White Paper

A Tour Beyond BIOS Launching Standalone SMM Drivers in the PEI Phase using the EFI Developer Kit II

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

May 2015

Executive Summary

In the current UEFI PI infrastructure, SMM drivers are loaded in the PI DXE phase. Usages such as the Intel® Firmware Support Package (FSP) may include requirements that the SMM initialization be done in the early PI PEI phase, namely since current FSP environments are in PEI. Intel FSP is a binary to encapsulate Intel silicon module initialization. In addition, server Reliability, Availability, Serviceability (RAS) features may also require some RAS SMM modules to be launched in PEI because portions of RAS are part of the silicon module set. This paper presents how to support launching silicon specific SMM drivers in the PI PEI phase, while at the same time maintaining compatibility to launch existing SMM drivers in the PI DXE phase.

Prerequisite

This paper assumes that audience has EDKII/UEFI firmware development experience [UEFI][UEFI PI Specification] and IA32 SMM knowledge [IA32 Manual]. He or she should be familiar with UEFI/PI firmware infrastructure (e.g., PEI/DXE) and know the IA32 SMM driver flow. [UEFI Book]

Table of Contents

Overview

Introduction to SMM

System Management Mode (SMM) is a special-purpose operating mode in Intel® architecture based CPUs. SMM was designed to provide for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware from the manufacturer and not intended to be 3rd party extensible. [IA32 Manual]

Introduction to PI SMM

In order to support SMM in system firmware, [UEFI PI Specification] Volume 4 describes detailed infrastructure on how to support SMM in UEFI PI-based firmware. See below figure 1 for details.
The SMM Initial Program Loader (IPL) will load the SMM Foundation into SMM memory (SMRAM) and the SMM services will start at that time until a system shutdown.

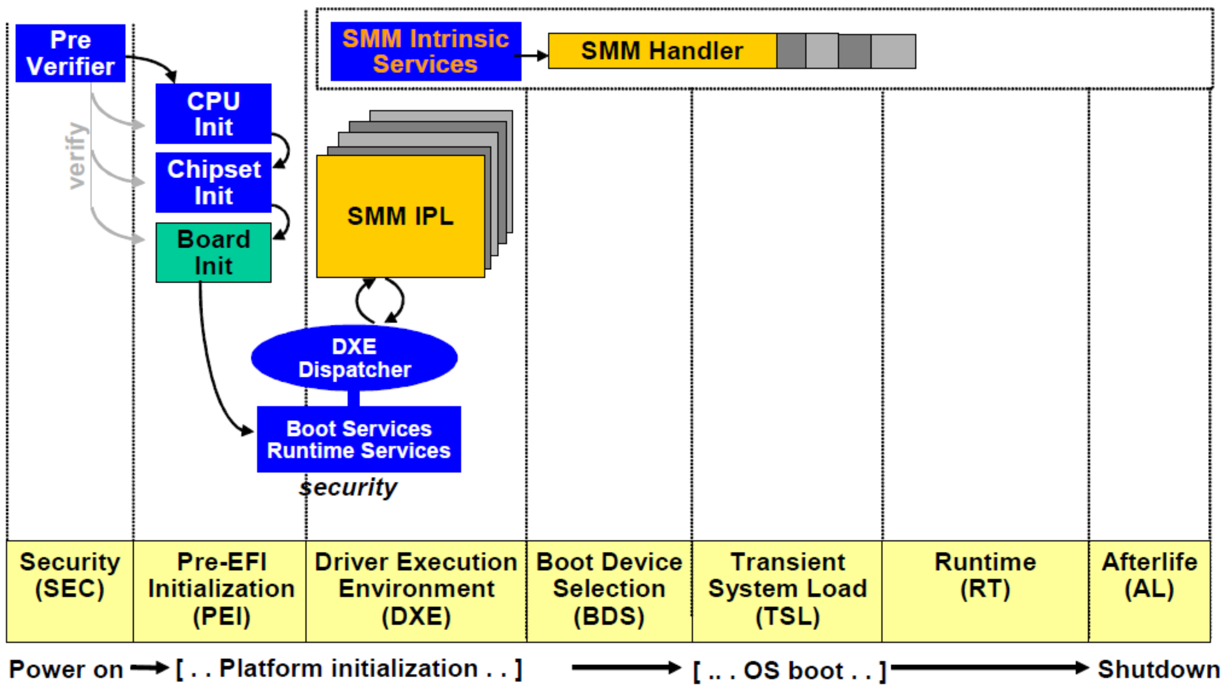

Figure 1 SMM architecture

Introduction to EDKII

EDKII is open source implementation of UEFI PI-based firmware which can boot multiple UEFI-aware operating systems. The EDKII open source includes the SMM infrastructure which follows the PI specification to provide capability of loading SMM drivers in the DXE phase of execution.

Summary
This section provided an overview of SMM and EDKII.

SMM Traditional Mode

In this document we refer to the current PI1.4 SMM infrastructure as the “SMM Traditional mode”. A brief introduction is given below. For more detail, please refer to [UEFI PI Specification] Volume 4.

Initialization

The current PI1.4 SMM infrastructure defines a rich set of interfaces. See figure 2.

