White Paper

A Tour Beyond BIOS – Capsule Update and Recovery in EDKII

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

December 2016

Executive Summary

Introduction

The firmware update capability represents an important feature for the system firmware on the mother board and the various device firmware instances, such as a host bus adapter-attached PCI option ROM, embedded controller (EC), baseboard management controller (BMC), etc. The firmware recovery is also a feature to support firmware boot in recovery mode in cases where the main flash image is errant or corrupt.

In [EDKII Security Design], we described general security design guidelines for firmware update and firmware recovery. In this paper, we will provides more detail on how we implement capsule update and recovery in EDKII.

Table of Contents

Firmware Update

A system may need to update the firmware image. There are some industry standards providing the guideline or the definition for the firmware update listed below.

NIST Standard

To begin, the National Institute of Standards and Technology (NIST) provides the security guidelines on BIOS update, such as 800-147: BIOS Protection Guidelines [NIST-1] and 800-147B: BIOS Protection Guidelines for Servers [NIST-2]. The summary is below:

- BIOS Update Authentication
    - Key storage in Root of Trust for Update (RTU)
    - Recovery mechanisms shall also use the authenticated update mechanism unless the recovery process meets the guidelines for a secure local update.
    - Rollbacks of the BIOS to an earlier version are permitted only if the update or rollback has been authorized by the organization.
- Secure Local Update (optional)
    - The local update mechanism be used only to load the first BIOS image or to recover from a corruption of a system BIOS
- Integrity Protection
    - The RTU and the system BIOS shall be protected from unintended modification.
- Non-Bypassability
    - Bus mastering that bypasses the main processor (e.g., Direct Memory Access to the system flash) shall not be capable of directly modifying the firmware.
    - Microcontrollers on the system shall not be capable of directly modifying the firmware.

This document defined the general security requirement for firmware update. It does not define in detail either the firmware update capsule format or the details of the update process.

Hardware Compatibility Specification for Systems for Windows 10

In order to maintain the integrity of the platform firmware, Microsoft describes a “Secure firmware update process” in [Win10 HCS]. The requirement is aligned with the NIST standard, such as including a signed bios update and rollback protection.

Windows UEFI Firmware Update Platform

Microsoft provides “Windows UEFI Firmware Update Platform [WFU]” to define how to perform firmware updates initiated by Windows. The summary is below:

- Support Capsule API
- Firmware updates initiated by Windows
- Specify firmware resource in ESRT

This document also defines a set of pre-installation criteria, such as:

- Power Check (25% battery charge)
- Security Check
- Integrity Check
- Version Check
- Storage Check

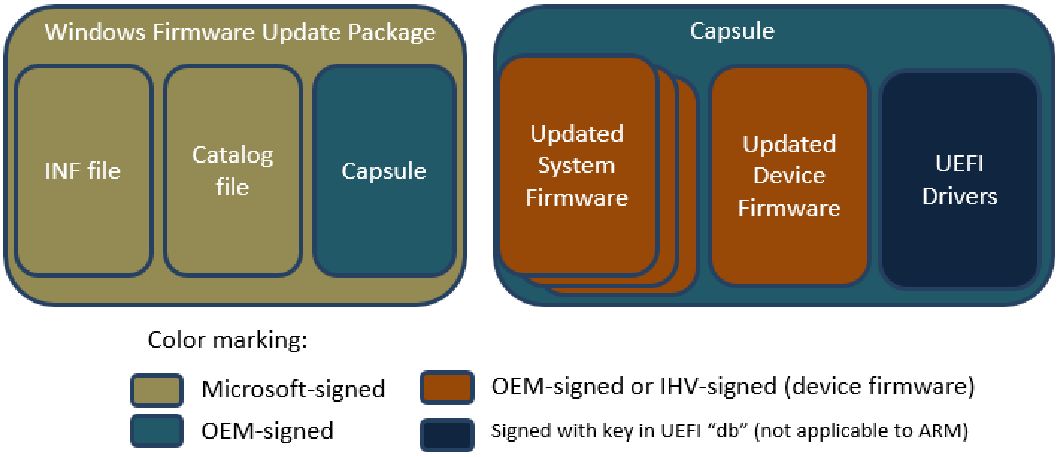

Figure 1 – Firmware Update Components and Signers
Source: Windows UEFI Firmware Update Platform [WFU]

Figure 1 shows the components involved in a Windows UEFI Firmware Update Platform. This document defines a capsule package format used in Windows, which is the “Microsoft Signed” box below. It does not define the capsule image format, which is the “OEM-signed” box below.

UEFI Specification

The UEFI specification [UEFI] defines **UEFI capsule services**, **EFI_FIRMWARE_MANAGEMENT_PROTOCOL**, **FMP capsule format** and **EFI System Resource Table (ESRT)** to support system firmware and device firmware update.

The platform may consume a system FMP and a device FMP protocol to get the firmware image information and report the ESRT to a UEFI OS.

An OS agent may call the UEFI service UpdateCapsule() to pass the capsule image from the OS to the firmware. Based upon the capsule flags, the firmware may process the capsule image immediately, or the firmware may reset the system and process the capsule image on the next boot.

The FMP capsule image format is shown in figure 2. It contains the update FMP drivers and the FMP payloads. The FMP payload contains the binary update image and optional vendor code. The platform may consume a FMP protocol to update the firmware image.

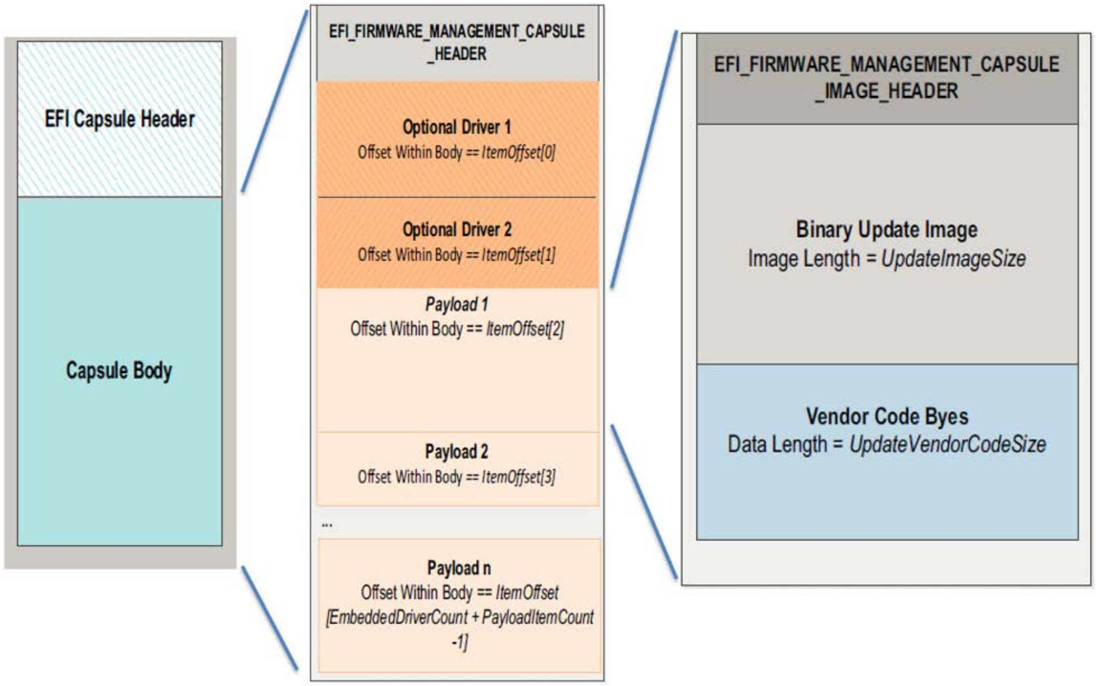
EFI Capsule Header
Capsule Body
EFI_FIRMWARE_MANAGEMENT_CAPSULE_HEADER
Optional Driver 1
Offset Within Body == ItemOffset[0]
Optional Driver 2
Offset Within Body == ItemOffset[1]
Payload 1
Offset Within Body == ItemOffset[2]
Payload 2
Offset Within Body == ItemOffset[3]
...
Payload n
Offset Within Body == ItemOffset [EmbeddedDriverCount + PayloadItemCount -1]
EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER
Binary Update Image
Image Length = UpdateImageSize
Vendor Code Byes
Data Length = UpdateVendorCodeSize

Figure 2 – UEFI FMP Capsule Format
Source: UEFI specification [UEFI]

This document defines the detailed processing of the capsule image and the capsule image format, but it does not define the format of binary update image. The latter can be implementation choice.

PI Specification

The UEFI Platform Initialization specification [PI] defines EFI_PEI_CAPSULE_PPI to handle UEFI capsule coalesce process (the detail will be discussed later). If there is a valid capsule, the PI boot mode is set to BOOT_ON_FLASH_UPDATE and the UEFI Capsule HOB is produced, so that at a later point the platform update driver may consume this information to process capsule images.

This document defines how to report a capsule image in the firmware. It also does not define the capsule image format.

Summary
This section introduces the firmware update related specifications.

UEFI FMP Capsule Update

For security considerations, the flash device is locked during the system boot. As such, it might not be possible to update the flash device directly in an operating system.

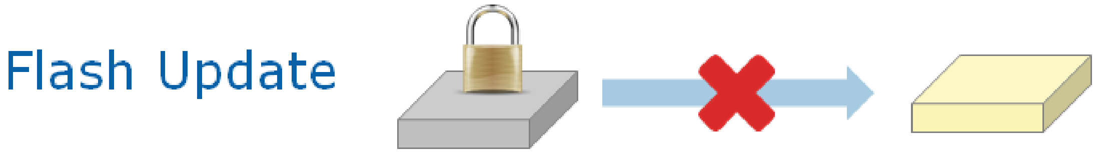

Figure 3 – Direct Flash Update

As an alternative, one possible way is that the operating system sends a firmware image to the system and lets the system update the flash device before the flash device is locked in next boot. See figure 4. This firmware image is called a "capsule". The UEFI specification defines a set of capsule services to let operating system pass the information to the firmware. The UEFI specification also defines the capsule image format for the EFI_FIRMWARE_MANAGEMENT_PROTOCOL (FMP).

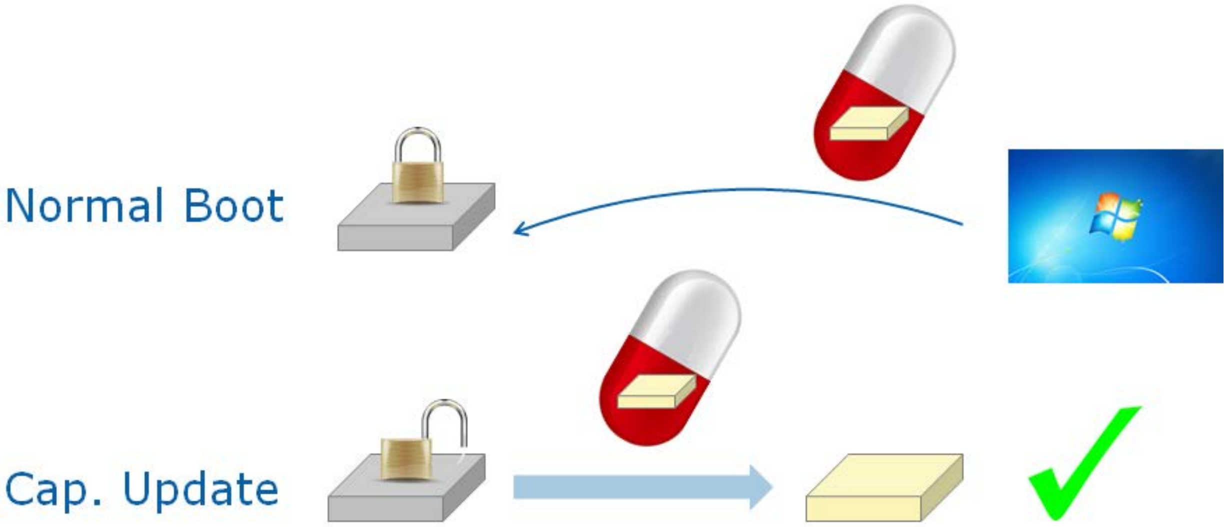

Figure 4 – UEFI Capsule Based Firmware Update

EDKII provides the sample drivers to performance the signed system firmware update via the capsule interface. It follows the NIST standard, Microsoft design and the UEFI specification.

Capsule Update General Flow

The capsule update general flow is below (See figure 5):

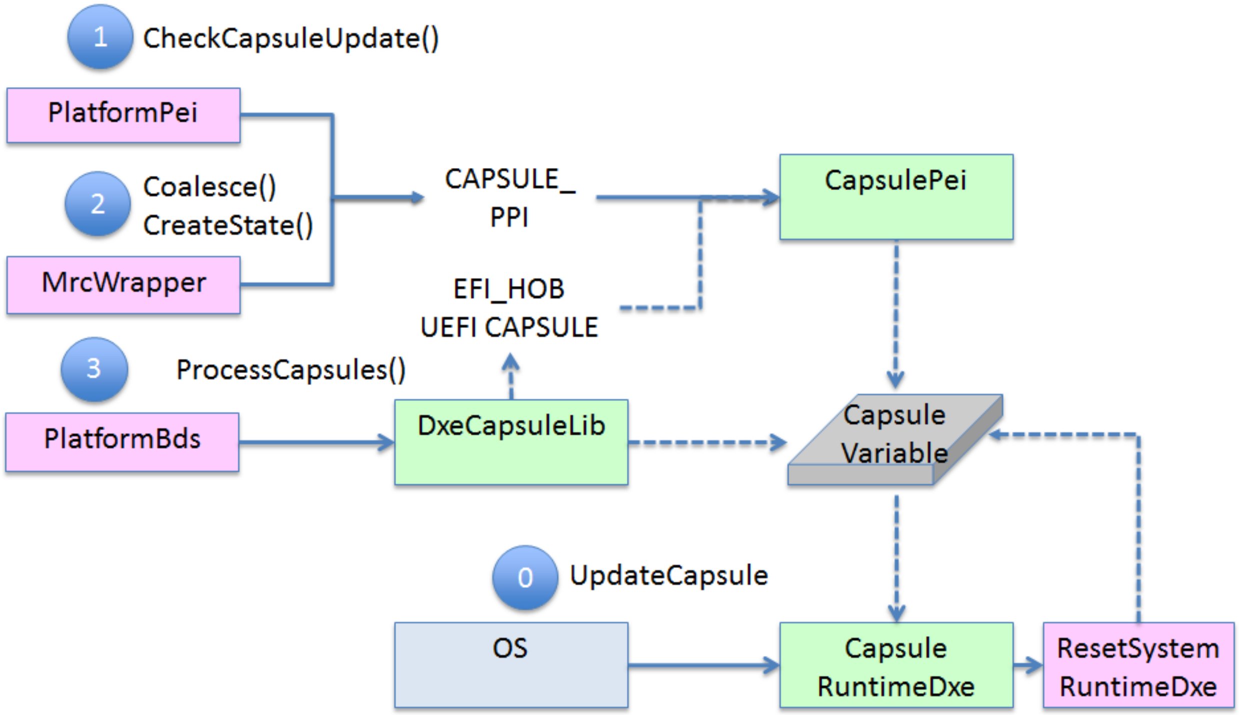

Figure 5 – Capsule Update General Flow

0) In a UEFI OS, an OS module calls Capsule runtime service **EFI_RUNTIME_SERVICES.UpdateCapsule()** to pass the capsule image to the firmware with the reset capability. (The service is provided by https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/CapsuleRuntimeDxe) After that, the system resets.

