White Paper

A Tour Beyond BIOS Implementing S3 Resume with EDKII

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

Star Zeng
Intel Corporation

September 2015

Executive Summary

This paper presents the internal structure and boot flow of PI S3 resume design, as implemented in the EDKII.

Prerequisite
This paper assumes that audience has EDKII/UEFI firmware development experience. He or she should also be familiar with UEFI/PI/ACPI firmware infrastructure, such as SEC, PEI, DXE, runtime phase, and S-states.

Table of Contents

Overview

Introduction to the S3 resume

S3 resume is a power saving feature defined in [ACPI] Advanced Configuration and Power Interface specification. ACPI defines a set power state of transition. S3 here means G1 sleeping – S3 sleeping state.

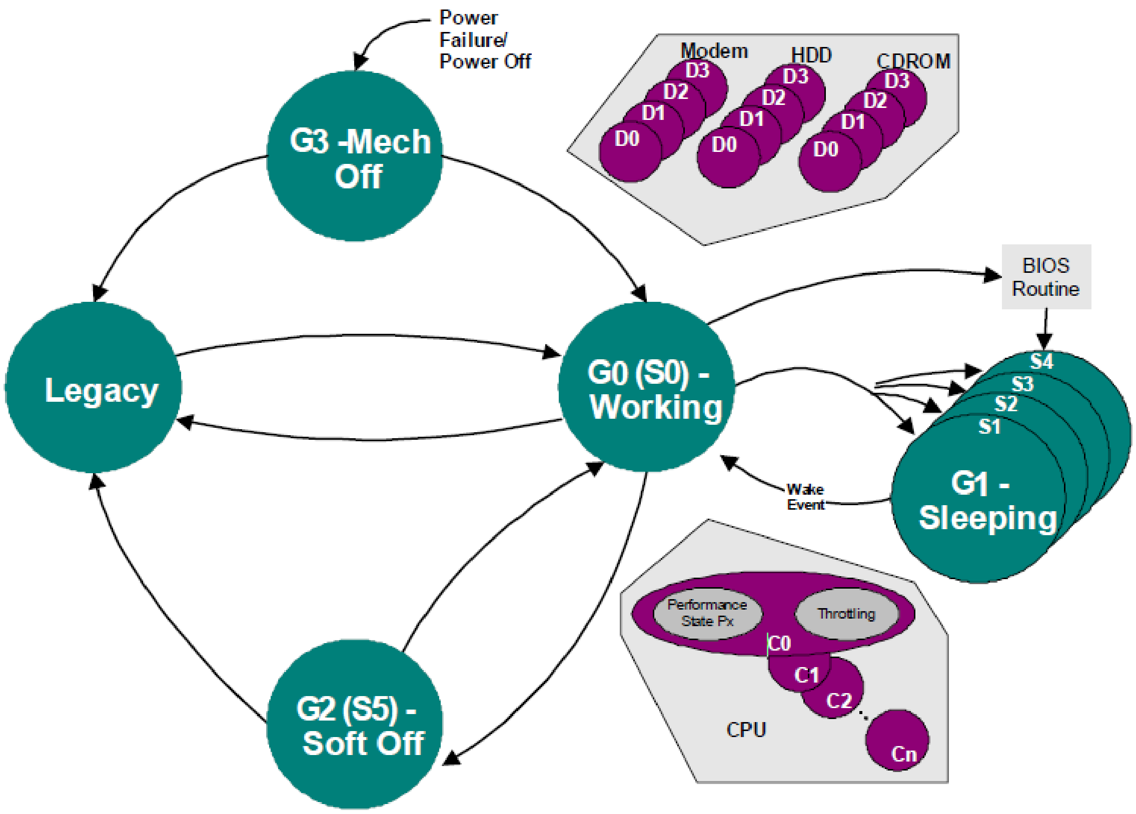

Figure 1 Global System Power States and Transitions

Totally, there are 4 Sx states defined in the G1 sleeping group. We only discuss S3 here because:

1) S1 resume is handled in SMM directly.
2) S2 resume is not used in most IA32 platforms.
3) S4 resume is similar to a normal boot.
   NOTE: There is one option named S4 BIOS, which means BIOS uses firmware to save a copy of memory to disk and then initiates the hardware S4. When the system wakes, the firmware restores memory from disk and wakes OSPM by transferring control to the FACS waking vector. But it is not used in most platforms since this is the OS-independent art from early APM.
4) S5 resume is similar to a normal boot.

Only S3 resume is a special boot path and has significant difference as normal boot path. So we will discuss this special boot mode in the next chapters.

Like other restart mechanisms, there are PI boot modes defined for these paths, which can be found in the 'boot paths' section of the volume 1 of the UEFI PI specification.

Threat Model

S3 means "suspend to memory". The OS context is in memory, and BIOS context is also in memory. If some malicious code can modify memory for BIOS S3 resume, it can attack the BIOS directly. In order to mitigate this threat, BIOS need design a way to protect the BIOS context, so that in S3 resume path, the integrity is maintained.

This white paper only covers the design of protecting the BIOS context. This white paper does not cover how to protect the OS context, like the FACS waking vector.

We will discuss security considerations and design in Part III. We will also discuss the current limitations and solution of a secure boot script library and LockBox.

EDKII module involved in S3

Below core modules are involved in S3 saving:

1) S3SaveStateDxe – Save Boot Script in DXE phase
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/S3SaveStateDxe
2) SmmS3SaveState – Save Boot Script in SMM
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/SmmS3SaveState
3) S3BootScriptLib – Boot Script library
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/PiDxeS3BootScriptLib
4) AcpiS3SaveDxe – prepare S3 context
https://github.com/tianocore/edk2/tree/master/IntelFrameworkModulePkg/Universal/Acpi/AcpiS3SaveDxe

Below core modules are involved in S3 resume:

1) S3Resume2Pei – Restore system configuration
https://github.com/tianocore/edk2/tree/master/UefiCpuPkg/Universal/Acpi/S3Resume2Pei
2) BootScriptExecutor – Execute boot script
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/BootScriptExecutorDxe

Below core modules provide security services:

1) SmmLockBoxLib – SMM based LockBox library.
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/SmmLockBoxLib
2) SmmLockBox driver – provide service for SMM based LockBox library DXE instance in normal boot path, or PEI instance after SMI enabled in S3 path.
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/LockBox/SmmLockBox

Below core modules provide compatibility support for EDKI/Framework architecture.

1) BootScriptSaveOnS3SaveStateThunk – Provide BootScriptSave protocol on top of S3SaveState protocol.
https://github.com/tianocore/edk2/tree/master/EdkCompatibilityPkg/Compatibility/BootScriptSaveOnS3SaveStateThunk
2) BootScriptThunkHelper – Save BootScriptThunk module to LockBox.
https://github.com/tianocore/edk2/tree/master/EdkCompatibilityPkg/Compatibility/SmmBaseHelper
3) SmmScriptLib – Provide EDKI/framework SmmScriptLib based on EDKII BootScriptLib.
https://github.com/tianocore/edk2/tree/master/EdkCompatibilityPkg/Foundation/Library/Smm/SmmScriptLib

Summary

This section provided an introduction of S3 resume and its threat model. In next sections, we will introduce the 4 parts of S3 in EDKII. Part I focuses on S3 context preparation, part II focuses on S3 resume boot path, part III focuses on security consideration in S3, and the final part focuses on compatibility support.

Part I – Preparing for S3

PI Architecture for Boot Script

The goal of the S3 resume process is to restore the platform to its pre-boot configuration. The PI Architecture still needs to restore the platform in a phased fashion as it does in a normal boot path. The figure below shows the phases in an S3 resume boot path.

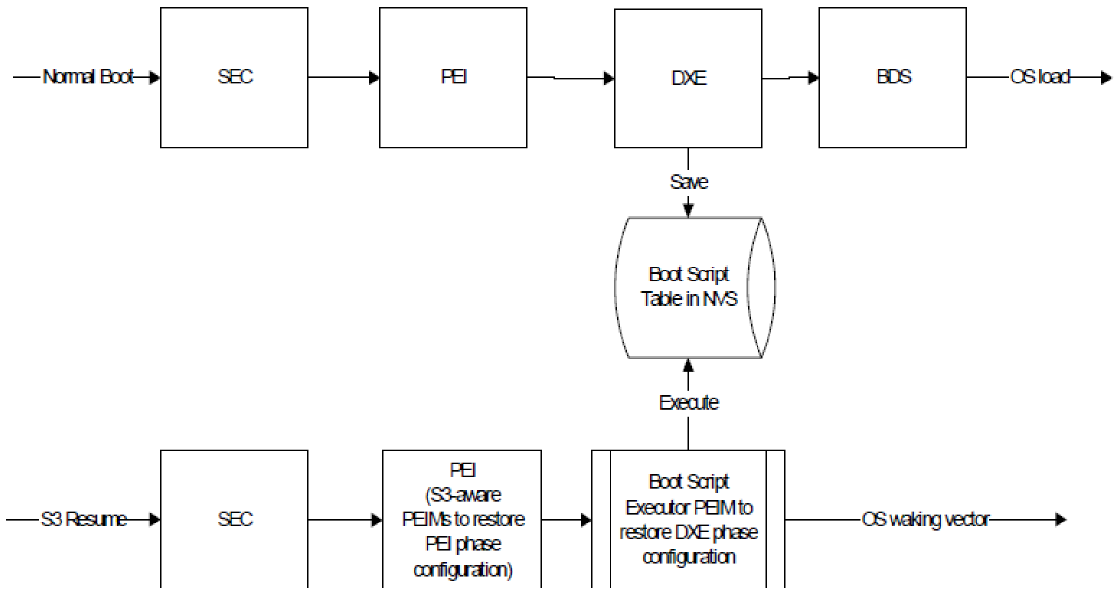

Figure 2 PI Architecture S3 Resume Boot Path

In normal boot, the PEI phase is responsible for initializing enough of the platform's resources to enable the execution of the DXE phase, which is where the majority of platform configuration is performed by different DXE drivers.

In S3 resume phase, bringing DXE in and making a DXE driver boot-path aware is very risky for the following reasons:

- The DXE phase hosts numerous services, which makes it rather large.
- Loading DXE from flash is very time consuming.

Instead, the PI Architecture provides a boot script that lets the S3 resume boot path avoid the DXE phase altogether, which helps to maximize optimum performance. During a normal boot, DXE drivers record the platform's configuration in the boot script, which is saved in NVS. During the S3 resume boot path, a boot script engine executes the script, thereby restoring the configuration.

The ACPI specification only requires the BIOS to restore chipset and processor configuration. The chipset configuration can be viewed as a series of memory, I/O, and PCI configuration operations, which DXE drivers record in the PI Architecture boot script. During an S3 resume, a boot script engine executes the boot script to restore the chipset settings.

Save State Protocol

Save State protocol defines how a DXE PI module can record IO operations (aka boot script) to be performed as part of the S3 resume. It has a DXE version and an SMM version.

In order to maintain the integrity, the DXE version of boot script save service is closed at SmmReadyToLock. The SMM version boot script save service exists and works at Smm S3 dispatch handler.

For details of the Save State Protocol API, please read PI specification vol 5, chapter 8 S3 resume, 8.7 S3 Save State Protocol.

The implementation of the DXE version Sava State protocol is at
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/S3SaveStateDxe

The SMM version is at
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/SmmS3SaveState

Both protocol implementations depend on the S3BotScriptLib at
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/PiDxeS3BootScriptLib
The S3BootScriptLib can be used by an DXE driver or a SMM driver.

The silicon code or platform code may use the Save State protocol, or Smm Save State protocol, or S3BootScriptLib to record boot script. No matter which interface is used, the boot script data will be saved into one common table finally.

