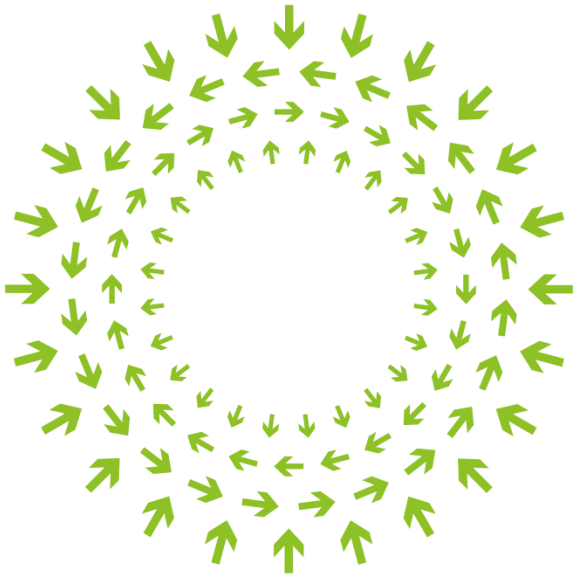

OPEN
Compute Project

Project Cerberus
Host Processor Firmware Requirements

Author:
Christopher Weimer, Senior Firmware Engineer, Microsoft
Bryan Kelly, Principle Firmware Engineering Manager, Microsoft

Revision History

Date	Description
1/9/2018	Version 0.1 – Initial Draft
4/18/2018	Updates for new R/W flash management and recovery details
9/13/2018	Changes to PFM information and SPI limitations
12/3/2018	Updated list of supported SPI flash commands Added recovery information when running without a PFM Added clarification around 4kB region alignment
12/6/2018	Clarification around 4-Byte address SPI commands
8/19/2019	Added section detailing host firmware update
9/26/2019	Added section detailing host firmware recovery

As of November 1, 2017, the following persons or entities have made this Specification available under the Open Web Foundation Final Specification Agreement (OWFa 1.0), which is available at http://www.openwebfoundation.org/legal/the-owf-1-0-agreements/owfa-1-0

Microsoft Corporation.

You can review the signed copies of the Open Web Foundation Agreement Version 1.0 for this Specification at http://www.opencompute.org/participate/legal-documents/, which may also include additional parties to those listed above.

Your use of this Specification may be subject to other third party rights. THIS SPECIFICATION IS PROVIDED "AS IS." The contributors expressly disclaim any warranties (express, implied, or otherwise), including implied warranties of merchantability, non-infringement, fitness for a particular purpose, or title, related to the Specification. The entire risk as to implementing or otherwise using the Specification is assumed by the Specification implementer and user. IN NO EVENT WILL ANY PARTY BE LIABLE TO ANY OTHER PARTY FOR LOST PROFITS OR ANY FORM OF INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION OR ITS GOVERNING AGREEMENT, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND WHETHER OR NOT THE OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CONTRIBUTORS AND LICENSORS OF THIS SPECIFICATION MAY HAVE MENTIONED CERTAIN TECHNOLOGIES THAT ARE MERELY REFERENCED WITHIN THIS SPECIFICATION AND NOT LICENSED UNDER THE OWF CLA OR OWFa. THE FOLLOWING IS A LIST OF MERELY REFERENCED TECHNOLOGY: INTELLIGENT PLATFORM MANAGEMENT INTERFACE (IPMI); I^2C IS A TRADEMARK AND TECHNOLOGY OF NXP SEMICONDUCTORS ; EPYC IS A TRADEMARK AND TECHNOLOGY OF ADVANCED MICRO DEVICES INC.; ASPEED AST 2400/2500 FAMILY PROCESSORS IS A TECHNOLOGY OF ASPEED TECHNOLOGY INC.; MOLEX NANOPITCH, NANO PICOBLADE, AND MINI-FIT JR AND ASSOCIATED CONNECTORS ARE TRADEMARKS AND TECHNOLOGIES OF MOLEX LLC; WINBOND IS A TRADEMARK OF WINBOND ELECTRONICS CORPORATION; NVLINK IS A TECHNOLOGY OF NVIDIA; INTEL XEON SCALABLE PROCESSORS, INTEL QUICKASSIST TECHNOLOGY, INTEL HYPER-THREADING TECHNOLOGY, ENHANCED INTEL SPEEDSTEP TECHNOLOGY, INTEL VIRTUALIZATION TECHNOLOGY, INTEL SERVER PLATFORM SERVICES, INTEL MANAGABILITY ENGINE, AND INTEL TRUSTED EXECUTION TECHNOLOGY ARE TRADEMARKS AND TECHNOLOGIES OF INTEL CORPORATION; SITARA ARM CORTEX-A9 PROCESSOR IS A TRADEMARK AND TECHNOLOGY OF TEXAS INSTRUMENTS; GUIDE PINS FROM PENCOM; BATTERIES FROM PANASONIC. IMPLEMENTATION OF THESE TECHNOLOGIES MAY BE SUBJECT TO THEIR OWN LEGAL TERMS.

Contents

1 Introduction

This document will describe the expected operation of a host processor whose firmware is being protected by Cerberus. It will also detail the additional requirements and limitations that Cerberus imposes on the host processor.

2 Normal Operation

For a host processor connected to Cerberus, it will appear as if there is a single flash device connected to the processor. This is true regardless of the actual number of flash devices present on the board or the number of SPI chip select lines coming from the host processor. Cerberus only uses a single chip select from the host processor to assert the chip select on the protected flashes. How the chip select is asserted on the flash devices is completely managed by Cerberus. For host processors with multiple chip select lines on the boot SPI bus, asserting any chip select other than the primary (CS0) will result in no operation being performed, in the same way that asserting a chip select that is not physically connected to a device would perform no operation.