1) On the next boot, a platform PEI module (such as https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Platform/Pei/PlatformInit/MrcWrapper.c) calls **PEI_CAPSULE_PPI.CheckCapsuleUpdate()** to determine if a capsule needs to be processed. The CapulePei module (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/CapsulePei) will read the **L"CapsuleUpdateData"** variable to determine if there are capsules in the memory. If the condition is true, this PEIM module calls **EFI_PEI_SERVICES.SetBootMode(**BOOT_ON_FLASH_UPDATE**).**

   NOTE: In many cases, a capsule is used for firmware update. As such, many platforms choose ***BOOT_ON_FLASH_UPDATE*** *as the Boot mode to indicate that there is capsules need to be processed. However, a capsule might be used for other purposes, such as information passing, and a capsule is only required to be populated in the EFI_SYSTEM_TABLE. Care must be taken to handle the capsule images other than firmware update.*

2) After the Memory Reference Code (MRC) module initializes system memory, the platform PEI module (such as https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Platform/Pei/PlatformInit/MrcWrapper.c) calls **PEI_CAPSULE_PPI.Coalesce()** to determine the current location of the various

capsule fragments and coalesce them into a contiguous region of system memory. Then the platform PEI module calls **EFI_PEI_SERVICES.InstallPeiMemory()**, to register the found memory configuration with the PEI Foundation. The contiguous region for capsule must be excluded in the PEI memory. Finally, the platform PEI module calls **PEI_CAPSULE_PPI.CreateState()** to copy the capsules into PEI memory and to create a **EFI_HOB_UEFI_CAPSULE**. At this point, the following PEI/DXE/SMM modules can know the location of the coalesced UEFI capsule memory pages.

3) After the platform enters the BDS phase, the BDS detects boot mode (such as https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Library/PlatformBootManagerLib/PlatformBootManager.c) If the current boot mode is **BOOT_ON_FLASH_UPDATE**, the BDS calls **CapsuleLib:ProcessCapsules()** (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/DxeCapsuleLibFmp) to process capsule images. BDS should call **ProcessCapsules()** twice. The first call must be before **EFI_END_OF_DXE_EVENT**, because the system flash part is locked after **EFI_END_OF_DXE_EVENT**. The system capsule must be processed in first call. If a device capsule FMP protocol is produced and the device capsule FMP has zero EmbeddedDriverCount, the device capsule is also processed.

4) **ProcessCapsules()** parses CAPSULE_HOB and processes the satisfied capsule images one by one. If a Windows UX capsule exists, it is processed first. If the capsule image is a UEFI defined FMP capsule, the **ProcessFmpCapsuleImage()** is called.

5) **ProcessFmpCapsuleImage()** locates all FMP protocols, calls **Fmp->GetImageInfo()** to get the firmware image information and compares with the FMP capsule. Only if the ImageTypeId, ImageIndex and HardwareInstance in FMP protocol and FMP capsule matched, then ProcessFmpCapsuleImage() calls **Fmp->SetImage()** to perform the firmware update. If the update returns success, ProcessFmpCapsuleImage() records the information in the capsule status variable so that the same image will not be processed twice.

6) After **EFI_END_OF_DXE_EVENT** and after ConnectAll(), the BDS calls **ProcessCapsules()** again. At that time, all device capsule FMP protocols are exposed. If a capsule is processed in the first call, it is not processed in the second call. Alternately, if a capsule is not processed in the first call, it is processed in the second call. After all the capsules are processed, the system may reset if the reset is required by one of the capsules processed in the first call or second call.

Execution Environment

Per the NIST guideline, the flash update operation must happen in a secure environment. It could be in firmware boot phase before EndOfDxe, or in SMM. The EDKII Capsule update solution chooses to update the system firmware before EndOfDxe because the flash region remains open until the EndOfDxe. This latter usage is sometimes called 'temporal isolation' since no 3rd party code should execute prior to the EndOfDxe. Only system board vendor PI modules and drives should execute.

Authentication Check

The capsule update solution needs to follow NIST standard to verify the image.

FMP Payload check

- **Signing check**: The firmware update capsule must be signed so that the update mechanism can check the integrity of the contents. The UEFI FMP capsule uses the **EFI_FIRMWARE_IMAGE_AUTHENTICATION** before the real payload image, and it requires the certificate type to be **EFI_CERT_TYPE_PKCS7_GUID**. The authentication information is the PKCS7 signature. The capsule update mechanism extracts the "Trusted Cert" from the firmware boot block (FvRecovery) and uses a "Trusted Cert" to verify the PKCS7 signature. If a capsule does not have **EFI_FIRMWARE_IMAGE_AUTHENTICATION** or the signing verification fails, this capsule should be ignored.
  In EDKII, the signature check is done in the FmpAuthenticationLib at https://github.com/tianocore/edk2/tree/master/SecurityPkg/Library/FmpAuthenticationLibPkcs7
- **Version check**: Besides the signature verification, the update mechanism should also check the capsule version to prevent a rollback attack. There are both **Version** and **LowestSupportedImageVersion** fields in **EFI_FIRMWARE_IMAGE_DESCRIPTOR**. The version check requires that the **Version** in capsule image must be greater than or equal to the **LowestSupportedImageVersion** in the current firmware. If the version in capsule image is lower than that in the current firmware, this capsule should be dropped.
  In EDKII, one image version check sample for capsule update is done in the EdkiiSystemCapsuleLib at https://github.com/tianocore/edk2/tree/master/SignedCapsulePkg/Library/EdkiiSystemCapsuleLib.
  If other signed capsule solution is used, it need provide similar function to check the version.

FMP driver check

- The FMP driver should be a normal UEFI driver. The recommendation is to just reuse UEFI secure boot to validate the integrity of the FMP driver.
  If the secure boot is disabled, there could be a platform policy to determine if the FMP driver should be launched. At a minimum, this FMP driver should be launched after EndOfDxe. At that time, the system flash is locked.

Below figure 6 shows a sample of device firmware capsule verification. It has a device update driver in capsule image. UEFI secure boot needs to verify the device update driver. The device update driver needs to verify the device firmware payload in the FMP capsule.

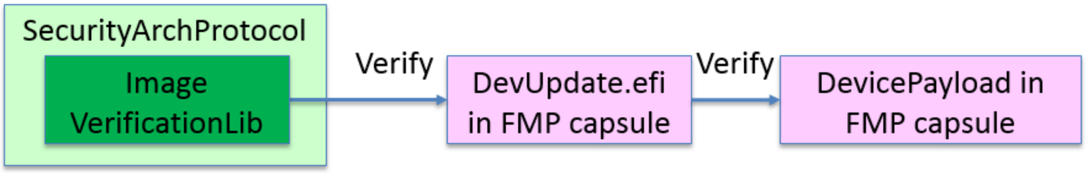

Figure 6 – FMP capsule Driver and Payload Authentication

The device payload verification is device implementation specific. It may use the UEFI specification defined **EFI_FIRMWARE_IMAGE_AUTHENTICATION** or use a hardware mechanism, such as Intel CPU Microcode Load. (The details of Microcode Update will be discussed later.)

Capsule image not for firmware update

The UEFI capsule is designed to allow a caller to pass information to the firmware. Update Capsule is commonly used to update the firmware FLASH. It can also be used for an operating system to have information persist across a system reset, such as crash dump information.

If a capsule has **CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE** flag, it should be populated into the EFI configuration table. This latter case might not be for firmware update.

Otherwise, the capsule image is process according to **EFI_CAPSULE_HEADER.CapsuleGuid**. For example, if the GUID is **WINDOWS_UX_CAPSULE_GUID**, this capsule contains BMP file information and is used to display the screen during firmware update. [WFU]. The code is also handled in https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/DxeCapsuleLibFmp

PI Capsule Coalesce

The capsule contents are designed to be communicated from an OS-present environment into the system firmware. The OS allocates contiguous virtual address space and describes this address space to the firmware as a discontinuous set of physical address ranges. The firmware is passed both physical and virtual addresses and pointers to describe the capsule so the firmware can process the capsule immediately or defer processing of the capsule until after a system reset.
See figure 7. The left side of the figure shows the OS view of the capsules as two separate contiguous virtual memory buffers. The center of the figure shows the layout of the data in system memory. The right hand side of the figure shows the scatter gather list passed into the firmware.
EFI_PEI_CAPSULE_PPI.Coalesce() parses the current location of the various capsule fragments and coalesces them into a contiguous region of system memory.

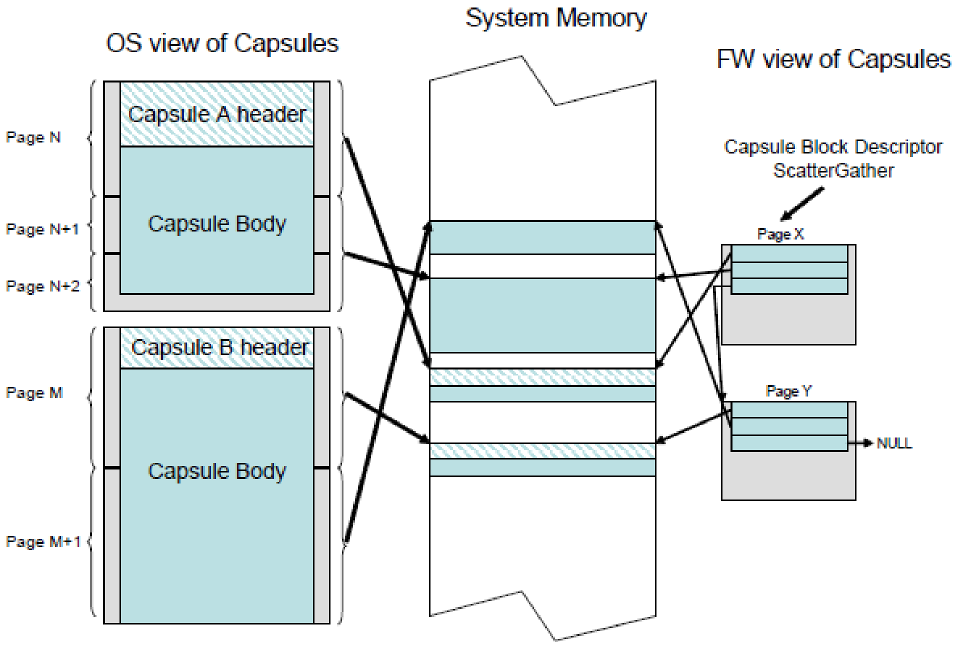

Figure 7 –Scatter-Gather List of UEFI Capsule
Source: UEFI specification [UEFI]

The capsule coalesce action needs to parse the scatter gather list, which is an external source. It must do some basic sanity checking of the scatter gather. The EDKII capsule PEIM driver does such checks based upon the EFI_RESOURCE_DESCRIPTION_HOB. The memory initialization component (such as MRC) should report all DRAM regions via EFI_RESOURCE_DESCRIPTION_HOB. If the capsule fragment is inside of DRAM, it is considered as valid, or it is invalid and this capsule should be rejected. See figure 8.

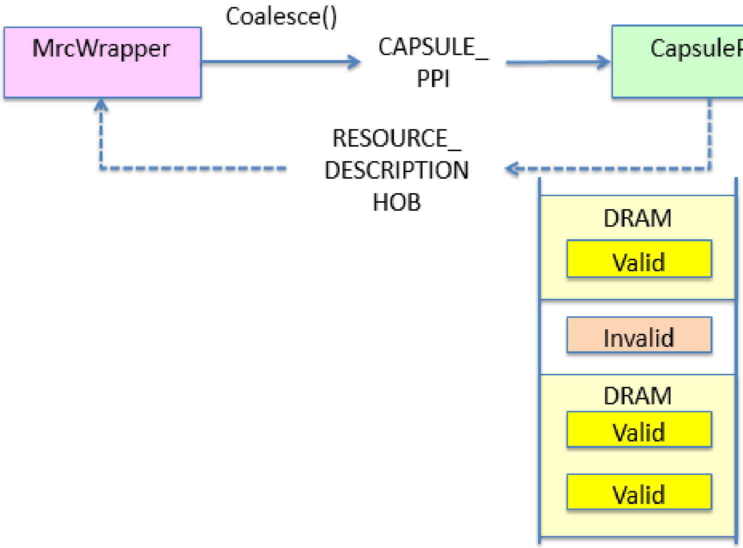

Figure 8 – Capsule Coalesce Check

Capsule Coalesce Flow

See figure 9.0 for the general coalesce flow. The CapsulePeim (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/CapsulePei) driver gets capsule memory from memory initialization component. It parses the capsule scatter gather list, puts all of the **EFI_CAPSULE_BLOCK_DESCRIPTOR** entries at the end of capsule memory, allocates free memory and puts all capsule fragments together in the capsule memory.