General Flow of Boot Script Save and Execution

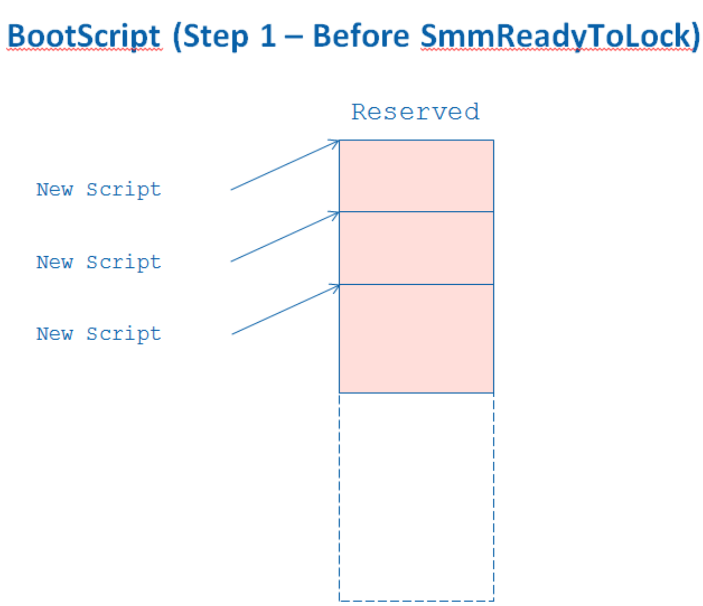

Figure 3 BootScript Flow Step 1 – Before SmmReadyToLock

Step 1: In general boot script entries are added by silicon driver and platform driver. Both DXE version boot script and SMM version boot script are allowed to write to one boot script memory. This boot script memory is in ACPI reserved memory, so that OS reserve this memory region. When BootScriptLib allocates boot script memory, it allocates additional `PcdS3BootScriptRuntimeTableReservePageNumber` pages for boot script at SMM runtime. So that SMM version boot script can add runtime boot script there.

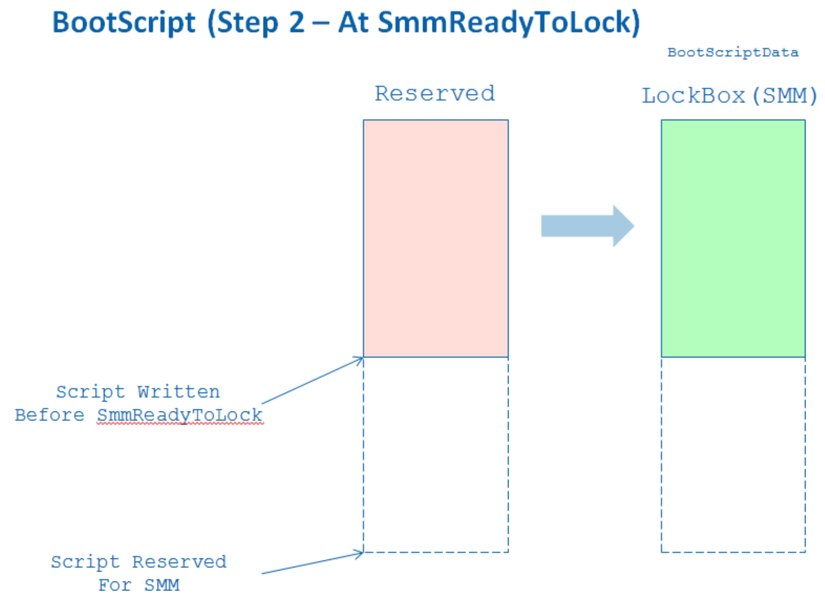

Figure 4 BootScript Flow Step 2 – At SmmReadyToLock

Step 2: When system gets SmmReadyToLock event, BootScript library need copy current boot script to **BootScriptData lockbox**. The reason is that ACPI reserved memory is considered as unsecure place, but system configuration is vital to S3 resume. We need save it to a secure place – LockBox. (The detail of lockbox will be discussed in Part III).

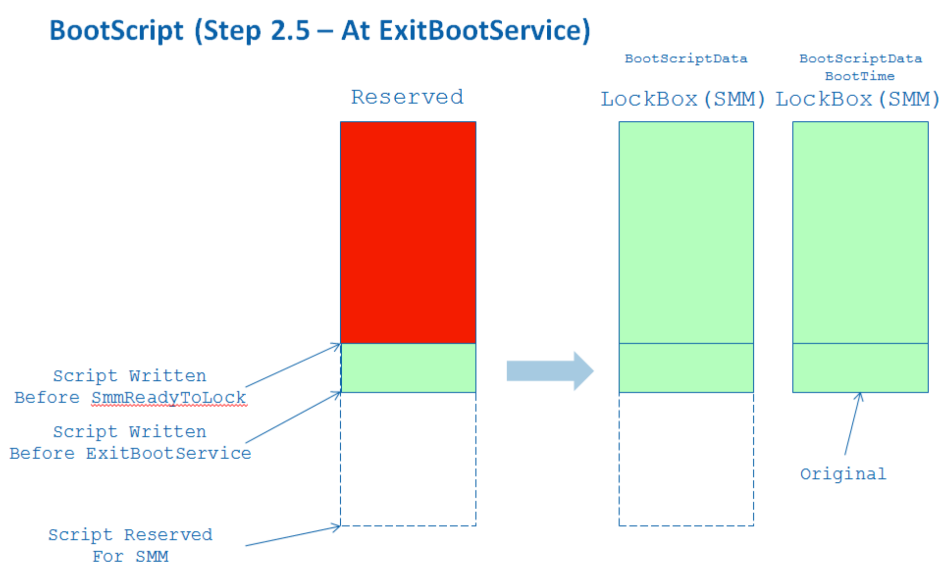

Figure 5 BootScript Flow Step 2.5 – At SmmExitBootServices

Step 2.5: When system gets SmmExitBootServices event or SmmLegacyBoot event, BootScript library need duplicate current boot script to another **BootScriptDataBootTime lockbox**. The reason is that boot script entry (DXE version or SMM version) in UEFI boot time is only added once, but boot script entry (SMM version) in UEFI runtime might be added multiple times. When system is back from S3, we need restore the UEFI boot time boot script table to remove all UEFI runtime boot script entry, so that the UEFI runtime boot script entry in this boot can be append at right place.

At this point of time the data in **BootScriptDataBootTime lockbox** is same with **BootScriptData lockbox**. All the boot script update between SmmReadyToLock and SmmExitBootServices is in both lockbox. After that, **BootScriptData lockbox** can still be updated for runtime boot script, while **BootScriptDataBootTime lockbox** is never changed.

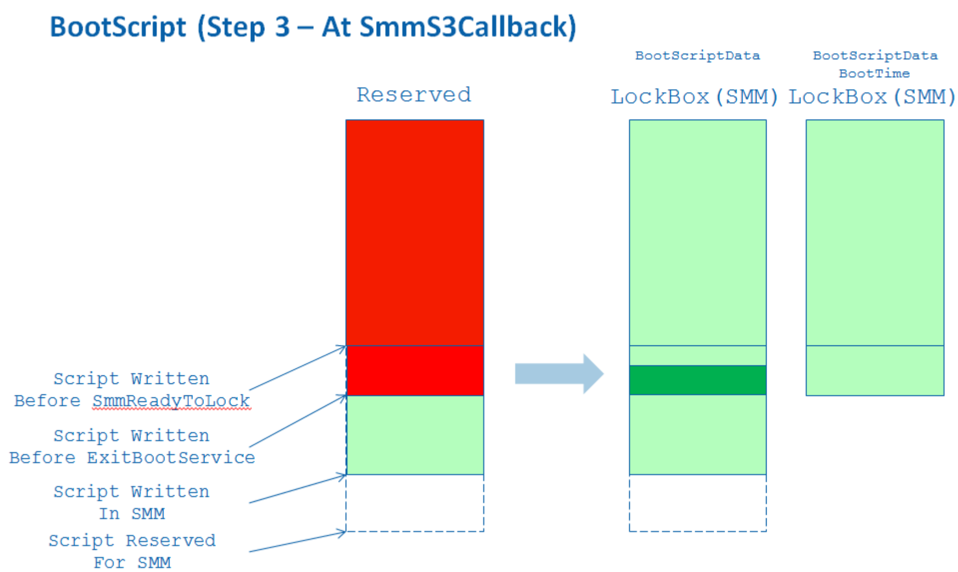

Figure 6 BootScript Flow Step 3 – At SmmS3Callback

Step 3: After system boots to OS and OS does S3 suspend action, SMM handler can get SmmS3Callback event. A platform or silicon handler may save the current system configuration to boot script as UEFI runtime boot script.

At this point of time, all UEFI runtime boot script is updated into **BootScriptData lockbox**. **BootScriptDataBootTime lockbox** is never changed.

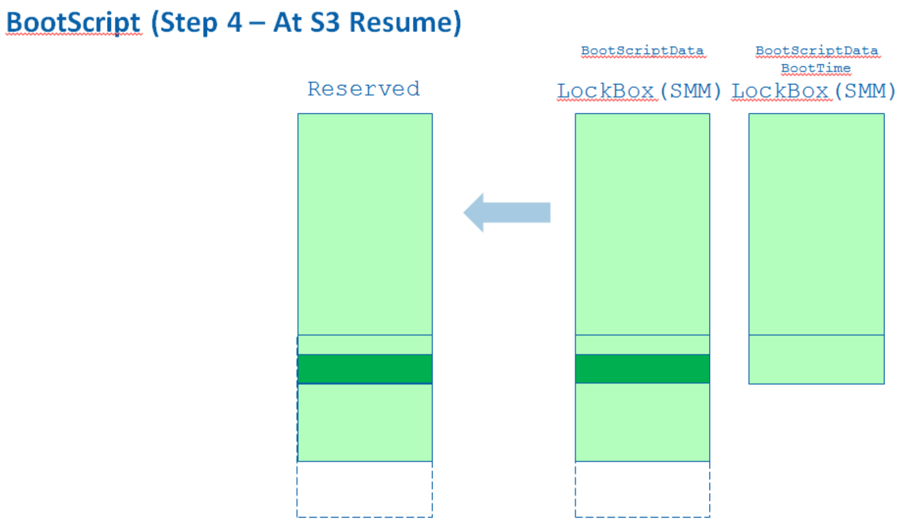

Figure 7 BootScript Flow Step 4 – At S3 Resume

Step 4: During S3 resume, the BIOS restores the **BootScriptData lockbox** data to ACPI reserved memory, to make sure the integrity of system configuration. The BIOS also restores **BootScriptTableBaseGuid lockbox** to know the ACPI reserved memory address of boot script. Finally, BootScriptExecutor executes boot script data to restore system configuration.

At same time, **BootScriptSmmPrivateDataGuid lockbox** is restored too, so that BootScript library can know that system is back from S3.

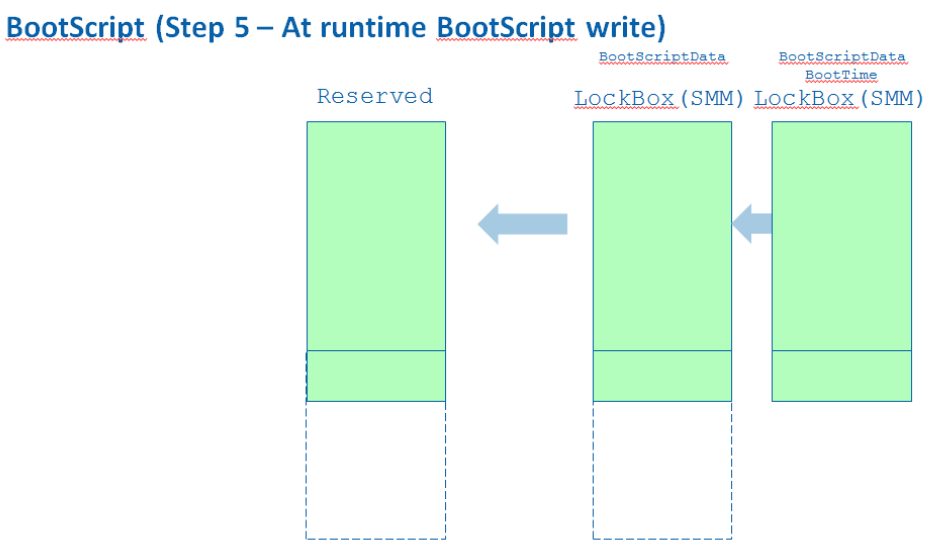

Figure 8 BootScript Flow Step 5 – At runtime BootScript write

Step 5: In first UEFI runtime boot script write action, BootScript library detects system back from S3. If so, BootScript library restores **BootScriptDataBootTime lockbox** to **BootScriptData lockbox** and boot script in ACPI reserved memory. Then the boot script memory becomes same as normal boot, and runtime boot script will be just appended.