- **`EFI_SMM_ACCESS2_PROTOCOL`** is used to control the visibility of the SMM memory (SMRAM) on the platform.
- **`EFI_SMM_CONTROL2_PROTOCOL`** is used initiate synchronous System Management Interrupt (SMI) activations.
- **`EFI_SMM_CONFIGURATION_PROTOCOL`** indicates which areas within SMRAM are reserved for use by the CPU for any purpose, such as stack, save state or SMM entry point.
- SMM IPL is to launch SMM foundation, and **`EFI_SMM_BASE2_PROTOCOL`** is used to locate the SMST during SMM driver initialization.

These modules work together to create the SMM infrastructure for SMM drivers in the DXE phase. The SMM foundation will load all of the SMM drivers, and the SMM drivers will register SMI handlers to service synchronous or asynchronous SMI activations. A synchronous SMI is generated by an overt action by ring 0 software running on the host, such as a write to port 0xB2, whereas an asynchronous SMI can be generated by a timer, GPI transition or other indicia from the platform without direct host software interaction.

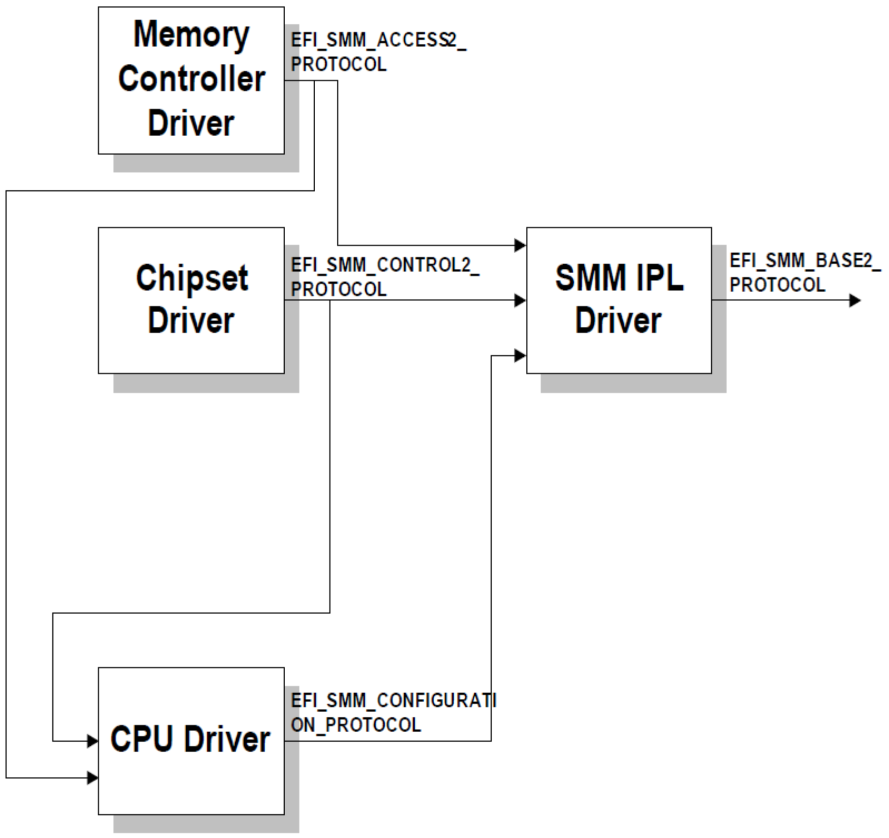

Figure 2 Example SMM Initialization Components

In EDKII, the SmmIpl driver is at https://github.com/tianocore/edk2-MdeModulePkg/tree/master/Core/PiSmmCore/PiSmmIpl.inf . The SmmFoundation driver is at https://github.com/tianocore/edk2-MdeModulePkg/tree/master/Core/PiSmmCore/PiSmmCore.inf.

SMI Handler

During runtime, when the SMI happens, the SMM Entry in the SmmCpu driver will be invoked initially. Then the SmmCpu driver will perform Boot-Strap Processor (BSP)/Application Processors (Aps) rendezvous, after which the BSP will invoke the SMM foundation entry point. The SMM foundation does not have knowledge on how to service SMI's, so it will call the registered SMI handlers. There are 3 types of SMI handlers. Details can be seen in figure 3 below.

- Root SMI Controller Handlers. These are handlers for devices which directly control SMI generation for the CPU. They are registered by calling **`SmiHandlerRegister`**() with **`HandlerType`** set to NULL.
- Child SMI Controller Handlers. These are SMI handlers which handle a single interrupt source from a Root or Child SMI handler. They are registered by calling the **`SmiHandlerRegister`**() function with **`HandlerType`** set to the GUID of the Parent SMI Controller SMI source.

- SMI Handlers. These SMI handlers perform basic software or hardware services based on the SMI source received.