2.1 Flash Layout

To simultaneously provide protection for the host processor's firmware image and allow it the ability to store configuration or other state in the flash, the image needs to be segmented into contiguous blocks of read-only and read/write regions. Because the read/write region is mutable, it cannot be protected by Cerberus since there is no expected state to compare against. Being unprotected, it must only contain configuration and other non-executable data. All executable parts of the firmware image should be stored in the read-only regions which are protected from modification by Cerberus.

Multiple regions of each type can be defined, but each region must be defined to align with increments of 4kB[1] or 64kB flash blocks.

While SPI flash allows a single read command to continue reading sequential data, it is important that single read operations are not used when crossing region boundaries. Doing so could result in reading invalid data since it is not guaranteed that the different regions contiguous or located on the same flash device. Ensuring that reads are not issued that would cross region boundaries will prevent against error. Similarly, properly defining regions in PFM is a critical part of firmware protection.

[1] 4kB alignment is not available with all Cerberus implementations.

2.2 Read/Write Regions

Special consideration may be required for handling read/write regions of flash. No protection of this data is provided by Cerberus, so the host processor's firmware must be resilient to corrupt or unexpected data. While in some scenarios, there may be multiple physical flash devices, only one device is used to store the read/write regions. If these regions get corrupted with bad are unexpected data, it is the responsibility of the host firmware to recover.

Cerberus does ensure that the contents of read/write regions are persisted across firmware authentication cycles. In a dual flash configuration, after an update has been authenticated and the new flash device is activated and marked as read-only, the contents of the read/write regions are copied into the flash device that was just made writable. After this is done, both flash devices will have the same contents in all read/write regions. Once the system boots, the read/write contents on the read-only flash will be inaccessible and remain unchanged. The read/write contents on the read/write flash will be accessed and updated by host firmware.

In addition to typical sources of flash corruption, there are a couple of additional ways to get corrupt or unexpected data due to Cerberus operation.

2.3 Unauthenticated Firmware Updates

If a firmware update process is executed with unsupported firmware, it is possible that the read/write regions could have also been written. When Cerberus authentication fails for this update, the flash devices remain unchanged. This means that:

1) The host will continue to boot from the previous version of firmware.
2) The read/write regions will contain whatever data was written during the unauthorized firmware update.

This is similar to standard cases of data corruption, since the read/write data is now not what it should be. Normal mechanisms for dealing with flash corruption in the host firmware will likely account for this scenario.

2.4 Boot Failover

If supported by the host, boot failover can also cause unexpected read/write data. If the active image fails to boot and the failover flow is triggered, the read/write data is not copied. This avoids attempting to boot with corrupt read/write data. The read/write data that is used during recovery is the version of this data at the time of the last authenticated update. For failover to work correctly, it is necessary to ensure that read/write data is compatible between different versions of host firmware.

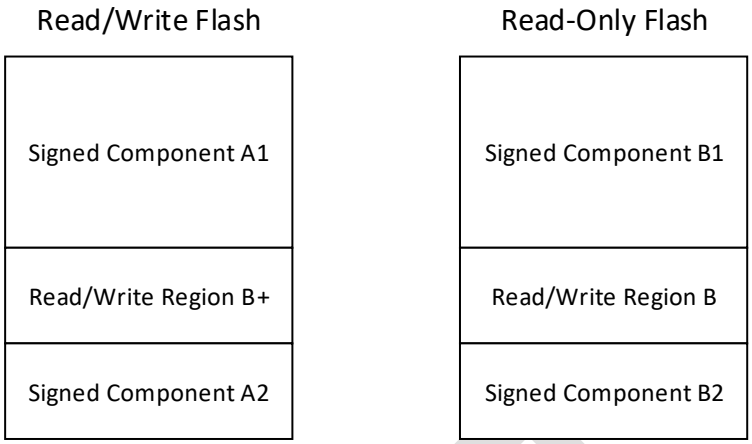

Figure 1 Read/Write State on Failover

In the example above, firmware image B was written to flash and authenticated. This caused the flash containing image B to marked as read-only, and the read/write data from B (Read/Write Region B) to overwrite the read/write data for image A (Read/Write Region B+). The read/write data contained in B+ will be the region that gets updated when image B runs. The read/write data contained in B remains unchanged. If image B fails to boot, triggering failover, the flash containing image A will be marked read-only and be used to boot, but it will boot using the read/write data contains in Read/Write Region B. The read/write data for image B needs to be compatible with image A.

2.5 Flash Layout Changes

If the flash layout changes between firmware releases, any possibility for rolling back to the previous version will be lost. The act of migrating the read/write regions from the old read/write flash device to the new read/write flash device will cause the old read-only image to be corrupted, resulting in authentication failures on rollback attempts. This corruption only occurs after the new image has been verified to be good.

Changes to the flash layout that will cause this type of corruption are any changes relating to size and/or address of the read-only and read/write regions relative to each other. Any layout changes that are completely encapsulated within an existing read-only or read/write region will not cause rollback corruption. In the example flash layouts below, only migration from A to B would cause rollback corruption. The read/write region is unchanged between A, C, and D, so no corruption would occur.