In summary, there are multiple Lockboxes are created:

– **BootScriptDataBootTimeGuid** LockBox: It is used to restore data after back from S3 to handle potential INSERT boot script at runtime.
– **BootScriptDataGuid** LockBox (IN_PLACE): It is used to restore data at S3 resume.
– **BootScriptTableBaseGuid** LockBox (IN_PLACE): It is used to save TableBase address.
– **BootScriptSmmPrivateDataGuid** LockBox (IN_PLACE): SMM private data with "BackFromS3 = TRUE" at runtime. S3 will help restore it to tell the Library the system is back from S3.

Boot Script Implementation

PI specification does not define the boot script format. It is implementation specific. In EDKII, the BootScript record entry format is defined at https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Library/PiDxeS3BootScriptLib/BootScriptInternalFormat.h

The boot script library internal data structure is below:

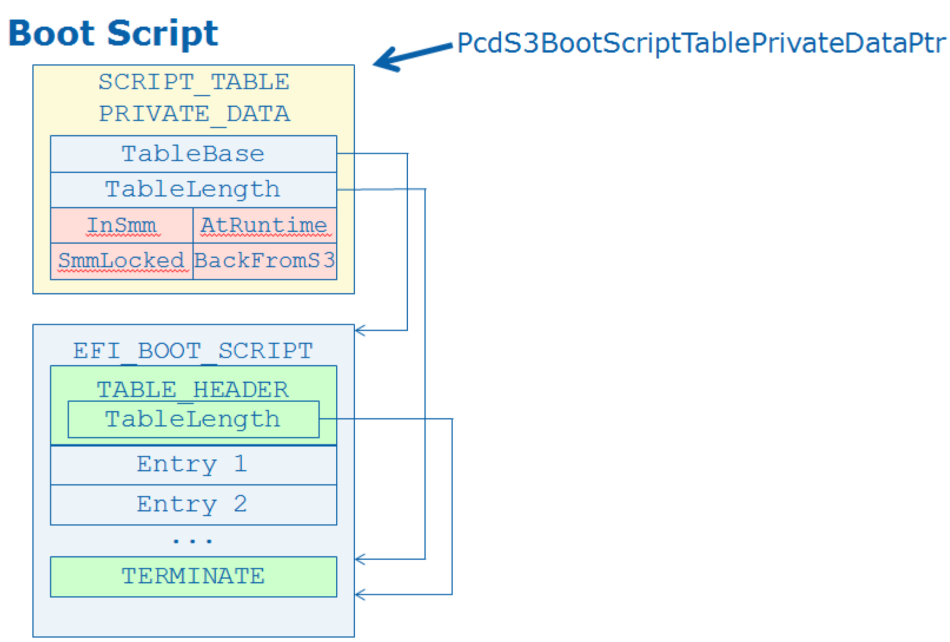

Figure 9 Boot Script: internal data structure

Every library instance will have a pointer to SCRIPT_TABLE_PRIVATE_DATA. This data structure is initialized by first library instance, and pointer is assigned as a dynamic PCD: PcdS3BootScriptTablePrivateDataPtr, to make sure there is only one copy in global. The SCRIPT_TABLE_PRIVATE_DATA will have a pointer to EFI_BOOT_SCRIPT, which contain the whole BootScript entries, like MMIO_WRITE, IO_WRITE, PCI_WRITE, with a header and a terminate entry. Both SCRIPT_TABLE_PRIVATE_DATA and EFI_BOOT_SCRIPT.HEADER have TableLength field. The difference is that SCRIPT_TABLE_PRIVATE_DATA.TableLength does not include final TERMINATE, while EFI_BOOT_SCRIPT.HEADER.TableLength include the TERMINATE.

There are 4 important fields of SCRIPT_TABLE_PRIVATE_DATA. They record the current boot script library status.

– **InSmm**: It records if this library is linked by SMM driver and in SMM.
– **AtRuntime**: It records if current state is after SmmExitBootServices or SmmLegacyBoot.
– **SmmLocked**: It records if current state is after SmmReadyToLock.
– **BackFromS3**: It indicates that the system is back from S3. It is used by **BootScriptSmmPrivateDataGuid** LockBox.

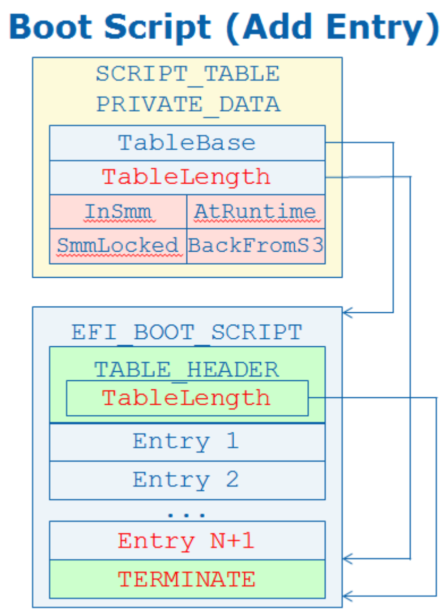

Figure 10 Boot Script: add entry

When silicon drivers use BootScript library or S3 Save State protocol to add a new boot script entry, the boot script lib will check if the length exceeds MAX table length. If the new length exceed the MAX table length, and **SmmLocked** is FALSE, the library will reallocate EFI_BOOT_SCRIPT with sufficient length. Then the library will append this new entry and update TableLength field. The TERMINATE field is added in SmmReadyToLock event, and it is updated after every new SmmBootScript entry is added.

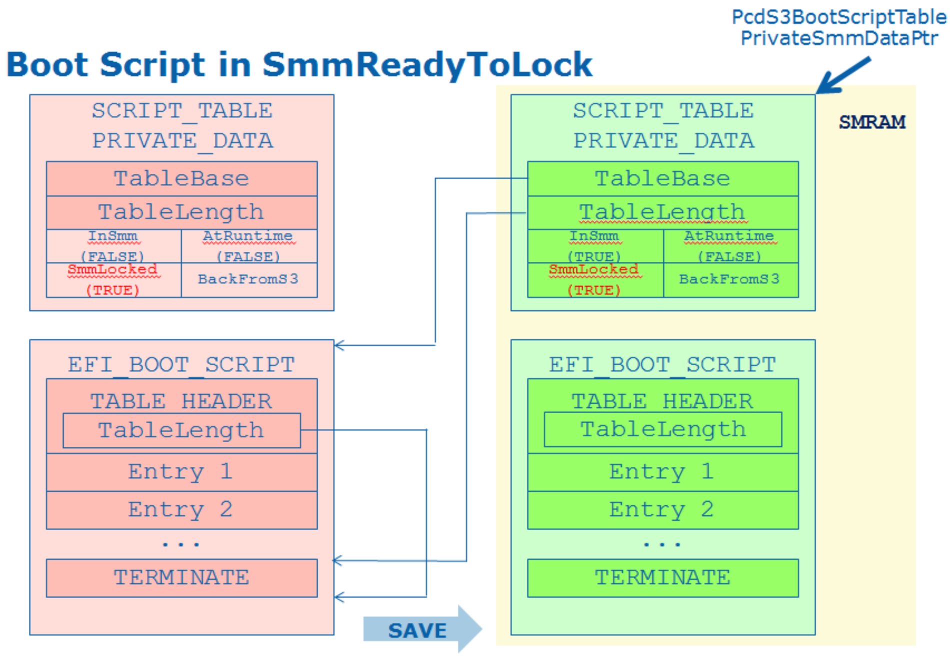

Figure 11 Boot Script: in SmmReadyToLock

When SmmReadyToLock event happens, the DXE version boot script library will set **SmmLocked** as TRUE, add final TERMINATE, and close the boot script service. The DXE version boot script also saves the whole EFI_BOOT_SCRIPT to **BootScriptDataGuid** LockBox (Detail of LockBox will be discussed later).

When SmmReadyToLock event happens, the SMM version boot script library will duplicate a copy of SCRIPT_TABLE_PRIVATE_DATA into SMRAM, because the DXE copy cannot be trusted any more. The pointer is assigned as another dynamic PCD: PcdS3BootScriptTablePrivateSmmDataPtr, to make sure there is only one copy in global SMM area. For any further boot script request, the SMM version will update the EFI_BOOT_SCRIPT table, and sync the update back to **BootScriptDataGuid** LockBox.

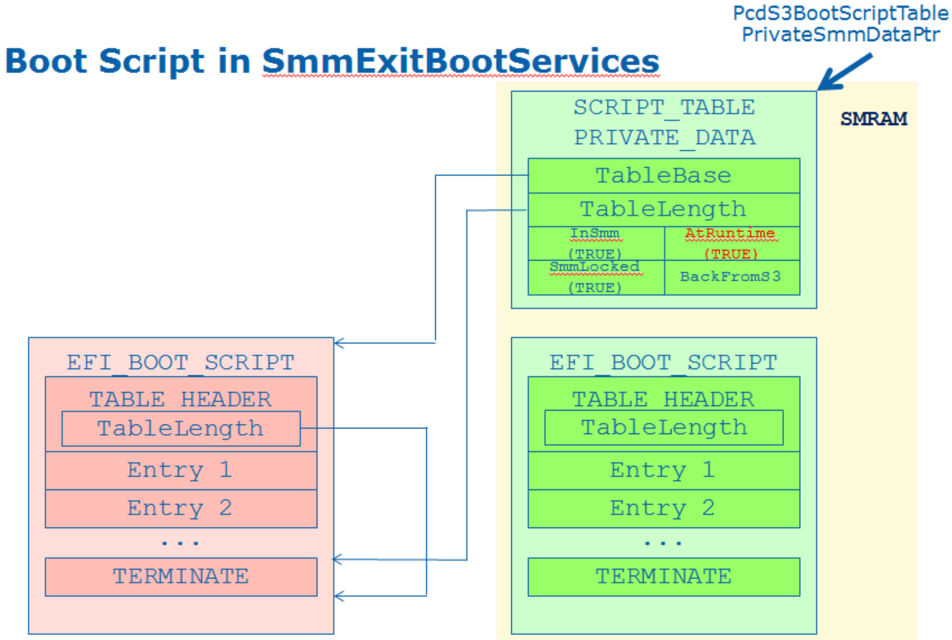

Figure 12 Boot Script: in SmmExitBootServices

When SmmExitBootServices or SmmLegacyBoot event happen, the SMM version boot script library will set **AtRuntime** as TRUE, and copy **BootScriptDataGuid** LockBox date to **BootScriptDataGuidBootTime** LockBox.

This library will also save **BootScriptSmmPrivateDataGuid** LockBox with **BackFromS3**=TRUE, so that when lockbox is restored by S3Resume module, this library can know such information, and will restore **BootScriptDataGuidBootTime** LockBox to **BootScriptDataGuid** LockBox later.