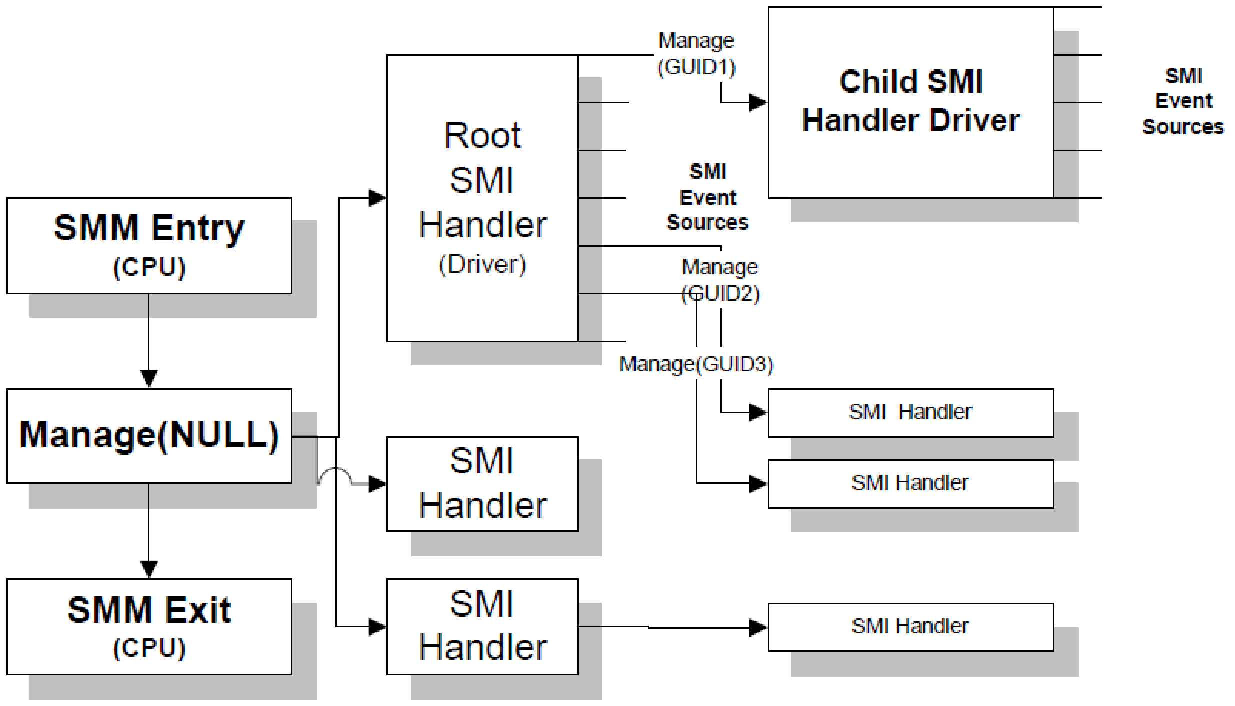

Figure 3 SMI Handler

Summary
This section provides a brief introduction to the current PI1.4 SMM infrastructure.

SMM Stand-alone Mode

Problems in the SMM Traditional mode

PI1.4 SMM mode is a reasonably complete infrastructure to support SMM drivers. At the same time, though, we also see some requirements to create an SMM environment earlier, e.g. in the PEI Phase. An example can include but is not limited to early Intel-only silicon initialization code, such as the Intel ® Firmware Support Package, wherein only PEI modules are executed prior to invoking an open source boot loader, such as open source EDK II or coreboot [EMBED] platform code. Intel FSP entails having an early container for Intel-only, IP-protected binary content, so having the ability to load content into SMM during this phase may be required, especially for servers and some of the reliability, availability, and serviceability (RAS) codes [APEI].

Now the problem is that the current PI1.4 SMM (traditional mode) does not provide such capability. For example, the entry point of the SMM driver is the same as a UEFI specification **`EFI_IMAGE_ENTRY_POINT`**, but EFI System Table is not available in the PEI phase.

SMM Standalone Mode

In order to resolve the problems listed above, we introduce the “SMM standalone mode” concept. SMM standalone mode can run in any firmware phase, whether it be DXE, PEI or even SEC.

An SMM standalone mode driver can only run in the SMM environment. It is not allowed to touch any other resource, such as a UEFI protocol in the DXE phase, or a PPI in PEI phase.

SMM standalone mode happens to resolve another security concern that some SMM drivers might call a UEFI protocol in SMM phase. Now, a standalone SMM driver does not have any interface to access UEFI protocols, so the risk does not exist.

Summary
This section provides the problem statement for the current SMM Traditional Mode, and introduce concept of SMM standalone mode.

In the next several sections, we will introduce SMM standalone mode components one by one.

Standalone SMM IPL

Load SMM Foundation

The goal of SMM IPL is to load the SMM foundation from the flash region to SMRAM, and jump to its entry point. Details can be found in figure 4 below.