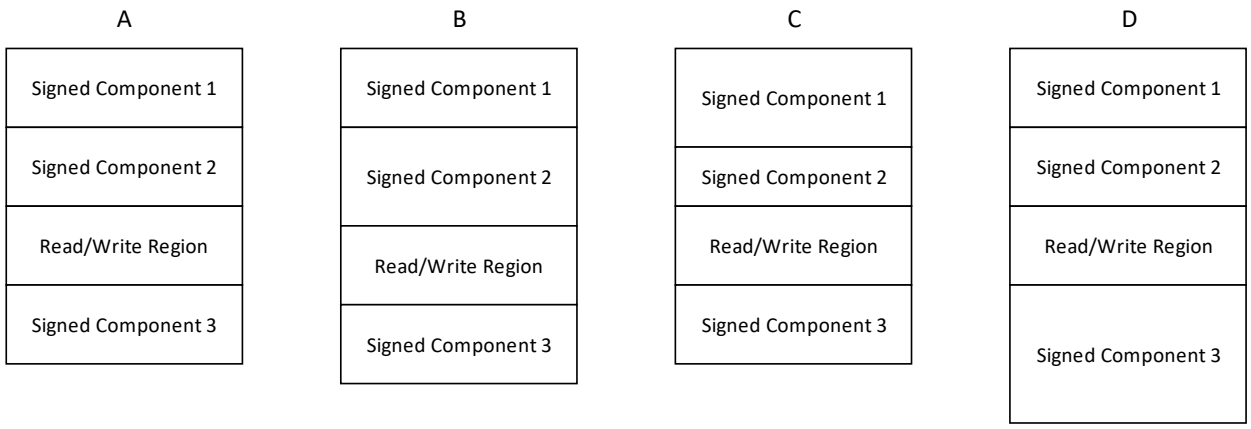

Figure 2 Flash Layout Examples

2.6 Flash Device State

After Cerberus has finished flash validation, the state of the flash device will be the same as it was before validation was started. Specifically, this means the address mode of the flash won't be changed. Hosts must be able to handle the flash address mode not changing on warm reset.

2.7 Boot-Time Verification

Where appropriate, the host processor firmware should implement self-verification of image components at boot-time. By having early boot image components (e.g. bootloaders) verify subsequent components (e.g. kernel), the amount of flash that Cerberus needs to verify on each boot is reduced. Instead of verifying the entire flash on each boot, only the bootloader will be validated. Cerberus acts as the hardware Root of Trust in this scenario for the subsequent verification stages inherent in the image boot process. An example of this self-verification is using FIT images for Linux kernel verification from U-Boot.

2.8 Update Verification

On the next boot after the host processor has written a firmware update to protected flash, Cerberus will verify the update to see if it can be used as the new firmware image. Before being made the active boot image, the flash is verified in its entirety. This means that every signed component will be verified, even components that would not get verified by Cerberus as part of a self-verification boot flow. Also, all flash that is neither part of a signed component nor a read/write region will be checked to ensure that it is blank. If either of these checks fail on the updated flash device, the image will be rejected and the old image will remain the active boot image.

2.9 Platform Firmware Manifest

Every firmware release for the host processor needs to have an associated set of information that can be put into the Platform Firmware Manifest (PFM) that gets sent to Cerberus. The PFM defines the

versions of firmware that the platform is allowed to run and contains all the validation and layout information for that version. In order for Cerberus to match an image stored on flash against an entry in the PFM, the firmware image must have the firmware version identifier stored as a string in the image. This identifier must be stored in a signed flash region that gets verified on every boot of the system. In the case of self-verification boot flows, this means it must be in a region of flash verified directly by Cerberus. The specifics of how and where this version string is stored is determined by the image format of the host processor and is specified in the PFM.

As part of the release process for new firmware, the following information needs to be captured so that a PFM can be generated that includes that version of firmware.

1) Firmware version identifier
2) The platform identifier for this firmware. When a PFM is created for a collection of firmware versions, all versions must report the same platform identifier.
3) Flash address of version identifier
4) Read/Write flash regions. Cerberus supports up to three read/write regions. However, multiple read/write regions that are contiguous with each other can be grouped together into one region for Cerberus.
    a. Start flash address
    b. End flash address
5) Signed firmware components
    a. PEM formatted public key used to sign the component
    b. Signature for the firmware component
    c. Flash regions used by the component. These regions are only the regions used to generate the signature. Unused regions of flash that are not part of the signature must not be included. Each region defines a contiguous block, but any number of blocks can be used to generate the signature.
        i. Start flash address
        ii. End flash address
    d. Indication of whether Cerberus is responsible for validating the component on each boot. In self-verification scenarios, only the bootloader would have this flag set.
6) Optionally, the byte value to look for in flash for any unused region. An unused region is one this neither part of a read/write region nor part of signed firmware. If no byte value is provided, it is assumed the unused regions will be blank.