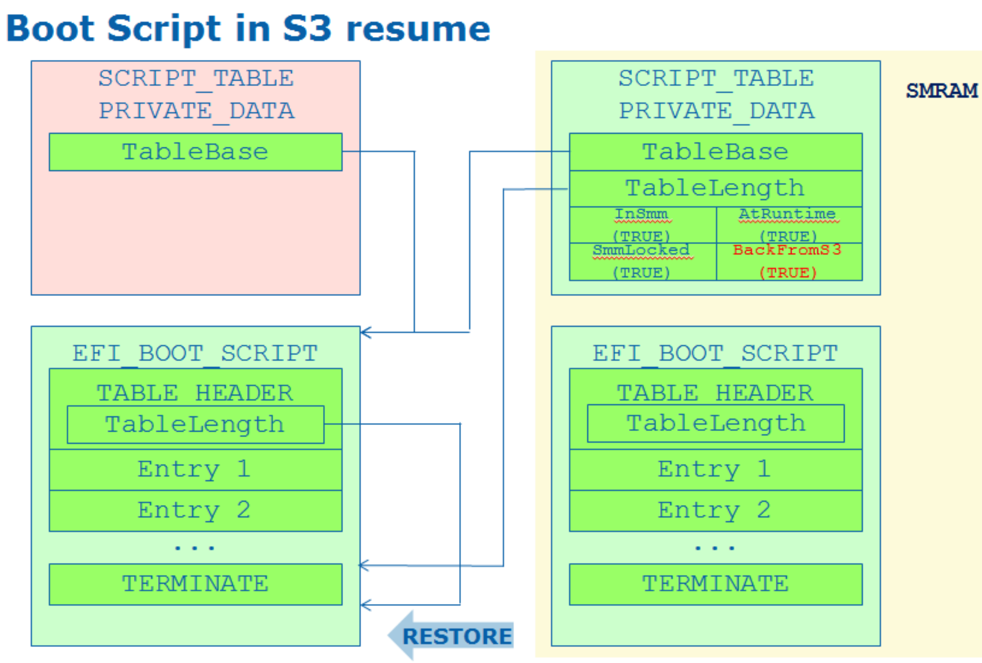

Figure 13 Boot Script: in S3 resume

Finally, during S3 resume phase, the EFI_BOOT_SCRIPT table is restored from **BootScriptDataGuid** LockBox, so the integrity is maintained. During LockBox restoration, the **BackFromS3** is also updated to TRUE.

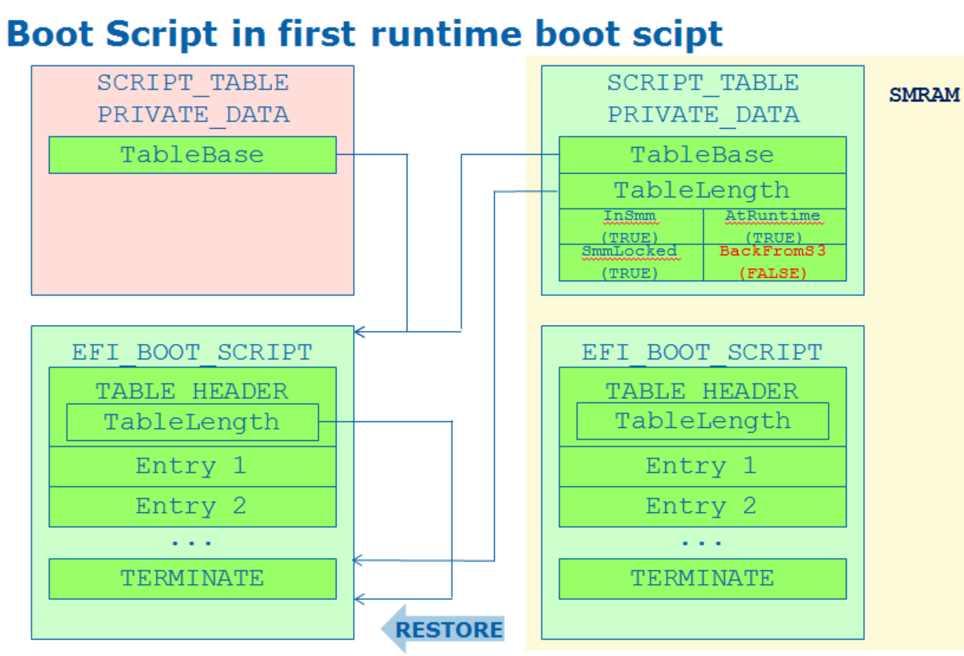

Figure 14 Boot Script: in first runtime boot script

In first UEFI runtime boot script write action, BootScript library detects **BackFromS3**. If it is TRUE, BootScript library restores **BootScriptDataBootTime lockbox** to **BootScriptData lockbox** and boot script in ACPI reserved memory. After that **BackFromS3** is set to FALSE.

Then the boot script memory becomes same as normal boot, and runtime boot script is just appended.

Processor Configuration

ACPI specification requires to the BIOS to restore processor configuration, which involves the following:

- Basic setup for System Management Mode (SMM)
- Microcode updates
- Processor-specific initialization
- Processor cache setting

In most platforms, SMM setup is done by SMMCPU driver. Microcode updates and processor initialization are done by CPU driver or SMMCPU driver. Processor cache setting is done by platform driver. Most of those drivers are close source.

There is one SMM interface (SMM_S3_RESUME_STATE) produced by platform, and consumed by SMM and S3 resume driver. It is defined in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/AcpiS3Context.h

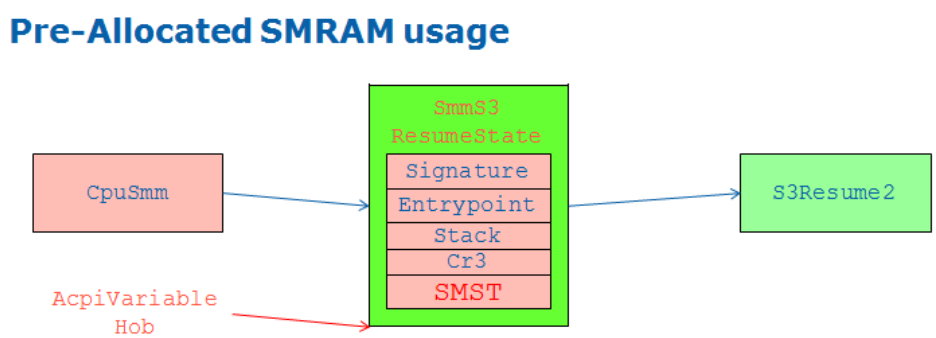

Figure 15 Pre-Allocated SMRAM for SMM S3

By design if a platform supports S3, it need pre-allocate a piece of SMRAM and report it via gEfiAcpiVariableGuid Hand-off-Block (HOB). Then SMMCPU driver can find this SMRAM and setup SMM_S3_RESUME_STATE data structure in normal boot.

In S3 boot mode, the S3resume driver will find same HOB, and jump to SMM_S3_RESUME_STATE.SmmS3ResumeEntryPoint to do SMM mode setup. Processor initialization may be done in SmmS3ResumeEntryPoint or done in a dedicated CPU PEIM.

S3 System Information

Besides Boot Script, there is some other information needed during S3, like FACS table address, the S3 page table address. This is done by AcpiS3SaveDxe driver https://github.com/tianocore/edk2/tree/master/IntelFrameworkModulePkg/Universal/Acpi/AcpiS3SaveDxe

A platform BSD should call EFI_ACPI_S3_SAVE_PROTOCOL.S3Ready() function to indicate that platform is ready to collect S3 information.

- AcpiFacsTable is required by S3 resume driver to find out waking vector.
- S3NvsPageTableAddress is required S3 resume driver to jump to X64 long mode boot script executor from IA32 mode.
- BootScriptStackBase and BootScriptStackSize are required by S3 resume driver to setup stack for boot script executor.
- IdtrProfile and S3DebugBufferAddress are required by boot script executor to setup IDT, and INT3 entry.

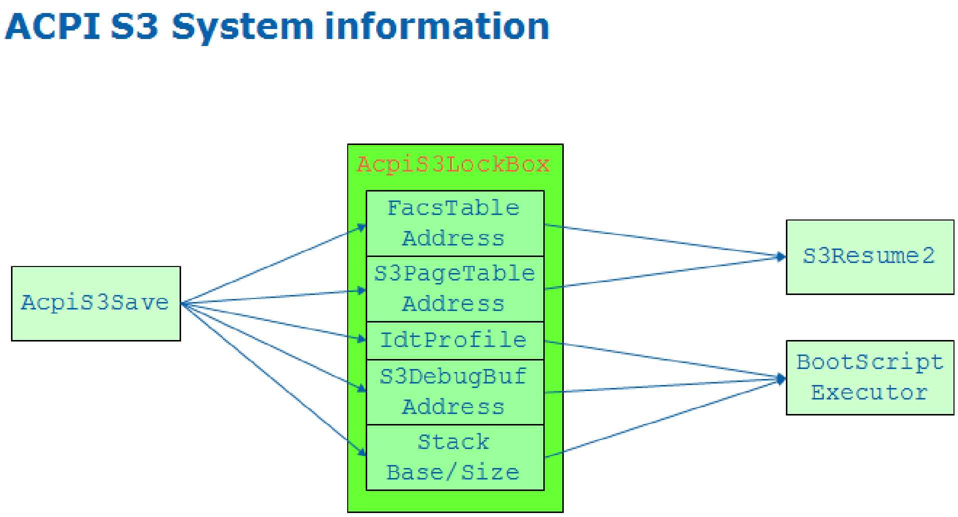

Figure 16 ACPI S3 system information

In order to maintain the integrity, the information is saved to LockBox with gEfiAcpiS3ContextGuid. In S3 resume mode, S3 resume driver can restore this LockBox to get the data.

S3 Sleep Trigger

If a platform supports S3, it should provide _S3 ACPI method in DSDT to define how to trigger S3. Also a platform may enable SXSMI so that when OS writes S3 to PM_CONTROL register, SMI will be triggered, and platform S3SmiHandle can do final boot script save action.

Normal Boot Flow for S3

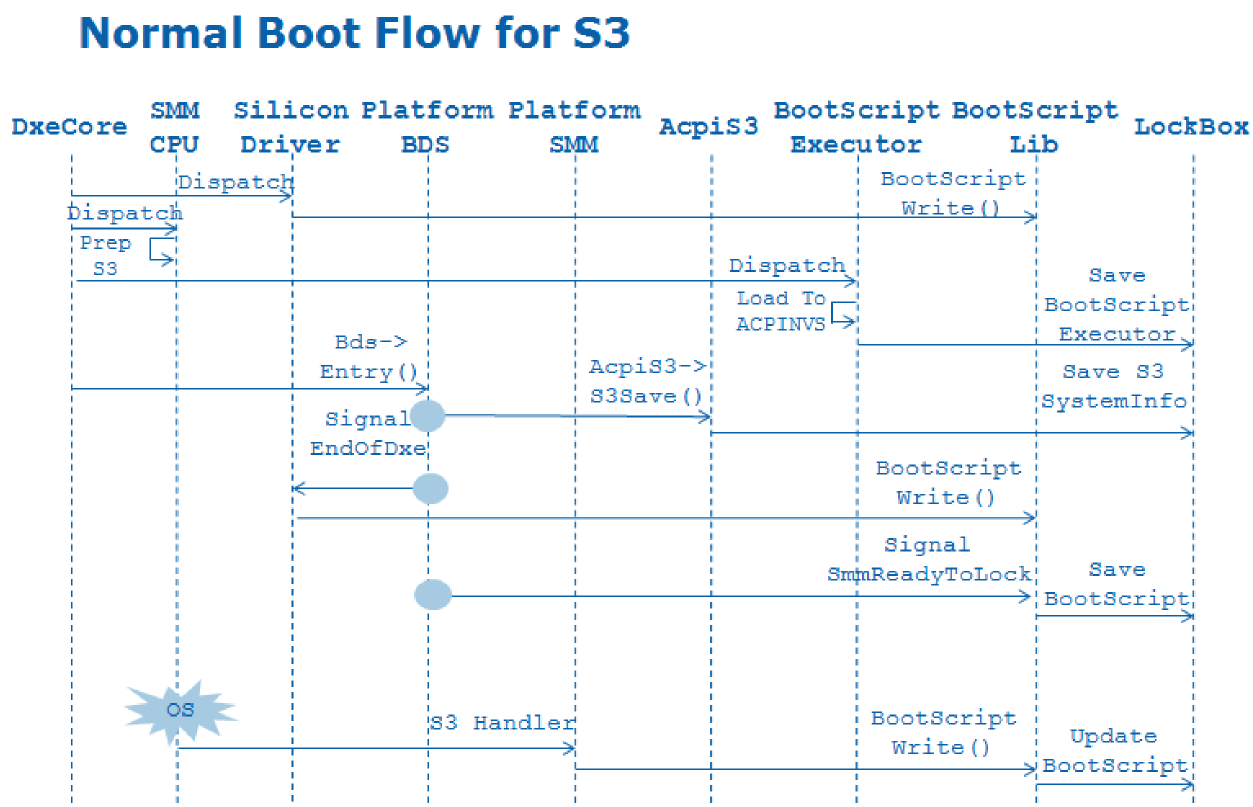

Figure 17 Normal Boot Flow for S3

Below is summary of normal boot flow for S3.