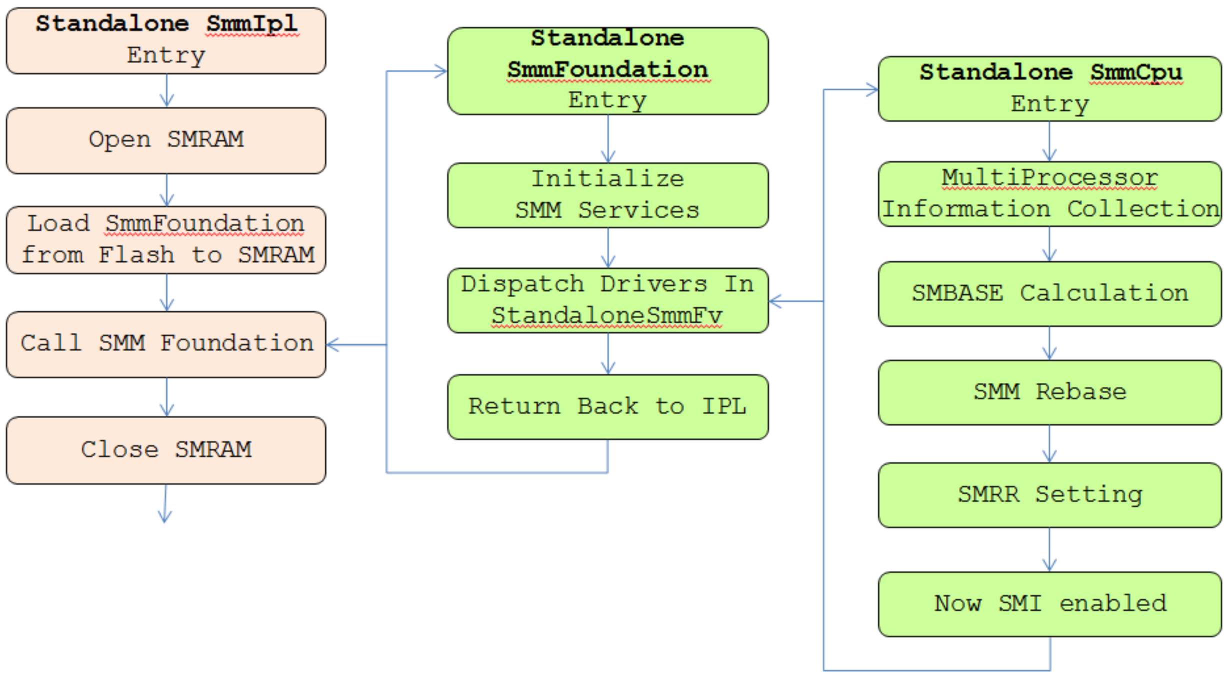

Figure 4 Standalone SMM Mode initialization

A sample SMM initialization in PEI might include:

1. A PEIM produces the **`EFI_PEI_SMM_ACCESS2_PPI`**, which describes the different SMRAM regions available in the system.
2. A PEIM produces the **`EFI_PEI_SMM_CONTROL2_PPI`**, which allows synchronous SMIs to be generated.
3. A PEIM (dependent on the **`EFI_PEI_SMM_ACCESS2_PPI`** and, perhaps, the **`EFI_PEI_SMM_CONTROL2_PPI`**), does the following:
• Initializes the SMM entry vector with the code necessary to meet the entry point requirements described in “Entering & Exiting SMM”.
• Opens SMRAM • Creates the SMRAM heap.
• Loads the SMM Foundation into SMRAM. The SMM Foundation produces the SMST.
• Optionally, closes SMRAM so that it is no longer visible.
• Optionally, locks SMRAM so that its configuration can no longer be altered.
• Publishes the **`EFI_PEI_SMM_COMMUNIATION_PPI`**
4. SMM Foundation does following
• Initialize all SMM services in SMST

• Dispatch drivers in StandaloneSmmFv
5. SMM CPU driver is dispatched by SMM Foundation. This driver does following:
• Collect multiprocessor information
• Calculate SMBASE and do SMM Rebase
• Set SMRR register
• Publish SMM_CONFIGURATION_PROTOCOL to let SMM foundation register SMM entry point.
6. SMM Foundation invokes the *RegisterSmmEntry()* function with the SMM Foundation SMM entry point.
• At this point SMM is initially configured and SMIs can be generated.
7. During the remainder of the PEI phase, additional SMM standalone drivers may load and be initialized in SMRAM.
8. During the remainder of the DXE phase, additional SMM standalone drivers may load and be initialized in SMRAM.

SMM Foundation Entry Point

Since SMM foundation cannot touch any UEFI or PEI services, but it does need some information like where is SMRAM. How can the SMM foundation ascertain such information?

The easiest way is to let the SMM IPL pass a set of parameters to the SMM Foundation via a Hand-off Block (HOB).

The SMM HOB can be used via invocation of the SMM Foundation as shown below:

```
typedef
EFI_STATUS
(EFIAPI *STANDALONE_SMM_FOUNDATION_ENTRY_POINT) (
  IN VOID  *HobStart
  );
```

Then SMM Foundation can find any information via a HOB, such as the SMRAM location, Standalone SMM Firmware Volume.
NOTE: SMM IPL just uses the “HOB” concept here. It does not mean that the IPL must pass full PEI HOB list to the SMM Foundation. The SMM IPL can construct a subset of the PEI HOBs and pass them into the SMM Foundation.

SMM Communication

The SMM IPL also needs to provide a capability for SMM communication. In the DXE phase, it is done via the **EFI_SMM_COMMUNICATION_PROTOCOL**. In the PEI phase, it could be accomplished via a corresponding **PEI_SMM_COMMUNICATION_PPI**.