2.10 Release Metadata Format

The metadata for each release should be stored in the following XML format for use in generating PFMs. The “UnusedByte” tag is optional. If it is not present, the default value of 0xff (blank) will be used.

```
<Firmware version="Identifier" platform=”Identifier”>
        <VersionAddr>0x00000000</VersionAddr>
        <UnusedByte>0xff</UnusedByte>
        <ReadWrite>
```

```
                <Region>
                        <StartAddr>0x00000000</StartAddr>
                        <EndAddr>0x00000000</EndAddr>
                </Region>
                <Region>
                        <StartAddr>0x00000000</StartAddr>
                        <EndAddr>0x00000000</EndAddr>
                </Region>
        </ReadWrite>
        <SignedImage>
                <PublicKey>
                        PEM key data
                </PublicKey>
                <Signature>
                        Image signature
                </Signature>
                <Region>
                        <StartAddr>0x00000000</StartAddr>
                        <EndAddr>0x00000000</EndAddr>
                </Region>
                <Region>
                        <StartAddr>0x00000000</StartAddr>
                        <EndAddr>0x00000000</EndAddr>
                </Region>
                <ValidateOnBoot>true</ValidateOnBoot>
        </SignedImage>
        <SignedImage>
                <PublicKey>
                        PEM key data
                </PublicKey>
                <Signature>
                        Image signature
                </Signature>
                <Region>
                        <StartAddr>0x00000000</StartAddr>
                        <EndAddr>0x00000000</EndAddr>
                </Region>
                <Region>
                        <StartAddr>0x00000000</StartAddr>
                        <EndAddr>0x00000000</EndAddr>
                </Region>
                <ValidateOnBoot>false</ValidateOnBoot>
        </SignedImage>
</Firmware>
```

2.11 Component Firmware Manifest

The Component Firmware Manifest (CFM) consists of a component manifest, describing device type and allowed measurements (versions, signatures) for each component in the system. This is a smaller version of the PFM for authenticating components.

The PFM and CFM are implemented based on the Platform Configuration Data (PCD). A file that directs the PFM Port configuration, Filter Registers, Failure and Remediation Actions for the platform. The PCD also contains location information for devices types (bus, address). This file is SKU specific.

2.12 Release Metadata Format

The metadata for each valid release of component firmware is stored in the following XML format used for generating the CFM, which is deployed as a PFM extension.

```
<DeviceType id="Identifier">
        <Firmware version="Identifier">
                <SignedImage>
                        <Digest>
                                Firmware Digest
                        </Digest>
                        <FailureAction>1</ FailureAction >
                </SignedImage>
        </Firmware>
</DeviceType >
```

FailureAction: 0 = Platform Defined, 1 = Report Only, 2 = Auto Recovery, 3 = Power Off

2.13 Platform Configuration Data

The Platform Configuration Data (PCD) structure component location and policy information for the Platform Active Root of Trust. The PCD is platform specific and applied independent of the PFM and CFM updates. The metadata for the PCD is stored in the following xml format used for generating the PCD update

```
<Configuration platform="Identifier">
        <Version>0x00000000</Version>                       #monatomic Id
        <RoT>                                               #RoT of Trust
                <IsPARoT>true</IsPARoT>                     #True if consumer is a PA RoT
                <Ports>
                        <Port id="#">                       #Ports listed are Active
                                <SPIFreq>48000000</SPIFreq> #Frequency of SPI bus
                        </Port>
                </Ports>
                <Interface>
                        <Address></Address>                 #Address of RoT
                        <BMCAddress></BMCAddress>           #BMC I2C address

                </Interface>
        </RoT>
        <CPLD>
                <Address></Address>                         #Motherboard CPLD Info
                <Channel></Channel>
        </CPLD>
        <Components>
                <Component>
                        <DeviceType>Identifier</DeviceType> #Component Id, See CFM
                        <Bus>Integer</Bus>                  #Component I2C Channel
                        <Address>Integer</Address>          #Component I2C Address
                        <I2CMode>0</I2CMode>                #0 for multimaster, 1 for master-slave
                        <EID>Integer</EID>                  #Component MCTP EID
                        <Muxes>
                                <Mux level="integer">       #Mux I2C Info
                                        <address></address>
                                        <channel></channel>
                                </Mux>
                        </Muxes>
                        <PwrCtrl>
                                <Register></Register>       #PWR CTRL on CPLD
                                <Mask></Mask>               #Bit Mask
                        </PwrCtrl>
                </Component>
        </Components>
        <Policy>
                <Active>true</Active>                       #Policy Active
```

```
            <DefaultFailureAction>1</DefaultFailureAction>#Default attestation Failure
        </Policy>
</Configuration>
```

3 Firmware Updates

Depending on how the firmware update is implemented for the host processor, the update may or may not be impacted by Cerberus.

3.1 Erasing Flash

All unused regions of flash must contain the static value that is specified in the manifest file for Cerberus validation to pass. Normally, the unused flash will be blank. If there is reason to suspect unused regions of flash may not contain this expected value (perhaps due to a change in flash layout), they will need to be erased and, if necessary, programed as part of the update.

3.2 Verifying Flash

During firmware updates, it is sometimes necessary to read the physical flash that is being programmed. With Cerberus, the flash that is read and the flash that is written are different. This means update algorithms that read back the programmed flash to verify its contents will fail. There are two ways the host processor can handle this situation:

1) Don't read the flash as part of the update procedure and delegate flash verification to Cerberus. On reboot after update, Cerberus will already be executing a full validation of the new flash image to see if it should be used as the new active boot image. If it fails verification, the invalid image would not be allowed to boot. The prior image would remain the active image and it would be as if the update never happened. In this case, there is no risk in skipping verification steps during update.
2) If the host can suspend flash accesses, Cerberus can be directed to do run-time attestation of the new firmware. This will give immediate feedback as to whether to new update will be accepted on the next reboot.

3.3 Verifying Read/Write Flash

Cerberus has no way to verify correctness of flash regions that are defined as read/write in the manifest file. Firmware update utilities on the host should verify the read/write contents during the firmware update. Reads and writes to these regions of flash are always to the same physical flash device.

