White Paper

A Tour Beyond BIOS Implementing Profiling in with EDK II

Jiewen Yao, Intel Corporation

Vincent J. Zimmer, Intel Corporation

Star Zeng, Intel Corporation

Fan Jeff, Intel Corporation

July 2016

Executive Summary

The Unified Extensible Firmware Interface (UEFI) and Platform Initialization (PI) specification defines rich execution environments such as Security (SEC), Pre-EFI Initialization (PEI), Driver Execution Environment (DXE), System Management Mode (SMM) and UEFI Runtime (RT) for firmware drivers. There are more and more features added into a firmware. At same time, the firmware still has a resource constrained environment. In addition to functionality, the size, performance, and security are three major concerns of a firmware engineer. This paper introduces several profiling features implemented in EDK II to help the UEFI firmware developer to analyze the size, performance and security of a UEFI firmware implementation.

Prerequisite

This paper assumes that audience has EDK II/UEFI firmware development experience. He or she should also be familiar with UEFI/PI firmware infrastructure, such as SEC, PEI, DXE, runtime phase.

Table of Contents

Table of Figures

Overview

Introduction to profiling features in EDK II

The UEFI specification defines a standard environment for booting an operating system (OS). The [UEFI] PI specification defines a standard environment for firmware building blocks to support constructing the UEFI environment. According to the PI specification, a typical firmware boot involves the SEC, PEI, DXE, and RT phases. The UEFI environment is created in the DXE phase.

EDK II is an open source implementation for UEFI PI firmware. EDK II includes many generic components, such as the PEI Core, DXE Core, SMM Core, Runtime Drivers, `PiSmmCpu`, PCI Bus, USB Bus, ATA Bus, etc. In addition, some silicon modules and platform modules are needed in order to build a full featured UEFI PI firmware for a given system board.

Besides the functionality in a given firmware implementation, the size, performance and security are three major additional concerns. In order to help a firmware developer perform an analysis of these metrics, EDK II provides a set of profiling features.

- **Memory Profile**: This profile feature can show the memory usage in each module during a firmware boot. As a subset feature of the memory profile capability, **memory leak detection** can help detect the instance of a memory leak in a UEFI driver or application, such as the UEFI Shell.
- **Performance Profile**: This profile feature can show the performance data for each module during a firmware boot. This performance profile can be used to construct the ACPI firmware performance data table (FPDT) for an ACPI OS.
- **SMM Profile**: This profile feature can record any SMM-based data access and code access to regions outside of SMRAM. This is important to help the firmware developer check if the data access originating from code within SMM can meet WSMT requirement and also check if there is an SMM code access to a region outside of SMRAM. Any violation is a critical firmware security issue.

NOTE: *Most profiling features are designed as debug features. It is improper to include all these profiling features in a final production firmware. Some profiling features, such as* `PiSmmCore` *database dump, may cause the SMM information leak. They are only for the debug purposes.*

Summary

This section provided an overview of the profile features in EDK II. In next several chapters, we will introduce each profiling features in detail.

Part I – Memory Profile

Problem statement

The UEFI specification defines a set of UEFI Runtime Services (RT), such as `GetTime/SetTime`, `GetVariable/SetVariable`. In order to support these RT services, a UEFI firmware implementation must allocate the `EfiRuntimeServicesCode` memory for the driver and `EfiRuntimeServicesData` memory for the associated data. A UEFI OS must allocate the virtual address for all `EfiRuntimeServicesCode` or `EfiRuntimeServicesData` memory regions, and the UEFI OS may not touch these memory regions.

There is a similar situation for `EfiACPIMemoryNVS` and `EfiReservedMemoryType` regions. A UEFI firmware implementation may allocate `EfiACPIMemoryNVS` to store an ACPI table, and it may allocate `EfiReservedMemoryType` to store its own private data, such as a S3 Boot Script. Once the UEFI firmware allocates these memory types, the OS may not use them. All those regions (RT code, RT data, ACPI NVS, reserved) are considered as hardware reserved memory type.

What this practically means is that the more hardware reserved memory a firmware allocates during the boot, the less usable memory an OS can use during the OS runtime.

In order to balance the memory usage between the Firmware/BIOS and the OS, some OS vendors like Microsoft* have a requirement for hardware memory reservation. See [*WHCK Requirement*], *System.Fundamentals.Firmware.HardwareMemoryReservation.*

- <=2GB systems – max of 3% of 2GB (61.5MB)
- 2-3GB systems – max of 3% of 3GB (92.2MB)
- For all other systems, a max of 120MB

If this requirement is broken, a developer may want a tool to check every hardware reserved memory allocation to see if it can be reduced. How can we achieve that?

Memory profile in DXE phase

The basic idea to record the memory allocation information is simple – Add a hook in `gBS->AllocatePage/AllocatePool` function and record this allocation.

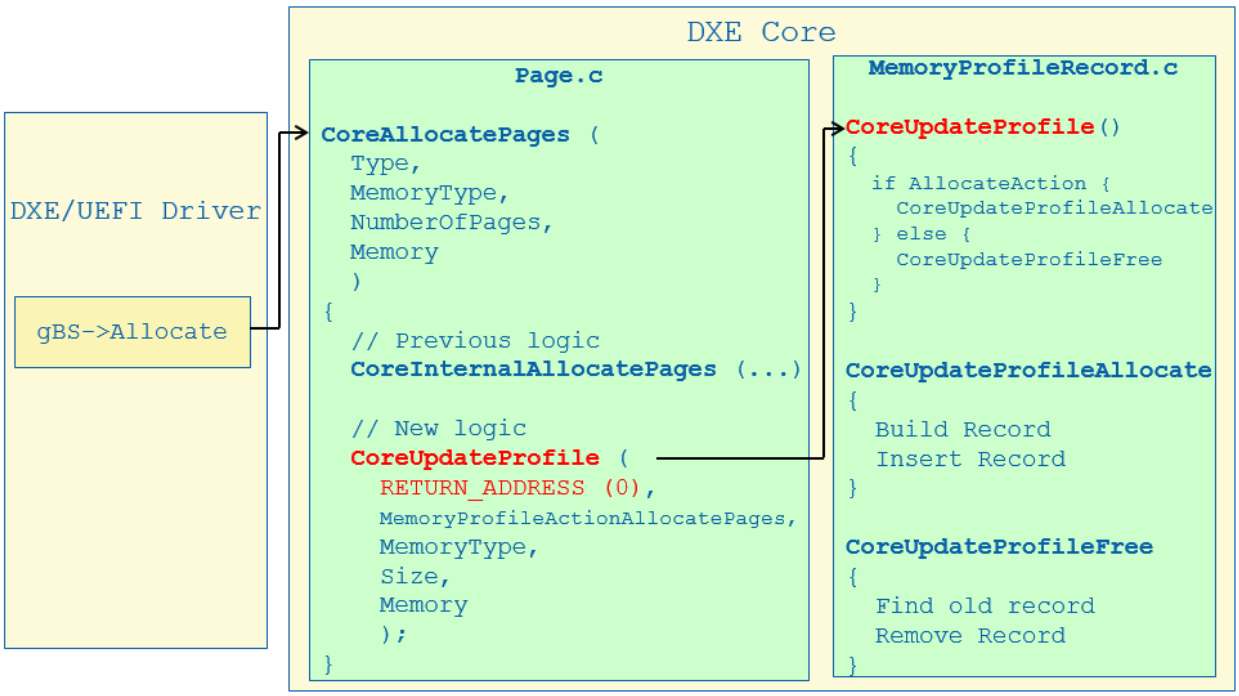

Figure 1 – memory profile data collection

This work is done in the `DxeCore`, so the nature place to add this profile feature is the `DxeCore`. See figure 1.

When a DXE/UEFI driver calls `gBS->AllocatePages`, it invokes the `DxeCore CoreAllocatePages()`. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/Page.c). This function calls `CoreInternalAllocatePages()` for the memory allocation. If the memory is allocated successfully, `CoreUpdateProfile()` is called to record this allocation action. Then in `CoreUpdateProfileAllocate()` function, a profile record is built and is inserted to the profile database.
(https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/MemoryProfileRecord.c)

The free action is similar. When a DXE/EFI driver calls `gBS->FreeFages`, it invokes the service `DxeCore CoreFreePages()`. This function calls `CoreInternalFreePages()`. Then `CoreUpdateProfileFree()` function is called to find the corresponding profile record and remove it from the profile database.

A special case for `FreePages()` involves this Free Action potentially only freeing partial regions of previous allocated pages. If so, when the old record is removed, `CoreUpdateProfileFree()` must insert a new record for the remaining pages.

So what does the profile record look like?

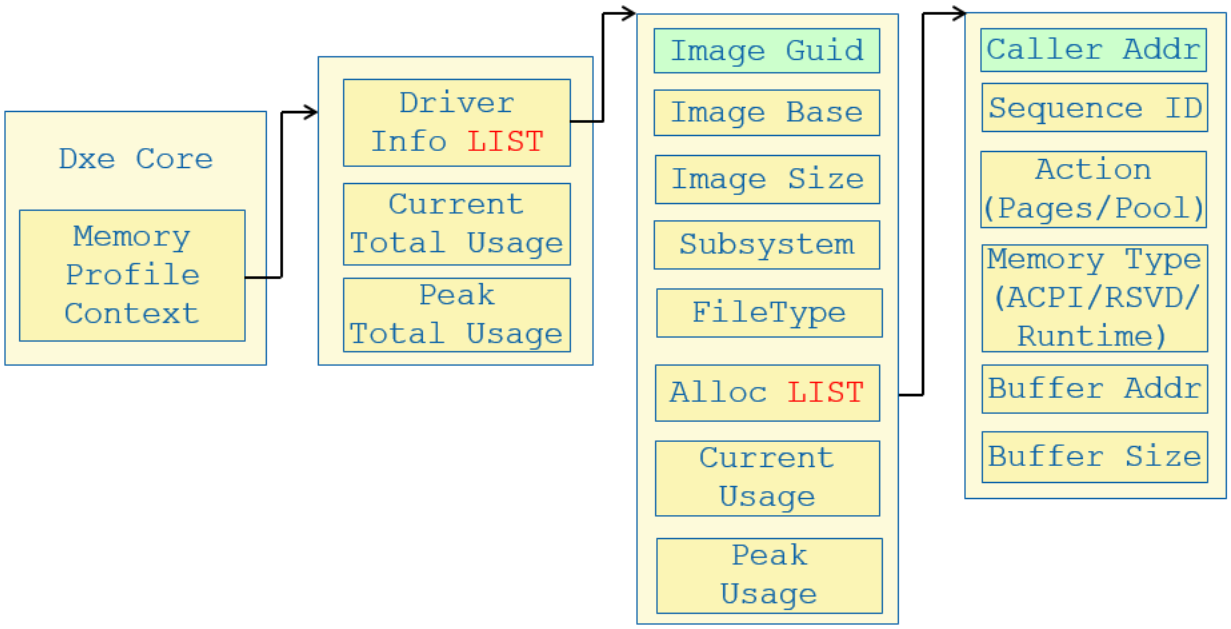

Figure 2: memory profile data structure

See figure 2. The memory profile record is maintained by the DxeCore.

The DxeCore allocates memory for the profile context - MEMORY_PROFILE_CONTEXT_DATA (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/MemoryProfileRecord.c). It includes a MEMORY_PROFILE_CONTEXT (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/MemoryProfile.h) and a driver information list. The driver information list includes a MEMORY_PROFILE_DRIVER_INFO (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/MemoryProfile.h) and an allocation information list – a list of MEMORY_PROFILE_ALLOC_INFO (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/MemoryProfile.h). The MEMORY_PROFILE_DRIVER_INFO records the driver information, such as image GUID, image base/size, file type, current memory usage and peak memory usage. The MEMORY_PROFILE_ALLOC_INFO records each individual allocation action, such as allocation action, memory type, and buffer address/size.

By using this structure, a user can clearly discover how the memory is allocated by a specific image. The final buffer allocation result is reflected by the current usage. Sometimes, the peak usage is also important to let people have a clear picture regarding how much memory was allocated in the history of the platform bootstrap.

Now, how does DxeCore record a list of images?

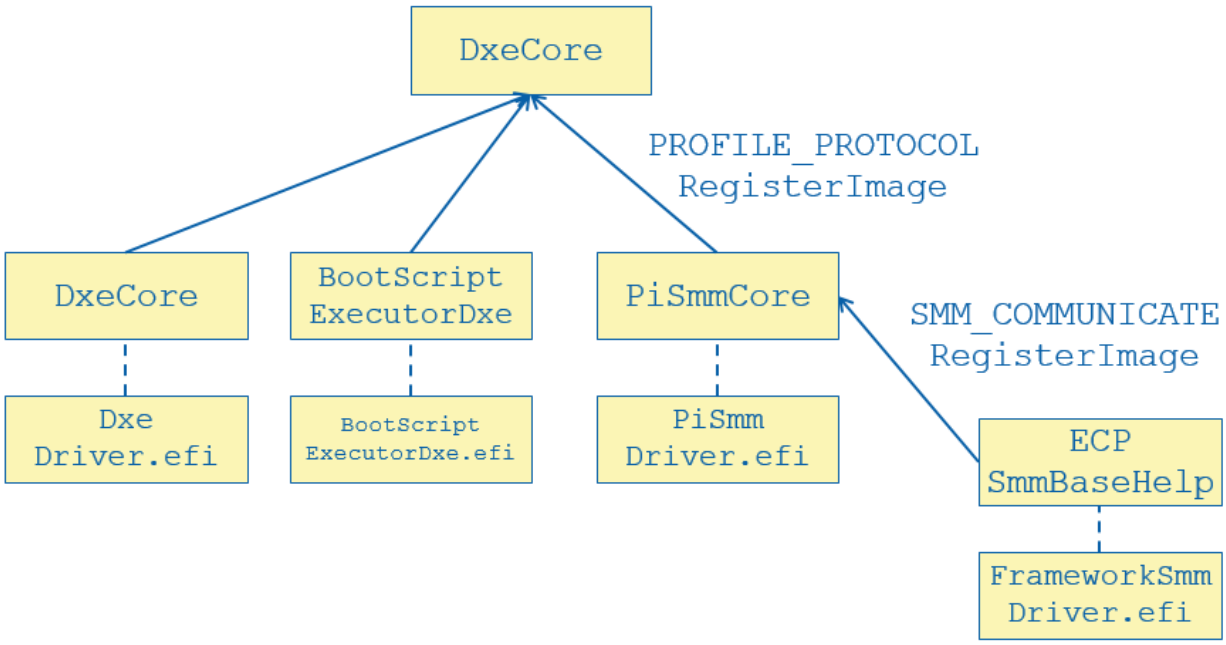

Figure 3: memory profile image registration

See figure 3. In most cases, the `DxeCore` calls `RegisterMemoryProfileImage()` on `StartImage()`.
(https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Image/Image.c).
`RegisterMemoryProfileImage()` builds a record for an image and inserts it to profile database
(https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/MemoryProfileRecord.c)

When an image is unloaded, the `DxeCore` calls `UnregisterMemoryProfileImage()`. This function finds the corresponding record for an image. If this driver does not have any allocation records, this driver record is removed. However, if this driver has some allocation records, this record is kept and only image base/size fields are cleared to indicate that this driver is unloaded. By using this mechanism, the allocation records are kept so that a user can know if there is potential memory leak in this driver.

Besides an image loaded by the `DxeCore`, there are some other ways to load an image. For example, `BootScriptExecutorDxe` needs to reload itself into an ACPI Reserved memory region. If so, how can the `DxeCore` record the reloaded image information?

The `EDKII_MEMORY_PROFILE_PROTOCOL` is used to resolve this problem.
(https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/MemoryProfile.h) The `BootScriptExecutorDxe` should call `EDKII_MEMORY_PROFILE_PROTOCOL.RegisterImage()` to register the reloaded image.
(https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/Acpi/BootScriptExecutorDxe/ScriptExecute.c).