See figure 5 for more details. The SMM IPL provides SMM private data to the SMM Foundation. In SMM private data, there are data objects such as the Communication Buffer pointer, size and return status. The SMM Communication function in SMM IPL will fill the communication buffer, and then trigger the SMI. Then SMM Foundation can read the GUID in communication buffer and dispatch control to the appropriate SMM handler.

The SMM IPL PEIM can save SMM private data into a HOB so that the SMM IPL DXE driver can get this SMM private data and know the communication buffer location.

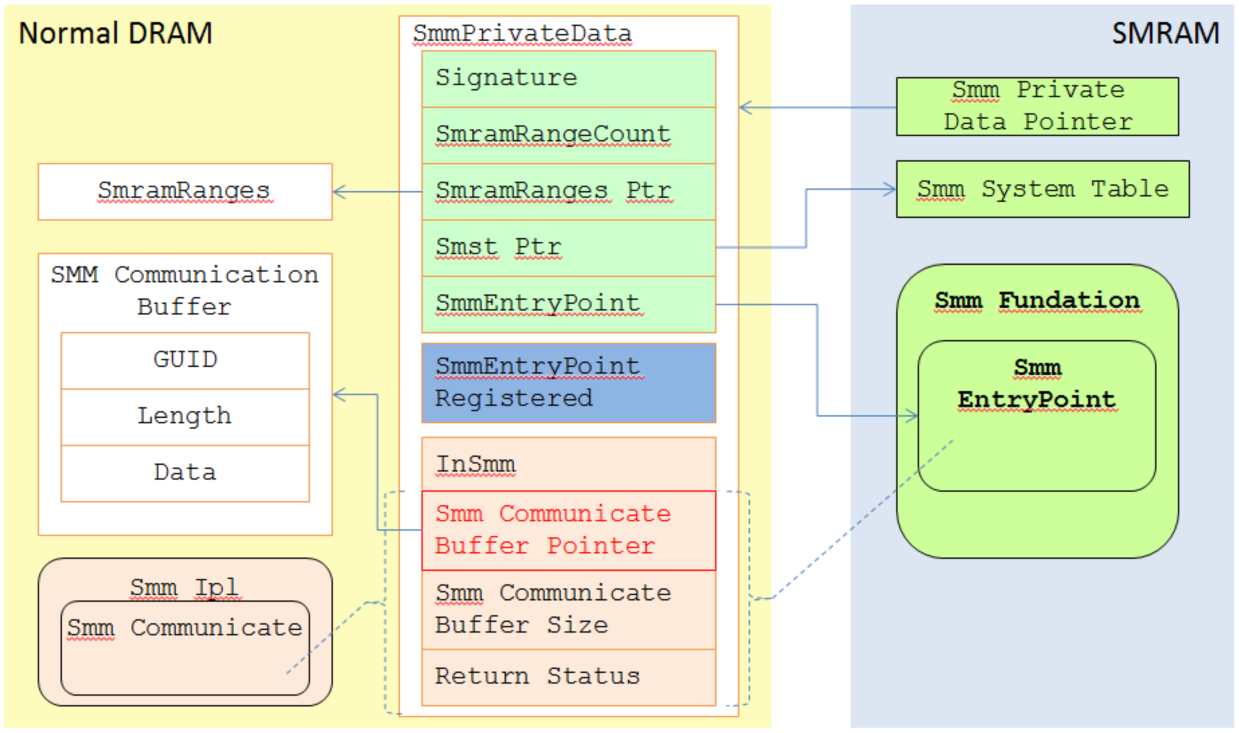

Figure 5 SMM Communication

IA32->X64 Mode Transition

In order to support loading an SMM driver in the DXE phase for compatibility considerations, or access above 4G resources, the SMM Foundation must be in X64 mode. However, most PEI phase is still 32bit, which means SMM IPL is still 32bit.

For this case, the SMM IPL must have thunk support to switch to X64 mode before calling the SMM Foundation entry point, and switch back to IA32 mode after SMM Foundation returns.

Summary

This section introduces the Standalone SMM IPL driver.

Standalone SMM Foundation

SMM HOB

Once SMM Foundation is invoked, the SMM Foundation will parse SMM HOB and find out the location of SMRAM, and then initialize memory services. Then the SMM Foundation will save the SMM HOB to the SMM configuration table. Because a Standalone SMM driver cannot access any UEFI services, the SMM HOB is used to pass information, such as MP_INFORMATION and SMRAM region information. Details can be found in figure 7 below.