- DxeCore dispatches silicon driver. Silicon driver calls boot script library to save boot script record.
- DxeCore dispatches SMM CPU driver. SMM CPU driver prepares processor configuration restore and SMM_S3_RESUME_STATE in SMRAM.
- DxeCore dispatches BootScriptExecutor driver. BootScriptExecutor reloads itself from DXE memory to ACPI reserved memory, and uses LockBox to protect itself.
- DxeCore finishes DXE driver dispatch and call BDS->Entry() to enter BSD phase.
- PlatformBds calls AcpiS3->S3Save() to prepare S3 system information.
- PlatformBds signals EndOfDxe, which is last chance for silicon driver to record boot script..
- PlatformBds signals SmmReadyToLock, which causes boot script service close and boot script driver load itself into LockBox.

In OS environment, OS triggers S3. Then SMI is signaled and S3handler in PlatformSmm driver takes control. Then S3 handler collects system configuration and saves to SMM boot script. Then boot script library updates LockBox content again.

There are 2 important notes:

- PlatformBDS need 1) call AcpiS3->S3Save() to save S3 system information, then 2) signal EndOfDxe to give DXE silicon driver last chance to save boot script, and finally 3) signal

SmmReadyToLock to close boot script and save boot script to LockBox. The 2nd call and 3rd call must be in this order, or the boot script might not be able to save correctly.

- SmmBootScript has the capability to update boot script even during OS runtime. The only place to call BootScriptWrite should be in S3 handler, because there is no more OS code running.

Summary

This section describes how an EDKII BIOS prepares the S3 environment in a normal boot.

Part II – S3 Resume Boot Path

S3 Boot Mode Detection

A platform needs to detect the boot mode at an early portion of the PEI phase. By checking PM_CONTROL register, a platform may know if hardware is resumed from S3 boot mode.

A special case is that even if the hardware is resumed from S3 boot mode, the system might be in some other boot mode, like flash capsule update. This solution might be chosen purposely, because capsule update requires memory persist across the system reset.

Memory Initialization

After memory initialization, the MRC wrapper needs to report the PEI memory base and size. In the S3 resume path, the PEI memory base and size should be in ACPI reserved memory allocated during normal boot. See “Part I – S3 System Information” for details on where these data items could be located.

S3 Resume2 PPI

The DXE Initial Program Load (IPL) PPI is architecturally the last PPI that is executed in the PEI phase. When resuming from S3, the DXE IPL PEIM will transfer control to the S3 Resume PPI, which is responsible for restoring the platform configuration and jumping to the waking vector.

In EDKII, the S3Resume2 PEIM is at
https://github.com/tianocore/edk2/tree/master/UefiCpuPkg/Universal/Acpi/S3Resume2Pei

S3Resume PEIM performs the below-listed major actions:

- Restore all LockBox to its original place.
- Call SMM entry point to restore processor configuration.
- Lock SMM. This must be done to maintain SMM integrity.
- Call Boot Script Executor to restore chipset configuration.
- Signal EndOfPei, to notify other PEIMs.
- Find FACS OS waking vector to resume.

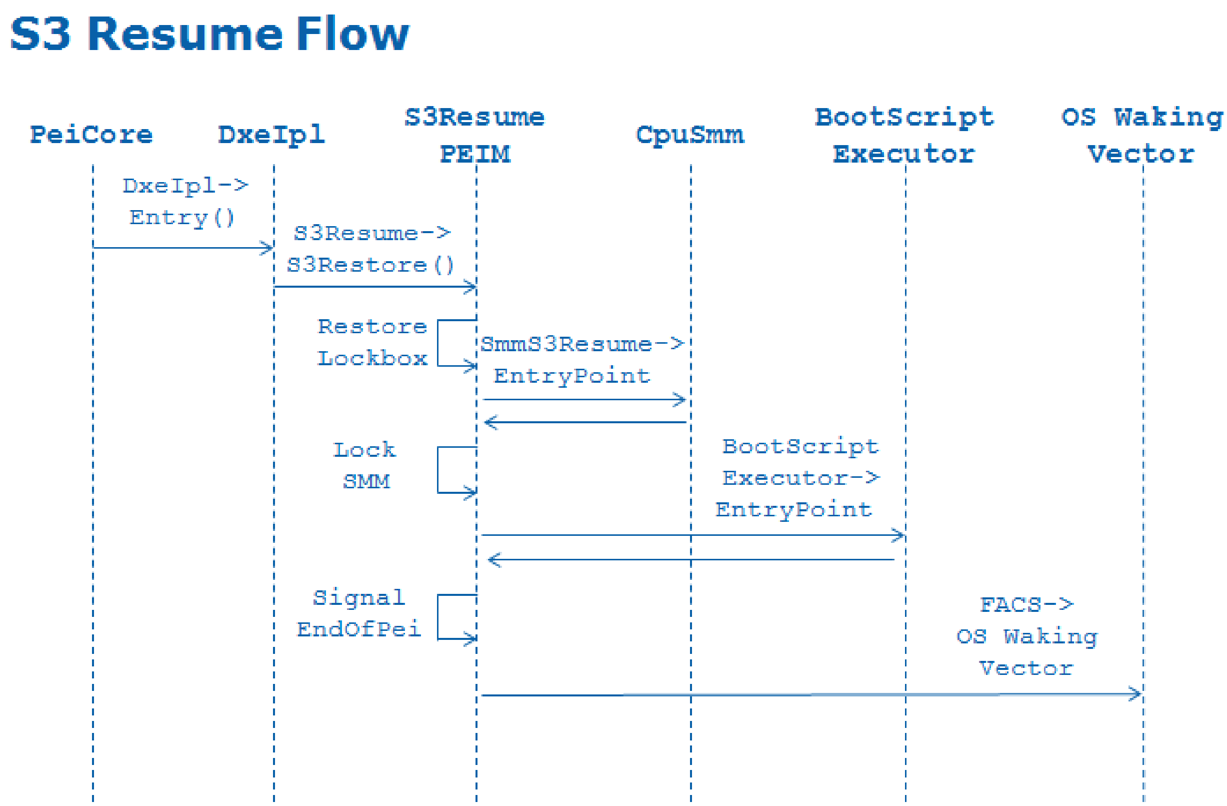

Figure 18 S3 Resume Flow

See above figure for detailed flow from DxeIpl to OS waking vector.

Performance Consideration

The PI specification requires the entry point of a boot script DISPATCH opcode to have the same calling convention as the PI DXE Phase. That means, if DXE is 64bit and PEI is 32bit, the boot script executor is 64bit instead of 32bit. When S3 resume driver to jump to X64 long mode boot script executor from IA32 mode, a new page table is needed.

In a normal boot, the AcpiS3SaveDxe needs to reserve memory page table memory. It is the S3 resume driver that creates page table content in S3 phase because the content in ACPI memory is not trusted in S3 resume phase. However, to create a page table for full system memory might be time consuming, especially for server platform.

The S3 resume driver does performance improvement by creating only a 4G page table by default, since most MMIO operation is below 4G. In order to support >4G MMIO access, the boot script executor will setup a page fault exception handle, and create >4G page table per request. This satisfies both performance and functionality.

Size Consideration

The ACPI specification requires the Paging mode is enabled and physical memory for waking vector is identity mapped for 64bit entry point. That means, we need setup a full page table (1G or 2M) for whole system memory and MMIO.

If a platform chooses 64bit waking vector, 2M page table, and full 48bit linear address, the page table size is huge. We recommend platform using 64bit waking vector with 1G page table enabled, or reporting small linear address. This can reduce overhead of reserved memory size for page table.

If a platform only supports 16bit or 32bit waking vector, there will be no page table size concern.

Processor Configuration Restoration

During normal boot path, the process drivers (CPU and SMMCPU) will prepare the S3 execution environment, and expose it in SMM_S3_RESUME_STATE HOB. See "Part I – Processor configuration" on where these data could be.

In S3 boot path, the same HOB is exposed by MRC. Since memory content is persist in S3 path, the SMM_S3_RESUME_STATE data in SMRAM is same as normal boot. S3 Resume PEIM finds the HOBs and passes control to the SMM_S3_RESUME_STATE. SmmS3ResumeEntryPoint to do SMM mode setup. Processor initialization may be done in SmmS3ResumeEntryPoint or done in a dedicated CPU PEIM. After SMM driver finishes the initialization, it will jump back to S3Resume PEIM.

The detail of processor configuration restore in SMM is silicon specific, and it is not discussed in this white paper.

Boot Script Executor

Boot Script Executor is a DXE module. In normal boot, it reloads itself to ACPI reserved memory, saves BootScriptExecutor entry point into LockBox, and uses LockBox to protect itself. In EDKII, BootScriptExecutorDxe is at https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/BootScriptExecutorDxe

The Boot Script Executor also links to S3BootScriptLibrary and uses the data structure defined in "Part I – Boot Script implementation".

During S3 resume phase, S3 Resume PEIM will restore the BootScript LockBox to get the BootScript entry point and jump to it. At same time, Boot Script Executor is restored from LockBox, so the integrity is maintained. Executor finds original Boot Script Table and executes all actions, like IO_WRITE, MMIO_WRITE, PCI_WRITE, etc.

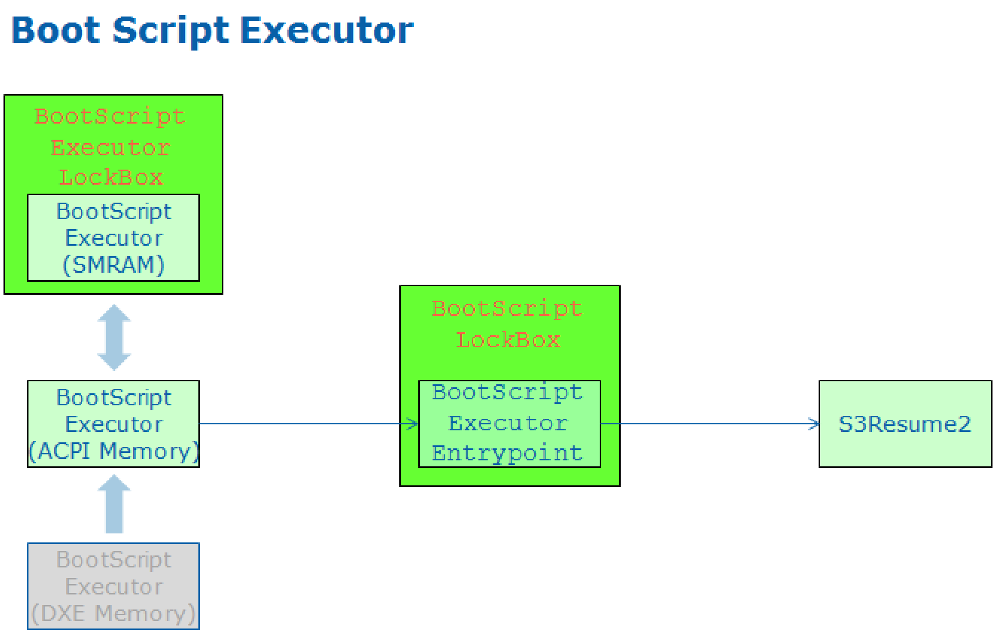

Figure 19 Boot Script Executor

Jump to OS waking vector

Finally, the S3Resume PEIM finds the OS waking vector in FACS, and resumes to the OS. The FACS table address is saved by the AcpiS3Save driver in a normal boot.

