image_id_t is now defined as a 32-bit integer. Functions no longer have a pointer type for this parameter.
- UUID concept dropped from function names and parameters.
- Added Vendor ID and Class ID to psa_image_info_t structure.
- Added Future changes section
- Added error code and success code definitions
- Fixed mistake: psa_fwu_abort return type changed from void to psa_status_t
- Clarifications to the text
- Replaced PSA_ERROR_ROLLBACK_DETECTED with PSA_ERROR_NOT_PERMITTED

- Remove standardized image IDs until we get more feedback
- Improvements to the Design Overview text

Index of API elements