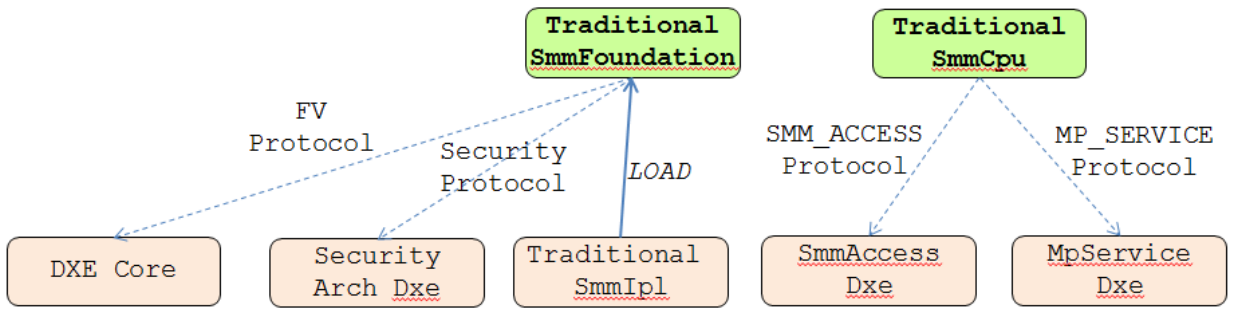

Figure 6 Traditional SMM Model Driver Relation

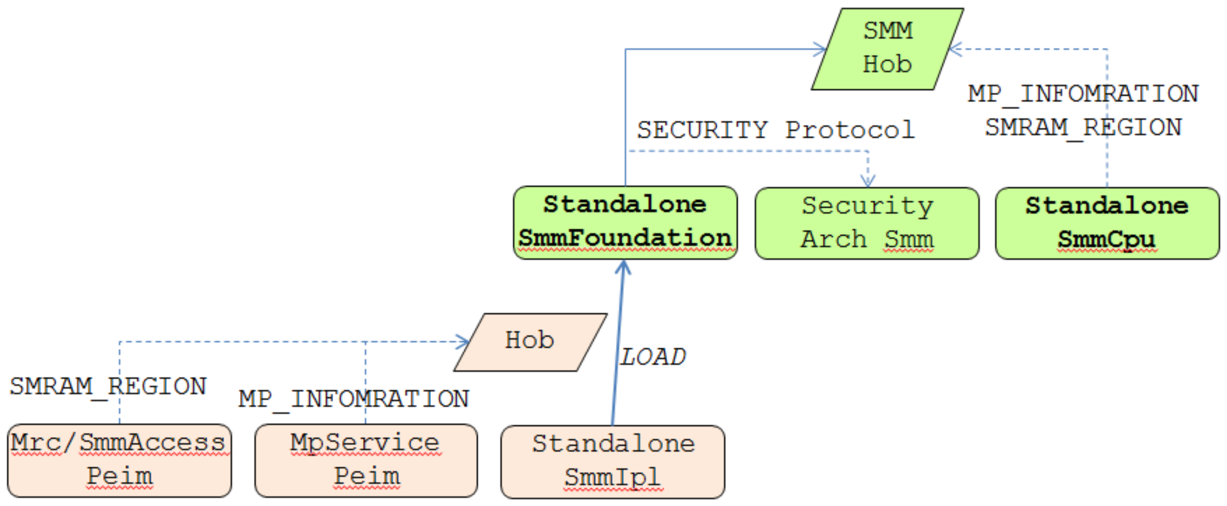

Figure 7 Standalone SMM Mode Driver Relation

Standalone SMM Firmware Volume

In traditional mode, the SMM Foundation will use the **EFI_FIRMWARE_VOLUME2_PROTOCOL** to find firmware volumes, and the**EFI_SECURITY_ARCH2_PROTOCOL** to check driver integrity. More details on this can be found in Figure 6.

In standalone mode, the SMM Foundation cannot use such services. As such, we let the SMM IPL pass the initial Standalone SMM Firmware Volume to SMM Foundation via a HOB so that the SMM Foundation can parse the HOB data to get the Firmware Volume location and thus discover each of the Standalone SMM drivers.

One standalone SMM driver needs to install **`EFI_SECURITY_ARCH2_PROTOCOL`** to the SMM database so that the standalone SMM Foundation can use this protocol to check driver integrity. See figure 7 for more details.

There might be one standalone SMM driver that installs the**`SMM_FIRMWARE_VOLUME2_PROTOCOL`** protocol to SMM protocol database later. After this, the SMM Foundation can use the SMM protocol to discover additional firmware volumes.

There might be one Standalone SMM driver that invokes the *`ProcessFirmwareVolume`*() API provided by the SMM Foundation to process more firmware volumes directly.

Dispatcher Standalone SMM driver

In traditional mode, the SMM Foundation runs SMM drivers with the entry point **`EFI_IMAGE_ENTRY_POINT`**.

In standalone mode, the SMM Foundation cannot use this methodology because the EFI System Table is not available. Instead, the Standalone SMM driver can have the below entry point definition:

```
typedef
EFI_STATUS
(EFIAPI *SMM_IMAGE_ENTRY_POINT) (
  IN EFI_HANDLE            ImageHandle,
  IN EFI_SMM_SYSTEM_TABLE2 *SmmSystemTable
  );
```

The SmmSystemTable is a pointer to services provided by the SMM Foundation. It is always available.

Compatibility Support

In order to maximum reuse of existing SMM drivers, the Standalone SMM Foundation also needs to support the current SMM driver defined in the PI SMM specification, namely those drivers whose entry point is **`EFI_IMAGE_ENTRY_POINT`**.

The first question is: How does the SMM Foundation know the EFI System table?

To answer this question, we create a DXE phase SMM IPL stub module which passes the EFI System Table pointer and communicates to the SMM Foundation. Then SMM Foundation can save EFI System Table pointer to SMRAM.

The second question is: How does the SMM Foundation know if a SMM driver is a standalone SMM driver or a traditional SMM model driver?