The UEFI memory allocation is not limited to DXE/UEFI drivers. Some SMM drivers may also call the UEFI memory allocation services to allocate the normal memory, instead of SMRAM. In such cases, the memory profile record should also include all SMM drivers.

This PI SMM driver registration is done by `PiSmmCore RegisterImageToDxe()` function. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/PiSmmCore/SmramProfileRecord.c). It calls `EDKII_MEMORY_PROFILE_PROTOCOL.RegisterImage()` to register the SMM image to the DXE profile database.

The ECP Framework SMM driver registration is a little complicated. Because an ECP Framework SMM driver does not have PI SMM knowledge, it is loaded by the DXE Core and calls `EFI_SMM_BASE_PROTOCOL.Register`. The ECP SmmBaseOnSmmBase2Thunk `SmmBaseRegister()` function is called and it triggers SWSMI for image registration (https://github.com/tianocore/edk2/tree/master/EdkCompatibilityPkg/Compatibility/SmmBaseOnSmmBase2Thunk). Then ECP `SmmBaseHelper LoadImage()` function is called to load the PE image into SMRAM and execute. (https://github.com/tianocore/edk2/tree/master/EdkCompatibilityPkg/Compatibility/SmmBaseHelper). `LoadImage()` function should call `RegisterSmramProfileImage()` to communicate with `PiSmmCore` to register the SMM image to SMM profile database. Then `PiSmmCore` calls `EDKII_MEMORY_PROFILE_PROTOCOL.RegisterImage()` to register the same SMM image to the DXE profile database.

As a last step, how can a user view the memory profile record?

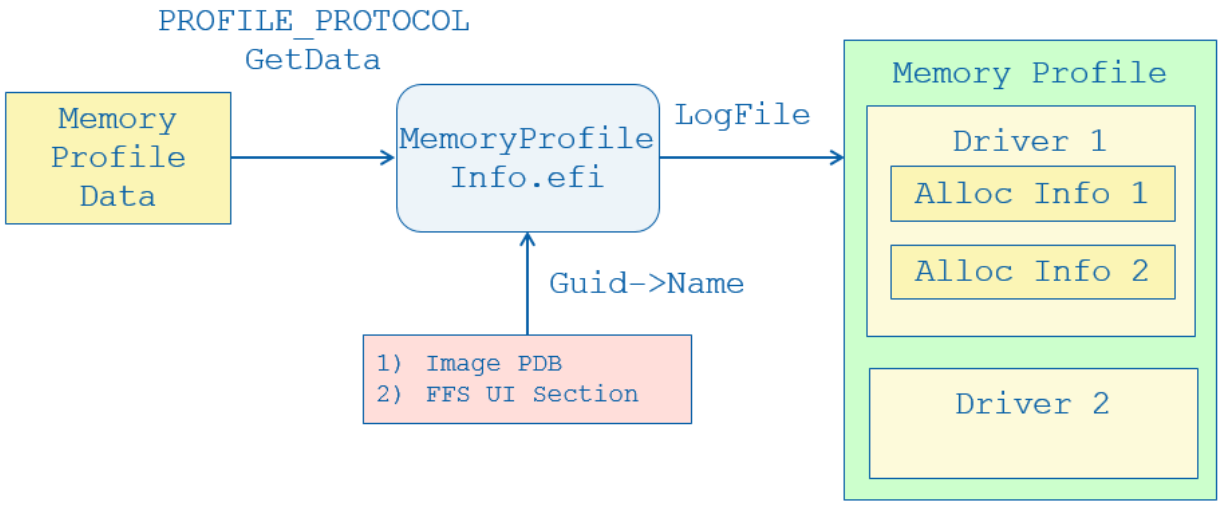

Figure 4: memory profile report

See figure 4. A user may use UEFI SHELL application `MemoryProfileInfo.efi` (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Application/MemoryProfileInfo) to dump the memory profile database.

In order to be user friendly, this application tries to convert the image GUID name to be a user readable name. First, it tries to get the PE COFF PDB information in the record. If this fails, it tries to find a FFS UI section corresponding to the GUID.

Memory profile in SMM phase

The memory profile in the SMM phase is similar to the one in the DXE phase. Here we name the SMM memory profile to be the SMRAM profile to avoid confusion.

If an SMM driver allocates a UEFI memory region (by using `gBS->AllocatePages`), this allocation is still recorded in the DXE memory profile database. If an SMM driver allocates a SMRAM memory region (by using `gSmst->SmmAllocatePages`), this allocation is recorded in the SMRAM profile database.

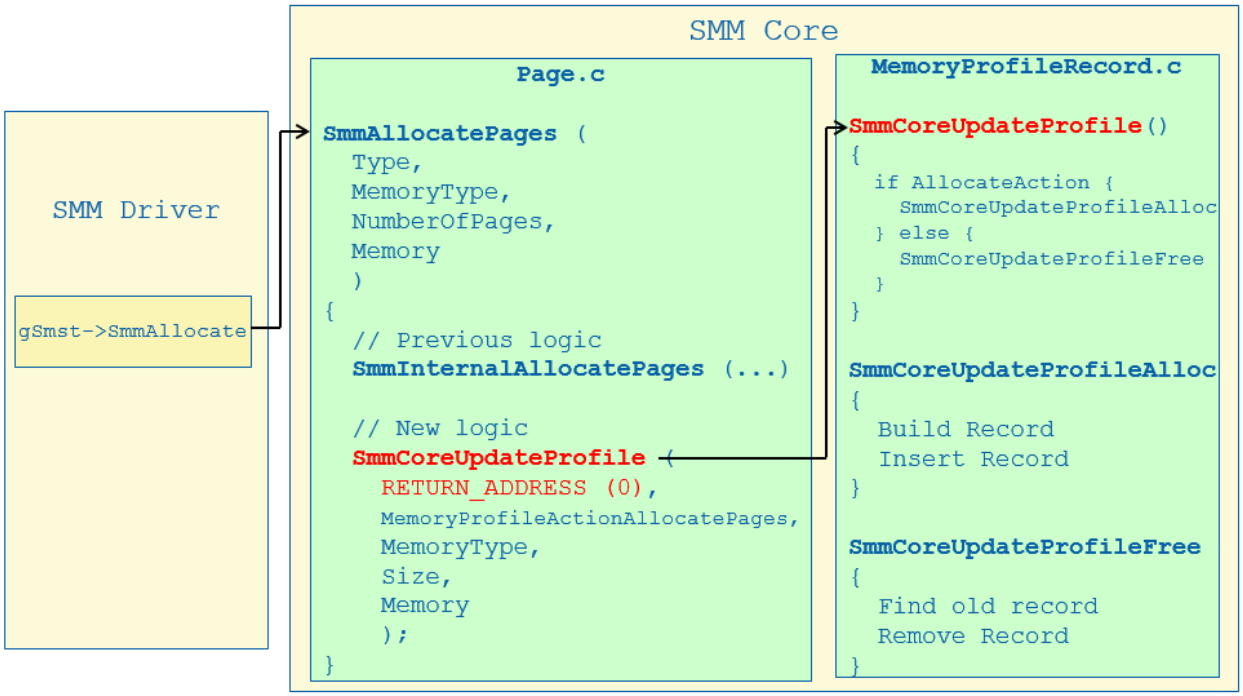

Figure 5: SMRAM profile data collection

See figure 5.

When an SMM driver calls `gSmst->SmmAllocatePages`, it invokes the `SmmCore SmmCoreAllocatePages()`. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/PiSmmCore/Page.c). This function calls `SmmInternalAllocatePages()` for the memory allocation. If the memory is allocated successfully, `SmmCoreUpdateProfile()` is called to record this allocation action. Then in `SmmCoreUpdateProfileAllocate()` function, a profile record is built and inserted into the SMRAM profile database. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/PiSmmCore/SmramProfileRecord.c)

The Free Action is similar. When an SMM driver calls `gSmst->SmmFreeFages`, it invokes `SmmCore SmmFreePages()`. This function calls `SmmInternalFreePages()`. Then `SmmCoreUpdateProfileFree()` function is called to find the corresponding profile record and remove it from the SMRAM profile database.

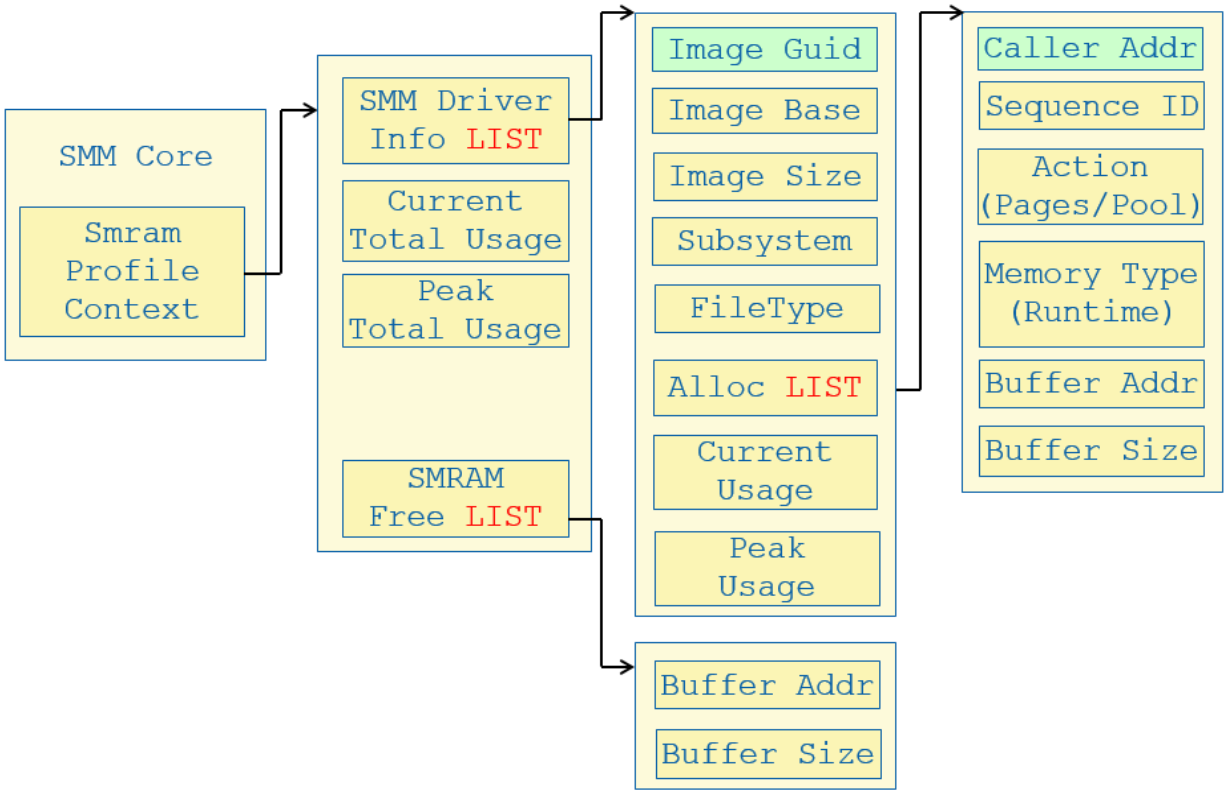

Figure 6: SMRAM profile data structure

See figure 6. The SMRAM profile record in `SmmCore` is similar to the memory profile record in `DxeCore`.

The only difference is that the SMRAM profile record also includes a SMRAM free list. As such, this information can be dumped later to show how much SMRAM is available.

The image registration in the `SmmCore` is similar to the one in the `DxeCore` as well. Because `PiSmmCore` discovers and loads all PI SMM drivers, it calls `RegisterSmramProfileImage()` on `StartImage()`. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/PiSmmCore/Dispatcher.c). `RegisterSmramProfileImage()` builds a record for an image and inserts it into profile database (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/PiSmmCore/SmramProfileRecord.c)

We also discussed the ECP Framework SMM image registration before. See figure 3.

NOTE: *The ECP support is deprecated and we strongly encourage a platform to convert all Framework SMM drivers to PI SMM drivers in order to follow the PI specification.*

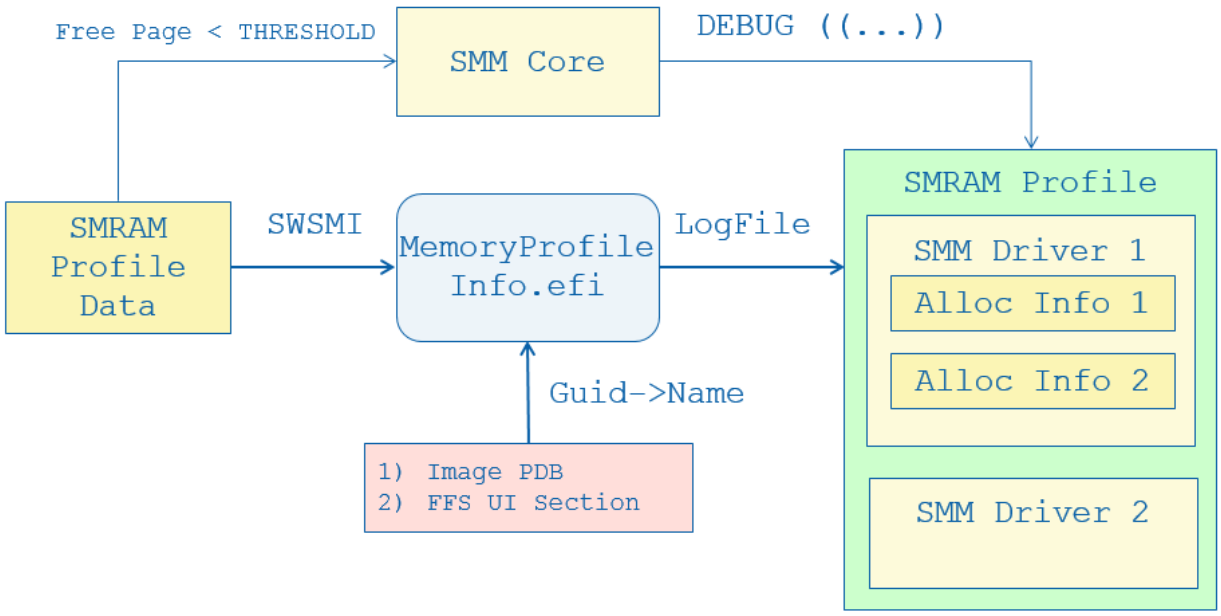

Figure 7: SMRAM profile report

Finally, a user may use the UEFI SHELL application `MemoryProfileInfo.efi` (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Application/MemoryProfileInfo) to dump the SMRAM profile database as well.

One additional feature in the SMRAM profile capability is that if the free pages in SMRAM are less than a predefined THRESHOLD (4 pages), then the `PiSmmCore` may dump the full SMRAM profile record as a debug message in order to notify the developer of an issue.

Memory profile in PEI phase

Memory profiling in the PEI phase is unsupported at this moment.
There are several reasons:

1) Most PEI memory allocations are `EfiBootServicesData / EfiBootServicesCode`. These are not hardware reserved memory.
2) We observe very few cases wherein a silicon or platform PEIM may allocate `EfiACPIMemoryNVS` and `EfiReservedMemoryType`. These allocations can easily be identified by use of the UEFI SHELL MEMMAP command.

As such, we chose to not enable the memory profile in the PEI phase at this moment. If we can see the value later, it can be enabled in the future.

How to enable

1) A user may set `gEfiMdeModulePkgTokenSpaceGuid.PcdMemoryProfilePropertyMask|0x3` to enable memory profiling for UEFI memory or SMRAM. BIT0 means to enable UEFI memory profiling. BIT1 means to enable SMRAM profiling.

2) A user may set `gEfiMdeModulePkgTokenSpaceGuid.PcdMemoryProfileMemoryType|0x661` to enable memory profile for `Reserved+ACPINvs+ACPIReclaim+RuntimeCode+RuntimeData`. The bit mask for this PCD uses same order as found in the UEFI specification.
3) After the system boots to UEFI shell, a user may use the UEFI shell tool `MemoryProfileInfo.efi` to dump the profile information.