3.4 Update Package

Host firmware updates will require both the new firmware image and a PFM that includes support for that image. The update package needs to contain both pieces, and some means to apply them. A typical update package will contain:

- Firmware image
- Firmware update application
- PFM
- Cerberus utility
- Update script

3.5 Update Execution

To apply the new firmware, the associated PFM needs to first be sent to Cerberus and the update image needs to be applied to host flash. A script or some other coordinating application would need to execute to ensure both components are updated. See Figure 3 for the expected flow of execution.

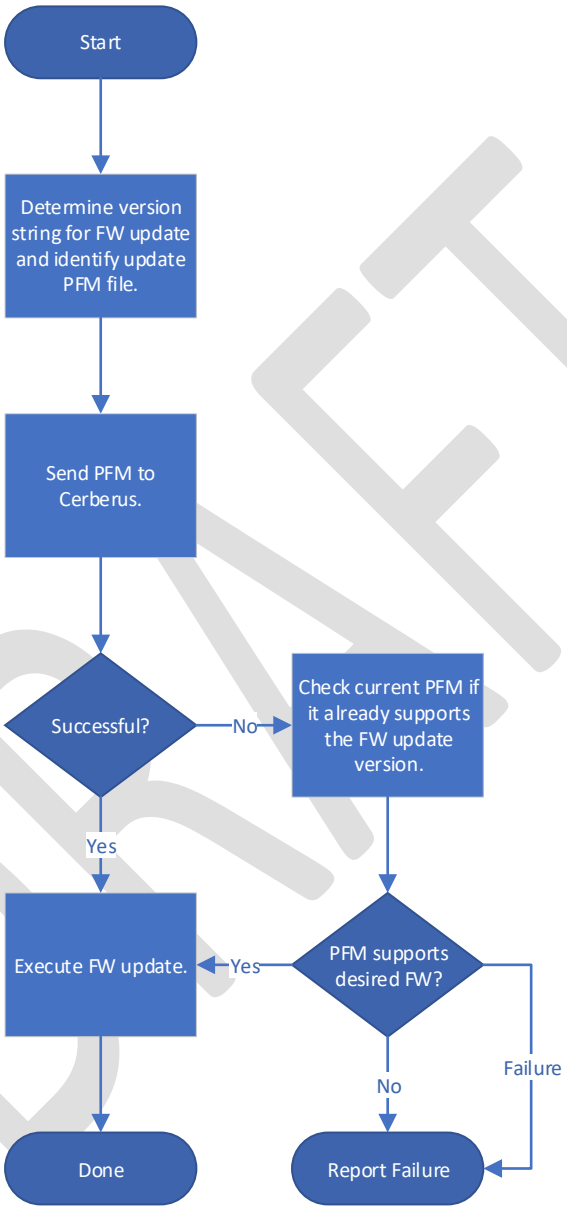

Figure 3 Update Script Execution Flow

After the firmware has been successfully updated, the host must reboot in order for the update to be validated and applied.

3.5.1 Cerberus Commands

The update flow shown in Figure 3 requires communication with Cerberus. If Cerberus communication is done using the standard Cerberus utility, the following commands will be used:

1. Send the PFM to Cerberus: pfmupdate <port> <file> 0
    a. The last argument should be 0 to skip PFM activation until after the FW update is complete.
2. Check current PFM for FW support: pfmcheckversion <port> 0 <version>

3.6 Run-time Verification

If the host processor is able to quiesce access to the SPI flash, it is possible for Cerberus to perform verification of the update without a reboot. This gives immediate feedback regarding the validity of the updated firmware and saves validation time during the next reboot. A reboot is still required for Cerberus to activate the new image and allow the host to boot from it. As part of activation, Cerberus will still perform operations on boot, including migration of read/write data.

The only change to the flow in Figure 3 for this operation comes after a successful FW update. Instead of simply ending the update flow, another command would be sent to Cerberus to immediately activate the PFM. This command can be used even if on PFM was sent to force validation of the image update.

3.6.1 Cerberus Commands

If Cerberus communication is handled with the standard utility, one of the following commands will be used to execute run-time verification:

- Verify without sending a PFM: pfmactivate <port> 1
- Send a new PFM and verify: pfmupdate <port> <file> 1
    - This should not be used prior to a firmware update. Follow the flow in Figure 3 when a firmware update is being sent.

4 Failover Booting

For host processors that support watchdog failover booting using a second SPI chip select, this process will be handled by Cerberus. Cerberus will monitor the host processor for reset events as well as the state of the two chip selects. If the primary chip select is the first to be asserted following a reset event, a normal boot is assumed, and Cerberus follows the regular boot flow. If the secondary chip select is the first to be asserted or is asserted while the processor is in reset, a failover boot is assumed, and Cerberus will hold the host processor in reset. Cerberus will then inspect the contents of the read/write flash device to see if it contains a valid image. If so, Cerberus will swap the read-only and read/write flash devices and release the processor from reset. At this point, the host processor will proceed to

boot normally using the primary chip select. The state machine for detecting watchdog failover can be seen below.

If the failover boot is detected before Cerberus is provisioned with a PFM, Cerberus will still provide recovery. In this case, the entire contents of the flash on CS1 will be copied into the flash on CS0. For this to work properly, there must be a good image programmed the flash on CS1. An unprovisioned Cerberus grants no access to CS1, so this image must be programmed onto this flash out of the system context.

This situation is most likely to occur immediately following a firmware update that was determined to be valid by Cerberus but had some bug in it that caused it to not boot properly. In this case, the read/write flash chip is the one that has the image that was just running on the host processor. It is highly likely to still be a good image if the new image triggers a failover case.