The easiest way to answer this question is to assume that the SMM driver loaded in the PEI phase is always a standalone mode driver, and to assume that SMM drivers loaded in the DXE phase are always traditional mode.

The other possible way is to extend PI1.4 vol 3, to add **EFI_FV_FILETYPE_SMM_STANDALONE** to explicitly indicate that the FFS file contains a Standalone SMM driver.

Summary

This section introduces the Standalone SMM Foundation, including how to support both the standalone SMM driver and the traditional SMM driver model.

Standalone SMM CPU Driver

The SMM CPU driver is the most important SMM driver because it will rebase SMRAM and enable SMI activations.

Getting multiprocessor information

The traditional SMM model CPU driver will call **`MP_SERVICES_PROTOCOL`** to collect multiprocessor information, including CPU number and APIC ID. (See figure 6)

The standalone SMM CPU driver cannot use this DXE protocol, so it will use MP information in SMM HOB. Our MpService PEIM not only produces the **`PEI_MP_SERVICE_PPI`**, but also creates the **`MP_INFORMATION_HOB`**. See figure 7 for more details.

Getting SMRAM region

The traditional SMM CPU driver model will call the **`SMM_ACCESS_PROTOCOL`** to collect SMRAM region information. See figure 6 for more details.

The standalone SMM CPU driver cannot use the protocol, so it will use the SMRAM region information in the SMM HOB. A silicon memory initialization driver or an SMM access PEIM may produce this information in the HOB. See figure 7 for more details.

SMRAM Rebase

The SMM CPU driver will calculate SMBASE according to the multiprocessor information and SMRAM region for each processor. Finally, the SMM CPU driver will rebase SMRAM, which is same between a traditional SMM driver model and a standalone SMM driver.

SMM_CORE_ENTRY

Last but not least in terms of importance, the SMM CPU driver will install the **`SMM_CONFIGURATION_PROTOCOL`** to the SMM database so that the SMM Foundation will register the SMM entry point.

SMI Flow

After the SMM CPU driver is loaded, SMI's are enabled. When the SMI is triggered, the processor invokes the 16-bit SMI entry in the SMM CPU Driver, and then the CPU will switch to IA32/X64 mode. The BSP/APs will rendezvous to make sure all processors are running at the same point. The BSP will call the SmmEntrypoint address registered by the Standalone SMM Foundation. After returning, all processors will rendezvous again and execute the RSM instruction to return to the normal CPU operation mode. See figure 8 for more details.

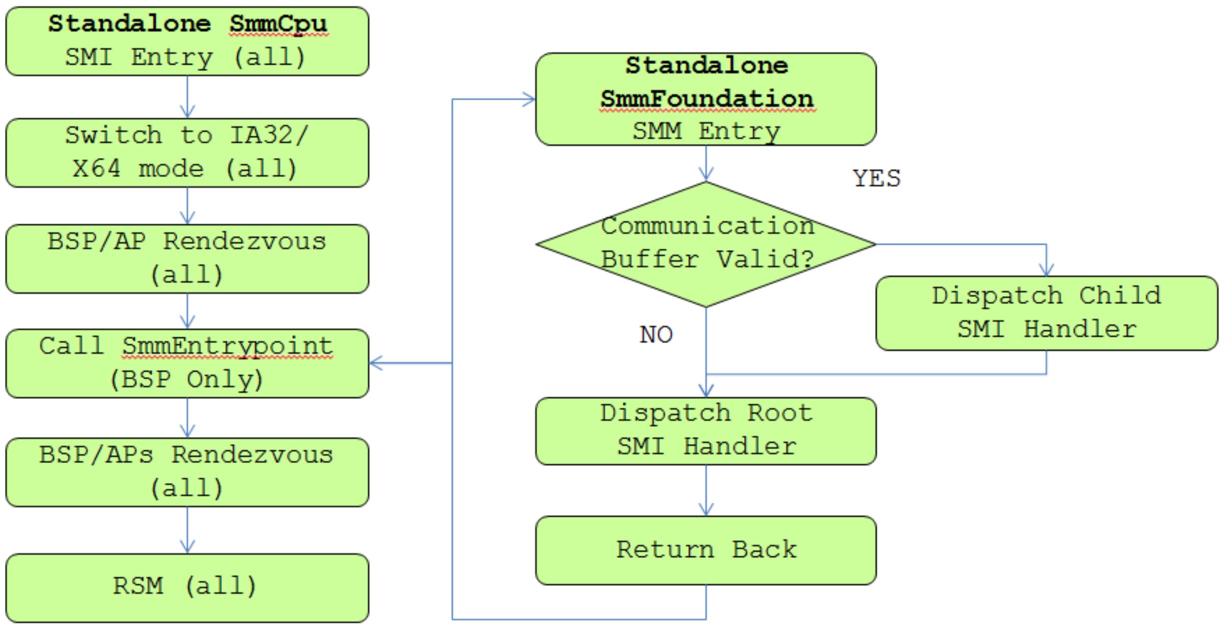
Standalone SmmCpu
SMI Entry (all)
Switch to IA32/ X64 mode (all)
BSP/AP Rendezvous (all)
Call SmmEntrypoint (BSP Only)
BSP/APs Rendezvous (all)
RSM (all)
Standalone SmmFoundation SMM Entry
YES
Communication Buffer Valid?
Dispatch Child SMI Handler
NO
Dispatch Root SMI Handler
Return Back

Figure 8 Standalone SMM CPU Flow

Summary

This section introduces the Standalone SMM CPU driver.