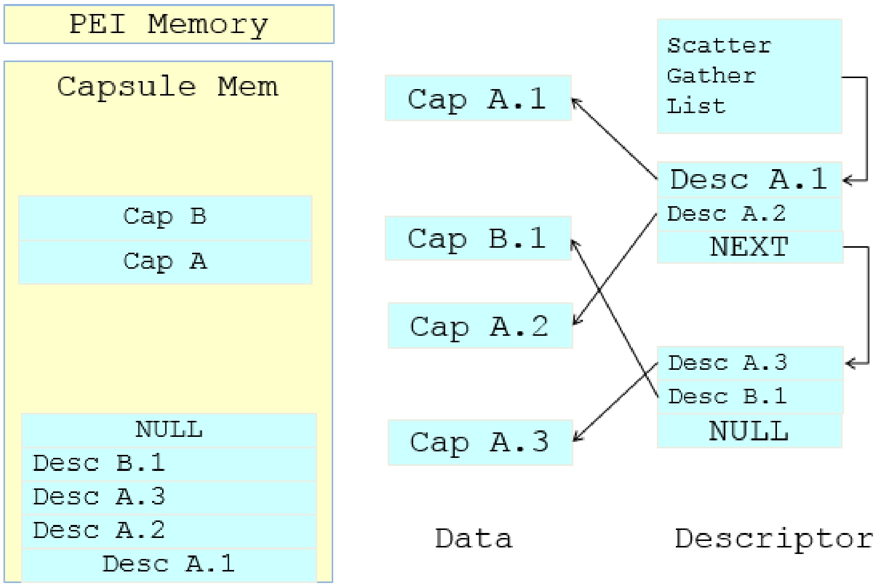

Figure 9.0 – Capsule Coalesce

The details of the capsule coalesce flow is below:

- Step 0: Collect capsule information. The CapsuleDataCoalesce() function calls BuildCapsuleDescriptors() to build **EFI_CAPSULE_BLOCK_DESCRIPTOR**. The capsule integrity is also checked here, when the **EFI_CAPSULE_BLOCK_DESCRIPTOR** is built. Then GetCapsuleInfo() is called to traverse the **EFI_CAPSULE_BLOCK_DESCRIPTOR** to collect capsule information, such as the number of descriptors, the number of capsules, and the capsule size. This information is saved to the variables, including the state EFI_CAPSULE_PEIM_PRIVATE_DATA and EFI_CAPSULE_BLOCK_DESCRIPTOR. This state information is still in temporary RAM so there are no capsule fragments in the stack.
- Step 1: There is an important function FindFreeMem() in capsule PEIM. The input parameters are a pointer to the capsule block list, the information on the available system memory, and the size of a candidate buffer. This function finds a free block inside of the available system memory, where the candidate buffer of the given size can be copied to safely. The FindFreeMem() use below logic:
  - Step 1.1: Initialize state. The free buffer candidate is on the bottom of the available memory.

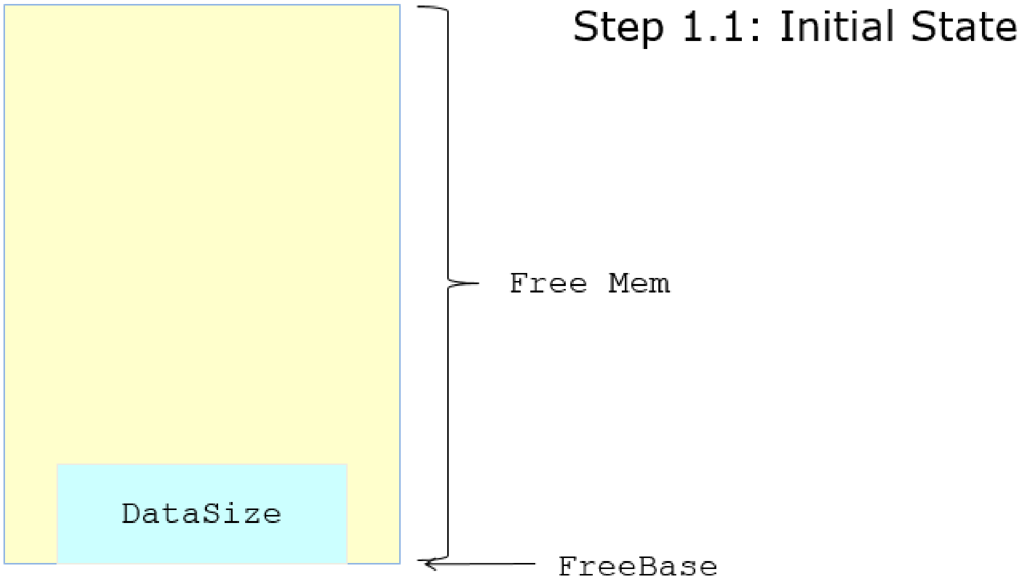

Figure 9.1 – Capsule Coalesce – FindFreeMem() step 1.1

  - Step 1.2: Overlap check for descriptor blocks. It goes through all the descriptor blocks to see if there is overlap between the free buffer candidate and descriptor blocks.

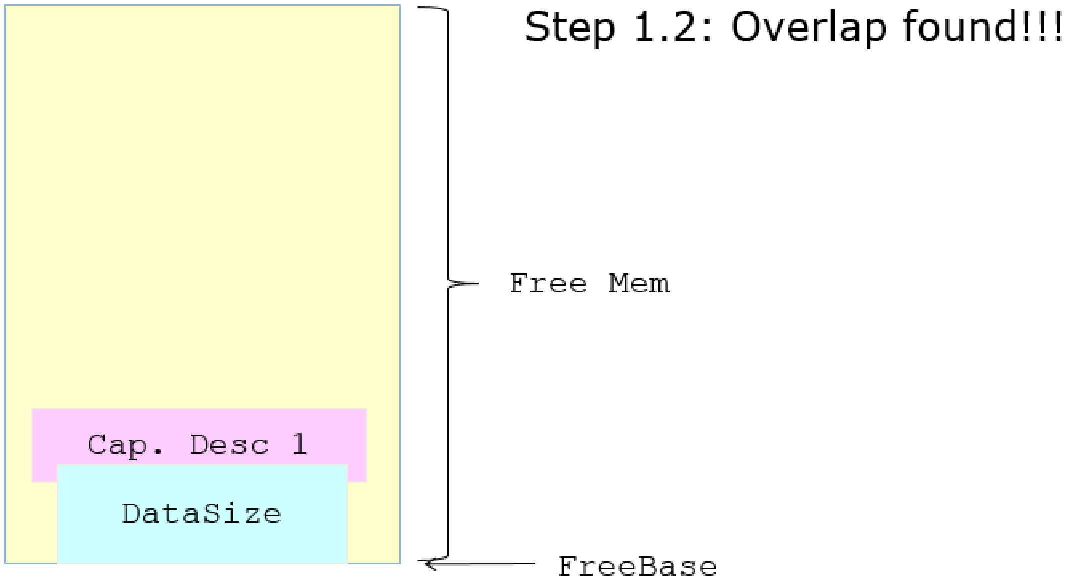

Figure 9.2 – Capsule Coalesce – FindFreeMem() step 1.2

- Step 1.3: Move data up in case of overlap. If there is overlap, then the free buffer candidate is moved up to eliminate the overlap.

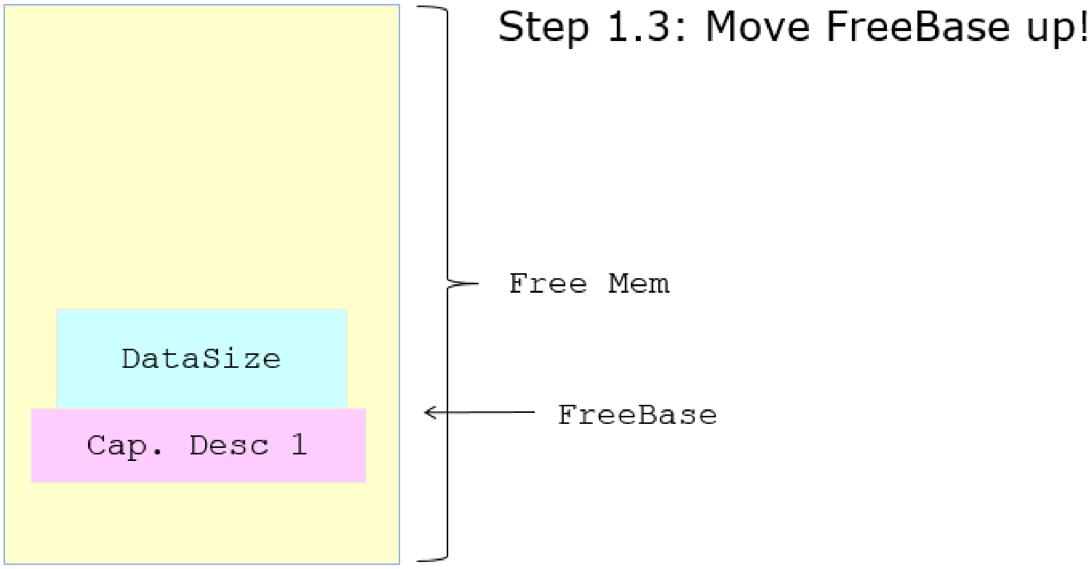

Figure 9.3 – Capsule Coalesce – FindFreeMem() step 1.3

- Step 1.4: Overlap check for capsule data fragment. It goes through all the capsule data fragment to see if there is overlap between the free buffer candidate and capsule data fragment.

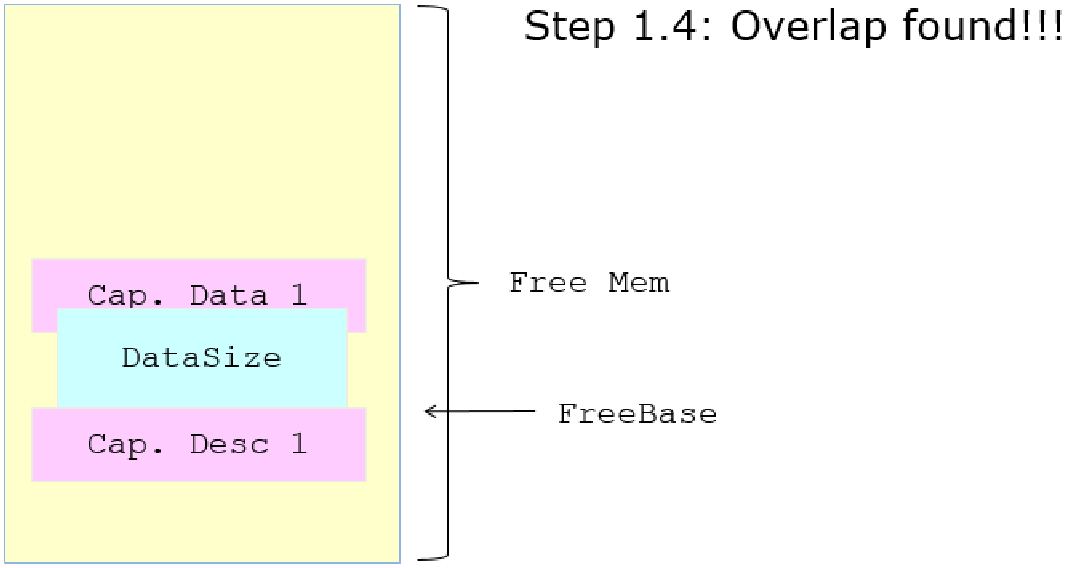

Figure 9.4 – Capsule Coalesce – FindFreeMem() step 1.4

- Step 1.5: Move data up in case of overlap. If there is overlap, then the free buffer candidate is moved up to eliminate the overlap.

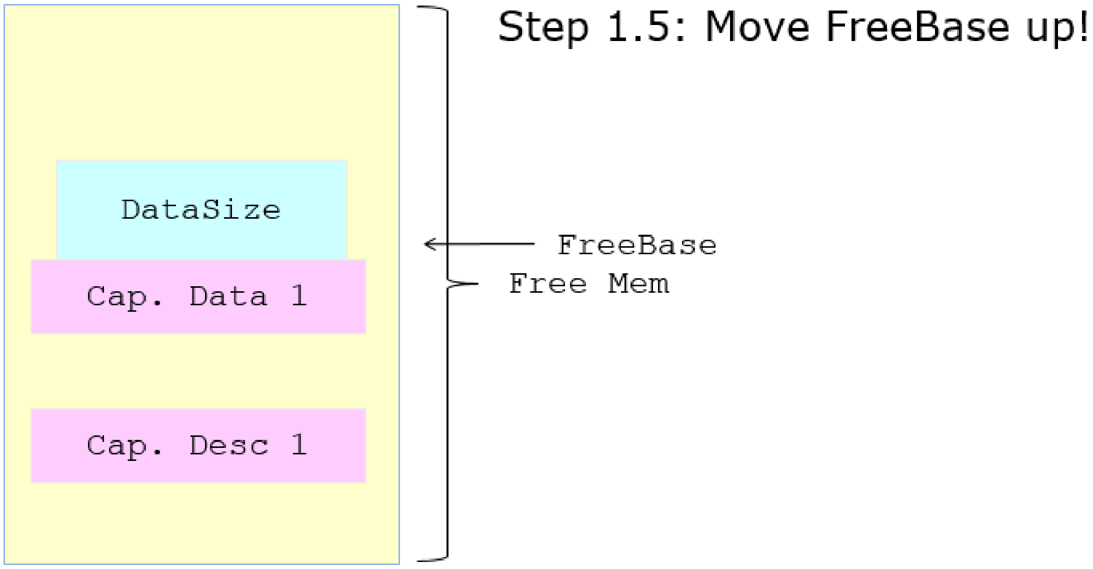

Figure 9.5 – Capsule Coalesce – FindFreeMem() step 1.5

- Step 2: Relocate capsule block descriptors.
  - Step 2.1: The Initialize state. The capsule block descriptors candidate is at the bottom of the available memory. NOTE: Here the candidate is only allocated and the memory content is not touched at this moment.