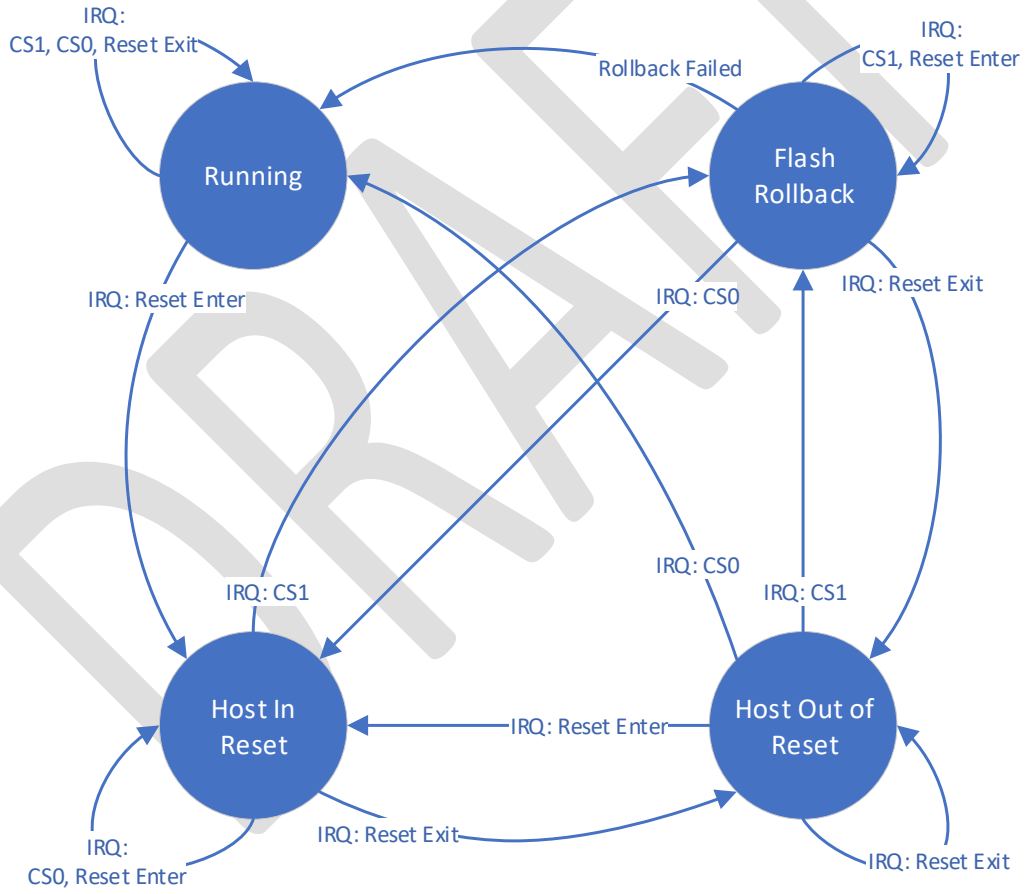

Figure 4 Watchdog Recovery State Machine

5 Host Firmware Recovery

In the event of complete authentication and/or boot failure, Cerberus will provide a recovery mechanism to restore the host back to an operational state. The recovery process will restore the host system back to a state where normal in-band (IB) and out-of-band (OOB) update flows are operational and communication with Cerberus is available.

5.1 Recovery Flow

When the host recovery flow is triggered, Cerberus will copy a bootloader recovery image stored on Cerberus flash to host flash. The bootloader recovery image will subsequently network download the kernel recovery image from the Rack Manager and execute the image out of host memory. The host system is expected to fully boot to the OS and restore the system back to a good state.

The host recovery flow will be triggered in the following scenarios:

1. POR authentication failure
2. The secondary chip select is the first to be asserted or is asserted while the processor is in reset and boot failover is not supported by the host
3. Boot failover is triggered and fails due to an authentication error

5.2 Recovery Image

Both a bootloader and kernel recovery image are necessary to execute host recovery. This section describes the requirements for both images.

5.2.1 Bootloader Recovery Image

Cerberus will store a bootloader recovery image (BRI) and provide a set of utility commands to update the recovery image. The BRI is signed with Cerberus keys. Cerberus will verify the recovery image during system initialization and as part of the image update process.

The bootloader recovery image (BRI) will initiate the boot sequence that will eventually restore the host system to a good state. Below are the requirements for the BRI:

1. The BRI is a single binary file that stores a bootloader script, additional images necessary to execute recovery (e.g. an environment image), and a signature of the entire BRI and associated data.
2. The BRI is formatted according to the format defined in this document.
3. The entire BRI and associated data will be signed and verified using Cerberus keys. Any updates to the BRI will require a new signature.
4. The BRI will network download from the Rack Manager images necessary to execute recovery.
5. The BRI will automatically detect the subnet portion of the management network address based on the assigned DHCP host address.
6. The recovery script is expected to continually seek the required firmware images until the requested images are successfully received by the host.

7. All parameters stored in the BRI are static.

5.2.2 Kernel Recovery Image

As part of the host firmware recovery process, the BRI will network download a kernel recovery image (KRI) from the Rack Manager. The KRI is expected to boot the host system to the OS and restore the host back to a good state. Below are the requirements for the KRI:

1. The KRI includes both the recovery kernel and root file system. The KRI may be one or multiple files.
2. The host will network download the KRI from the Rack Manager directly to main memory and will execute out of host memory.
3. The KRI will identify itself as a recovery image in some way to indicate the host is running in recovery mode.
4. Each platform that supports Cerberus will have a KRI.
5. The Rack Manager will host a KRI for each platform.
6. The name of the KRI for each platform is static.