NOTE: A user may use UEFI SHELL `MEMMAP` command to dump all of the UEFI memory map information. There might be cases that an `EfiACPIMemoryNVS` or `EfiReservedMemoryType` allocation is shown by MEMMAP, but it is not recorded in memory profile. If so, these memory regions might be:
1) An `EfiACPIMemoryNVS` or `EfiReservedMemoryType` memory allocated in PEI Phase.
2) An `EfiGcdMemoryTypeReserved` memory allocated by GCD services.

The PEI phase memory can be shown by the `HOB.EFI` tool (https://github.com/jyao1/EdkiiShellTool/tree/master/EdkiiShellToolPkg/HobList), and the GCD memory can be shown by the `GCD.EFI` tool (https://github.com/jyao1/EdkiiShellTool/tree/master/EdkiiShellToolPkg/Gcd).

Summary

This chapter introduced the memory profile feature to help a developer analyze the hardware memory reservation in a UEFI firmware implementation.

Part II – Memory Leak Detection

Problem statement

Memory leak detection is an important topic for any computer programming exercise. Some languages introduce a garbage collection algorithm to free memory automatically when there is no longer any reference. Some system standard libraries record the memory allocated by the application and free these resources automatically when the application is unloaded.

Unfortunately, the C language does not provide any garbage collection, and the UEFI or EDK II infrastructure does not yet define a standard way to free resource automatically. (It will be discussed later in chapter V) In the current EDK II implementation, we have to rely on each driver or application to free resources by itself to prevent memory leaks in the BIOS boot phase.

In a UEFI BIOS, a UEFI Shell might be provided to support invocation of some analysis tools and manufacturing tools before a machine is shipped. In this environment, tests may run for a long time and entail many operations. If there is some memory leak in the UEFI shell or UEFI shell application, the leak may become visible to end user for these usages. For example, the UEFI shell may die after 24 hours due to a memory leak & prior to completion of all of the tests.

How can a shell or shell application developer know if there is a memory leak and the location of the errant leaking code?

The memory profile feature can be used for memory leak detection. The feature we discussed before is necessary but not sufficient. We had better include the prior features in order to support memory leak detection.

- A user can know which line of code calls `AllocatePool()` API. (Instead of `gBS->Allocate()`)
- A user can know which line of code calls a specific API. (for example, `StrnCatGrow()` in SHELL)
- A user can know the total memory allocated by a specific line of code. (For comparison later)
- A user can configure to record a single module only. (Reduce number)
- A user can configure when to start recording and stop recording. (Reduce number)
- A user can know the symbol of a RVA (Function, Source, Line)

Memory profile protocol and library

To begin, we need to introduce a way to record which line of code calls `AllocatePage/AllocatePool` API as well as a specific API.

See Figure 8. In the shell, `ProcessCommandLine()` calls `StrnCatGrow()`, `StrnCatGrow()` calls `AllocatePool()`, and `AllocatePool()` calls `gBS->AllocatePool()` finally. We hope to record the `AllocatePool()` called by `StrnCatGrow()`, and the `StrnCatGrow()` invocation called by `ProcessCommandLine()`.

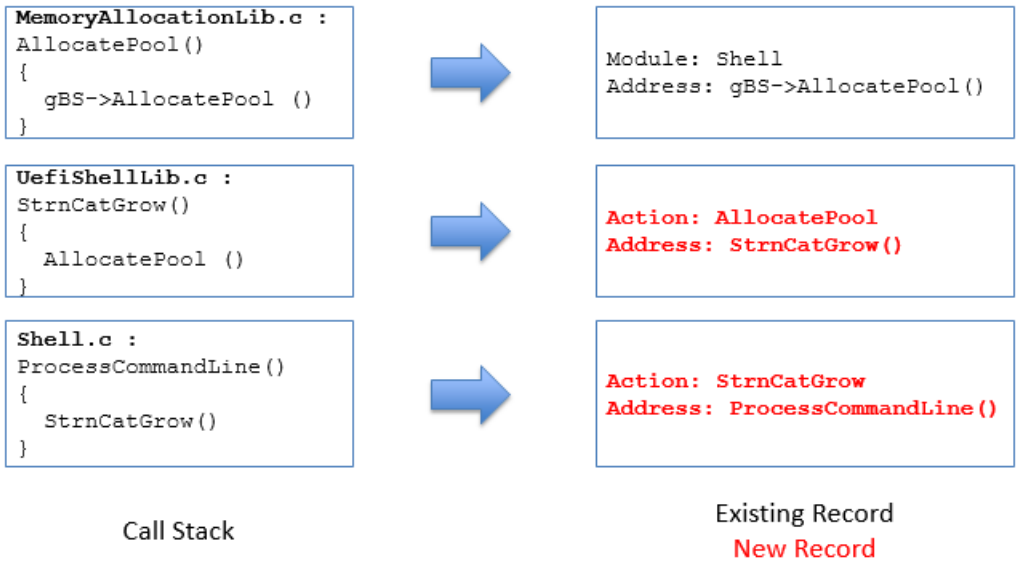

Figure 8: Allocation record

In order to achieve that goal, we need to introduce a way to enable other modules to record the memory profile entry beyond the `DxeCore`. This work is done by `EDKII_MEMORY_PROFILE_PROTOCOL.Record()`. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/MemoryProfile.h) In order to help other modules record the information, a `MemoryProfileLibrary` is introduced (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Library/MemoryProfileLib.h) with 1 API - `MemoryProfileLibRecord()`. By default, the `MemoryAllocationLib` can call this API to record all the APIs provided in the `MemoryAllocationLib`, such as `AllocatePages`, `AllocateRuntimePages`, `AllocateReservedPages`, `AllocatePool`, `AllocateZeroPool`, `ReallocatePool`, `AllocateCopyPool`, etc.

In addition, `PcdMemoryProfileDriverPath` is introduced to indicate which driver needs to be recorded. `PcdMemoryProfileDriverPath` is a device path list. Only if the device path of a driver matches a subset of `PcdMemoryProfileDriverPath` will this allocation action in this driver be recorded. NULL `PcdMemoryProfileDriverPath` means to record all allocations.

Finally, in order to support starting and stopping records, we introduce `EDKII_MEMORY_PROFILE_PROTOCOL.SetRecordingState()/GetRecordingState()`. The allocation action is only recorded after the recording is enabled.

Profile for a specific API

The `MemoryProfileLib` provides the capability to record the allocation action for any specific API. For example, the below patch adds profiling for the `StrnCatGrow()` function in `ShellPkg`.

```
===================================================================
--- Library/UefiShellLib/UefiShellLib.c (revision 19529)
+++ Library/UefiShellLib/UefiShellLib.c (working copy)
@@ -17,6 +17,9 @@
 #include <Library/SortLib.h>
```

```
 #include <Library/BaseLib.h>

+#include <Library/MemoryProfileLib.h>
+#define ModuleProfileActionStrnCatGrow  (MEMORY_PROFILE_ACTION_USER_DEFINED_MASK |
MemoryProfileActionAllocatePool | 0x10)
+
 #define FIND_XXXXX_FILE_BUFFER_SIZE (SIZE_OF_EFI_FILE_INFO + MAX_FILE_NAME_LEN)

 //
@@ -3211,7 +3214,10 @@
 {
   UINTN DestinationStartSize;
   UINTN NewSize;
+  UINTN ReturnAddress;

+  ReturnAddress = (UINTN)RETURN_ADDRESS(0);
+
   //
   // ASSERTs
   //
@@ -3260,11 +3266,17 @@
         NewSize += 2 * Count * sizeof(CHAR16);
       }
       *Destination = ReallocatePool(*CurrentSize, NewSize, *Destination);
+      if (*Destination != NULL) {
+        MemoryProfileLibRecord(ReturnAddress, ModuleProfileActionStrnCatGrow, EfiBootServicesData,
*Destination, NewSize, “StrnCatGrow”);
+      }
       *CurrentSize = NewSize;
     }
   } else {
     NewSize = (Count+1)*sizeof(CHAR16);
     *Destination = AllocateZeroPool(NewSize);
+    if (*Destination != NULL) {
+      MemoryProfileLibRecord(ReturnAddress, ModuleProfileActionStrnCatGrow, EfiBootServicesData,
*Destination, NewSize, “StrnCatGrow”);
+    }
   }

   //
Index: Library/UefiShellLib/UefiShellLib.inf
===================================================================
```

Symbol generation

The `MemoryProfileInfo.efi` utility (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Application/MemoryProfileInfo) dumps the profile data information in the UEFI shell environment. But the profile information only contains the PE/COFF image relative virtual address (RVA). In order to make the tool user friendly, we have to convert the RVA to a user readable symbol.

This work is done in an OS environment. In Linux*, “**nm**” is used to parse the debug information generated by the GCC compiler. In Windows, Microsoft Visual Studio* “**DIA SDK**” is used to parse the PDB information generated by the Microsoft Visual Studio compiler. A python* tool - **MemoryProfileSymbolGen.py** (https://github.com/tianocore/edk2/blob/master/BaseTools/Scripts/MemoryProfileSymbolGen.py) is used to parse the profile log file and add a symbol for each RVA address.

The full working flow is shown in figure 9.

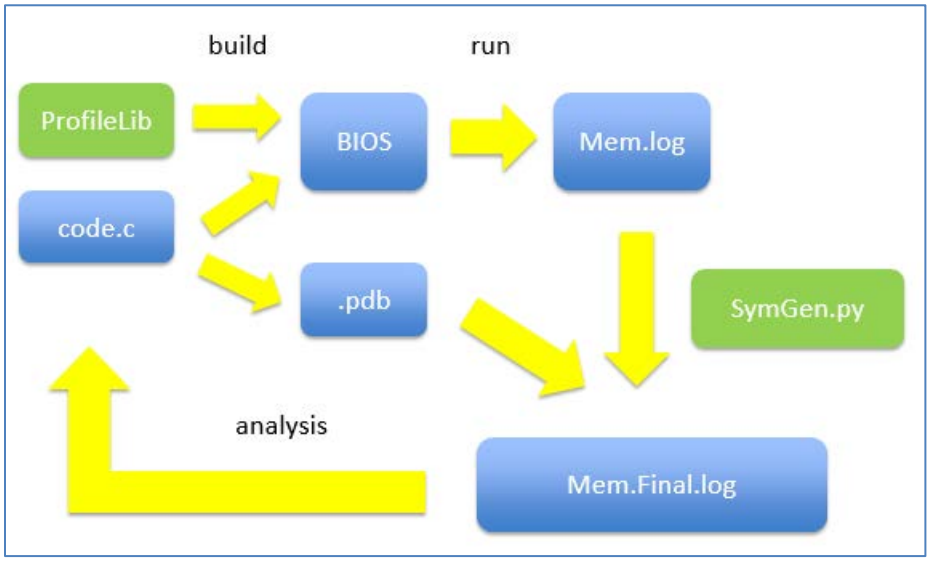

Figure 9: Memory leak detection working flow

How to enable

1) A user may use the earlier-describing mechanism to enable memory profiling. (See Memory Profile chapter). In addition, a user may set BIT7 for `gEfiMdeModulePkgTokenSpaceGuid.PcdMemoryProfilePropertyMask` to disable profiling on starting up.
2) A user may set a valid `gEfiMdeModulePkgTokenSpaceGuid.PcdMemoryProfileDriverPath` to only enable profiling for some specific drivers, such as the SHELL.

The `gEfiMdeModulePkgTokenSpaceGuid.PcdMemoryProfileDriverPath` defines a device path list.

For example:

```
{0x04, 0x06, 0x14, 0x00,  0x83, 0xA5, 0x04, 0x7C, 0x3E, 0x9E, 0x1C, 0x4F,
0xAD, 0x65, 0xE0, 0x52, 0x68, 0xD0, 0xB4, 0xD1, 0x7F, 0xFF, 0x04, 0x00}
```

is one device path for the shell image.

```
{0x04, 0x06, 0x14, 0x00,  0x83, 0xA5, 0x04, 0x7C, 0x3E, 0x9E, 0x1C, 0x4F,
0xAD, 0x65, 0xE0, 0x52, 0x68, 0xD0, 0xB4, 0xD1, 0x7F, 0x01, 0x04, 0x00,
0x04, 0x06, 0x14, 0x00,  0x8B, 0xE1, 0x25, 0x9C, 0xBA, 0x76, 0xDA, 0x43,
0xA1, 0x32, 0xDB, 0xB0, 0x99, 0x7C, 0xEF, 0xEF, 0x7F, 0xFF, 0x04, 0x00}
```

is a device path list for the shell image and `WinNtSimpleFileSystem` driver.

3) If the target module is a UEFI driver, the below library instances should be used in the DSC file:

```
MemoryProfileLib|MdeModulePkg/Library/UefiMemoryAllocationProfileLib/UefiM
emoryAllocationProfileLib.inf
MemoryAllocationLib|MdeModulePkg/Library/UefiMemoryAllocationProfileLib/Ue
fiMemoryAllocationProfileLib.inf
```

If the target module is an SMM driver, the below library instances should be used in the DSC file:

```
MemoryProfileLib|MdeModulePkg/Library/SmmMemoryAllocationProfileLib/SmmMem
oryAllocationProfileLib.inf
```

MemoryAllocationLib|MdeModulePkg/Library/SmmMemoryAllocationProfileLib/SmmMemoryAllocationProfileLib.inf

4) After the system boots to the UEFI shell, a user may use a UEFI shell tool, such as MemoryProfileInfo.efi, to dump the profile information. (First log)
5) In order to check for a memory leak, a user may do some operations in the shell, such as move a file or reconnect a driver. Then the user can dump the profile information again. (Second log)
6) In the OS, the user can run MemoryProfileSymbolGen.py to convert RVA address to symbols, and compare these 2 logs to see if there is any difference. If the second log shows more memory is allocated, that means it is a potential memory leak.