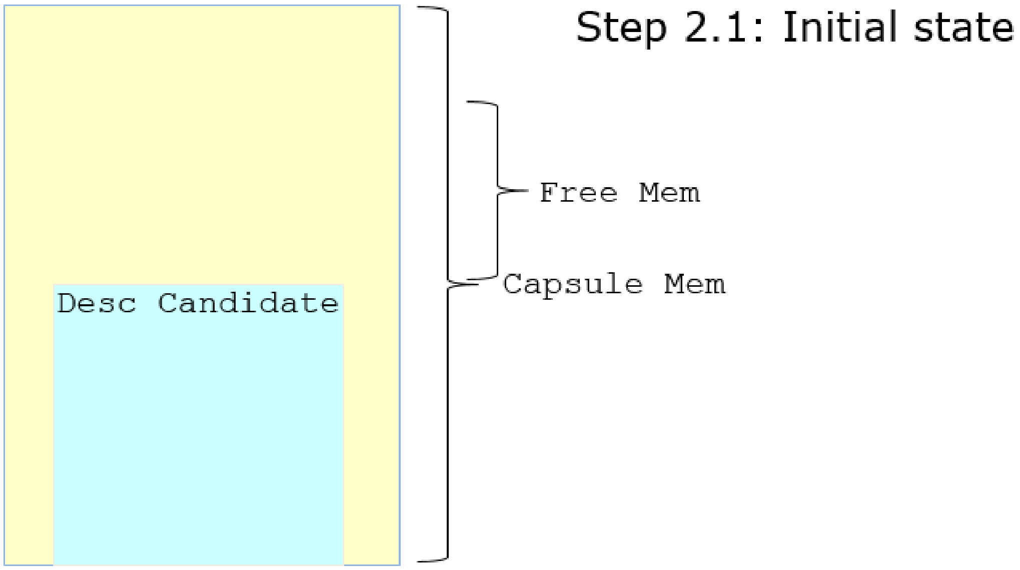

Figure 9.6 – Capsule Coalesce – RelocateBlockDescriptors () step 2.1

– Step 2.2: Overlap check. It goes through all the descriptor blocks and capsule fragments to see if there is overlap between the capsule block descriptor candidate and descriptor blocks or the capsule fragments.

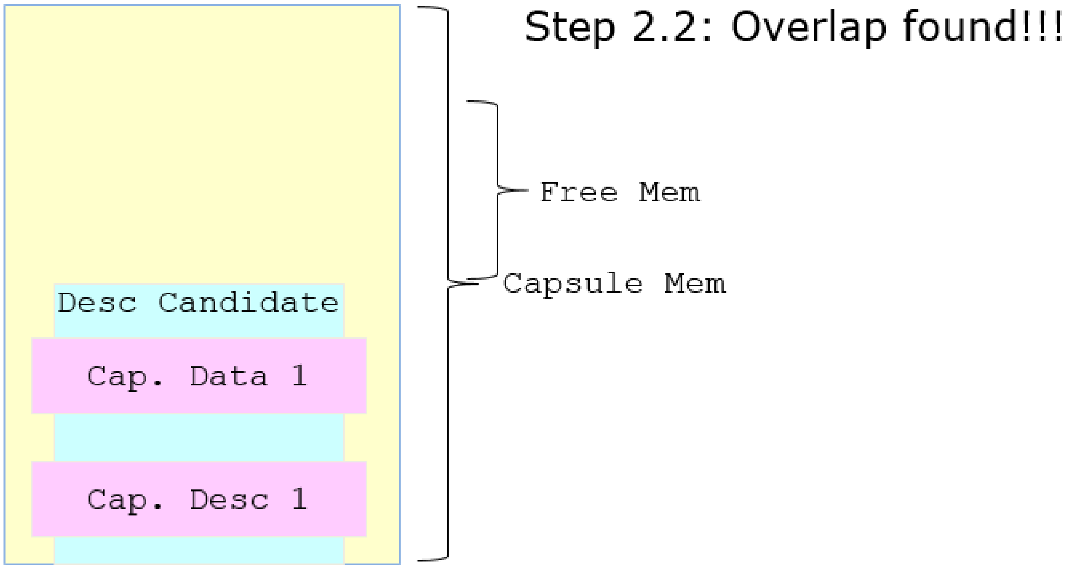

Figure 9.7 – Capsule Coalesce – RelocateBlockDescriptors () step 2.2

– Step 2.3: Move data up in case of overlap. If there is overlap, then the overlapped descriptor blocks or capsule fragments are moved up to free memory. FindFreeMem() is used to find out the free memory location. The candidate memory is still in its original place and untouched.

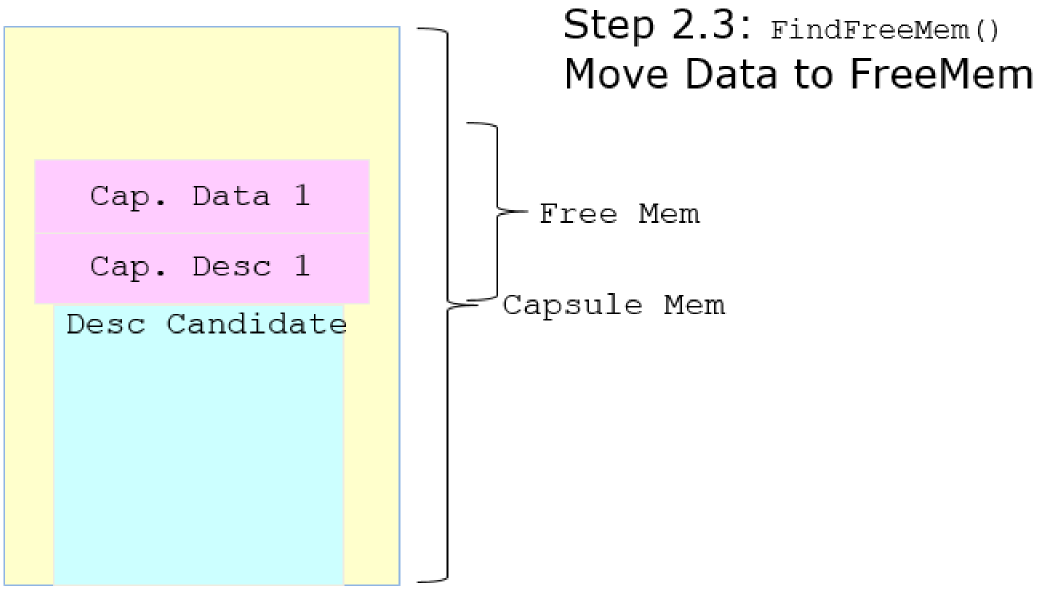

Figure 9.8 – Capsule Coalesce – RelocateBlockDescriptors () step 2.3

- Step 2.4: Relocate. Now there is no useful content in candidate memory. The capsule block descriptors are relocated into the candidate memory.

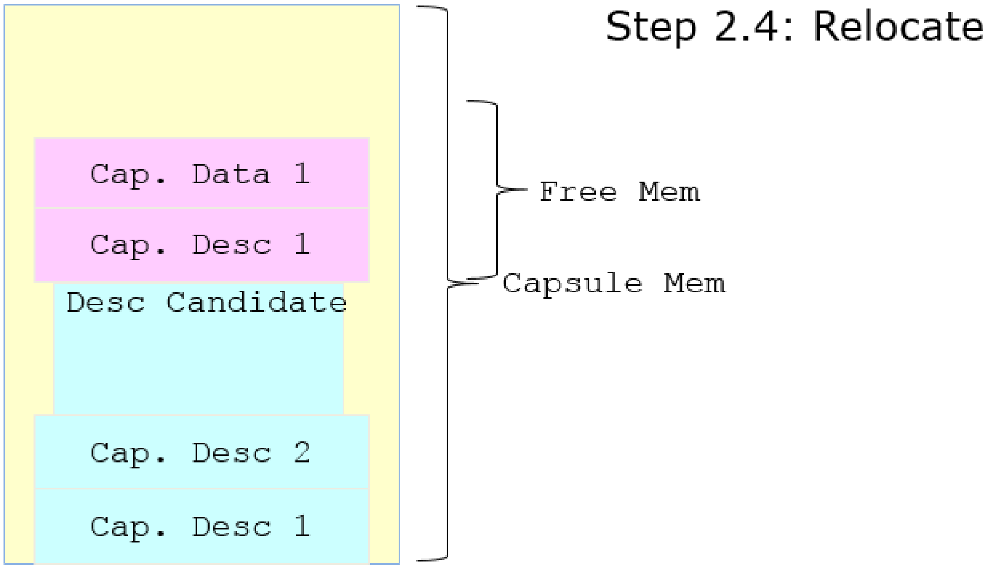

Figure 9.9 – Capsule Coalesce – RelocateBlockDescriptors () step 2.4

- Step 3: Relocate capsule data fragment.
  - Step 3.1: The Initialize state. The capsule data fragment candidate is on the top of the available memory. NOTE: Here the candidate is only allocated, the memory content is not touched at this moment.

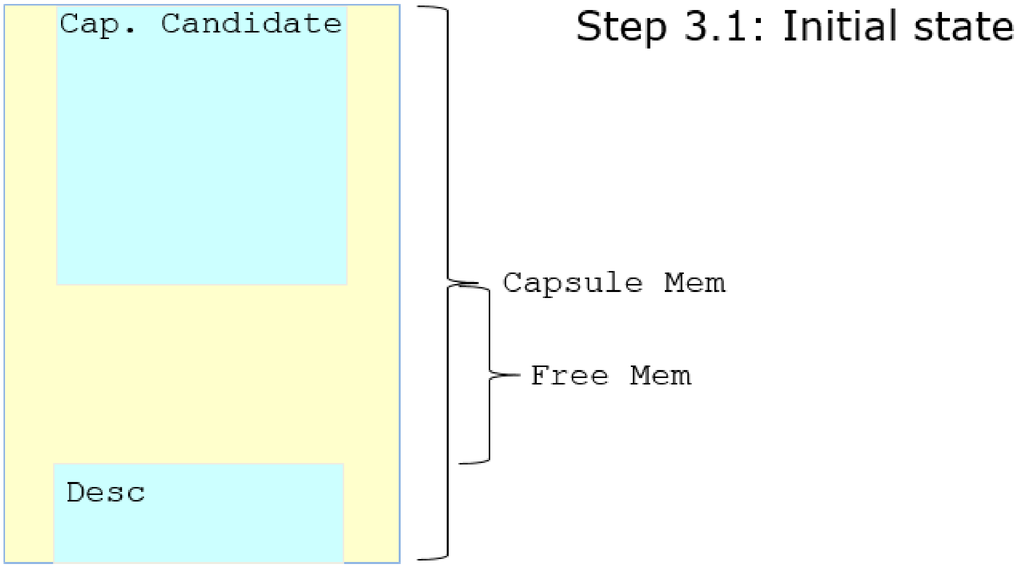

Figure 9.10 – Capsule Coalesce – CapsuleDataCoalesce () step 3.1

– Step 3.2: Overlap check. It goes through all the capsule fragment to see if there is overlap between the capsule block descriptor candidate and capsule fragment.

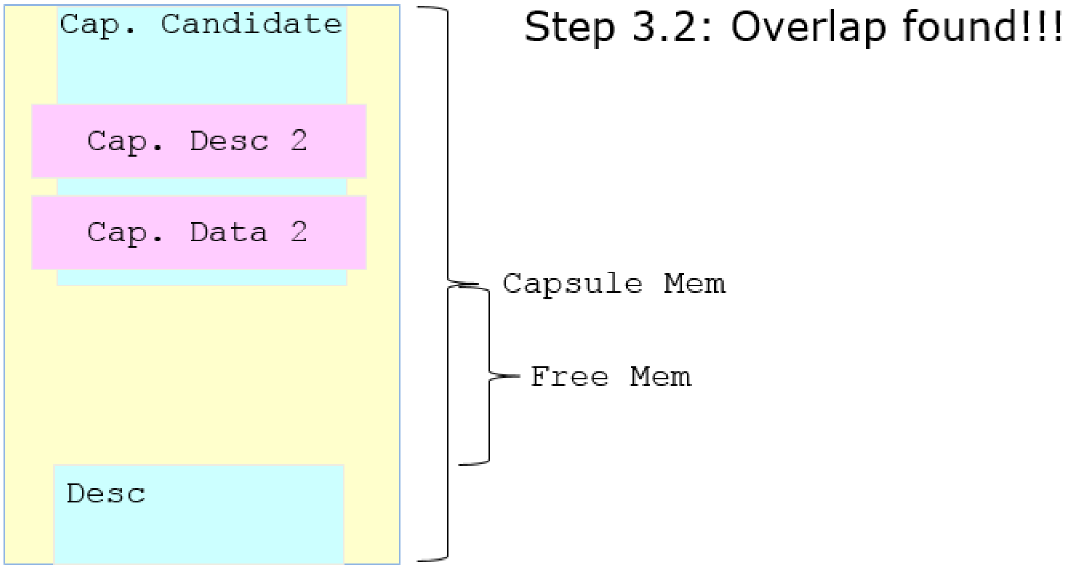

Figure 9.11 – Capsule Coalesce – CapsuleDataCoalesce () step 3.2

– Step 3.3: Move data down in case of overlap. If there is overlap, then the overlapped capsule fragment is moved down to free memory. FindFreeMem() is used to find out the free memory location. The candidate memory is still in original place and untouched.