5.3 Bootloader Recovery Image Format

A BRI is a variable length structure consisting of a top-level image header followed by one or more image sections and the image signature (see Figure 5). All header data in the BRI should be stored in little endian format (top-level header, section headers, etc.).

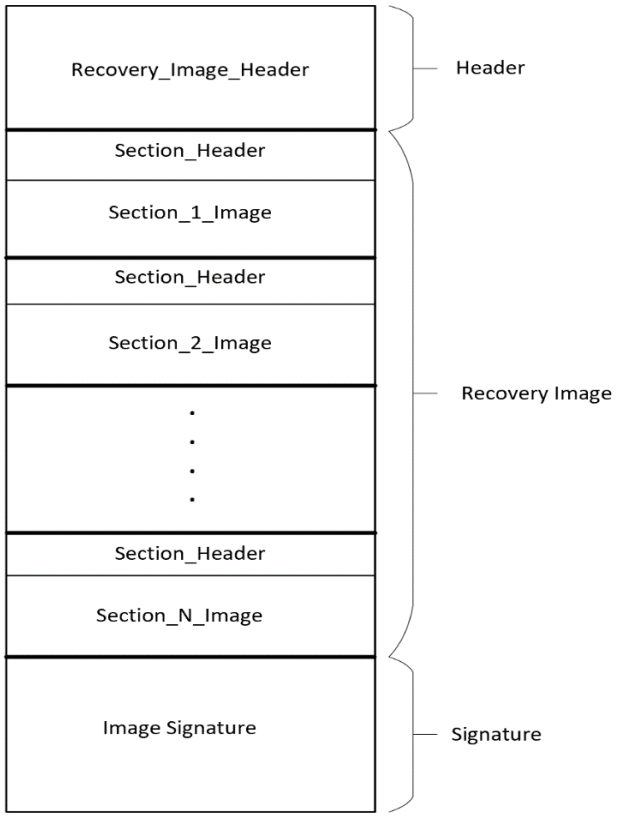

Figure 5: BRI with top-level header, image, and signature.

5.3.1 Top-Level Header

A BRI includes a top-level header with several fields (see Figure 6). Table 1 lists and describes the top-level image header fields in more detail.

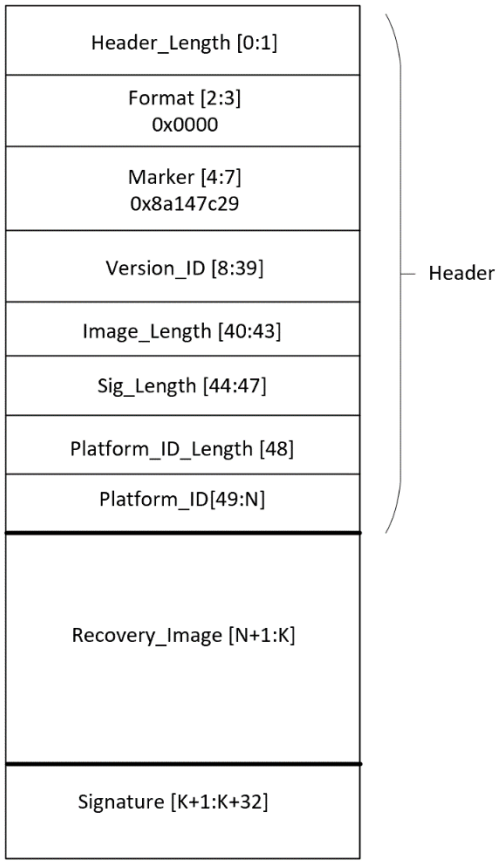

Figure 6: BRI with top-level header fields.

Name	Size (B)	Description	Value (s)
Header Length	2	Length of the header, including the length bytes	49+Platform ID Length
Format	2	Format of the header data	0x0000
Marker	4	Marker for a valid header	0x8a147c29
Version ID	32	Version identifier. Must be a null-terminated string.	variable
Image Length	4	Recovery image length, including header and signature length bytes.	$0 \leq n \leq 2^{32} - 1$
Signature Length	4	Signature length in bytes	$0 \leq n \leq 2^{32} - 1$
Platform ID Length	1	Platform identifier length in bytes	$0 \leq n \leq 2^{8} - 1$
Platform ID	Platform ID length	Platform identifier. This should match the identifier in the PFM for the component. Must be a null-terminated string.	variable
Recovery Image	variable	Recovery image binary	variable
Image Signature	Signature Length	Signature of recovery image	variable

Table 1: Description of top-level BRI header fields.

5.3.2 Recovery Section

Following the top-level header in a BRI is one or more recovery sections. Each recovery section includes a section header followed by the section image (see Figure 7). Table 2 lists and describes the valid values and ranges for the section header fields. The host write address field in the section header for each successive recovery section should be in ascending order. Figure 8 shows an example of a BRI with two sections. During the recovery process, Cerberus will copy each section image to the physical host flash address specified in the section header.

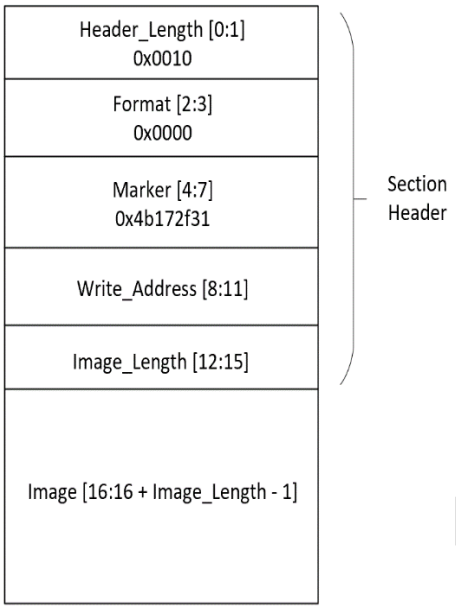

Figure 7: Recovery image section with header values.