Below is the sample output of the UEFI shell, after we enabled memory leak detection feature.
It tells us: the gBS->AllocatePool() happens 0xE6 times and the total allocated size is 0xB18C. At c:\home\edk-ii\shellpkg\library\uefishellcommandlib\uefishellcommandlib.c:546, the AllocateZeroPool() is called 0x3F times, and the total allocated size is 0x7E0.

```
==========================
Driver - Shell (Usage - 0x0000B18C) (Pdb - c:\home\edk-
ii\Build\NT32IA32\DEBUG_VS2010x86\IA32\ShellPkg\Application\Shell\Shell\DEBUG\Shell.pdb)
Caller List:
  Count            Size                   RVA             Action
==========  ==================     ================== (================================)
0x000000E6  0x000000000000B18C <== 0x000000000001663C (gBS->AllocatePool) (InternalAllocatePool() - c:\home\edk-
ii\mdemodulepkg\library\uefimemoryallocationlibprofile\memoryallocationlib.c:440)
0x0000003F  0x00000000000007E0 <== 0x0000000000023479 (Lib:AllocateZeroPool) (ShellCommandRegisterCommandName()
- c:\home\edk-ii\shellpkg\library\uefishellcommandlib\uefishellcommandlib.c:546)
0x0000003F  0x000000000000030C <== 0x00000000000234C2 (Lib:AllocateCopyPool) (ShellCommandRegisterCommandName()
- c:\home\edk-ii\shellpkg\library\uefishellcommandlib\uefishellcommandlib.c:548)
0x0000000B  0x00000000000000B0 <== 0x00000000000239A7 (Lib:AllocateZeroPool) (ShellCommandRegisterAlias() -
c:\home\edk-ii\shellpkg\library\uefishellcommandlib\uefishellcommandlib.c:807)
0x0000000B  0x000000000000005C <== 0x00000000000239F0 (Lib:AllocateCopyPool) (ShellCommandRegisterAlias() -
c:\home\edk-ii\shellpkg\library\uefishellcommandlib\uefishellcommandlib.c:809)
0x0000000B  0x0000000000000060 <== 0x0000000000023A0F (Lib:AllocateCopyPool) (ShellCommandRegisterAlias() -
c:\home\edk-ii\shellpkg\library\uefishellcommandlib\uefishellcommandlib.c:810)
0x00000007  0x0000000000000134 <== 0x000000000001DF0E (Lib:ReallocatePool) (StrnCatGrow() - c:\home\edk-
ii\shellpkg\library\uefishelllib\uefishelllib.c:3262)
0x00000001  0x0000000000000078 <== 0x000000000000B825 (Lib:AllocateZeroPool) (ConsoleLoggerInstall() -
c:\home\edk-ii\shellpkg\application\shell\consolelogger.c:40)
0x00000001  0x000000000000300C <== 0x000000000000D16C (Lib:AllocateZeroPool) (ConsoleLoggerResetBuffers() -
c:\home\edk-ii\shellpkg\application\shell\consolelogger.c:1217)
0x00000001  0x00000000000005DC0 <== 0x000000000000D191 (Lib:AllocateZeroPool) (ConsoleLoggerResetBuffers() -
c:\home\edk-ii\shellpkg\application\shell\consolelogger.c:1223)
0x00000001  0x0000000000000014 <== 0x000000000000D7B7 (Lib:AllocateZeroPool)
(CreatePopulateInstallShellParametersProtocol() - c:\home\edk-
ii\shellpkg\application\shell\shellparametersprotocol.c:340)
0x00000003  0x0000000000000060 <== 0x00000000000240A9 (Lib:AllocateZeroPool)
(ShellCommandAddMapItemAndUpdatePath() - c:\home\edk-
ii\shellpkg\library\uefishellcommandlib\uefishellcommandlib.c:1182)
0x00000003  0x000000000000001E <== 0x00000000000240E3 (Lib:AllocateCopyPool)
(ShellCommandAddMapItemAndUpdatePath() - c:\home\edk-
ii\shellpkg\library\uefishellcommandlib\uefishellcommandlib.c:1187)
0x00000003  0x0000000000000024 <== 0x00000000000040AB (Lib:AllocateZeroPool) (AddLineToCommandHistory() -
c:\home\edk-ii\shellpkg\application\shell\shell.c:1336)
0x00000003  0x0000000000000090 <== 0x00000000000040F4 (Lib:AllocateCopyPool) (AddLineToCommandHistory() -
c:\home\edk-ii\shellpkg\application\shell\shell.c:1338)
0x00000001  0x0000000000000FA0 <== 0x0000000000003E94 (Lib:AllocateZeroPool) (DoShellPrompt() - c:\home\edk-
ii\shellpkg\application\shell\shell.c:1217)
0x0000000F  0x00000000000000FA <== 0x000000000000A067 (Lib:AllocateZeroPool) (EfiShellGetEnvEx() - c:\home\edk-
ii\shellpkg\application\shell\shellprotocol.c:2675)
0x00000016  0x0000000000000108 <== 0x0000000000003FC5 (Lib:AllocateZeroPool) (AddBufferToFreeList() -
c:\home\edk-ii\shellpkg\application\shell\shell.c:1275)
0x00000002  0x000000000000016C <== 0x0000000000009F2A (Lib:AllocateZeroPool) (EfiShellGetEnvEx() - c:\home\edk-
ii\shellpkg\application\shell\shellprotocol.c:2635)
```

```
0x00000001  0x0000000000000044 <== 0x00000000000045B9 (Lib:AllocateCopyPool) (ShellConvertVariables() -
c:\home\edk-ii\shellpkg\application\shell\shell.c:1542)
0x00000001  0x0000000000000044 <== 0x0000000000005D26 (Lib:AllocateZeroPool) (RunShellCommand() - c:\home\edk-
ii\shellpkg\application\shell\shell.c:2577)
0x00000001  0x0000000000000038 <== 0x0000000000011C9B (Lib:AllocateZeroPool) (CreateFileInterfaceFile() -
c:\home\edk-ii\shellpkg\application\shell\filehandlewrappers.c:1845)
0x00000001  0x0000000000000034 <== 0x000000000000FC20 (Lib:AllocateZeroPool) (CreateSimpleTextOutOnFile() -
c:\home\edk-ii\shellpkg\application\shell\consolewrappers.c:451)
0x00000001  0x0000000000000018 <== 0x000000000000FCA5 (Lib:AllocateZeroPool) (CreateSimpleTextOutOnFile() -
c:\home\edk-ii\shellpkg\application\shell\consolewrappers.c:466)
0x00000001  0x000000000000002C <== 0x000000000001DF33 (Lib:AllocateZeroPool) (StrnCatGrow() - c:\home\edk-
ii\shellpkg\library\uefishelllib\uefishelllib.c:3267)
0x00000001  0x000000000000002C <== 0x000000000000855F (Lib:AllocateCopyPool) (InternalShellExecuteDevicePath() -
c:\home\edk-ii\shellpkg\application\shell\shellprotocol.c:1447)
0x00000001  0x0000000000000004 <== 0x000000000000D5B4 (Lib:AllocateZeroPool) (ParseCommandLineToArgs() -
c:\home\edk-ii\shellpkg\application\shell\shellparametersprotocol.c:249)
==========================
```

NOTE: *In order to return correct information, we strongly encourage that a user disables compiler optimization. Or the address might be inaccurate.*

NOTE: *We call it POTENTIAL memory leak because it might be legal in some cases. For example, a SHELL may choose to cache the recent command, or a variable. This must be investigated case by case.*

NOTE: *We strongly encourage that the user does not enable all memory types or* `EfiBootServicesCode/EfiBootServicesData` *for all drivers. Our experience shows that if this action is taken, the system will boot extremely slowly because there are many allocate/free actions in all of those drivers. In order to detect a memory leak, we recommend just enabling all memory type or* `EfiBootServicesCode/EfiBootServicesData` *for some specific drivers.*

Memory leak prevention

We may introduce a way to record memory allocation for each driver and application, and free resource when one driver or application is unloaded.

However, we also found that it is an incompatible change. According to the UEFI specification, a driver needs to register an `EFI_LOADED_IMAGE_PROTOCOL.Unload` API to do cleanup when the `DxeCore` unloads this image. The `DxeCore` only needs to free the resource allocated by `DxeCore` itself.

A DXE driver may purposely exit without freeing a memory resource because these memory regions might be used by other drivers. If the `DxeCore` frees the memory for an unloaded driver or application then this may break the previous assumption.

Also, it will become a bigger problem if only the memory is freed but the source using this memory is not freed, such as a protocol. A protocol may become a dangling pointer.

In the DXE phase, since the image unload is handled by the `DxeCore`, the `DxeCore` is best place to know which resource is invalid.

We did a prototype in the DXE core to free the protocol and event associated with the image automatically when the image is unloaded. But it is not proper to include it unless these behaviors are clearly documented in a specification.

Summary

This section describes the memory leak detection feature to help a developer analyze memory leak issues in a UEFI firmware implementation.

Part III – Performance Profile

Problem statement

End users hope to have a system that boots as quickly as possible.

OS vendors like Microsoft* have time requirements on both a system boot and a S3 resume. This information also needs to be reported via the ACPI FPDT table. [*WHCK requirement*]

How can a developer know if this requirement is met and which piece of code takes the longest time?

ACPI Firmware Performance Data Table

The ACPI specification defines a firmware performance data table (FPDT) which provides information to describe the platform initialization performance records. See figure 10 for the ACPI FPDT structure.

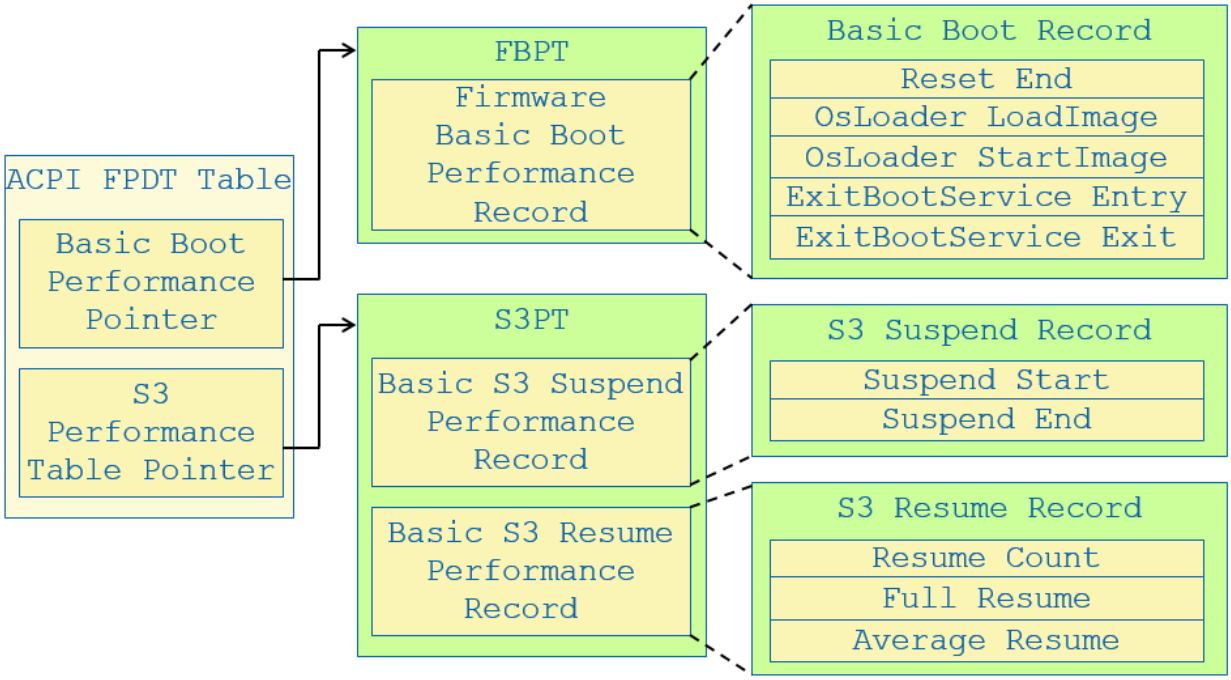

Figure 10: ACPI FPDT structure

This FPDT table includes 2 pointers. One points to a firmware basic boot performance record table (FBPT) which includes the time for `ResetEnd`, `OsLoader`, and `ExitBootService`. The other one points to an S3 performance record table (S3PT) which includes the time for S3 suspend, and S3 resume. These records are the ***basic*** records. A platform vendor, a hardware vendor, or a platform firmware vendor may define his or her own record types and formats as extensions. Please refer to [ACPI] for more detail.

So what are these FPDT data fields are how are they filled in by EDK II?

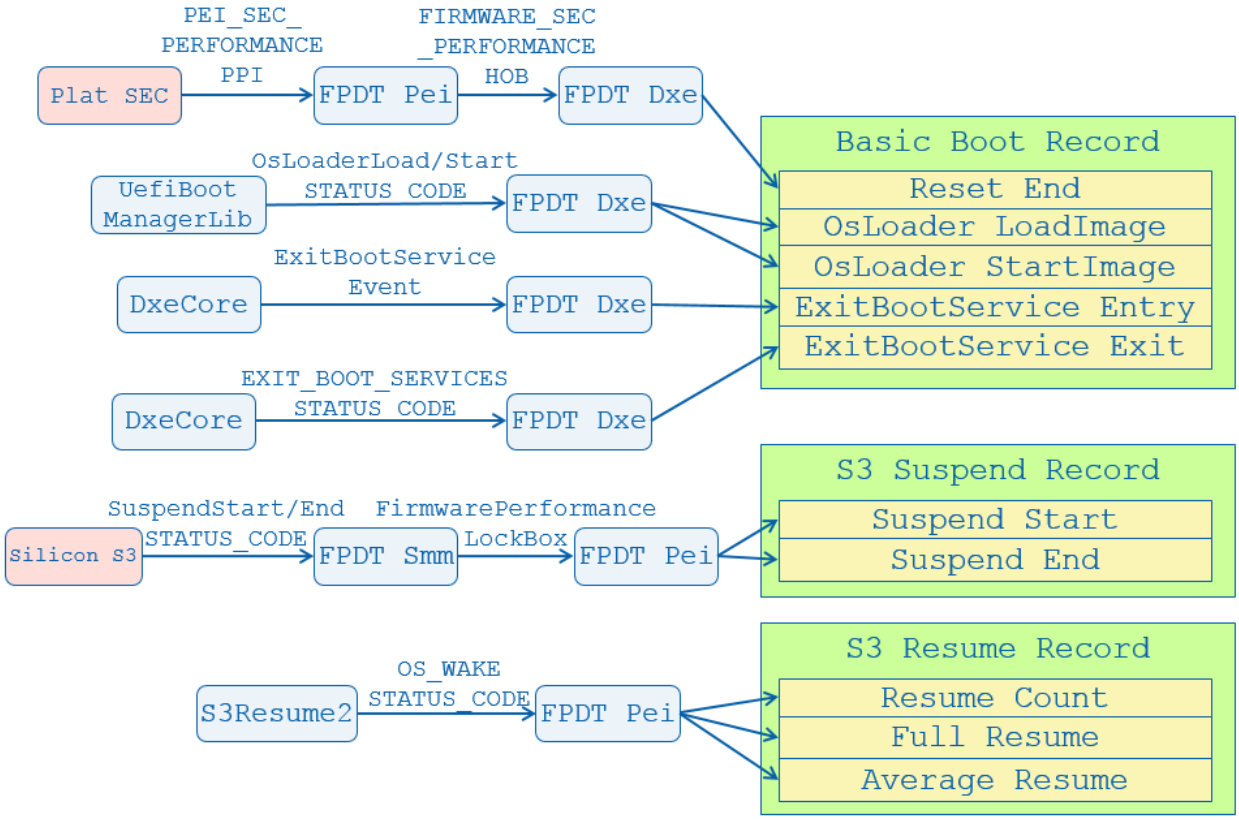

Figure 11: ACPI FPDT modules and entries

In EDK II, there are 3 FPDT related modules - https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/FirmwarePerformanceDataTablePei, https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/FirmwarePerformanceDataTableDxe, https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/FirmwarePerformanceDataTableSmm. See figure 11.

The FPDT DXE module fills the basic boot record. After a system boot, the platform SEC module records the `ResetEnd` time and reports the information to the FPDT PEI module via the `PEI_SEC_PERFORMANCE_PPI` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Ppi/SecPerformance.h). Then the FPDT PEI builds a `FIRMWARE_SEC_PERFORMANCE_HOB` which will be consumed by a FPDT DXE module in order to fill in the `ResetEnd` field.

The OS Loader `LoadImage/StartImage` action happens in the `BDS/UefiBootManagerLib`. They are reported via a `STATUS_CODE`. The FPDT DXE driver installs a `ReportStatusCode` listener to get the information and fill in the OS Loader `LoadImage/StartImage` field.

The FPDT DXE module registers for an `ExitBootService` event to fill in the `ExitBootServiceEntry` field. The `ExitBootServiceExit` action is reported as a `STATUS_CODE`

by the DxeCore. The ReportStatusCode listener in FPDT can know when it happens and fill in the ExitBootServiceExit field.

The FPDT DXE module also saves the S3 performance table pointer into a FirmwarePerformanceS3 LockBox. In the S3 resume path, the FPDT PEIM reads the same LockBox to know where the S3 performance table.

The S3 suspend action happens in a silicon SMM module. We assume that a silicon SMM module registers for an S3 Callback and reports the SuspendStart/End via a STATUS_CODE - PcdProgressCodeS3SuspendStart and PcdProgressCodeS3SuspendEnd (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/MdeModulePkg.dec). The FPDT SMM driver installs a ReportStatusCode listener to get the information and saves the SuspendStart/End time into a FirmwarePerformance LockBox. In the S3 resume path, the FPDT PEIM reads the same LockBox to retrieve the value and fills the S3 suspend record.

The FPDT PEIM also installs a ReportStatusCode listener for S3 resume. Before the BIOS jumps to the OS waking vector, the S3Resume2 PEIM reports OS_WAKE STATUS_CODE. Then the FPDT PEIM calculates the S3 resume time and fills in the S3 resume record.