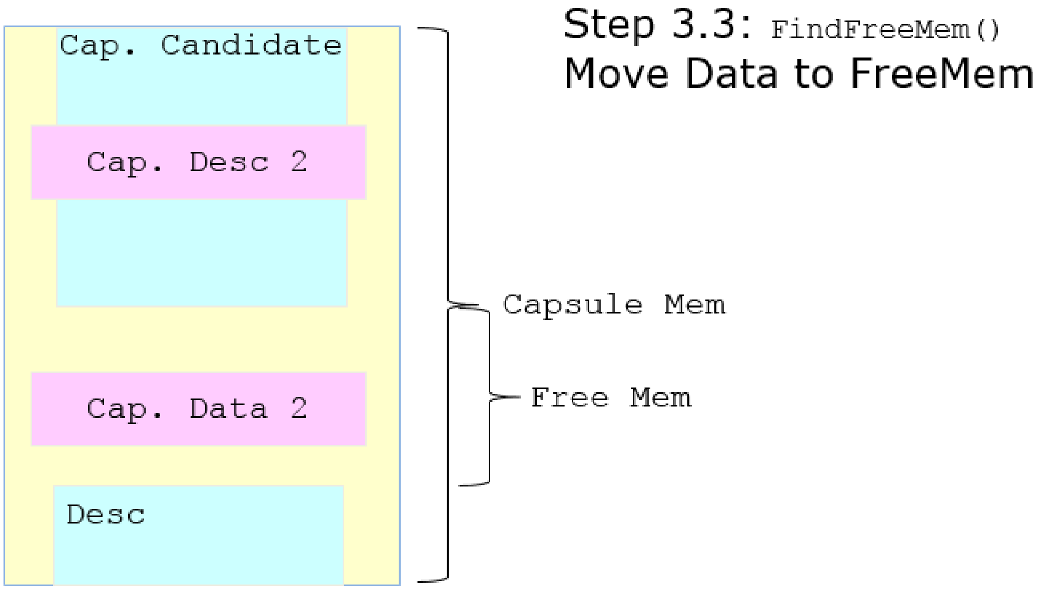

Figure 9.12 – Capsule Coalesce – CapsuleDataCoalesce () step 3.3

- Step 3.4: Relocate. Now there is no useful content in candidate memory. The capsule data fragments are relocated into the candidate memory.

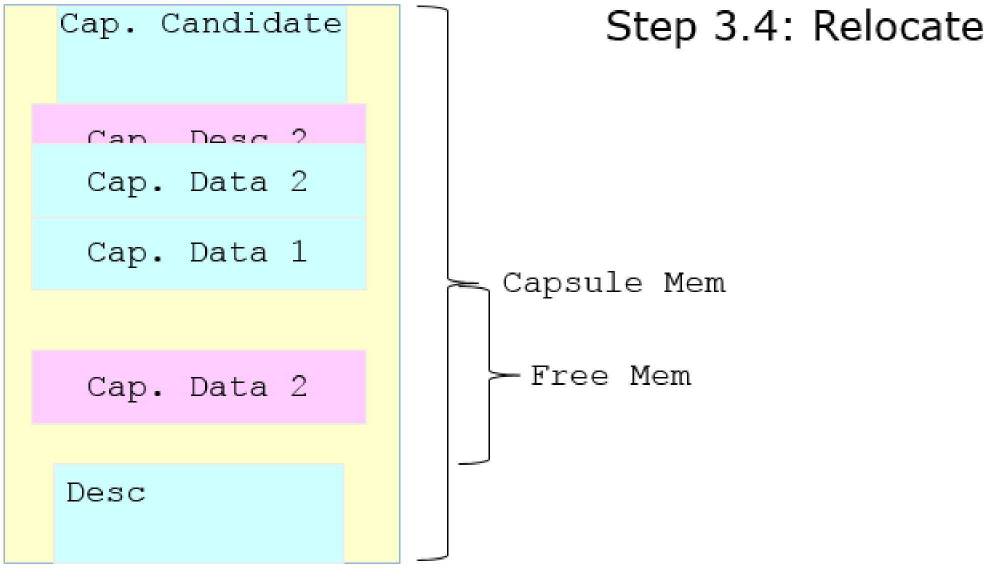

Figure 9.13 – Capsule Coalesce – CapsuleDataCoalesce () step 3.4

- Finally, there is a list of EFI_CAPSULE_BLOCK_DESCRIPTOR on the bottom of capsule memory. There is an EFI_CAPSULE_PEIM_PRIVATE_DATA on the top of the capsule memory, where the capsule images are coalesced. The free memory is in the middle.

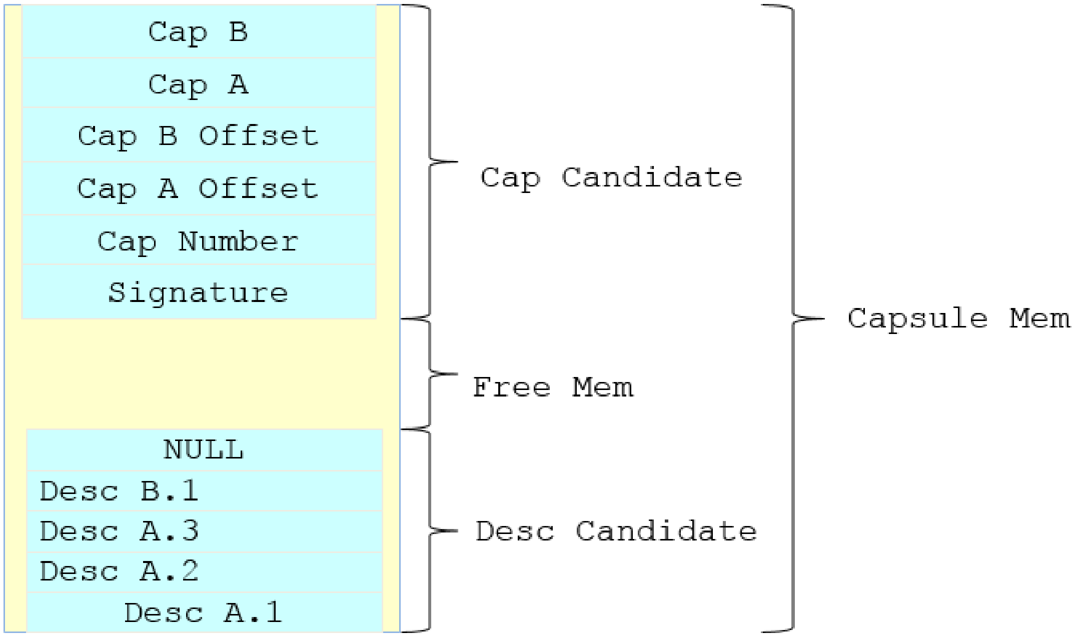

Figure 9.14 – Capsule Coalesce – Internal Data Structure

Capsule Coalesce for IA32 PEI and X64 DXE

When UEFI capsule services are invoked in the OS, the OS assumes the firmware can handle the memory reported via the UEFI memory map. As such, the capsule images should be covered by the UEFI memory map. There is a special configuration in Intel X86 firmware - the PEI phase is 32bit and the DXE phase is 64bit. The mode transition happens in DXE IPL. In such a case, the 32bit PEI Capsule module should handle the 64bit address space because the OS may put the capsule block descriptor or data fragments above 4GiB memory.

In EDKII, if the system configuration is **PcdDxeIplSwitchToLongMode** is TRUE, it means PEI is IA32 mode and DXE is X64 mode. In a normal boot, the CapsuleRuntimeDxe driver (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/CapsuleRuntimeDxe) prepares the X64 mode execution context and saves it into **L"CapsuleLongModeBuffer"** variable. In BOOT_ON_FLASH_UPDATE boot mode, the CapsulePei driver (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/CapsulePei) consumes the **L"CapsuleLongModeBuffer"** variable, constructs the X64 context, and switches to X64 mode to coalesce the capsule scatter gather list. This **L"CapsuleLongModeBuffer"** variable content should be locked to keep the integrity.

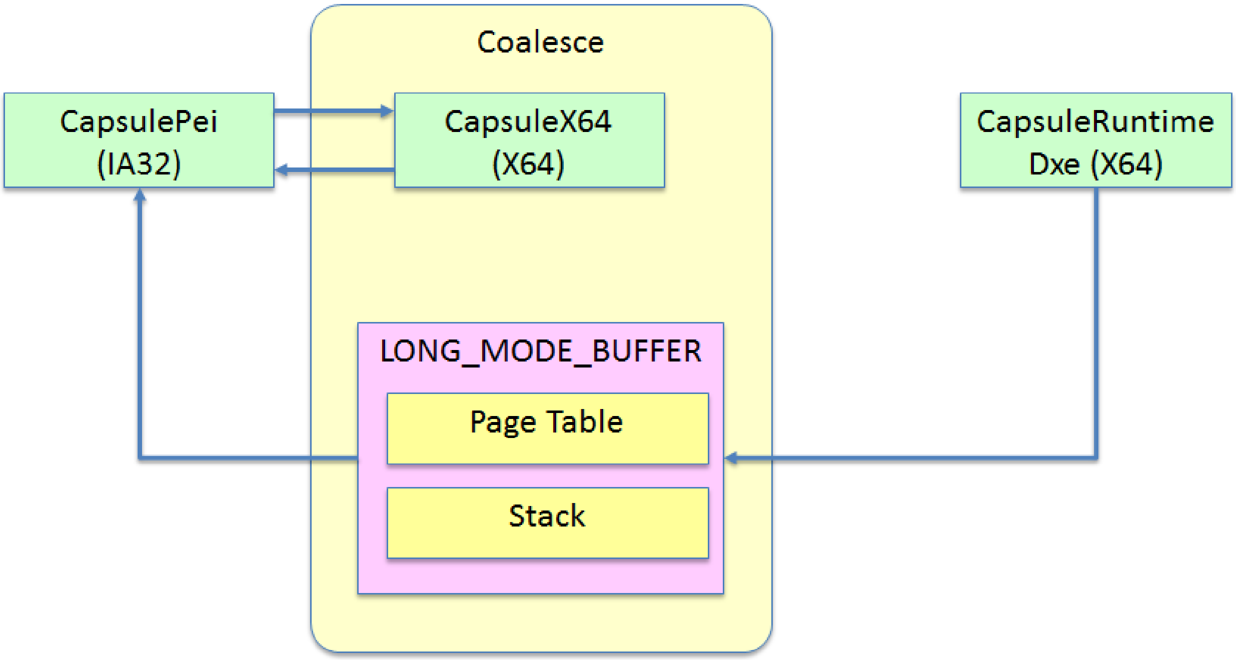

Figure 10 – Capsule Coalesce for IA32 PEI/X64 DXE

Summary

This section introduces the capsule update in the UEFI/PI firmware.

EDKII System FMP Capsule Update

Because the UEFI specification does not define detailed format for the FMP binary update image, EDKII defines its own system firmware binary update image format. This format is EDKII specific. The IBV/OEM may choose different image formats to perform the same function.

EDKII System Firmware Binary Update Image Format

EDKII system FMP capsule reuses the UEFI specification defined FMP capsule with **EFI_FIRMWARE_IMAGE_AUTHENTICATION.** The payload image is one PI Firmware Volume. It includes:

- Firmware update configuration file (SysFirmUpdateConfig.ini). This configuration file describe the content in the system firmware. (Such as https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Feature/Capsule/SystemFirmwareUpdateConfig/SystemFirmwareUpdateConfig.ini)
  The format of the capsule update configuration is defined at https://github.com/tianocore/edk2/blob/master/SignedCapsulePkg/Include/Guid/EdkiiSystemFmpCapsule.h. User may choose to use one entry to describe the whole FD image directly, or to use multiple entries to describe each FV regions one by one. User must make sure that the FirmwareType field is filled correctly to describe if the component is used as a variable region. More detail is discussed in “Variable Update Consideration” later.
- Driver FV. This is a nested FV, which contains a system firmware update FMP driver. (SysFirmUpdate.efi). (https://github.com/tianocore/edk2/tree/master/SignedCapsulePkg/Universal/SystemFirmwareUpdate)
- System Firmware Binary. This is raw binary data to be updated into flash region.
- Image Description. In FvRecovery of the system firmware, there is an image description data. It contains the data in **EFI_FIRMWARE_IMAGE_DESCRIPTOR**. (Such as https://github.com/tianocore/edk2/tree/master/QuarkPlatformPkg/Feature/Capsule/SystemFirmwareDescriptor)

See figure 11.

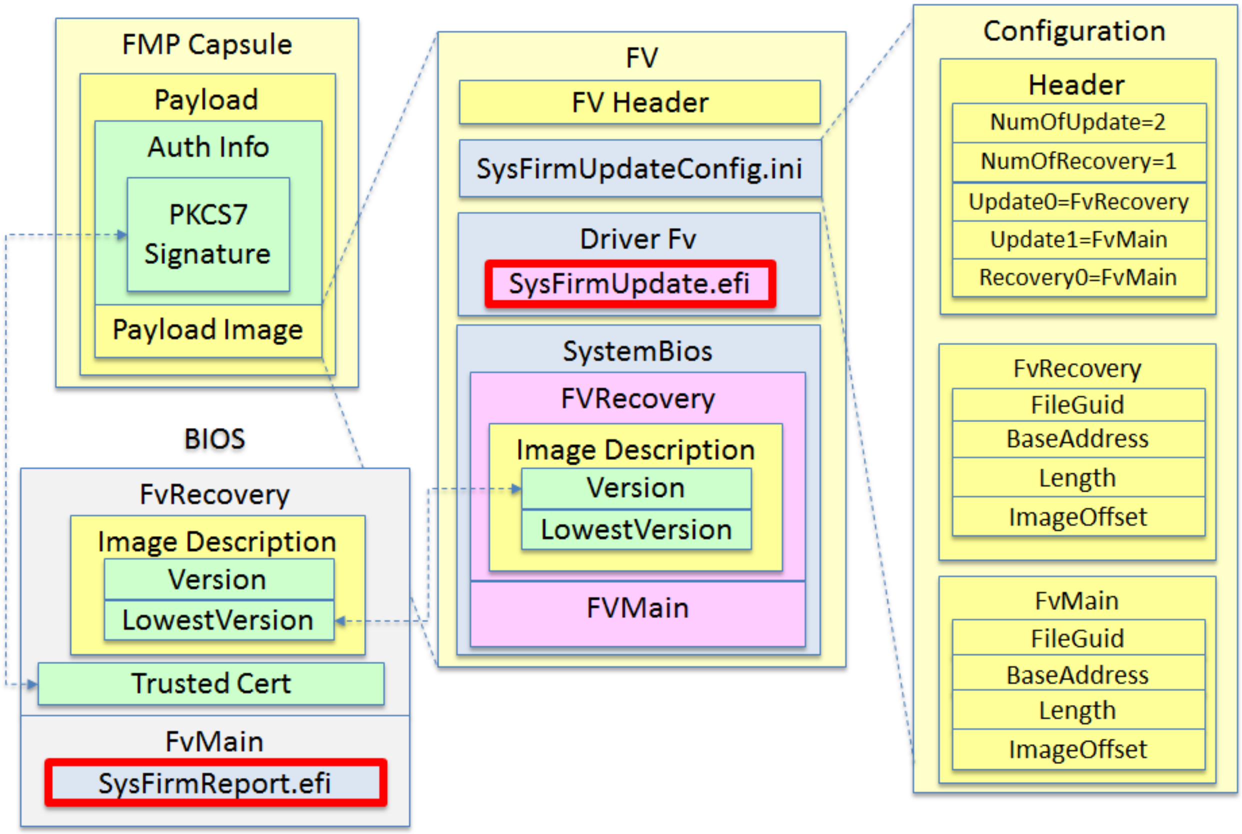

Figure 11 – EDKII Signed System FMP capsule format

Authentication Check

Below figure 12 shows EDKII system FMP capsule verification. SysFirmUpdate.efi (https://github.com/tianocore/edk2/blob/master/SignedCapsulePkg/Universal/SystemFirmwareUpdate/SystemFirmwareUpdateDxe.inf) is insides of FMP capsule, which supports SetImage() to update firmware. SysFirmReport.efi (https://github.com/tianocore/edk2/blob/master/SignedCapsulePkg/Universal/SystemFirmwareUpdate/SystemFirmwareReportDxe.inf) is in the current firmware, which supports SetImage() to verify and dispatch the SysFirmUpdate.efi, then pass through the SetImage() request to SystemFirmUpdate.efi. The SysFirmReport.efi calls EdkiiSystemCapsuleLib to verify the **EFI_FIRMWARE_IMAGE_AUTHENTICATION** and version in system FMP capsule image, then load SysFirmUpdate.efi driver to perform the update request.