There are 3 possible waking vector defined by ACPI specification.

- 16 bit: Firmware_Waking_Vector, if X_Firmware_Waking_Vector does not exist or is zero. Real-mode address = (Physical address>>4) : (Physical address and 0x000F)
- 32 bit: X_Firmware_Waking_Vector, if either 64BIT_WAKE_SUPPORTED_F or 64BIT_WAKE_F flag is not set.
- 64 bit: X_Firmware_Waking_Vector, if both 64BIT_WAKE_SUPPORTED_F and 64BIT_WAKE_F flags are set.

S3Resume PEIM checks support for all of the above combination.

Summary

This section describes the S3 resume path in an EDKII BIOS.

Part III – Security Consideration

LockBox

We have discussed the S3 context preparation and the S3 resume path. Some components, like BootScript table and BootScript executor, are stored in ACPI reserved memory. There might be a risk that malicious software attacks the ACPI reserved memory and updates the content. The consequence is that silicon may be restored into a wrong condition, like a register based address stored wrong, or a register left unlocked. The former may cause the system to fail at booting, and the latter may cause security holes to be exposed.

In order to mitigate above threats, the EDKII designed a LockBox solution. **LockBox is a container to maintain the integrity of data, but not the confidentiality of data.** LockBox is a concept. There could be different implementation of LockBox, like SMM based LockBox, a Read-Only variable based LockBox, or an EC-based LockBox.

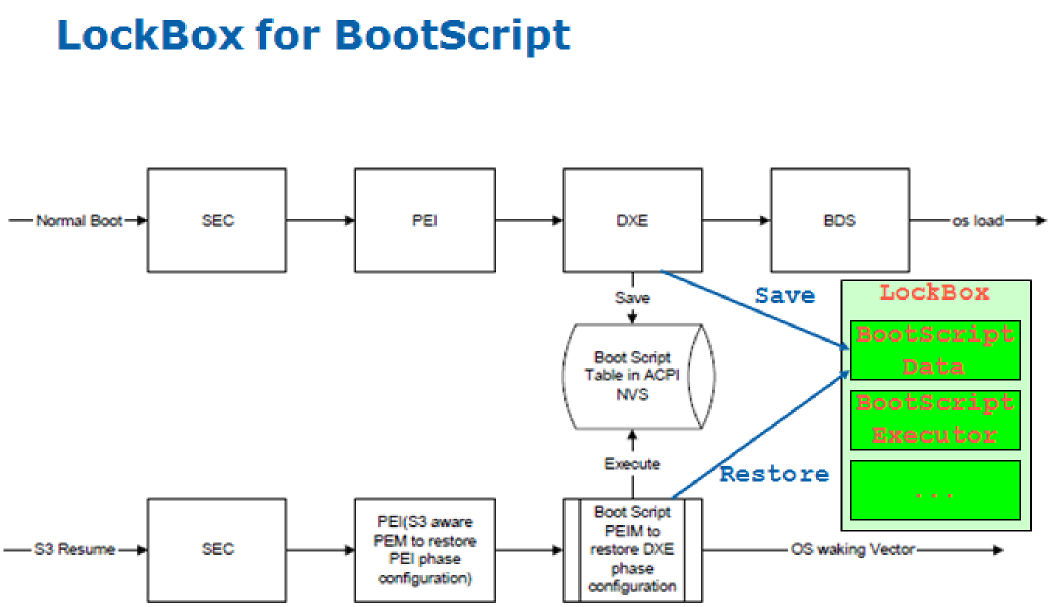

Figure 20 S3 Resume Boot Path with BootScript in LockBox

In EDKII, the API definition of LockBox is at https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Library/LockBoxLib.h
It provides below services:

1) SaveLockBox() - set data to LockBox.
2) UpdateLockBox() - update data to LockBox.
3) SetLockBoxAttributes() - set LockBox attributes. LOCK_BOX_ATTRIBUTE_RESTORE_IN_PLACE means this LockBox can be restored to original address with RestoreAllLockBoxInPlace().
4) RestoreLockBox() – get data from LockBox to caller provided buffer address, or original buffer address.

5) RestoreAllLockBoxInPlace() - restore data from all lockboxes which have LOCK_BOX_ATTRIBUTE_RESTORE_IN_PLACE attribute.

Not all LockBox services are available in all BIOS phases. In the next several sections, we will discuss details of a SMM-based LockBox implementation in EDKII.

SMM as LockBox

EDKII provides a default LockBox implementation – SMM based LockBox, since SMM is a standard execution environment defined in PI specification volume 4.

SMM LockBox involves below 2 modules.

1) SmmLockBoxLib – SMM based LockBox library. It has a PEI instance, a DXE instance, and a SMM instance.

https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/SmmLockBoxLib

2) SmmLockBox driver – provides services for a SMM based LockBox library and a DXE instance in the normal boot path, or a PEI instance after SMI enabled in S3 path.

https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/LockBox/SmmLockBox

SMM Instance

The SMM instance of SmmLockboxLib is the main functional part. It maintains the LockBox infrastructure in SMRAM. See below figure.

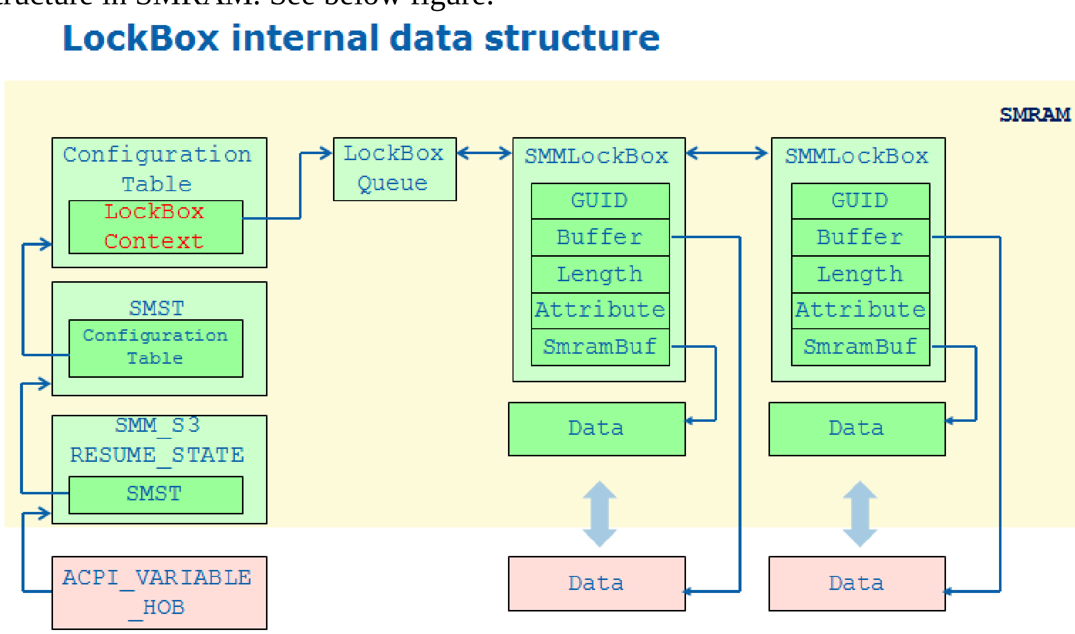

Figure 21 SmmLockBox: internal data structure

The LockBoxQueue is the header of SmmLockBox link list. Every LockBox below data structure:

1) GUID – it is the identity of LockBox.
2) Buffer – A pointer to original data buffer, it is used when caller request to restore to original buffer address.
3) Length – the size of data in LockBox.
4) Attribute – the attribute of LockBox,
5) SmramBuffer – the data buffer in SMRAM.

The address of LockBoxQueue is also saved as an SMM configuration table in SMST. The reason is that even when an SMI is not enabled yet, the PEI phase LockBox library can search the SMRAM region to find the LockBoxQueue and to find all LockBox content. This makes the PEI LockBox service available before SMM is ready in the S3 resume phase.

All 5 LockBox services are supported by the SMM instance, because SMM is trusted.

DXE Instance

The DXE instance of SmmLockboxLib calls SMM_COMMUNICATION protocol to communicate with the SmmLockbox service provider. The SmmLockBox driver provides LockBox software SMM handler to service the request from the DXE instance.

Let's take the boot script as an example: When a SMM version boot script lib requests a LockBox services, the code calls into the LockBoxSmmLib. The SMM instance allocates SMRAM, saves data and returns directly. When a DXE version boot script lib requests LockBox services, the code calls SMM_COMMUNICATE.Communicate(). Then a SWSMI is triggered, and the SmmCore finds the services handler registered by SmmLockBox driver. The LockBox SMM handler calls into SMM LockBox instance to allocate SMRAM and save data, then returns from SMM.

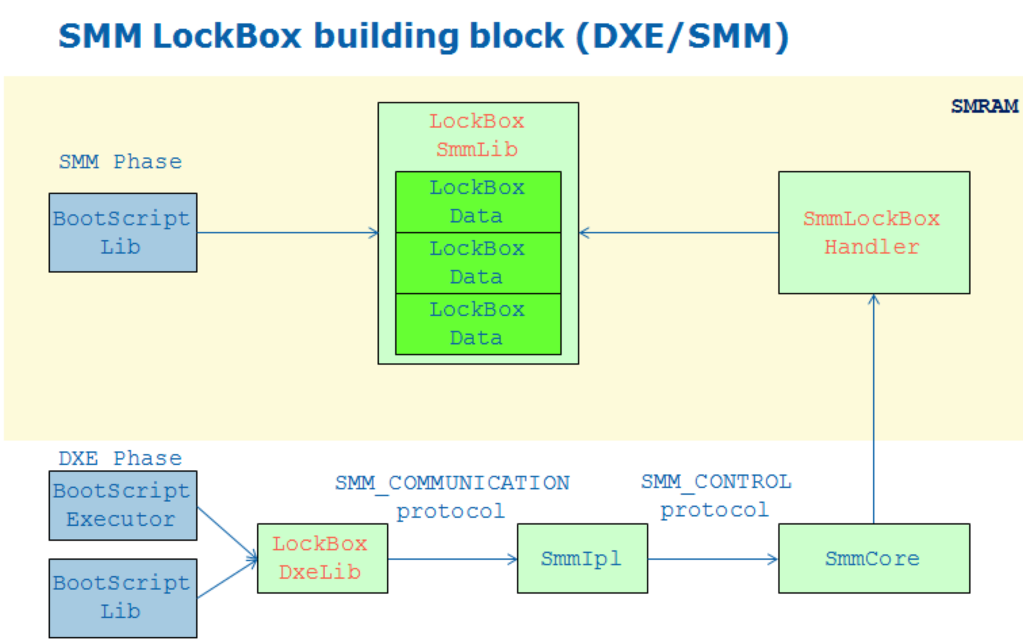

Figure 22 SmmLockBox: DXE/SMM

The DXE instance supports 5 LockBox services before SmmReadyToLock event. After SmmReadyToLock, SaveLockBox ()/UpdateLockBox ()/SetLockBoxAttribute() are closed and rejected by the LockBox SMM handler for security considerations.

PEI Instance

The PEI instance of SmmLockboxLib has two ways to communicate with the LockBox in SMRAM.
1) It uses the SMM_COMMUNICATION PPI to communicate the SmmLockbox service provider, similar as DXE instance.
2) When the PEI instance is used before SMM ready, the SMM_COMMUNICATION PPI will return EFI_NOT_STARTED. In this case, PEI SmmLockBoxLib needs to search the SMRAM region directly to find LockBox content.

See the LockBox internal data structure in SMM instance section. PEI SmmLockBoxLib can find ACPI_VARIABLE_HOB to get the SMM_S3_RESUME_STATE location, then get the SMST pointer. The address of LockBoxQueue is saved as SmmConfigurationTable in SMST. Care must be taken when PEI is 32bit while SMM/DXE is 64 bit, because all UINTN/VOID * defined in SMST must be parsed as UINT64 even in 32-bit PEI execution environment.