From the OS perspective, it might be enough to know the full BIOS boot or S3 resume time. But for a developer, he or she might want to know the detailed performance data for some specific code to facilitate performance tuning. As such, the ACPI FPDT is not enough.

EDK II Performance profile

In the EDK II MdePkg, there is a PerformanceLib (https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Library/PerformanceLib.h). It can be used to record the detailed performance data for the specific code.

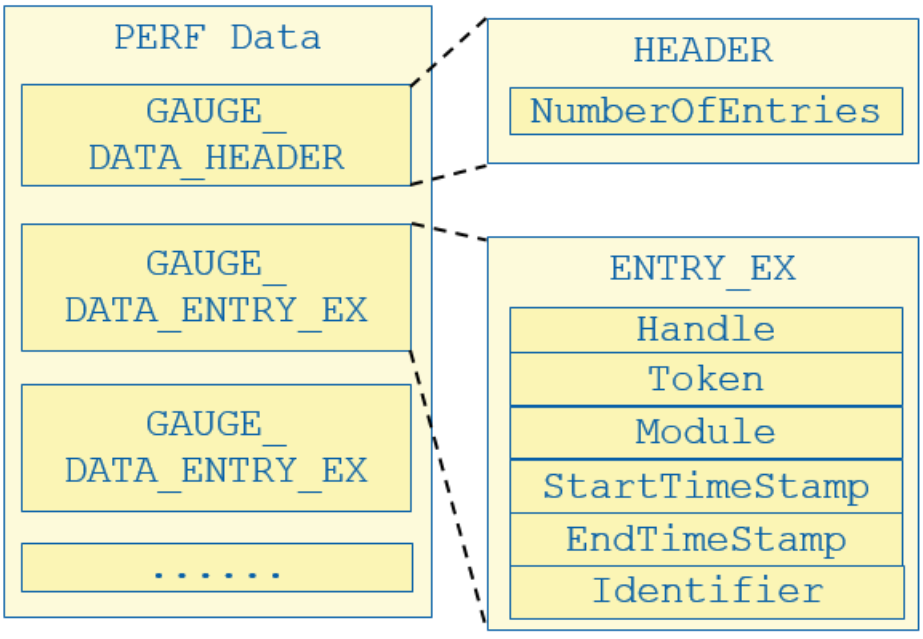

Figure 12: EDK II performance log structure

The final performance record is shown in figure 12. The GAUGE_DATA_HEADER includes a count of the GAUGE_DATA_ENTRY_EX records (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/Performance.h) Each GAUGE_DATA_ENTRY_EX includes the `Handle/Token/Module/StartTimeStamp/StopTimeStamp/Identifier` for a piece of code. The `Handle/Token/Module/Identifier` are used to match a Start/End pair. Handle is a pointer for the context. Token and Module are the human readable strings. Identifier is a system unique ID to identify the code.

How are these performance data items constructed in EDK II?

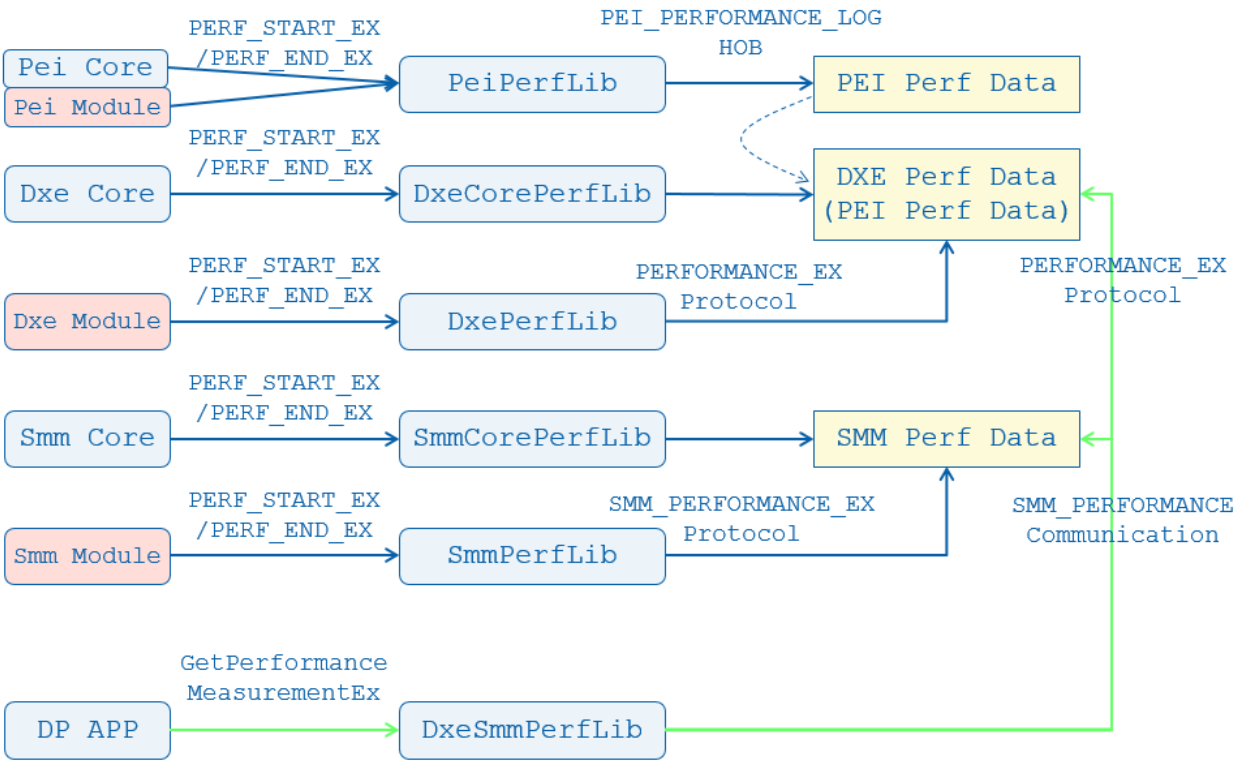

Figure 13: EDK II performance profile modules

See figure 13. EDK II provides a set of XXX`PerfLib`.

In order to record the performance data, a PEIM needs to link against `PeiPerformanceLib`. The performance data is recorded in PEI_PERFORMANCE_LOG HOB as PEI performance data.

The `DxeCore` links against `DxeCorePerformanceLib`. It allocates memory for the DXE performance data, parses the PEI_PERFORMANCE_LOG HOB, and converts the PEI data to DXE performance data. `DxeCorePerformanceLib` also installs a PERFORMANCE_EX protocol to provide services to other performance libraries.

The DXE module may link against `DxePerformanceLib`. This library just calls the PERFORMANCE_EX protocol to record the performance data.

On the SMM side, the SmmCore links against SmmCorePerformanceLib. It allocates memory for SMM performance data and installs the SMM_PERFORMANCE_EX protocol.

The SMM module may link against SmmPerformanceLib. This library calls SMM_PERFORMANCE_EX protocol to record the performance data in SMM.

Finally, in order to dump the performance data, the DP tool (PerformancePkg) needs to link against DxeSmmPerformanceLib. This library calls the PERFORMANCE_EX protocol to get the PEI and DXE performance data, and this library calls the SMM_PERFORMANCE communication protocol to get the SMM performance data.

DP is good UEFI shell tool, but in some cases we need to dump the performance data in the OS for events that occur after ReadyToBoot, or in order to ascertain performance data in S3 resume.

So we need another way to dump the performance data.

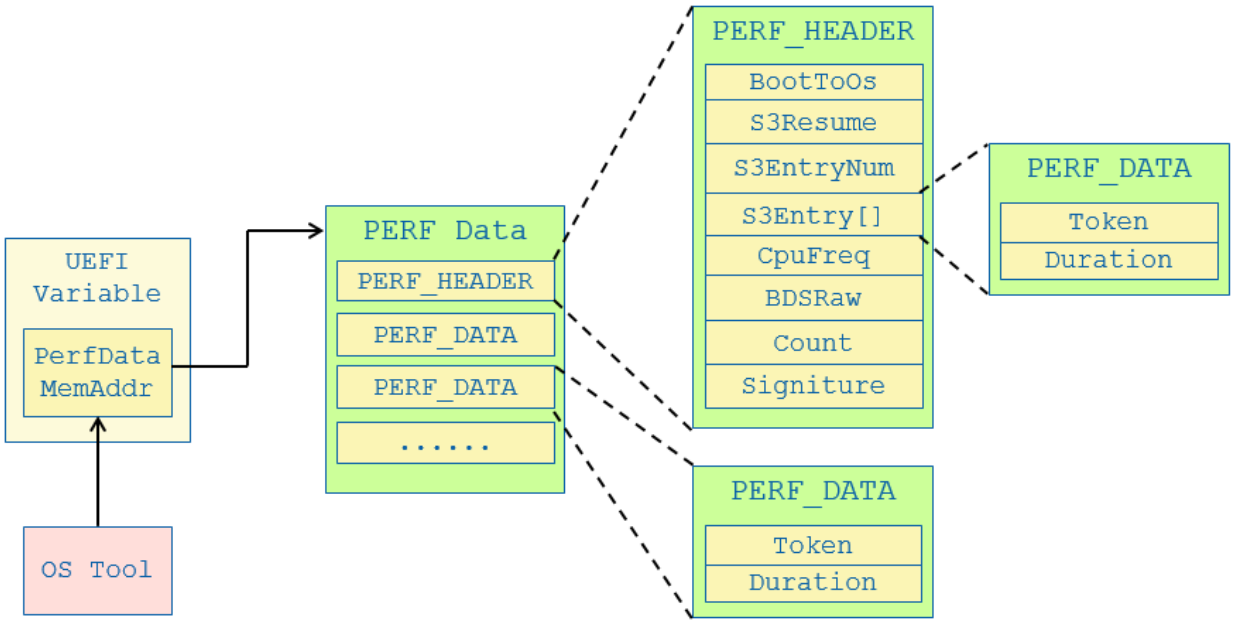

Figure 14: EDK II performance data for OS

Figure 14 shows the performance data for OS.

A variable "PerfDataMemAddr" is defined to point to the performance data for the OS. It has a PERF_HEADER to record general data, such as BootToOs, CpuFreq, with PERF_DATA followed by all of the additional boot records.

The S3 resume data items are included into the PERF_HEADER S3Entry. They are filled in during S3 resume time.

In order to dump these data items in the OS environment, an OS tool needs to get the "PerfDataMemAddr" and parse the PERF_HEADER and PERF_DATA fields.

The last question is: which modules should be included to fill these performance data items for the OS?

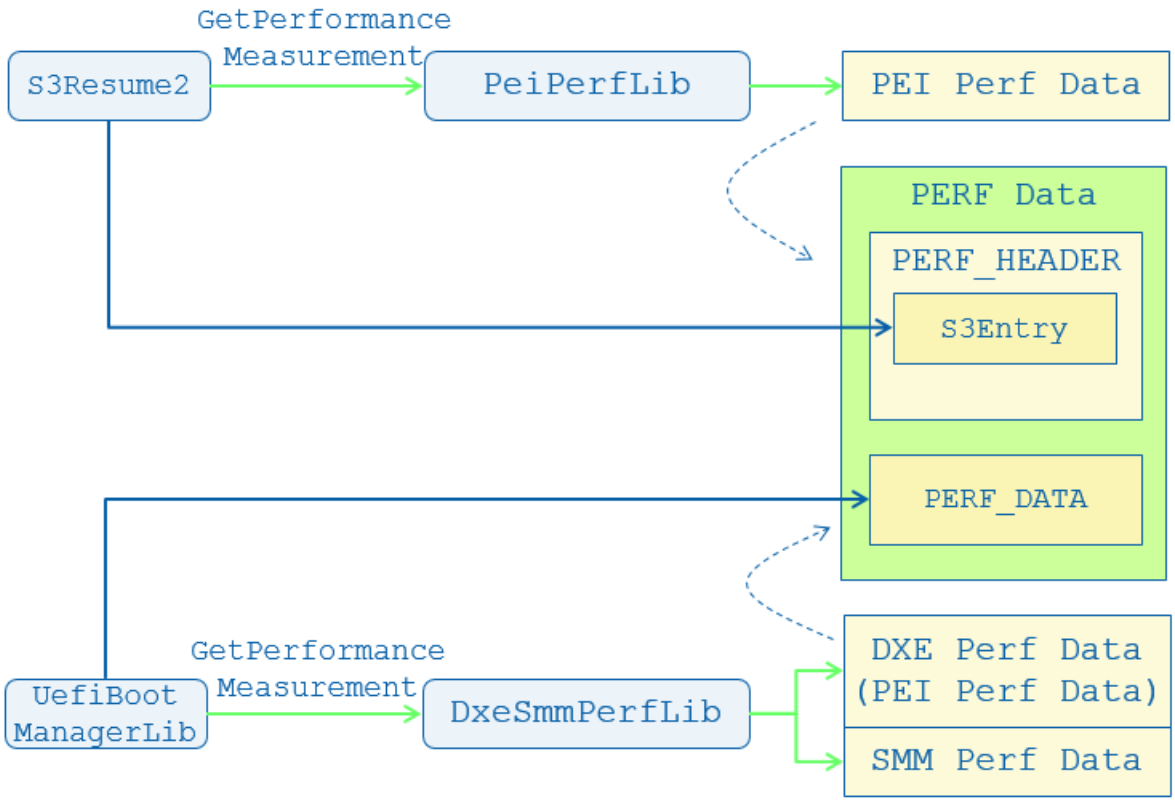

Figure 15: EDK II performance boot/S3 modules

See figure 15. During Boot, the UefiBootManagerLib (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Library/UefiBootManagerLib/BmPerformance.c) links against the DxeSmmPerformanceLib to get the boot record via the GetPerformanceMeasurement() API. It also allocates enough memory for PERF data, fills the boot record PERF_DATA and sets "PerfDataMemAddr" variable. In the S3 resume, the S3Resume2 module links against PeiPerformanceLib to get the S3 PEI record. It gets the "PerfDataMemAddr" variable and saves the S3 record to the S3Entry. After S3 resume, the OS tool can get the S3 record in this S3 resume.

How to enable

1) In order to enable ACPI FPDT, a user need add below components in DSC and FDF.

```
MdeModulePkg/Universal/Acpi/FirmwarePerformanceDataTablePei/FirmwarePerformancePei.inf
MdeModulePkg/Universal/Acpi/FirmwarePerformanceDataTableDxe/FirmwarePerformanceDxe.inf
MdeModulePkg/Universal/Acpi/FirmwarePerformanceDataTableSmm/FirmwarePerformanceSmm.inf
```

The report status code PCD must be enabled and report status code modules must be included, because some data are passed via report status code handler.

```
MdeModulePkg/Universal/ReportStatusCodeRouter/Pei/ReportStatusCodeRouterPei.inf
```

```
MdeModulePkg/Universal/ReportStatusCodeRouter/RuntimeDxe/ReportStatusCodeRouterRuntimeDxe
.inf
MdeModulePkg/Universal/ReportStatusCodeRouter/Smm/ReportStatusCodeRouterSmm.inf
# PCD
  gEfiMdePkgTokenSpaceGuid.PcdReportStatusCodePropertyMask|0x3
```

The platform SEC module must record the ResetEnd time and report the information to the FPDT PEI module via the PEI_SEC_PERFORMANCE_PPI.

If S3 is supported, a silicon SMM module must register for an S3Callback and report the SuspendStart/End via a STATUS_CODE - PcdProgressCodeS3SuspendStart and PcdProgressCodeS3SuspendEnd.