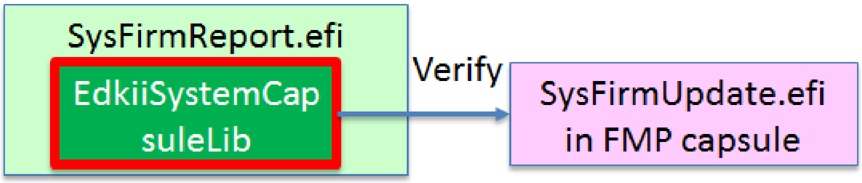

Figure 12 – EDKII System FMP capsule Payload Authentication

EDKII System FMP Capsule Update Driver

In the current firmware image, there is a system firmware report driver (SysFirmReport.efi). Its job is to get the image description and install a simple FMP to report the current firmware information via **EFI_FIRMWARE_MANAGEMENT_PROTOCOL.GetImageInfo()**.

When the FMP capsule is processed, the system firmware report driver (SysFirmReport.efi) **EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage()** function verifies and dispatches the DriverFv inside of the FMP capsule image. Then the SysFirmUpdate.efi is started. Then the SysFirmReport.efi passes the SetImage() request to the SysFirmUpdate.efi. Finally, SysFirmUpdate.efi driver gets the FMP capsule, does the verification again processes the FMP payload, and update the system firmware image on flash.

See figure 13 for the flow.

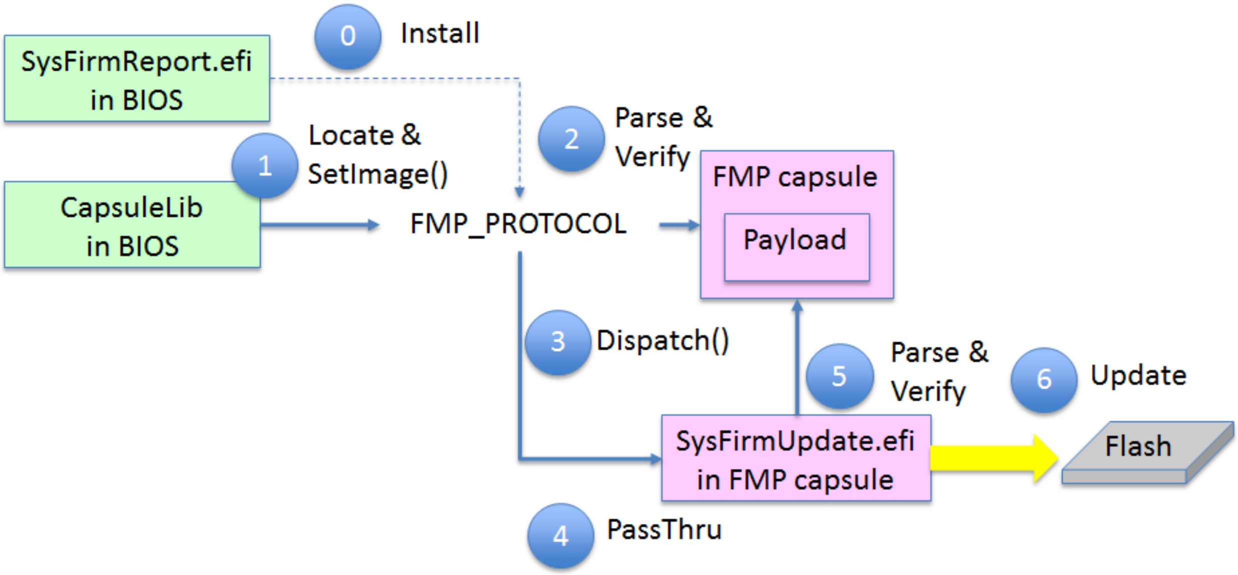

Figure 13 – EDKII System FMP Capsule Update Flow

- Step 0: When a system boots, a system firmware report driver (https://github.com/tianocore/edk2/blob/master/SignedCapsulePkg/Universal/SystemFirmwareUpdate/SystemFirmwareReportDxe.inf) installs an FMP protocol. This is a minimal feature FMP, and its **EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage()** API passes through the SetImage() request to a real function one. The purpose of this driver is to provide **EFI_FIRMWARE_MANAGEMENT_PROTOCOL.GetImageInfo()** interface, so that ESRT driver may consume this interface to build ESRT table for OS. The firmware update driver is put to EDKII system FMP capsule, because the firmware image component or layout might change later. So that it is better choice to deliver an update driver with the new image.
- Step 1: When a platform BDS detects **BOOT_ON_FLASH_UPDATE** boot mode, it calls CapsuleLib (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/DxeCapsuleLibFmp) **ProcessCapsules()** to process the capsules. The **ProcessCapsules()** need check capsule images one by one. If a capsule has **CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE** flag, this capsule is installed to UEFI configuration table. If the capsule GUID is **WINDOWS_UX_CAPSULE_GUID**, this capsule will be processed at first to display the BMP image on screen. If the capsule GUID is **EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID**, this is FMP capsule. It will be processed by

CapsuleLib **ProcessCapsuleImage()**. The later locates all **EFI_FIRMWARE_MANAGEMENT_PROTOCOL** and calls corresponding **EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage()** if **EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER.UpdateImageTypeId** matches **EFI_FIRMWARE_IMAGE_DESCRIPTOR.ImageTypeId**.

- Step 2: If the FMP capsule is an EDKII system FMP capsule, the SetImage() in SysFirmReport.efi (https://github.com/tianocore/edk2/blob/master/SignedCapsulePkg/Universal/SystemFirmwareUpdate/SystemFirmwareReportDxe.c) is invoked. The later will parse and verify the FMP capsule to check the integrity, because the capsule is treated as an external source and it is not trusted.
- Step 3: After FMP capsule passes the authentication, this FMP capsule payload is treated as trusted. The SetImage() in SysFirmReport.efi extracts the driver FV from FMP capsule payload and calls **EFI_DXE_SERVICES.ProcessFirmwareVolume()** and **EFI_DXE_SERVICES.Dispatch()** to dispatch the drivers inside of the driver FV. If there is a system firmware update driver - SysFirmUpdate.efi (https://github.com/tianocore/edk2/blob/master/SignedCapsulePkg/Universal/SystemFirmwareUpdate/SystemFirmwareUpdateDxe.inf) inside of the capsule, it is started now. This module installs **EFI_FIRMWARE_MANAGEMENT_PROTOCOL** with an internal **SYSTEM_FMP_PROTOCOL_GUID**.
- Step 4: After CapsuleLib dispatches the FV, the SetImage() in SysFirmReport.efi locates **SYSTEM_FMP_PROTOCOL_GUID** and passes through the EDKII system FMP capsule to the SetImage() in SysFirmUpdate.efi.
- Step 5: Then the SetImage() in SysFirmUpdate.efi parses and verifies payload, and extracts the real firmware binary.
- (https://github.com/tianocore/edk2/blob/master/SignedCapsulePkg/Universal/SystemFirmwareUpdate/SystemFirmwareUpdateDxe.c)
- Step 6: Finally, the real firmware binary is updated to flash area in PerformUpdate(). A platform must link a correct PlatformFlashAccessLib to perform the flash update. (Such as https://github.com/tianocore/edk2/tree/master/QuarkPlatformPkg/Feature/Capsule/Library/PlatformFlashAccessLib)

Pre-Installation Check

The EDKII system firmware FMP capsule should also support pre-installation check which is defined by Microsoft [WFU]. The integrity/security/version check requirement is similar to NIST guideline. They are already discussed in “Authentication Check” section. The power check should be platform specific. If a platform supports battery, we expect the platform BDS should add such battery check before call CapsuleLib **ProcessCapsules()**.

Variable Update Consideration

When we perform a firmware update, we might need to consider if and how we update the variable region. For device firmware, there is no such issue because the variable region is managed by system firmware. The CapsuleLib may record the firmware update state into UEFI variable region, such as LastAttemptVersion and LastAttemptStatus. This information is still available in the next boot.

However, for system firmware update, it becomes a problem, because the variable region might inside of the system firmware region. If the system firmware update also updates the variable region, the variable data should never be used after update. The EDKII System firmware update will hook the variable services and return an error directly to prevent accessing original variable region.

Then the following question arises: How can the system firmware update driver know if the variable region is updated or not? This information is recorded in the Firmware update configuration file. If the firmware type is NvRam, then the update driver knows the variable region will be updated.

Sometimes, when the system performs an update, it wants to preserve some of the current configuration setting, such as setup configuration, or the secure boot variable. Because it is platform policy on which variable should be preserved, the system firmware update driver leaves the work to the PlatformFlashAccessLib. One possible way is described below:

- When the PlatformFlashAccessLib finds it needs to update the variable region, it copies all of the UEFI variables to a local cache.
- The PlatformFlashAccessLib selects some of the variables to be preserved based upon the platform policy.
- The PlatformFlashAccessLib parses the variable region of the new firmware source buffer and patches the variable region with the preserved variable data.
- The PlatformFlashAccessLib writes the final patched variable region onto the system flash region.

Summary

This section introduces the EDKII system FMP capsule update solution in EDKII.

EDKII EFI System Resource Table

EFI System Resource Table Driver

The EFI System Resource Table (ESRT) is an UEFI configuration table produced by system firmware to report all potentially updatable firmware objects to the OS.

The EDKII ESRT driver (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/EsrtDxe) produces **ESRT_MANAGEMENT_PROTOCOL** protocol, which is an EDKII implementation protocol. If a platform module wants to add a ESRT table, it may calls **ESRT_MANAGEMENT_PROTOCOL.RegisterEsrtEntry**.

The EDKII ESRT driver consumes the **EFI_FIRMWARE_MANAGEMENT_PROTOCOL** instance and converts **EFI_FIRMWARE_IMAGE_DESCRIPTOR** to **EFI_SYSTEM_RESOURCE_ENTRY** automatically. If a firmware update driver produces **EFI_FIRMWARE_MANAGEMENT_PROTOCOL**, then there is no need to call **ESRT_MANAGEMENT_PROTOCOL**.

Figure 14 shows ESRT reporting.

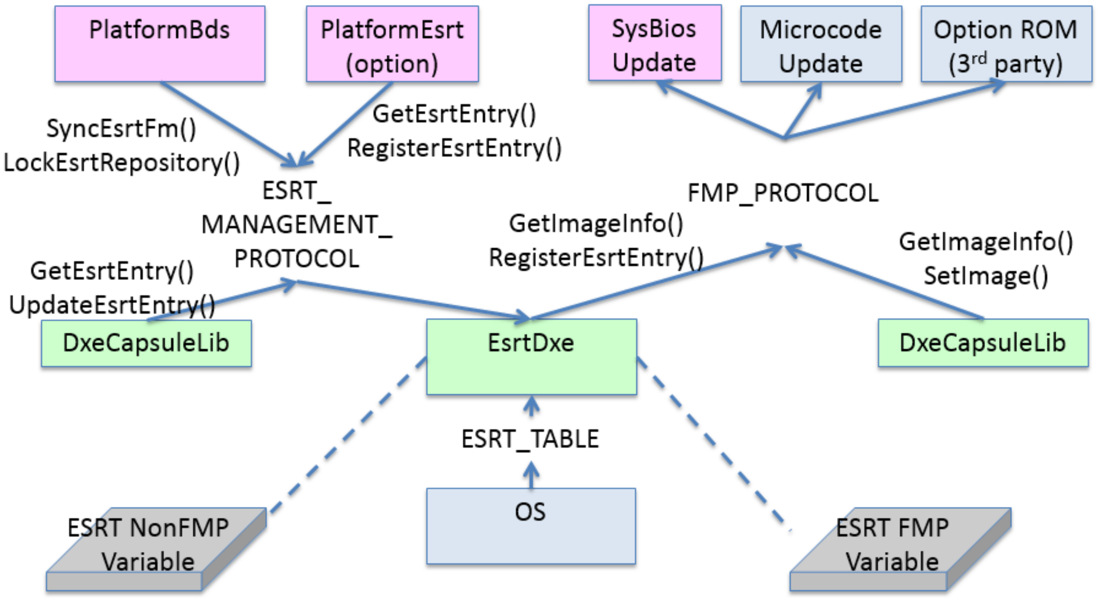

Figure 14 – ESRT reporting

On left hand side, ESRT driver produces **ESRT_MANAGEMENT_PROTOCOL**. A platform ESRT driver may call **ESRT_MANAGEMENT_PROTOCOL.GetEsrtEntry()** to get all ESRT entries, and **RegsiterEsrtEntry()** to add ESRT entry. The non FMP ESRT information is saved to **L"EsrtNonFmp"** variable.