In S3PEI instance it only supports 2 LockBox services in S3 phase - RestoreLockBox() and RestoreAllLockBoxInPlace().

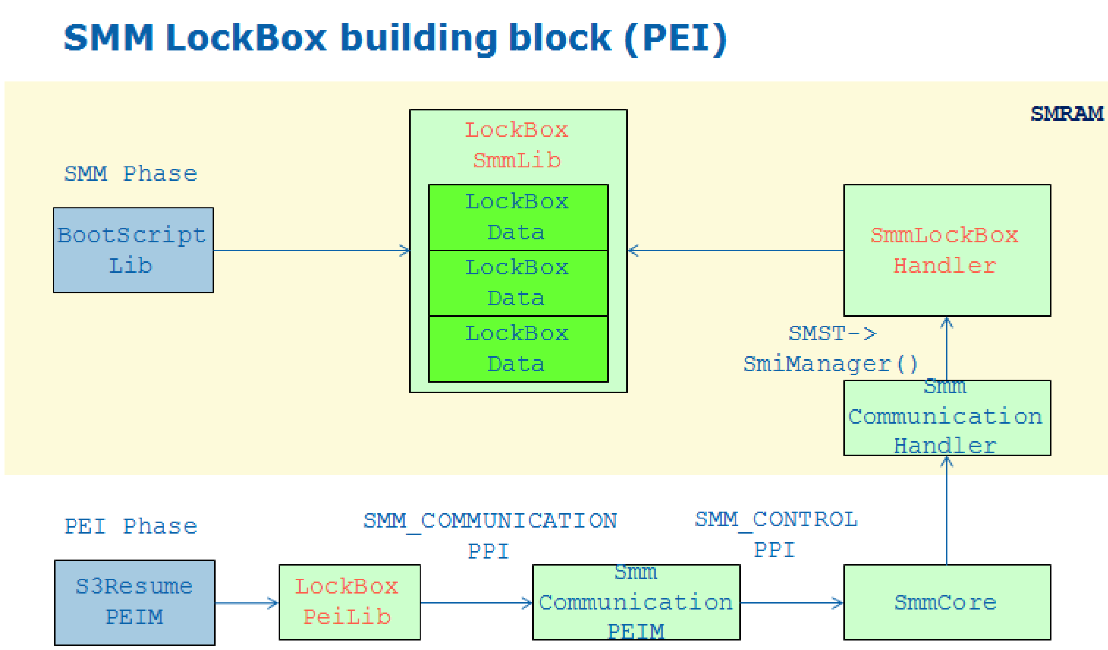

Figure 23 SmmLockBox: PEI/SMM

SMM Communication in PEI

In the DXE phase, the PI specification defines a standard API SMM_COMMUNICATION_PROTOCOL.Communicate() to let a DXE driver talk to a SMM

handler. It is implemented in SmmIpl at
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/PiSmmCore

In OS runtime, the UEFI specification defines a standard way - SMM Communication ACPI Table, to let the OS driver talk to a SMM handler.

In the PEI phase, EDKII also defines a SmmCommunication PPI to let a PEIM communicate with the SMM handler. The PPI definition is at
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Ppi/SmmCommunication.h This interface is used by LockBox PEI instance.

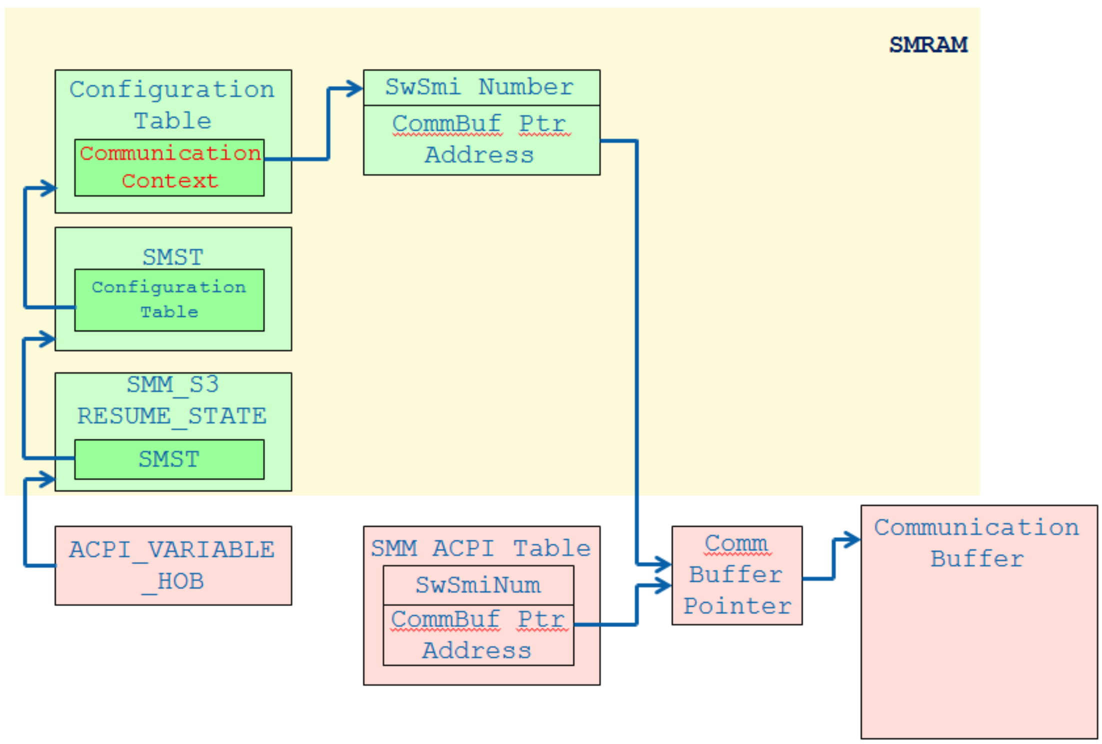

Figure 24 SmmCommunication (ACPI and PEI)

The PEI SmmCommunication implementation reuses the UEFI SMM Communication ACPI table. At boot time, a SmmCommunication driver installs the SMM ACPI table and prepares a Communication context in SMM ConfigurationTable.

At the S3 phase, the SmmCommunication PEIM can find the ACPI_VARIABLE_HOB to get SMM_S3_RESUME_STATE location, then get the SMST pointer. The SMM Communication Context is saved as SmmConfigurationTable in SMST. Then SmmCommunication PEIM can update the SMM ACPI table Communication Buffer pointer to let it point to data buffer, then signal SwSmi. Then a generic SMM communication handle is triggered and it will call SMST->SmiManager() to dispatch the SMM communication according to the GUID defined in communication buffer.

Using LockBox in Boot Script Driver

EDKII boot script implementation uses LockBox to protect:

1) Boot script executor (part II – Boot Script Executor)
2) Boot script metadata - SCRIPT_TABLE_PRIVATE_DATA.TableBase (part I – Boot Script Implementation)
3) Boot script data record - EFI_SCRIPT_TABLE (part I – Boot Script Implementation)

Secure boot script limitation and solution: Using LockBox in Silicon Driver

EDKII BootScript implementation saves all BootScript to LockBox. However, care must be taken for 2 special OPCODE: DISPATCH_OPCODE, and DISPATCH_OPCODE_2. (See PI specification Vol5) These 2 API records EntryPoint and Context to be executed in S3 resume path. The EntryPoint is just a pointer of code to be run, and Context is argument to be passed into EntryPoint.

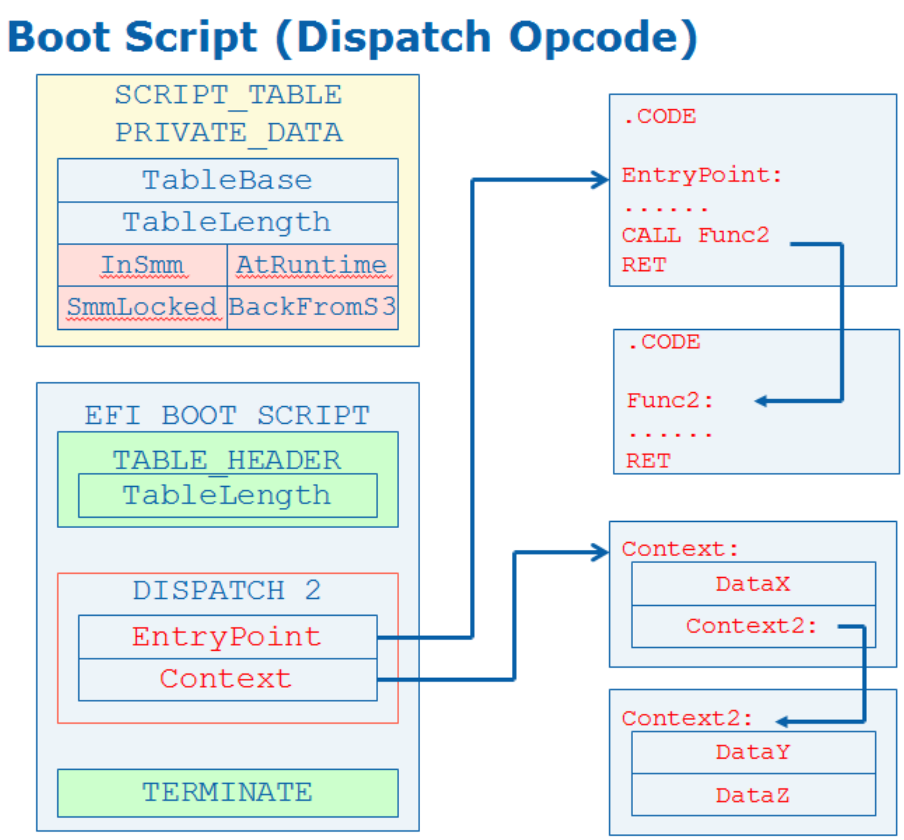

Figure 25 Boot Script: Dispatch Opcode 2

BootScriptLib only protects the 8 bytes EntryPoint, but cannot protect the whole code used by EntryPoint. The technical reason is that BootScriptLib does not have knowledge on the range of all code. BootScriptLib cannot use UEFI defined LoadedImage protocol, because this piece of code might be loaded by PE/COFF loader only, without gBS->LoadImage. BootScriptLib cannot search PE/COFF header, because the search-up is risky. Even PE signature is found, it cannot guarantee the 100% correctness, because the PE signature might be in .CODE segment or .DATA segment. Also there is no requirement that all code should be in same PE/COFF file, and there is no requirement that all code must have PE/COFF header.

The BootScriptLib only protects the 8 bytes of Context, but it cannot protect the whole Context data. The technical reason is that BootScriptLib does not know the length of Context data. It does not know how much data should be protected. Also the Context might contain another data pointer, but the BootScriptLib does not have such knowledge for 2nd layer pointer, and has no way to protect the data pointed by Context data.

Based on above reason, **EDKII BootScriptLib requires the producer of DISPATCH_OPCODE and DISPATCH_OPCODE_2 to call LockBox explicitly to protect 1) all code used by EntryPoint, and 2) the whole data region used as Context.**

LockBox limitation and solution: ReadOnly Variable and pre-allocated SMRAM

LockBox can be used to protect data in most case, but not in all case. It depends on when the LockBox is ready. In SmmLockBox implementation, the dependency is DRAM initialization.

- Case 1:

In most IA platforms, MRC (memory reference code) is the component to initialize memory. So LockBox cannot be used in MRC before DRAM initialization, because SMM is not ready. However, MRC module in S3 path depends on memory configuration data, and the configuration data must be saved in secure place. So the solution is to save to variable and lock it by using EDKII_VARIABLE_LOCK_PROTOCOL. Since this is variable is read only after EndOfDxe, the integrity is maintained.