2) In order to enable EDK II Performance profile, a user need choose below performance library instance.

```
# PEIM / PEI_CORE
PerformanceLib|MdeModulePkg/Library/PeiPerformanceLib/PeiPerformanceLib.inf
# DXE_DRIVER
PerformanceLib|MdeModulePkg/Library/DxePerformanceLib/DxePerformanceLib.inf
# DXE_CORE / DXE_RUNTIME_DRIVER / UEFI_DRIVER
PerformanceLib|MdeModulePkg/Library/DxeCorePerformanceLib/DxeCorePerformanceLib.inf
# DXE_SMM_DRIVER
PerformanceLib|MdeModulePkg/Library/SmmPerformanceLib/SmmPerformanceLib.inf
# SMM_CORE
PerformanceLib|MdeModulePkg/Library/SmmCorePerformanceLib/SmmCorePerformanceLib.inf
# UEFI_APPLICATION
DxeSmmPerformanceLib|MdeModulePkg/Library/DxeSmmPerformanceLib/DxeSmmPerformanceLib.inf
```

Below PCD should be configured properly.

```
# PCD
  gEfiMdePkgTokenSpaceGuid.PcdPerformanceLibraryPropertyMask|0x1
  gEfiMdeModulePkgTokenSpaceGuid.PcdMaxPeiPerformanceLogEntries|140
```

The BDS time out value can be set to 0 to indicate boot immediately.

```
gEfiMdePkgTokenSpaceGuid.PcdPlatformBootTimeOut|L"Timeout"|gEfiGlobalVariableGuid|0x0|1 #
Variable: L"Timeout"
```

And finally, the DP tool shell application need be built with the current platform.

```
# Shell application to dump
  PerformancePkg/Dp_App/Dp.inf
```

3) In order to support the performance profile recording, a user need select a proper timer lib.

- In most platforms, we recommend using “ACPI Timer + TSC Count”, by using below library instances.

```
TimerLib|PcAtChipsetPkg/Library/AcpiTimerLib/BaseAcpiTimerLib.inf
TimerLib|PcAtChipsetPkg/Library/AcpiTimerLib/DxeAcpiTimerLib.inf
```

- If there is no ACPI timer, the Local APIC Timer (Depend on PcdFSBClock) can be used instead.

```
TimerLib|UefiCpuPkg/Library/SecPeiDxeTimerLibUefiCpu/SecPeiDxeTimerLibUefiCpu.inf
```

4) After the system boot to UEFI shell, the user may use a UEFI shell tool to dump performance data. (PerformancePkg\Dp_App)

Summary

This chapter introduced the performance profile feature to help a developer evaluate and tune the boot performance.

Part IV – SMM Profile

Problem statement

System Management Mode (SMM) is a special-purpose operating mode in Intel® Architecture Processors. SMM was designed to provide for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code.

SMM was introduced in the Intel® 80386SL Processor. Since then, SMM has been widely used by firmware or hardware-assisted debuggers. SMM mode is a highly privileged operating mode, even higher than a normal hypervisor. If the hypervisor is regarded as RING -1, the SMM is regarded as RING -2. There are many SMM based attacks that have occurred in the world. For example, [SMM Attack01][SMM Attack02] discussed how to attack a system via outside SMRAM code access (See figure 16, 17). [SMM Attack03][SMM04][SMM05] discussed how to attack a system via outside SMRAM data access (See figure 18, 19). As a result, some OS vendors like Microsoft* define the firmware requirement for SMM. [WSMT][WSMT2]

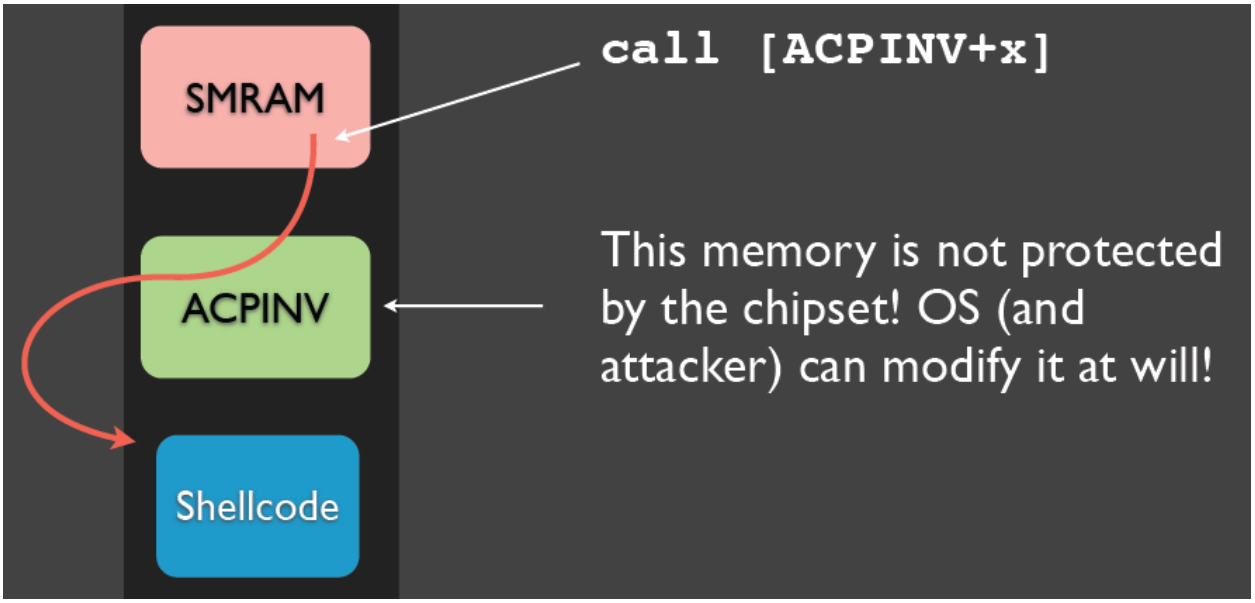

Figure 16: Attack system via outside SMRAM code access

(Source: Attacking Intel Trusted Execution Technology, 2009)

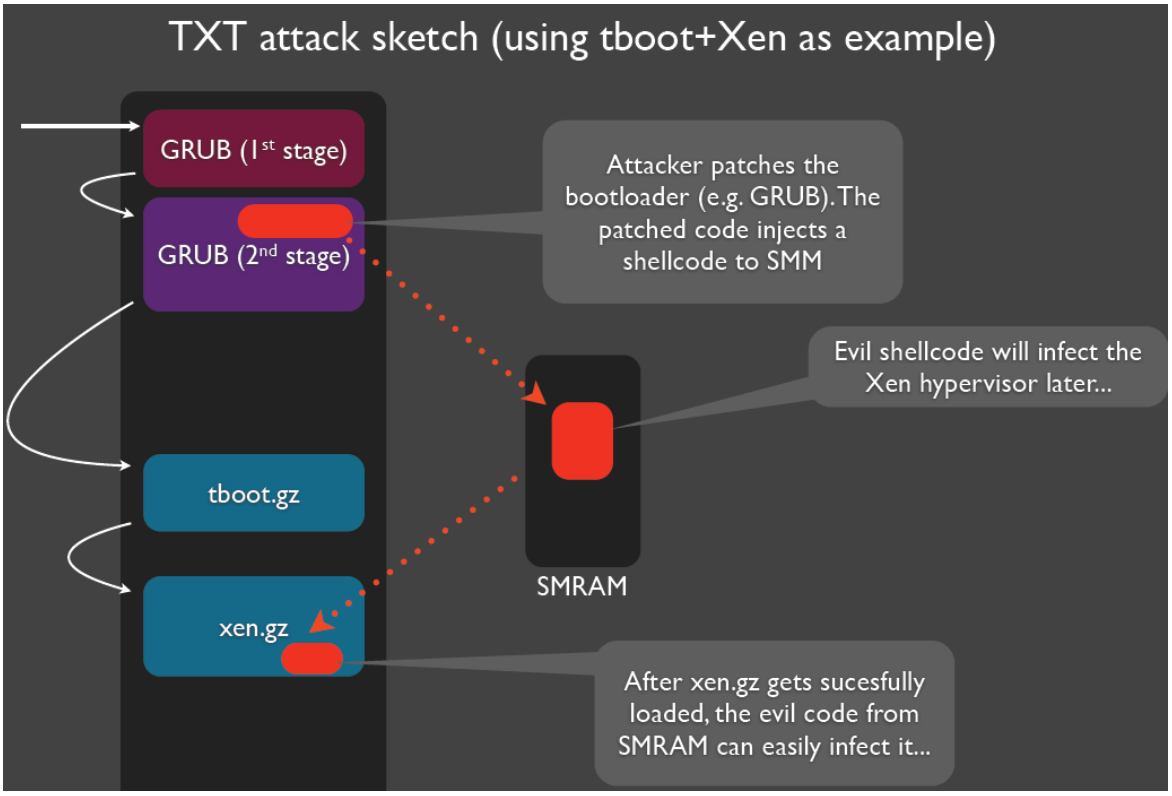

Figure 17: Attack system via outside SMRAM code access

(Source: Attacking Intel Trusted Execution Technology, 2009)

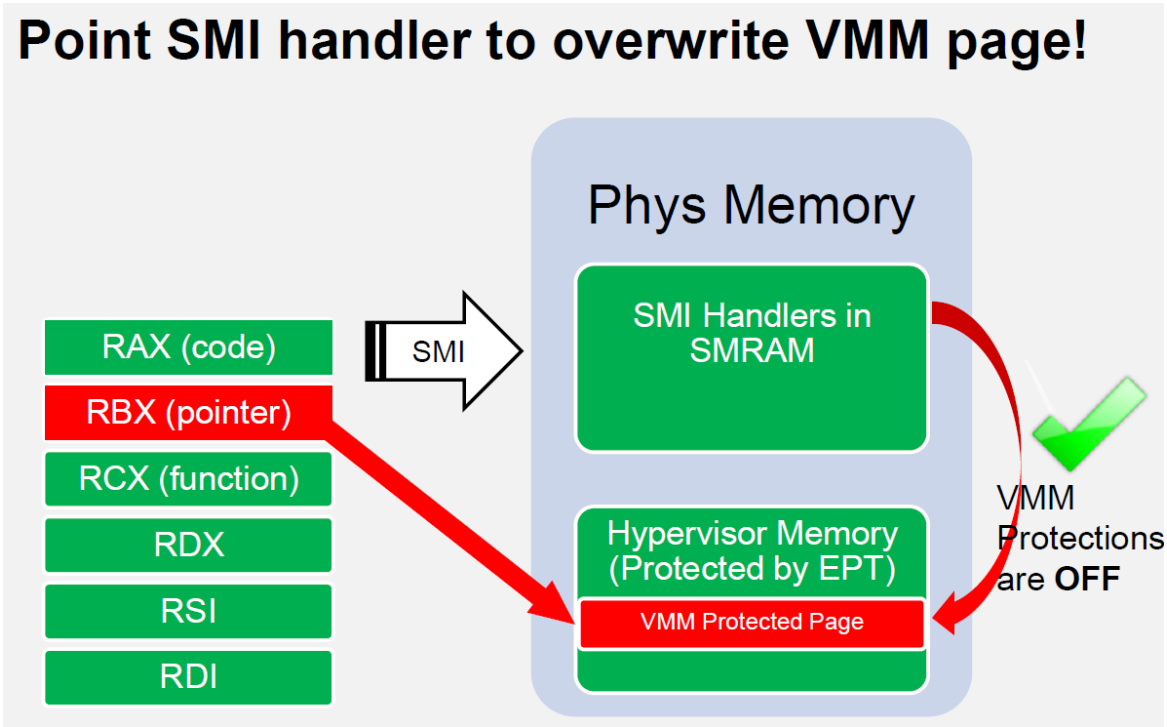

Figure 18: Attack system via outside SMRAM data access

(Source: Attacking Hypervisors via Firmware and Hardware, 2015)

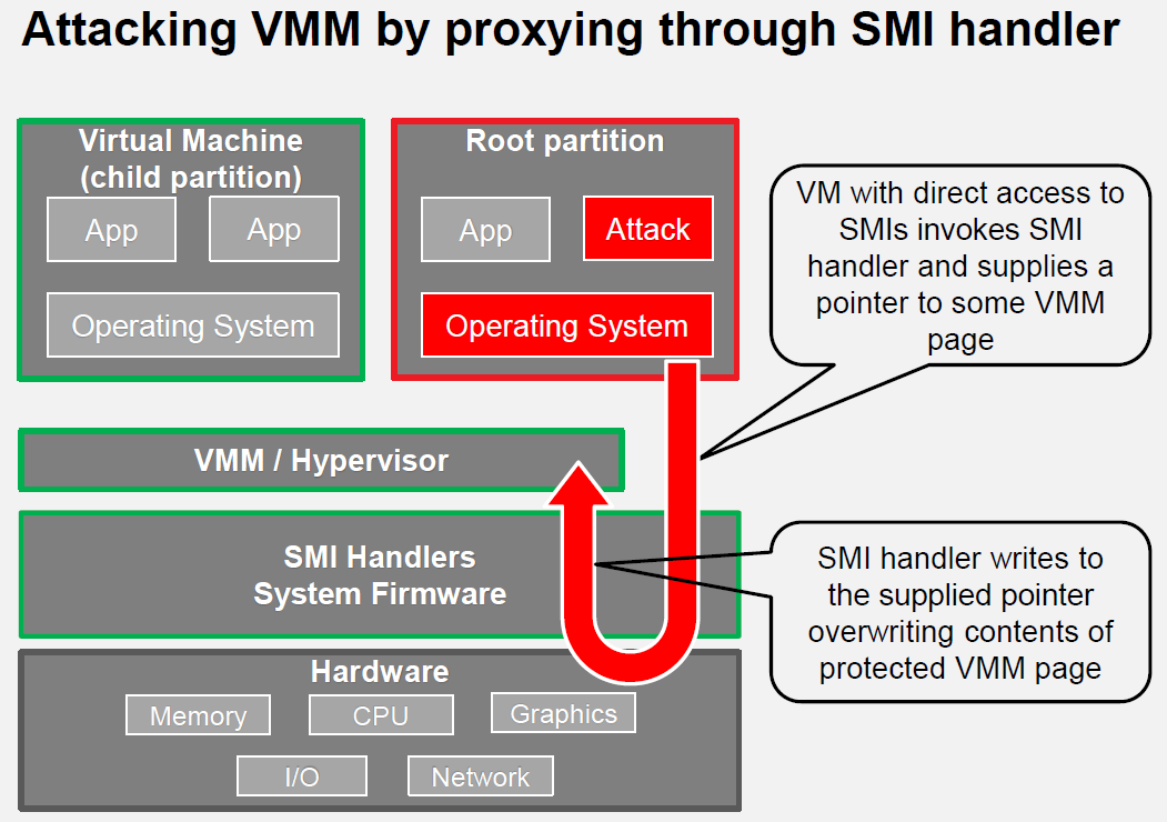

Figure 19: Attack system via outside SMRAM data access

(Source: Attacking Hypervisors via Firmware and Hardware, 2015)

We have discussed some features enhancements in EDK II, such as [SMM Comm Buffer], [SMM Monitor] to mitigate such attacks. Even though we have enhancements available, a developer may still want to see if there is any violation occuring in the firmware. How can we do that?

SMM profile Table

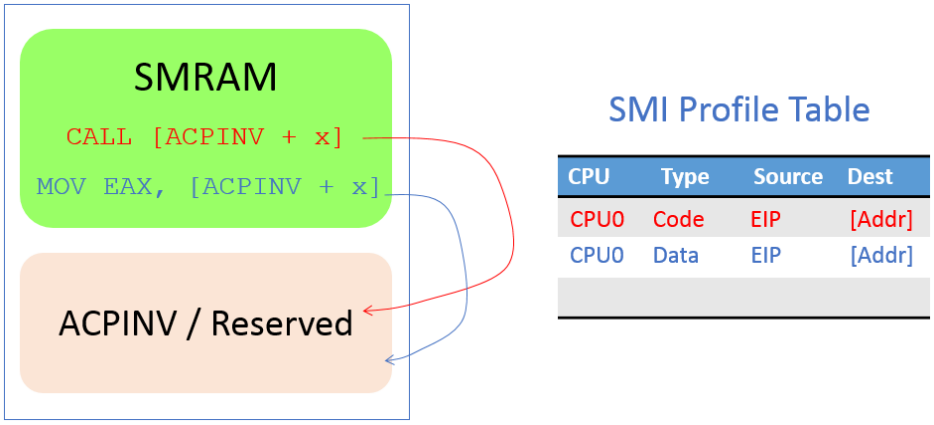

Figure 20: SMM profile table

The SMM profile table should record any outside SMRAM code access or data access. See figure 20. The natural way is capture this information is to leverage a CPU page table.