On right hand side, the ESRT driver consumes **EFI_FIRMWARE_MANAGEMENT_PROTOCOL**. A firmware update driver may produce **EFI_FIRMWARE_MANAGEMENT_PROTOCOL**. ESRT driver calls **EFI_FIRMWARE_MANAGEMENT_PROTOCOL,GetImageInfo()** to get the firmware information and converts this data into a ESRT entry. The FMP ESRT information is saved to the **L"EsrtFmp"** variable.

The reason for caching ESRT information into variable is that the capsule update may cause a system reset as the last step of the firmware update. As such, the ESRT must record the update state in the next boot in order to report such information to the OS.

Summary

This section introduces the ESRT solution in EDKII.

Microcode Update

"Intel® 64 and IA-32 Architectures Software Developer's Manual" [IA32SDM] defines an "Optional Processor Microcode Update Specifications" in the "Microcode Update Facilities" chapter. An INT15 D042 interface is defined to read and write the Microcode from and to a system firmware non-volatile area.

The INT15 services is only available on a legacy BIOS, or a UEFI firmware with the Compatibility Support Module (CSM) to support the legacy OS boot. However, there are more and more UEFI firmware without CSM because these UEFI firmware support the UEFI OS only without a CSM. As a result, there is no way to produce the INT15 service in these UEFI OS instances. We need to define a similar UEFI style interface for the UEFI firmware.

The UEFI specification defines an "EFI_FIRMWARE_MANAGEMENT_PROTOCOL" in the "Firmware Update and Reporting" chapter. An OEM system firmware instance produces an EFI_FIRMWARE_MANAGEMENT_PROTOCOL for Microcode update. A generic UEFI application tool could consume this EFI_FIRMWARE_MANAGEMENT_PROTOCOL interface to manage Microcode update area.

Microcode Update Function	Description	Legacy Interface	UEFI Interface
Presence test	Returns information about the supported functions.	INT15 D042 – Func 00h	EFI_FIRMWARE_MANAGEMENT _PROTOCOL is located. GetImageInfo() returns ImageTypeId: MICROCODE_FMP _IMAGE_TYPE_ID_GUID
Write update data	Writes one of the update data areas (slots).	INT15 D042 – Func 01h	EFI_FIRMWARE_MANAGEMENT _PROTOCOL.SetImage()
Update control	Globally controls the loading of updates.	INT15 D042 – Func 02h	N/A
Read update data	Reads one of the update data areas (slots).	INT15 D042 – Func 03h	EFI_FIRMWARE_MANAGEMENT _PROTOCOL.GetImage()

EFI_FIRMWARE_MANAGEMENT_PROTOCOL

The Microcode update module in UEFI firmware produces an **EFI_FIRMWARE_MANAGEMENT_PROTOCOL** and returns the **EFI_FIRMWARE_IMAGE_DESCRIPTOR** with the below **MICROCODE_FMP_IMAGE_TYPE_ID_GUID** as **ImageTypeId** in **GetImageInfo()**. If the system has more than one Microcode, each Microcode maps to one **EFI_FIRMWARE_IMAGE_DESCRIPTOR**.

The meaning of each field in the **EFI_FIRMWARE_IMAGE_DESCRIPTOR** is described below:

EFI_FIRMWARE_ IMAGE_DESCRIPTOR	Microcode Update Field	Comment

ImageIndex	N/A	Index of each Microcode
ImageTypeId	N/A	Must be **MICROCODE_FMP_IMAGE_TYPE_ID_GUID**
ImageId	**ProcessorFlags** << 32 | **ProcessorSignature**	Used to identify a Microcode
ImageIdName	N/A	
Version	**UpdateRevision**	Microcode version
VersionName	N/A	

Size	If (**DataSize** == 0) {
	2048
	} else {
	TotalSize
	}
		The total size of the Microcode binary, including header, data and optional header.

AttributesSupported	N/A	
AttributesSetting	N/A	
Compatibilities	N/A	
LowestSupported ImageVersion	**UpdateRevision**	Do not support rollback
LastAttemptVersion	N/A	
LastAttemptStatus	N/A	
HardwareInstance	N/A	0

#define **MICROCODE_FMP_IMAGE_TYPE_ID_GUID { 0x96d4fdcd, 0x1502, 0x424d, { 0x9d, 0x4c, 0x9b, 0x12, 0xd2, 0xdc, 0xae, 0x5c } }**

The EFI_FIRMWARE_MANAGEMENT_PROTOCOL consumer (an UEFI application or driver) may call GetImageInfo() to get the basic information, call GetImage() to read the current Microcode from NV storage, or call SetImage() to replace a new Microcode to NV storage with new version.

The EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage() must perform the below check when it appends or replaces a new Microcode to NV storage: (Similar to INT15 D042 Function 01H—Write Microcode Update Data)

1) The update **header version** should be equal to an update header version recognized by the firmware. The violation causes:
   a) Status: EFI_INCOMPATIBLE_VERSION
   b) AbortReason: L“InvalidHeaderVersion”
   c) LastAttemptVersion: The update version.
   d) LastAttemptStatus: LAST_ATTEMPT_STATUS_ERROR_INVALID_FORMAT
2) The update **loader version** in the update header should be equal to the update loader version contained within the firmware image. The violation causes:
   a) Status: EFI_INCOMPATIBLE_VERSION
   b) AbortReason: L“InvalidLoaderVersion”
   c) LastAttemptVersion: The update version.
   d) LastAttemptStatus: LAST_ATTEMPT_STATUS_ERROR_INVALID_FORMAT
3) The update block must **checksum**. This checksum is computed as a 32-bit summation of all double words in the structure, including the header, data, and processor signature table. The violation causes:

   a) Status: EFI_VOLUME_CORRUPTED
   b) AbortReason: L“InvalidChecksum”
   c) LastAttemptVersion: The update version.
   d) LastAttemptStatus: LAST_ATTEMPT_STATUS_ERROR_INVALID_FORMAT
4) The **processor signature** in the proposed update must be equal to the current processor signature. The violation causes:
   a) Status: EFI_UNSUPPORTED
   b) AbortReason: L“UnsupportedProcessSignature”.
   c) LastAttemptVersion: The update version.
   d) LastAttemptStatus: LAST_ATTEMPT_STATUS_ERROR_INCORRECT_VERSION
5) The **processor flags** in the proposed update must match the current platform ID. The violation causes:
   a) Status: EFI_UNSUPPORTED
   b) AbortReason: L”UnsupportedProcessorFlags” for the flags mismatch.
   c) LastAttemptVersion: The update version.
   d) LastAttemptStatus: LAST_ATTEMPT_STATUS_ERROR_INCORRECT_VERSION
6) The **update revision** in the proposed update should be greater than the update revision in the header of the current update in NVRAM. The violation causes:
   a) Status: EFI_INCOMPATIBLE_VERSION
   b) AbortReason: L“IncorrectRevision”
   c) LastAttemptVersion: The update version.
   d) LastAttemptStatus: LAST_ATTEMPT_STATUS_ERROR_INCORRECT_VERSION
7) The firmware must verify the authenticity of the update via the mechanism described in IA32 SDM “Microcode Update Loader”. The violation causes:
   a) Status: EFI_SECURITY_VIOLATION
   b) AbortReason: L“InvalidData”
   c) LastAttemptVersion: The update version.
   d) LastAttemptStatus: LAST_ATTEMPT_STATUS_ERROR_AUTH_ERROR

EFI_FIRMWARE_MANAGEMENT_CAPSULE

The EFI_FIRMWARE_MANAGEMENT_PROTOCOL is a UEFI boot time protocol, so it cannot be called at OS runtime. In order to support initiating Microcode update from the OS, a Microcode FMP capsule can be used.

A Microcode FMP capsule uses **EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID** to indicate it is an FMP capsule. One Microcode FMP capsule may contain one or more Microcode FMP payloads.

Each Microcode FMP payload uses **MICROCODE_FMP_IMAGE_TYPE_ID_GUID** as **UpdateImageTypeId** in **EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER** to indicate this FMP payload is for Microcode. The payload data area is the Microcode binary data, including header, data and optional header.

An OEM firmware capsule process driver needs to parse the Microcode FMP capsule as a normal FMP capsule, and uses **UpdateImageTypeId** to locate the correct Microcode **EFI_FIRMWARE_MANAGEMENT_PROTOCOL**. Then the capsule process driver calls **EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage()** for the new Microcode in the FMP payload.

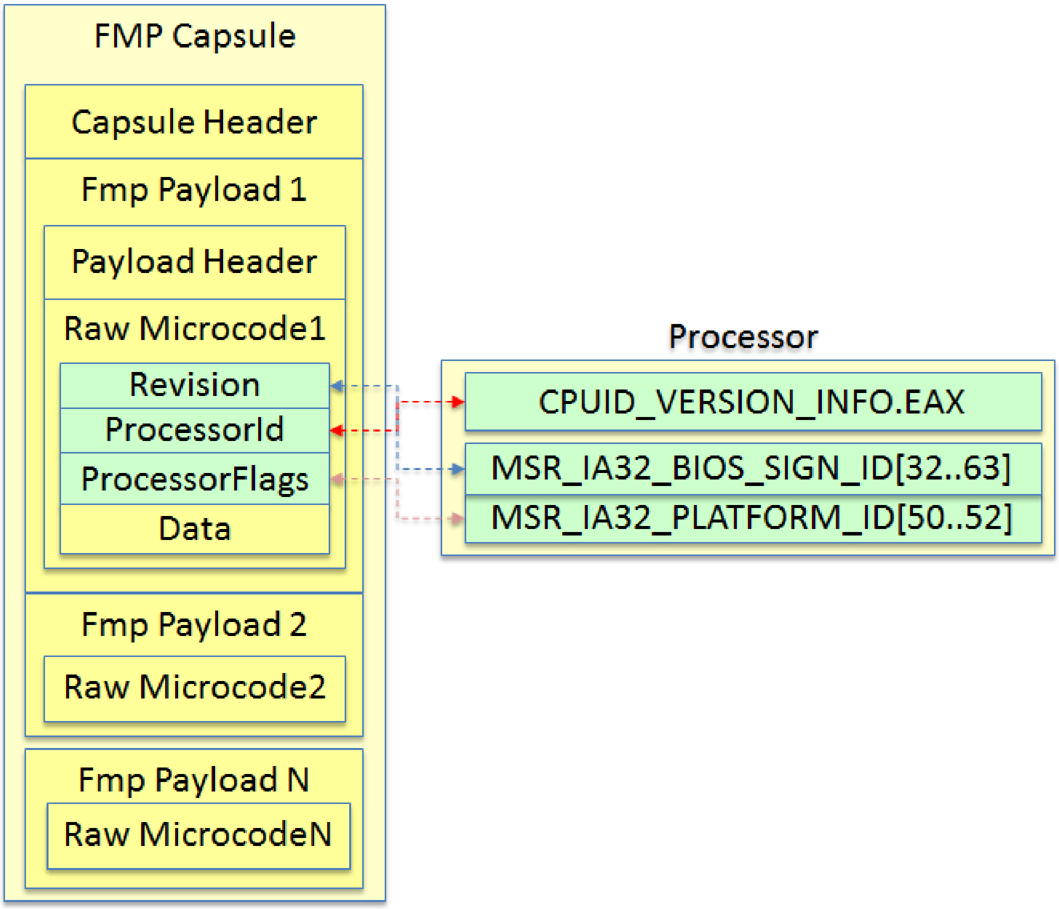

Figure 15 – Microcode FMP capsule

EFI System Resource Table

During system boot, the OEM firmware also needs to produce a **EFI System Resource Table** for Microcode. This work can be done by a generic ESRT driver which consumes **EFI_FIRMWARE_MANAGEMENT_PROTOCOL** to fill in the information.

The meaning of each field in the **EFI_SYSTEM_RESOURCE_ENTRY** is below:

EFI_SYSTEM_ RESOURCE_ENTRY	**Microcode Update Field**	Comment
FwClass	N/A	Must be **MICROCODE_FMP_IMAGE_TYPE_ID_GUID**
FwType		**ESRT_FW_TYPE_DEVICEFIRMWARE**
FwVersion	**UpdateRevision**	Microcode version
LowestSupported ImageVersion	**UpdateRevision**	Do not support rollback
CapsuleFlags	N/A	

LastAttemptVersion	N/A	
LastAttemptStatus	N/A	

NOTE: There might be multiple Microcode on a system firmware non-volatile storage area.

The ***EFI_FIRMWARE_MANAGEMENT_PROTOCOL*** *reports all of them, but at most one* ***EFI_FIRMWARE_IMAGE_DESCRIPTOR*** *sets* ***IMAGE_ATTRIBUTE_IN_USE*** *flag in* **AttributesSetting** *for one processor. For multi-socket system, there might be more than one* ***IMAGE_ATTRIBUTE_IN_USE*** *set for different processors.*

The ***EFI_SYSTEM_RESOURCE_ENTRY*** *only reports the microcode in use. Only* ***EFI_FIRMWARE_IMAGE_DESCRIPTOR*** *with* ***IMAGE_ATTRIBUTE_IN_USE*** *flag is reported in final* ***EFI_SYSTEM_RESOURCE_ENTRY****.*

UEFI Microcode Update Driver

The EDKII UEFI Microcode update driver is at https://github.com/tianocore/edk2/tree/master/UefiCpuPkg/Feature/Capsule/MicrocodeUpdateDxe. It links a platform specific MicrocodeFlashAccessLib, such as https://github.com/tianocore/edk2/tree/master/Vlv2TbltDevicePkg/Feature/Capsule/Library/PlatformFlashAccessLib.

Summary

This section discussed the Microcode Update in a UEFI firmware.

Recovery

Firmware recovery is a feature to allow a system to continue booting if the part of flash region is corrupt or errant. The flash region corruption should not happen, but in case of such an occurrence, the end user may use a recovery image on a CDROM or a USB key to boot the system and perform a flash update.

Recovery General Flow

The detailed general recovery flow is below. See figure 16.