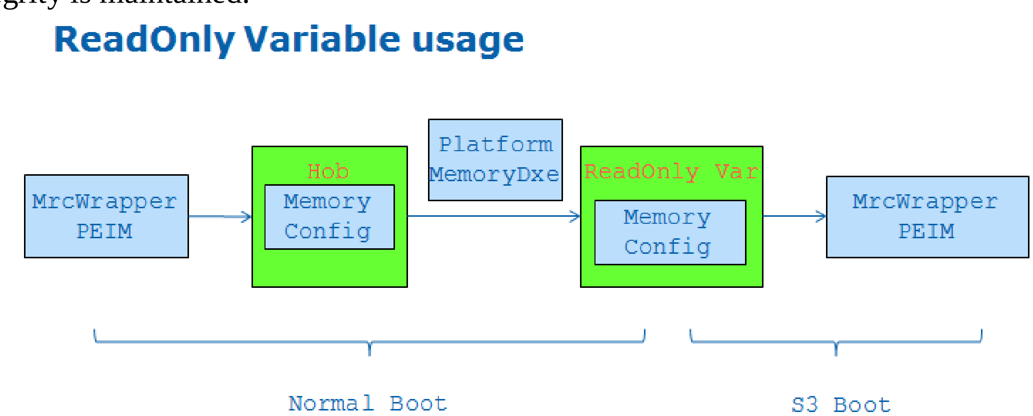

Figure 26 ReadOnly Variable Usage for Memory Configuration Data

- Case 2:

In S3 resume path, MRC wrapper also need install PEI memory used in S3. This information is collected in normal boot path, and should be saved securely. It is tricky that SmmLockBox might not be able to be used here because of software dependency. There are 2 options to save the data:

1) Use read only variable. EDKII variable driver has EDKII_VARIABLE_LOCK_PROTOCOL. A platform S3 driver may allocate this reserved memory base and size, then save all the data into a PlatformS3 state variable and lock it. This is similar as case 1.
2) In pre-allocated SMRAM. Since a platform already pre-allocate SMRAM for SMMS3 resume state, it can allocate larger piece for platformS3 state. Then a platform S3 driver can allocate and record reserved memory base and size for MRC in S3 boot path. Since SMRAM cannot be modified without authorization, the integrity is maintained.

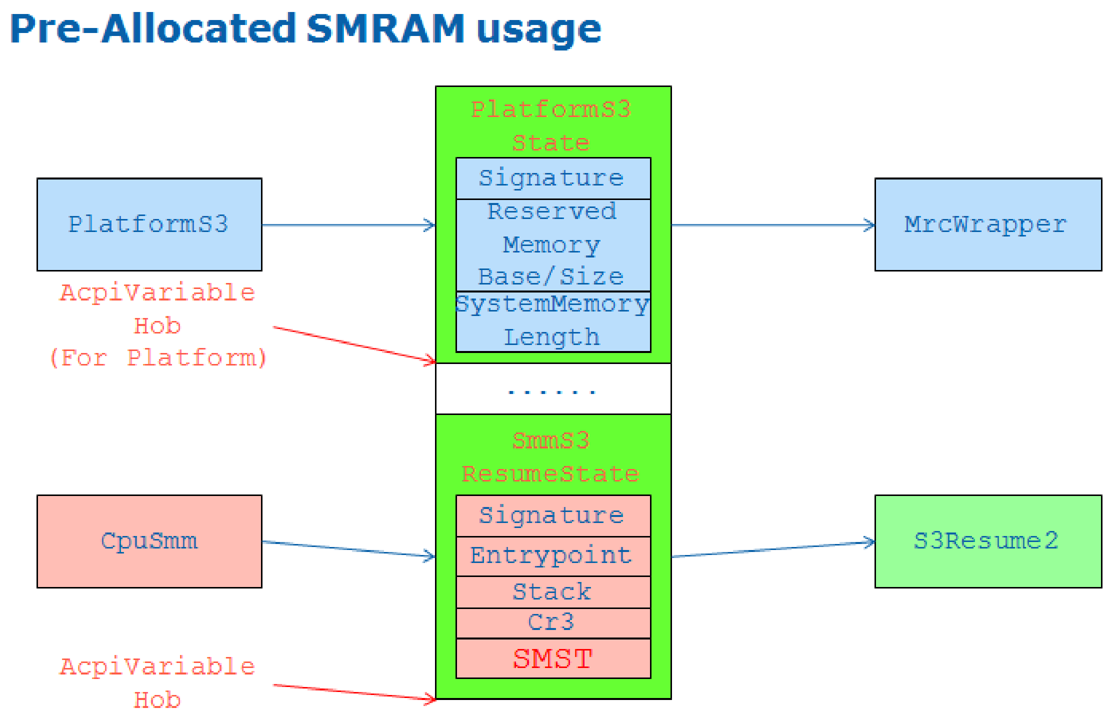

Figure 27 Pre-Allocated SMRAM for SMM S3 and Platform S3

As a conclusion, the **PEI driver needs to use a ReadOnly variable (EDKII_VARIABLE_LOCK_PROTOCOL) before DRAM initialization instead of SmmLockBox. The PEI driver may use pre-allocated SMRAM after DRAM is initialized and before the SmmLockBox driver is ready.**

Summary

This section describes the security considerations in S3 and introduced the details of the LockBox, including the limitations and details of the solution of a secure boot script and Lockbox.

Part IV – Compatibility Support

The S3SaveState protocol is a PI defined standard. However, a platform developed before PI specification uses another Intel Framework defined protocol – BootScriptSave protocol. EDKII provides compatibility support for EDKI driver in EdkCompatibilityPkg.

Boot script Save Protocol

BootScript Save protocol interface is provided by BootScriptSaveOnS3SaveStateThunk driver https://github.com/tianocore/edk2/tree/master/EdkCompatibilityPkg/Compatibility/BootScriptSaveOnS3SaveStateThunk

It consumes S3SaveState protocol and produces BootScriptSave protocol.

Framework SmmBootScript Lib

Although Intel Framework BootScriptSave protocol does not have SMM support, EDKI Framework implementation has SmmBootScript library, which provides support for EDKI SmmPlatform driver to save boot script in SMM phase.

In order to provide same functionality, EdkCompatibilityPkg also defined SmmScriptLib https://github.com/tianocore/edk2/tree/master/EdkCompatibilityPkg/Foundation/Library/Smm/SmmScriptLib

EDKI SmmPlatform driver can link against this library to work in EDKII BIOS.

Boot Script Executor

Most Boot Script OPCODEs defined in Intel Framework BootScriptSave protocol are the same as those supported by the PI S3SaveState protocol. However, the DISPATCH OPCODE execution environment is different.
The Intel Framework [FRAMEWORK] BootScriptSave protocol requires DISPATCH OPCODE running in PEI environment, while PI S3SaveState protocol requires DISPATCH OPCODE running in DXE environment. If a platform needs 32bit PEI and 64bit DXE, a thunk is required to convert the boot script execution environment.

However, since all OpCode is same, it is hard to identify a DISPATCH_OPCODE is framework one or PI one. EDKII BootScript thunk uses OpCode translation to resolve it.

Once a framework driver uses BootScriptSave protocol to record DISPATCH_OPCODE, the thunk driver will record a DISPATCH_OPCODE2, with Address being fixed ThunkStub, and Context being original Entrypoint32. The ThunkStub is a fixed entry point in BootScriptThunk driver. And since Framework BootScript does not support DISPATCH_OPCODE2, there is no ambiguity for DISPATCH_OPCODE2.

Figure 28 Framework Dispatch OpCode translation

In S3 resume path, when BootScript executor runs Dispatch Function 64, it may call into a PI version silicon boot script DISPATCH, or it may call into ThunkStub provided by BootScriptThunk. If it is latter case, BootScriptThunk driver will find real Dispatch Function 32 in Context, switch to IA32 mode to run it, then return back to X64 mode.

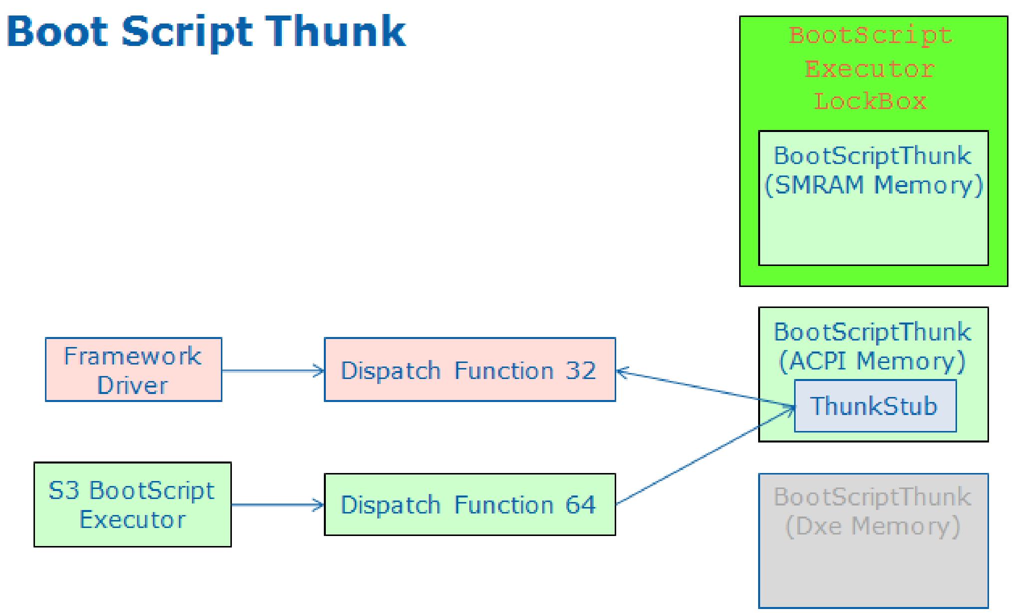

Figure 29 Boot Script Thunk

BootScript Thunk is a DXE module. In normal boot, it reloads itself to ACPI reserved memory. A BootScript Thunk helper will uses LockBox to protect the BootScriptThunk. In EDKII, BootScriptSaveOnS3SaveStateThunk driver is https://github.com/tianocore/edk2/tree/master/EdkCompatibilityPkg/Compatibility/BootScriptSaveOnS3SaveStateThunk

BootScriptThunkHelper is https://github.com/tianocore/edk2/tree/master/EdkCompatibilityPkg/Compatibility/BootScriptThunkHelper

Summary
This section describes the compatibility support on how to run EDKI driver in EDKII BIOS environment.

Conclusion

The S3 resume boot path is an important boot flow in most PC and mobile platforms. This paper describes the detailed work flow in an S3 resume boot path and the security considerations involved in S3 resume path.

Glossary

ACPI – Advanced Configuration and Power Interface. Static tables and ACPI Machine Language (AML) interpreted byte code. Preferred OS runtime interface to the platform.

Boot mode – PI token describing the restart type.

MRC – Memory Reference Code. This is silicon specific code to initialize memory controller.

PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications.

S3 – Sleep State 3.

SMM – System Management Mode. x86 CPU operational mode that is isolated from and transparent to the operating system runtime.

UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system. Predominate interfaces are in the boot services (BS) or pre-OS. Few runtime (RT) services.

References

[ACPI] Advanced Configuration and Power Interface www.uefi.org

[EDK2] UEFI Developer Kit www.tianocore.org

[FRAMEWORK] The Intel Platform Innovation Framework for the Extensible Firmware Interface Specifications http://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-specifications-general-technology.html

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.5 www.uefi.org

[UEFI Book] Zimmer, et al, "Beyond BIOS: Developing with the Unified Extensible Firmware Interface," 2nd edition, Intel Press, January 2011

[UEFI Overview] Zimmer, Rothman, Hale, "UEFI: From Reset Vector to Operating System," Chapter 3 of *Hardware-Dependent Software*, Springer, February 2009

[UEFI PI Specification] UEFI Platform Initialization (PI) Specifications, volumes 1-5, Version 1.4 www.uefi.org

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 and has been working on the EFI team at Intel since 1999.

Star Zeng (star.zeng@intel.com) is EDKII BIOS engineer with Software and Services Group at Intel Corporation.