In EDK II, `PiSmmCpuDxeSmm` driver constructs the SMM execution environment. This driver also builds a page table for SMM. (https://github.com/tianocore/edk2/tree/master/UefiCpuPkg/PiSmmCpuDxeSmm)

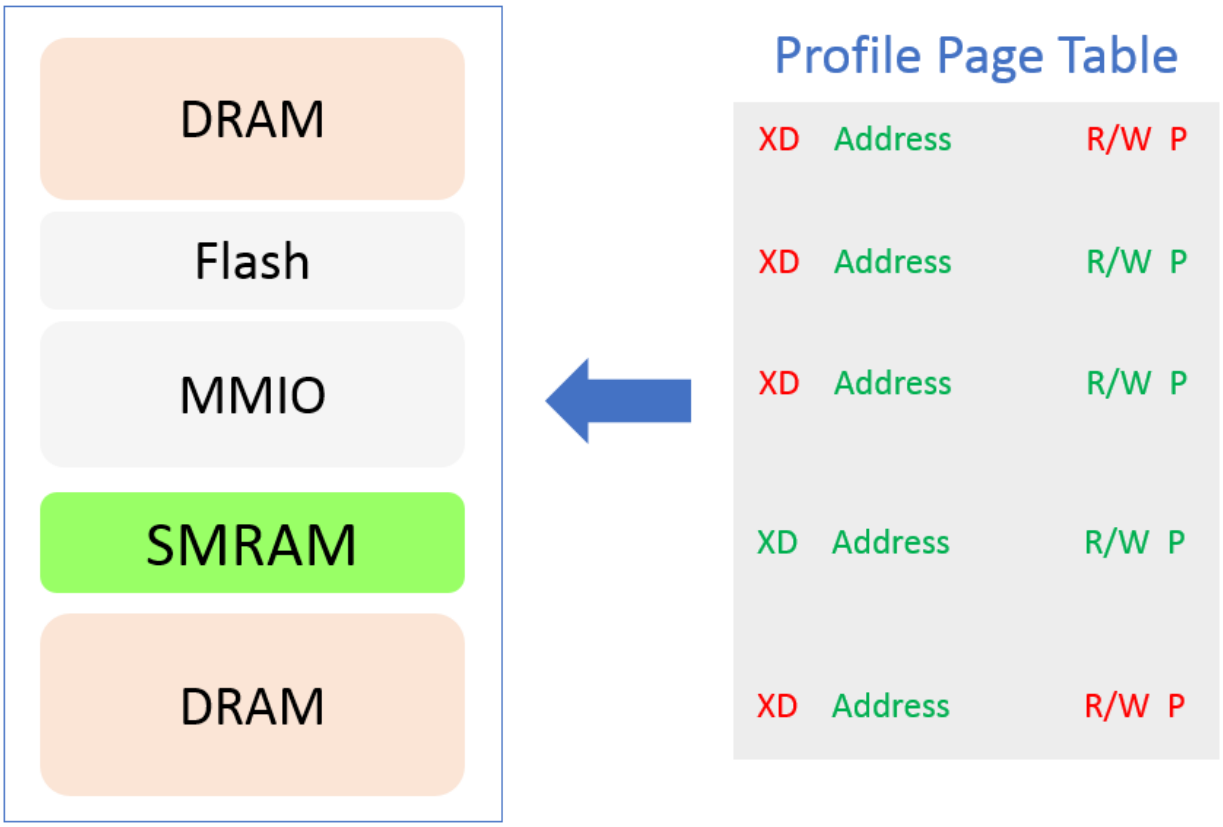

Figure 21: SMM profile initialization

In the `PiSmmCpuDxeSmm` entry point, `InitSmmProfile()` is called to prepare the profile execution environment.
(https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/SmmProfile.c)
`InitSmmProfileInternal()` allocates a reserved memory region for the profile database. The database cannot be in SMRAM because the whole profile database might be quite large. In `InitSmmProfileCallBack()`, the profile database header is recorded as a UEFI variable. This variable allows a SHELL tool or OS tool ability to read the variable and parse the profile data.

In `SmmProfileStart()`, a profile page table is activated in `PerformRemainingTasks()`. This happens after the `SmmReadyToLock` event because the `PiSmmCore` needs to leverage many DXE/UEFI services to load an image from flash into SMRAM before the SMM ready to lock. There are many legal outside SMRAM code and data accesses. We do not want to record these legal actions.

The `PiSmmCpuDxeSmm` driver only marks the SMRAM to be executable and also marks only SMRAM, the flash region and the MMIO region to be present. As a result, any outside of SMRAM code access will cause a page fault exception, and any normal DRAM data access will cause a page fault exception. See figure 21. The GREEN text in the profile page table means the data is valid. The RED text in the profile page table means the data is invalid.

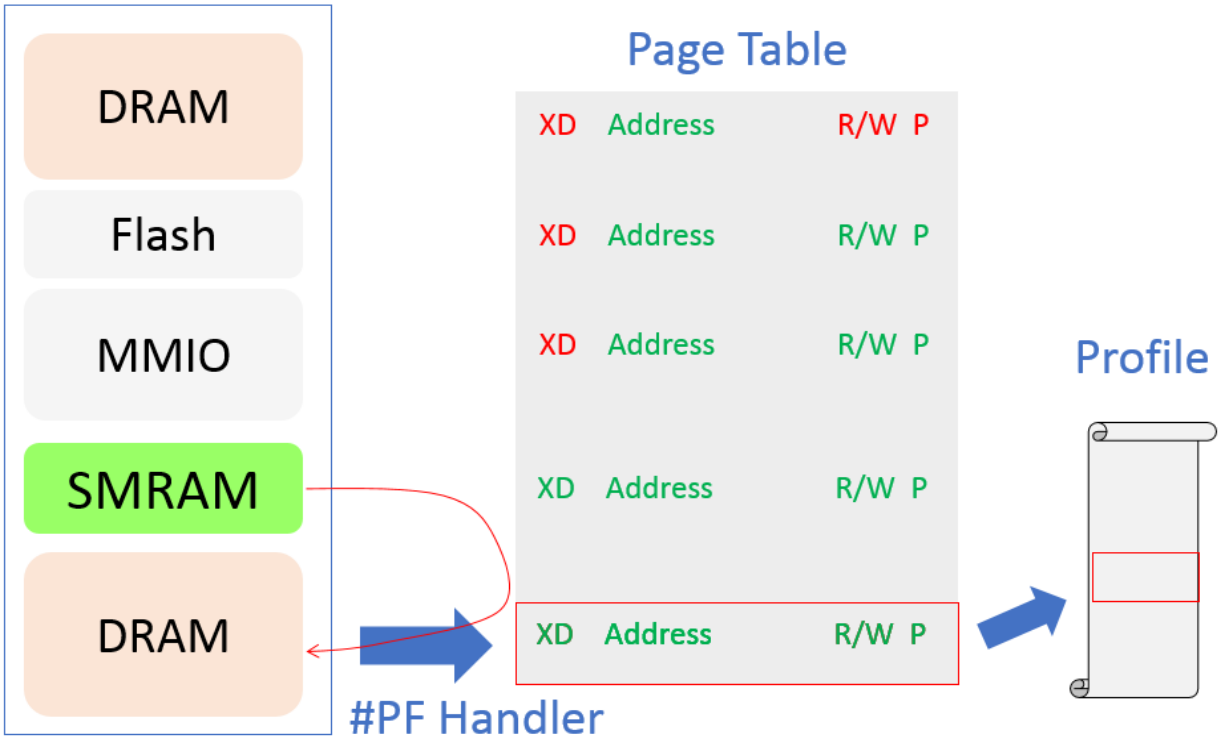

Figure 22: SMM profile runtime – catch and record

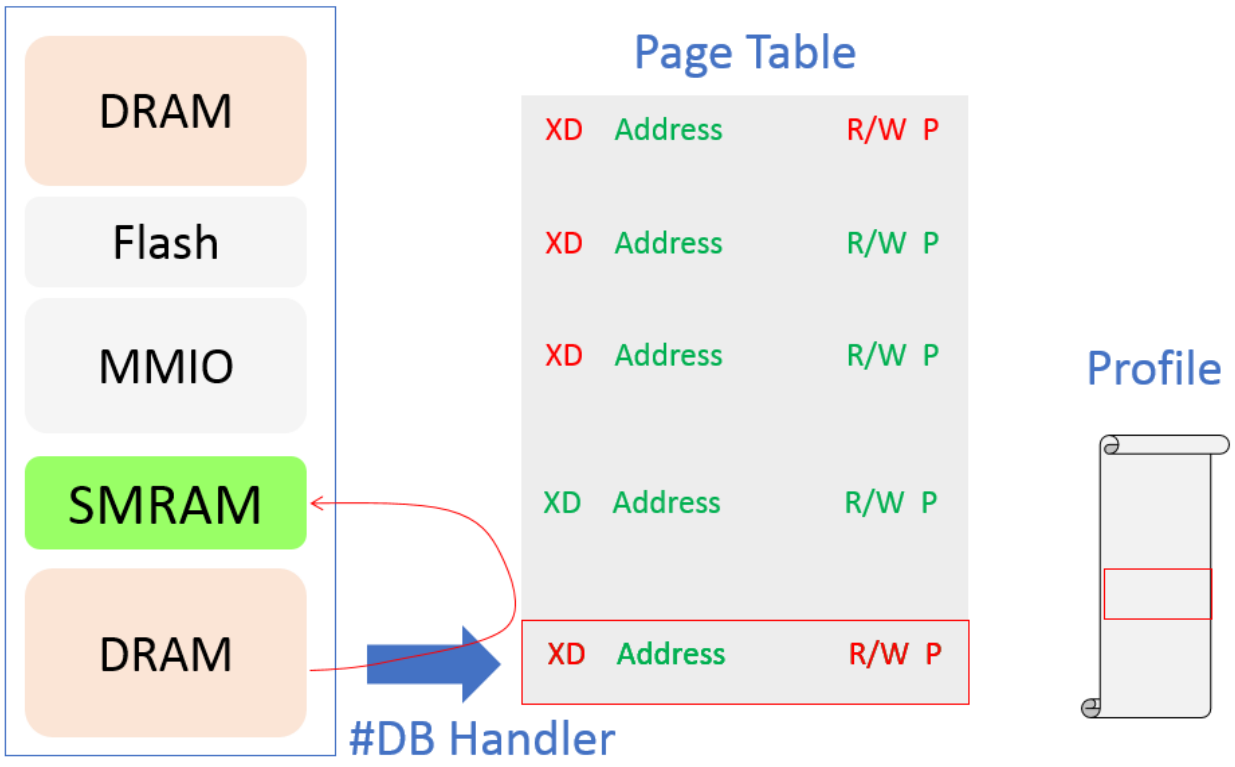

Figure 23: SMM profile runtime – restoration

During runtime, when there is an outside SMRAM code or data access, the page fault exception (#PF) is triggered. (https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/SmmProfile.c) The #PF handler `SmmProfilePFHandler()` inserts a record in the profile table and patches the corresponding page table entry to be valid. Then the root #PF handle `PageFaultIdtHandlerSmmProfile()` enables the single step debug exception and returns to the original instruction. See figure 22.

The original code can execute one instruction and then triggers a debug exception (#DB). The #DB handler `DebugExceptionHandler()` restores the original page table entry in order to catch the page fault exception again. See figure 23.

SMM profile Table in S3 resume.

In S3 resume, the `PiSmmCpuDxeSmm` performs the SMM rebase action and the SMM profile is still enabled.

How to enable

1) A user may set `gUefiCpuPkgTokenSpaceGuid.PcdCpuSmmProfileEnable|TRUE` to enable the SMM profile feature.
2) After the system boots to the UEFI shell, the user may use a UEFI shell tool to dump the `SmmProfile` information. (https://github.com/jyao1/EdkiiShellTool/tree/master/EdkiiShellToolPkg/SmmProfileDump)
3) In the OS, the user may use an OS tool to get the UEFI variable. There is no open source tool yet.

NOTE: *This SMM profile feature is a debug feature only. It should be disabled in a production build because it may expose SMM internal information, including function calls or data references.*

NOTE: *This SMM profile feature is just to provide the profile information. It is not used to protect SMM from being attacked, to prevent outside SMRAM access, or to check the correctness of hardware configuration. These protection features are discussed in [SMM Comm Buffer], [SMM Monitor].*

Summary

This chapter introduced the SMM profile feature to help a developer analyze the outside SMRAM data access or code access in a UEFI firmware implementation.

Part V – DXE/SMM/PEI Core Resource Profile

Besides memory resources, all other system resources, such as the protocol installed, the event created, and the callback function registered are also important.

If a driver or an application exits without freeing these resources, it is considered as UEFI resource leak.

A developer may want to check how many protocols, notifications, events, SMI handlers are installed on a firmware implementation. In order to review these core database, we developed a tool to dump the EDK II `DxeCore/SmmCore/PeiCore` internal data structures. (https://github.com/jyao1/EdkiiShellTool/tree/master/EdkiiShellToolPkg/EdkiiCoreDatabaseDump). This tool can be used by a developer to check if the resource is correct or not.

NOTE: *This* `EdkiiCoreDatabaseDump` *tool can only be used for the EDK II version* `DxeCore/SmmCore/PeiCore`. *If a firmware implementation does not use EDK II* `DxeCore/SmmCore/PeiCore`, *please do not use this* `EdkiiCoreDatabaseDump` *tool.*

DXE core resource profile

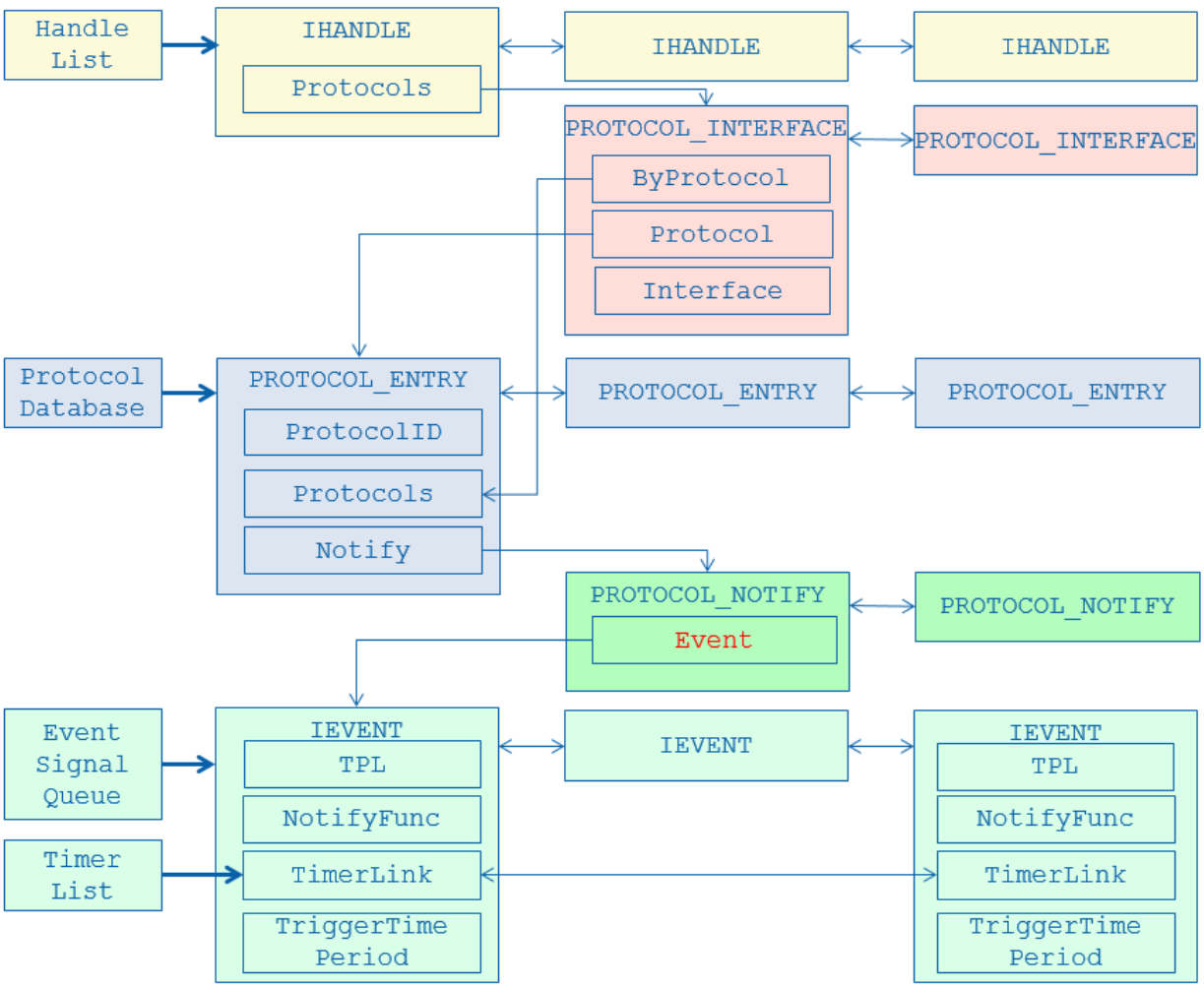

Figure 24: DXE Core internal structure

The EDK II DxeCore (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/Dxe) maintains a database for all resources, such as handles, protocols, and events. See figure 24. For the handle database, the DxeCore has gHandleList as the linked list header for all handles (IHANDLE). mProtocolDatabase is the linked list header for all protocols (PROTOCOL_ENTRY). The PROTOCOL_NOTIFY includes an Event field which points to IEVENT structure.