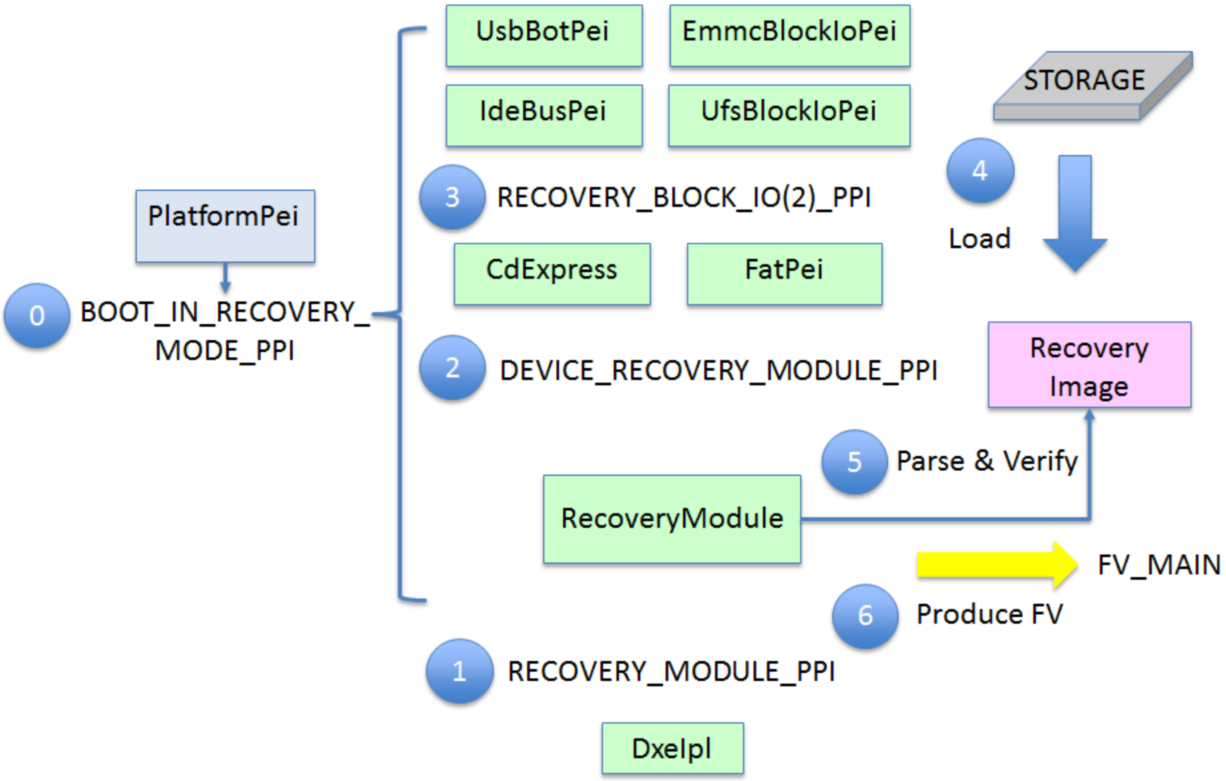

Figure 16 – Recovery General Flow

- Step 0: During system boot, a platform PEI module (such as https://github.com/tianocore/edk2/tree/master/QuarkPlatformPkg/Platform/Pei/PlatformInit/BootMode.c) detects the boot mode. If the condition is true, this PEIM module calls EFI_PEI_SERVICES.SetBootMode(BOOT_IN_RECOVERY_MODE). It also installs EFI_PEI_BOOT_IN_RECOVERY_MODE_PPI, so that modules with this dependency are dispatched in recovery mode.
- Step 1: As the final step of the PEI phase, the DxeIpl (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/DxeIplPeim) calls EFI_PEI_RECOVERY_MODULE_PPI.LoadRecoveryCapsule if the system boot mode is BOOT_IN_RECOVERY_MODE.
- Step 2: A RecoveryModule is the producer of EFI_PEI_RECOVERY_MODULE_PPI. It consumes

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.

- Step 3: The PEI file system driver is the producer of EFI_PEI_DEVICE_RECOVERY_MODULE_PPI. In EDKII, these modules include CdExpressPei (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Disk/CdExpressPei) and FatPei (https://github.com/tianocore/edk2/tree/master/FatPkg/FatPei). They consume EFI_PEI_RECOVERY_BLOCK_IO_PPI or EFI_PEI_RECOVERY_BLOCK_IO2_PPI.
- Step 4: The PEI block IO driver is the producer of EFI_PEI_RECOVERY_BLOCK_IO_PPI or EFI_PEI_RECOVERY_BLOCK_IO2_PPI. In EDKII, these modules are UsbBotPei (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Usb/UsbBotPei), IdeBusPei (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci/IdeBusPei), EmmcBlockIoPei (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Sd/EmmcBlockIoPei), and UfsBlockIoPei (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Ufs/UfsBlockIoPei). These PEIM's are the modules to load the capsule from a storage device into memory.
- Step 5: Once the RecoveryModule retrieves the recovery image, it will parse and verify the recovery image to check the integrity and extract the Firmware Volume for the DXE phase.
- Step 6: Finally, the RecoveryModule installs the extracted Firmware Volume for DXE. It builds **EFI_HOB_FIRMWARE_VOLUME** and installs **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI** or **EFI_PEI_FIRMWARE_VOLUME_INFO2_PPI**.

Authentication Check

The authentication check in the recovery is similar to the one in the firmware update process. The recovery image should be signed and version should be checked.

- **Signing check**: The signing check for the recovery is similar to the one for the capsule update. The UEFI FMP capsule uses the **EFI_FIRMWARE_IMAGE_AUTHENTICATION** before the real payload image. However the certificate type can be different one such as **EFI_CERT_TYPE_RSA2048_SHA256_GUID**. The reason is that the boot block may only have limited space to hold the singing verification algorithm. The PKCS1 RSA verification need smaller runtime memory than the PKCS7 verification.
  In EDKII, the RSA2048SHA256 signature check is at https://github.com/tianocore/edk2/tree/master/SecurityPkg/Library/FmpAuthenticationLibRsa2048Sha256.
- **Version check**: The version check for recovery is different with the one for the capsule update. In recovery boot mode, the version in the capsule image must be the same as the version in current firmware. The reason is that the DXE FV might have a special dependency on the PEI FV.
  1) PCD database: The DXE PCD database is inherited from the PEI PCD database. If the DXE FV and PEI FV are different, there might be mismatches in the PCD database. In this case the behavior is undefined.
     The internal PCD database information can be found at https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/PCD/Pei/Pcd.inf.
     There exists a tool to dump the PCD information and PCD internal database at https://github.com/jyao1/EdkiiShellTool/tree/master/EdkiiShellToolPkg/PcdDump.
  2) Hob: DXE FV may depend on some HOB information produced by the PEI FV.
  3) System State: the DXE FV may depend upon some system state initialization in the PEI phase.

Because of the above known potential issues, it is strongly recommend to use the same BIOS image for the recovery flow.
In EDKII, one image version check sample for capsule update is done in the EdkiiSystemCapsuleLib at https://github.com/tianocore/edk2/tree/master/SignedCapsulePkg/Library/EdkiiSystemCapsuleLib.
If other signed recovery solution is used, it need provide similar function to check the version.

Summary

This section introduces the recovery design in UEFI/PI firmware.

EDKII System FMP Based Recovery

EDKII Recovery Image Format

If the recovery image is from an external source, it should also be signed as part of the capsule update image. The only difference is that the recovery image can use any other algorithm, such as RSA2048_SHA256.

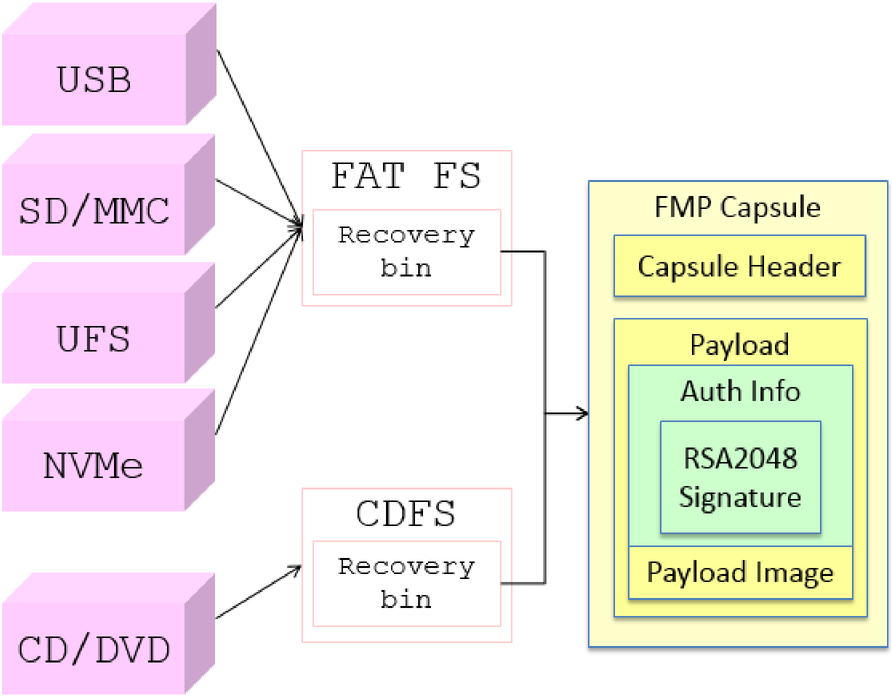

Figure 17 – EDKII Signed Recovery Image Format

Authentication Check

The authentication check in the recovery is similar to the one in the firmware update. We discussed the difference in previous chapter. RecoveryModuleLoadPeim links EdkiiSystemCapsuleLib to perform the check.

FMP Capsule Recovery Driver

The FMP capsule recovery flow is below. See figure 18.

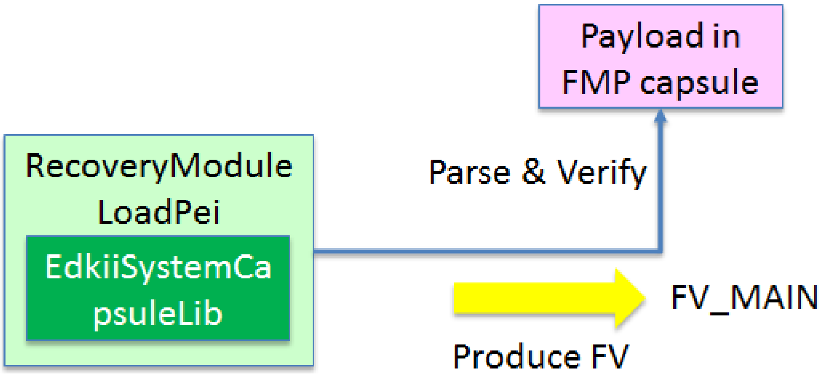

Figure 18 – Recovery Flow

- Once RecoveryModuleLoadPeim (https://github.com/tianocore/edk2/tree/master/SignedCapsulePkg/Universal/RecoveryModuleLoadPei) retrieves the FMP capsule. It will invoke EdkiiSystemCapsuleLib (https://github.com/tianocore/edk2/tree/master/SignedCapsulePkg/Library/EdkiiSystemCapsuleLib) to parse and verify the FMP capsule to check the integrity, because the capsule is treated as an external source and it is not trusted.
- If the verification succeeds, the real recovery binary is extracted. The RecoveryModuleLoadPeim needs to parse the same capsule configuration file (Such as https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Feature/Capsule/SystemFirmwareUpdateConfig/SystemFirmwareUpdateConfig.ini) to know which region is the DXE FV for recovery. The format of the recovery configuration is defined at https://github.com/tianocore/edk2/blob/master/SignedCapsulePkg/Include/Guid/EdkiiSystemFmpCapsule.h.
- Finally, the DXE FV for recovery is found. The RecoveryModuleLoadPeim builds **EFI_HOB_FIRMWARE_VOLUME** and installs **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI** or **EFI_PEI_FIRMWARE_VOLUME_INFO2_PPI**. If there are multiple FVs needed for DXE phase, the user must define all those regions in the configuration file one by one.

Summary

This section introduces the EDKII system FMP based recovery solution in EDKII.

Acknowledgments

We would like to thank *Michael D Kinney, Chao Zhang, Liming Gao, Star Zeng, David Wei, Karunakara Kotary, Nandagopal Sathyanarayanan, Sugumar Govindarajan, Rock Cao, Qing Huang and Sean Brogan* who reviewed the EDKII capsule update solution and gave valuable technical feedback when we designed the EDKII signed capsule and signed recovery solution.

References

[NIST-1] *800-147: BIOS Protection Guidelines*, http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-147.pdf

[NIST-2] *800-147B: BIOS Protection Guidelines for Servers*, http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-147B.pdf

[WFU] *Windows UEFI Firmware Update Platform*, https://www.microsoft.com/en-us/download/details.aspx?id=38405

[UEFI] *Unified Extensible Firmware Interface (UEFI) Specification*, Version 2.6 www.uefi.org

[PI] *UEFI Platform Initialization Specification*, Version 1.5 http://www.uefi.org/sites/default/files/resources/PI%201.5.zip

[IA32SDM] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, www.intel.com

[EDKII Security Design] Jiewen Yao, Vincent Zimmer, *A Tour Beyond BIOS - Security Design Guide in EDK II*, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Securiy_Enhancement_to_Mitigate_Buffer_Overflow_in_UEFI.pdf

[Win10 HCS] *Hardware Compatibility Specification for Systems for Windows 10*, https://msdn.microsoft.com/windows/hardware/commercialize/design/compatibility/systems

Authors

Jiewen Yao (jiewen.yao@intel.com) is EDKII BIOS architect, EDKII FSP package maintainer, EDKII TPM2 module maintainer, EDKII ACPI S3 module maintainer, with Software and Services Group at Intel Corporation. Jiewen is member of UEFI Security Sub-team and PI Security Sub-team in the UEFI Forum.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer with the Software and Services Group at Intel Corporation. Vincent chairs the UEFI Security and Network Sub-teams in the UEFI Forum.

This paper is for informational purposes only. THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.