Standalone SMM Drivers

Entry Point

A standalone SMM driver is different from the traditional SMM driver model. It has the below-listed entry point:

```
typedef
EFI_STATUS
(EFIAPI *SMM_IMAGE_ENTRY_POINT) (
  IN EFI_HANDLE            ImageHandle,
  IN EFI_SMM_SYSTEM_TABLE2 *SmmSystemTable
  );
```

Because the SMM driver does not have a pointer to the EfiSystemTable, it cannot access any UEFI services.

The SMM driver can access below:

1) SMM Protocol database – a database maintained by SMM Core.
2) SMM HOB – the Hand-off-Block from SMM IPL to SMM Core.

SMI Handler

The SMI handler in SMM standalone mode is same as the SMI handler in traditional mode.

Porting Traditional SMM driver to Standalone SMM driver

If a solution needs to run a SMM driver in PEI or SEC phase, we need to port this SMM driver to a standalone SMM driver. We can use the 2 steps below:

1) Make sure it does not access any UEFI services. If this SMM driver needs to access UEFI protocols, we need to use an alternative solution, for example, SMM HOB or SMM Protocol database. A real example is the SMM CPU driver introduced in previous section. It needs the MP_SERVICE_PROTOCOL previously, but now it can use the MP_INFORMATION_HOB.

2) Link other library instances in platform DSC file.

Library Class	**Library Instance**
UefiDriverEntryPoint	StandaloneSmmDriverEntryPoint
HobLib	StandaloneSmmHobLib
SmmServicesTableLib	StandaloneSmmServicesTableLib
MemoryAllocationLib	StandaloneSmmMemoryAllocationLib
SmmMemLib	StandaloneSmmMemLib

- StandaloneSmmDriverEntryPoint library – ModuleEntryPoint() calls ProcessLibraryConstructorList() and ProcessModuleEntryPointList(). Finally calls ProcessLibraryDestructorList() if there is an error.

- StandaloneSmmHobLib – Constructor gets SmmHob pointer from SMM Configuration Table.
- SmmServicesTableLib – Constructor gets SmmServicesTable pointer from 2nd parameter SystemTable.
- StandaloneSmmMemoryAllocationLib – FreePool() always uses gSmst to free memory in SMRAM.
- StandaloneSmmMemLib – Constructor gets SMRAM region from SMM Hob.

Then this SMM driver becomes a standalone SMM driver.

Summary
This section introduces the Standalone SMM driver, and how to port SMM driver from tradition mode to standalone mode.

Conclusion

SMM Standalone mode provides a firmware capability to run SMM driver in any phase, even PEI/SEC, as long as SMRAM is ready. The early launch capability might be needed by Intel FSP [FSP-EAS] and server RAS [APEI] features. A Standalone SMM Foundation can support the ability to load both the SMM standalone drivers and traditional SMM drivers.

Glossary

FV – Firmware Volume, a logical firmware device. See [UEFI PI Specification].

IPL – Initial program loader.

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

SMI – System Management Interrupt. The interrupt to trigger processor into SMM mode.

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

SMRAM – System Management RAM. The memory reserved for SMM mode.

UEFI – Unified Extensible Firmware Interface.   Firmware interface between the platform and the operating system.

References

[ACPI] Advanced Configuration and Power Interface, vesion 6.0, www.uefi.org

[APEI] Sakthikumar, Zimmer, "A Tour Beyond BIOS Implementing the ACPI Platform Error Interface with the Unified Extensible Firmware Interface," January 2013, https://firmware.intel.com/sites/default/files/resources/A_Tour_beyond_BIOS_Implementing_APEI_with_UEFI_White_Paper.pdf

[EDK2] UEFI Developer Kit www.tianocore.org

[EDKII specification] A set of specification describe EDKII DEC/INF/DSC/FDF file format, as well as EDKII BUILD. http://tianocore.sourceforge.net/wiki/EDK_II_Specifications

[EMBED] Sun, Jones, Reinauer, Zimmer, "Embedded Firmware Solutions: Development Best Practices for the Internet of Things," Apress, January 2015, ISBN 978-1-4842-0071-1

[FSP-EAS] Intel (R) Firmware Support Package (FSP) External Architecture Specification, Revision 1.1, April 2015 http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/fsp-architecture-spec-v1-1.pdf

[IA32 Manual] Intel® 64 and IA-32 Architectures Software Developer Manuals http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.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 with Software and Services Group (SSG) at Intel Corporation.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer with the Software and Services Group (SSG) at Intel Corporation. Vincent invented the original SMM infrastructure http://www.google.com/patents/US6848046 and wrote the Framework Interface Specification for SMM http://www.intel.com/content/www/us/en/processors/itanium/efi-smm-cis-v09.html in addition to working with the industry on UEFI PI SMM evolution for PI1.0 to the recently published PI1.4 specification at http://www.uefi.org.