Name	Size (B)	Description	Value (s)
Header Length	2	Length of the section header, including the length bytes	0x0010
Format	2	Format of the section header data	0x0000
Marker	4	Marker for a valid section header	0x4b172f31
Write Address	4	Physical host address to write section image	$0 \leq n \leq 2^{32} - 1$
Image Length	4	Size of section image	$0 \leq n \leq 2^{32} - 1$

Table 2: Description of recovery image section header fields.

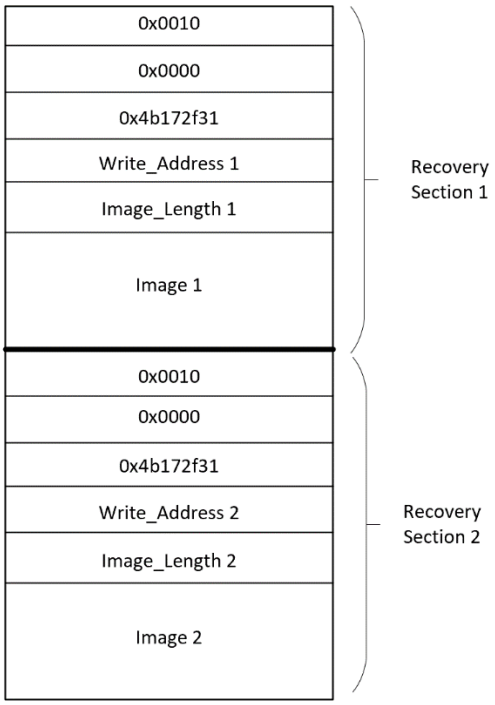

Figure 8: BRI with two sections (excluding the top-level image header and signature).

5.3.3 Release Metadata

The metadata for each release should be stored in the following XML format for use in generating a BRI binary. Table 3 describes the BRI XML tags in more detail.

```
<RecoveryImage version=”Identifier” platform=”Identifier”>
        <RecoverySection>
                <WriteAddress>0x00000000</WriteAddress>
                <EncodedImage>
                        Base64 Encoded Image Data
                </EncodedImage>
        </RecoverySection>
        <RecoverySection>
                <WriteAddress>0x00000001</WriteAddress>
                <EncodedImage>
                        Base64 Encoded Image Data
                </EncodedImage>
        </RecoverySection>
</RecoveryImage>
```

Name	Description
RecoveryImage	A recovery image with version ID and platform ID attributes. The recovery image includes N recovery image sections, where N ≥ 1.
RecoverySection	A recovery image section that includes the write address and the Base64 encoded image section data
WriteAddress	Defines a 4-byte physical host address to store the recovery section image
EncodedImage	A Base64 encoding of the recovery section binary image data

Table 3: Description of BRI XML tags.

6 SPI Flash Limitations

As part of the flash protection, Cerberus will block most SPI flash commands from being executed. This is a list of SPI flash operations that will be permitted by host processor firmware. Any SPI access not in this list will be blocked.

1) Read data in SPI, Dual, and Quad modes
    a. 1-1-1 Read (0x03)
    b. 1-1-1 4-Byte Address Read (0x13)[2]
    c. 1-1-1 Fast Read (0x0B)
    d. 1-1-1 4-Byte Address Fast Read (0x0C)[2]
    e. 1-1-2 Read (0x3B)
    f. 1-1-2 4-Byte Address Read (0x3C)[2]
    g. 1-1-4 Read (0x6B)
    h. 1-1-4 4-Byte Address Read (0x6C)[2]
    i. 1-2-2 Read (0xBB)[1]
    j. 1-2-2 4-Byte Address Read (0xBC)[1 2]
    k. 1-4-4 Read (0xEB)[1]
    l. 1-4-4 4-Byte Address Read (0xEC)[1 2]
2) Register reads
    a. Read Status Register (0x05)
    b. Read Status2 Register (0x35)
    c. Read Configuration/Status3 Register (0x15)
    d. Read ID (0x9F and 0x9E)
    e. Read Serial Flash Discoverable Parameters (0x5A)
    f. Read Extended Address Register (0xC8)[3]
3) Write data
    a. 1-1-1 Page Program (0x02)
    b. 1-1-1 4-Byte Address Page Program (0x12)[2]
    c. Write Disable (0x04)

   d. Write Enable (0x06)
4) Erase blocks
   a. 4kB Erase (0x20)
   b. 4-Byte Address 4kB Erase (0x21)[2]
   c. 32kB Erase (0x52)
   d. 4-Byte Address 32kB Erase (0x5C)[2]
   e. 64kB Erase (0xD8)
   f. 4-Byte Address 64kB Erase (0xDC)[2]
   g. Chip Erase (0x60 and 0xC7)
5) 4-Byte Address Mode
   a. Enter 4-Byte Mode (0xB7)
   b. Exit 4-Byte Mode (0xE9)
6) Soft Reset (0x66 and 0x99)

[1] For Cerberus ports that only support SPI, Dual and Quad I/O commands are not supported. Cerberus QSPI ports have full support.

[2] 4-Byte Address commands are not supported on all Cerberus implementations.

[3] Command not supported on all Cerberus implementations