For the event database, the DxeCore has gEventSignalQueue as the linked list header for SIGNAL event. mEfiTimerList is the linked list header for TIMER event. Once an event is notified, it is put onto gEventQueue[Tpl].

The sample output for DxeCoreDump on Nt32Pkg is below:

```
==================================
##################
# IMAGE DATABASE #
##################
Image: DxeCore (0x5F04000 - 0x30000, EntryPoint:0x601000, Base:0x600000)
       (FvFile(D6A2CB7F-6A18-4E2F-B43B-9920A733700A))
Image: DevicePathDxe (0x5E2B000 - 0x13000, EntryPoint:0x371040, Base:0x370000)
       (FvFile(9B680FCE-AD6B-4F3A-B60B-F59899003443))
Image: PcdDxe (0x5E1F000 - 0xC000, EntryPoint:0xA7B1040, Base:0xA7B0000)
       (FvFile(80CF7257-87AB-47F9-A3FE-D50B76D89541))
Image: ReportStatusCodeRouterRuntimeDxe (0x5ED6000 - 0x9000, EntryPoint:0xA7C1040, Base:0xA7C0000)
       (FvFile(D93CE3D8-A7EB-4730-8C8E-CC466A9ECC3C))
Image: StatusCodeHandlerRuntimeDxe (0x5ECB000 - 0xB000, EntryPoint:0x531040, Base:0x530000)
       (FvFile(6C2004EF-4E0E-4BE4-B14C-340EB4AA5891))
Image: WinNtOemHookStatusCodeHandlerDxe (0x5E15000 - 0xA000, EntryPoint:0xA7D1040, Base:0xA7D0000)
       (FvFile(CA4233AD-847E-4E5D-AD3F-21CABFE5E23C))
......

###################
# HANDLE DATABASE #
###################
Handle - 0x5A62F90
  Protocol - 4C8A2451-C207-405B-9694-99EA13251341, Interface - 0x62B000 (DxeCore)
  Protocol - gEfiLoadedImageProtocolGuid, Interface - 0x62B988 (DxeCore)
    (FvFile(D6A2CB7F-6A18-4E2F-B43B-9920A733700A))
......
Handle - 0x5648F10
  Protocol - 4C8A2451-C207-405B-9694-99EA13251341, Interface - 0xA8D4280 (HiiDatabase)
  Protocol - gEfiLoadedImageDevicePathProtocolGuid, Interface - 0x5648B10
    (Fv(6D99E806-3D38-42C2-A095-5F4300BFD7DC)/FvFile(348C4D62-BFBD-4882-9ECE-C80BB1C4783B))
  Protocol - gEfiLoadedImageProtocolGuid, Interface - 0x5649428
    (FvFile(348C4D62-BFBD-4882-9ECE-C80BB1C4783B))
Handle - 0x5648A10
  Protocol - gEfiHiiImageProtocolGuid, Interface - 0xA8D416C (HiiDatabase)
  Protocol - gEfiConfigKeywordHandlerProtocolGuid, Interface - 0xA8D41D8 (HiiDatabase)
  Protocol - gEfiHiiConfigRoutingProtocolGuid, Interface - 0xA8D41C0 (HiiDatabase)
  Protocol - gEfiHiiDatabaseProtocolGuid, Interface - 0xA8D4194 (HiiDatabase)
  Protocol - gEfiHiiStringProtocolGuid, Interface - 0xA8D4180 (HiiDatabase)
  Protocol - gEfiHiiFontProtocolGuid, Interface - 0xA8D415C (HiiDatabase)
Handle - 0x5648410
  Protocol - 4C8A2451-C207-405B-9694-99EA13251341, Interface - 0x1740D0 (WinNtThunkDxe)
  Protocol - gEfiLoadedImageDevicePathProtocolGuid, Interface - 0x5647F10
    (Fv(6D99E806-3D38-42C2-A095-5F4300BFD7DC)/FvFile(0C95A916-A006-11D4-BCFA-0080C73C8881))
  Protocol - gEfiLoadedImageProtocolGuid, Interface - 0x5647028
    (FvFile(0C95A916-A006-11D4-BCFA-0080C73C8881))
......
```

```
####################
# PROTOCOL DATABASE #
####################
Protocol - gEfiLoadedImageProtocolGuid
  Interface - 0x62B988 (DxeCore)
    (FvFile(D6A2CB7F-6A18-4E2F-B43B-9920A733700A))
  Interface - 0x57097A8
    (FvFile(9B680FCE-AD6B-4F3A-B60B-F59899003443))
......
Protocol - gEfiHiiConfigAccessProtocolGuid
  Interface - 0xAA5A4C0 (DriverHealthManagerDxe)
  Interface - 0x4DEAD20 (PlatDriOverrideDxe)
  Interface - 0xAC7B6F0 (SecureBootConfigDxe)
  Interface - 0x4DBDB94 (SecureBootConfigDxe)
Protocol - gEdkiiVariableExProtocolGuid
  Interface - 0xABB71A4 (VariableSmmRuntimeDxePw)
Protocol - gEdkiiVarCheckProtocolGuid
  Interface - 0xABB7250 (VariableSmmRuntimeDxePw)
Protocol - gEfiPlatformDriverOverrideProtocolGuid
  Interface - 0x4DEAD2C (PlatDriOverrideDxe)
......

##################
# EVENT DATABASE #
##################
# 1. NOTIFY_SIGNAL #
Event - 0x5A62790
  Type - 0x200, TPL - 0x1E, Function - 0x60DA30 (DxeCore)
Event - 0x5A62510 (gIdleLoopEventGuid)
  Type - 0x200, TPL - 0x10, Function - 0x612540 (DxeCore)
Event - 0x5A62610 (gEfiEndOfDxeEventGroupGuid)
  Type - 0x200, TPL - 0x10, Function - 0x6196F0 (DxeCore)
......
Event - 0x5E70B90 (gEfiEventVirtualAddressChangeGuid)
  Type - 0x60000202, TPL - 0x10, Function - 0xABB2A20 (VariableSmmRuntimeDxePw)
Event - 0x4CD3310
  Type - 0x80000200, TPL - 0x10, Function - 0x1215D0 (WatchdogTimer)
Event - 0x5E70C90 (gEfiEventVirtualAddressChangeGuid)
  Type - 0x60000202, TPL - 0x10, Function - 0xAC92C60 (MonotonicCounterRuntimeDxe)
Event - 0x4CD2C10 (gEfiEventExitBootServicesGuid)
  Type - 0x201, TPL - 0x10, Function - 0xAC92C50 (MonotonicCounterRuntimeDxe)
Event - 0x5E70D90 (gEfiEventVirtualAddressChangeGuid)
......
# 2. TIMER #
Event - 0x5649010 (TiggerTime:230000000, Period:10000000)
  Type - 0x80000200, TPL - 0x10, Function - 0xA891DF0 (EbcDxe)
Event - 0x4C74090 (TiggerTime:224560000, Period:200000)
  Type - 0x80000200, TPL - 0x10, Function - 0xAD646B0 (WinNtGopDxe)
Event - 0x4C87810 (TiggerTime:224560000, Period:200000)
  Type - 0x80000200, TPL - 0x10, Function - 0xAD646B0 (WinNtGopDxe)
====================================
```

SMM core resource profile

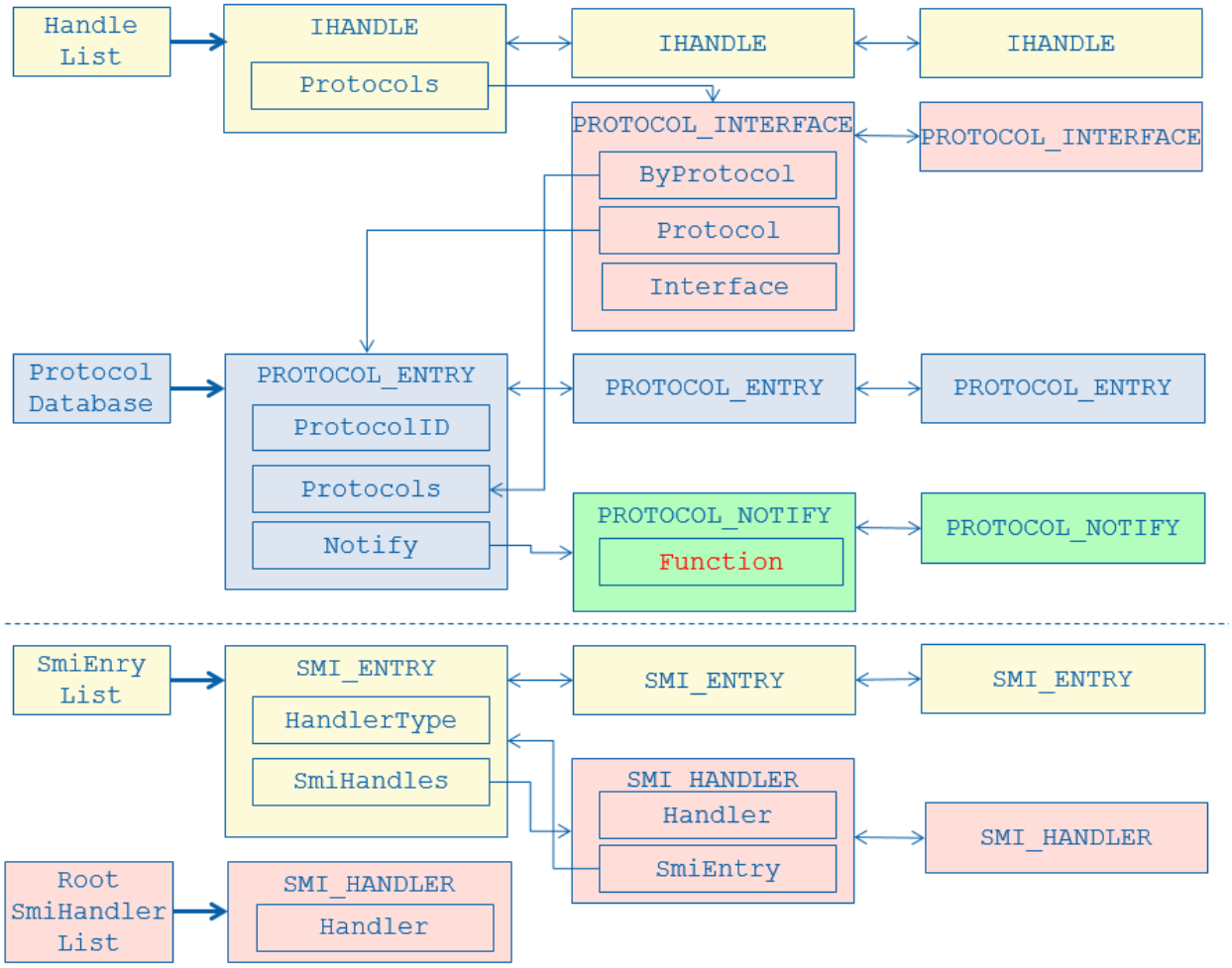

Figure 25: SMM Core internal structure

The EDK II `SmmCore` (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/PiSmmCore) is similar to the EDK II `DxeCore`. See figure 25. The difference is that the PROTOCOL_NOTIFY does not have an Event field in SMM, but instead it uses the Function pointer directly. The reason is that `SmmCore` does not define an Event concept. The protocol notification is a function.

The `SmmCore` also maintains a list of SMI `handlers. mSmiEntryList` is a linked list of `SMI_ENTRY` for all child SMI `handlers. mRootSmiHandlerList` is a special linked list for all root SMI handlers.

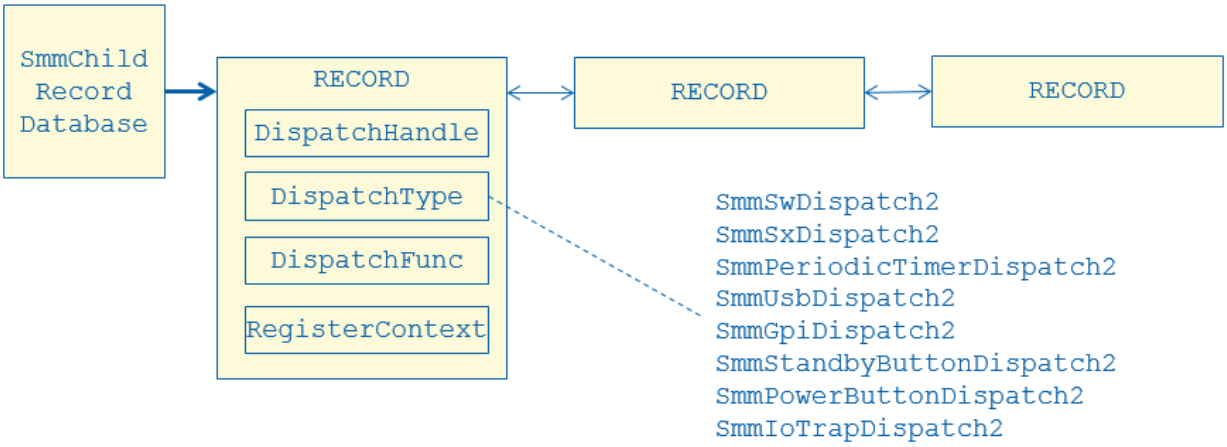

Figure 26: SMM Child internal structure

The PI specification also defines SMM child dispatch protocols. Usually, these protocols are installed by a silicon specific SMM child dispatcher module. This module maintains a linked list for all registered SMM child dispatch handler, such as `SmmSw`, `SmmSx`, `SmmPeriodicTimer`, `SmmUsb`, `SmmGpi`, `SmmStandbyButton`, `SmmPowerButton`, `SmmIoTrap`, and etc.

Figure 26 shows a possible internal structure in a SMM child dispatcher module. The `SmmChild` record database is the linked list for all SMM child dispatch handler.

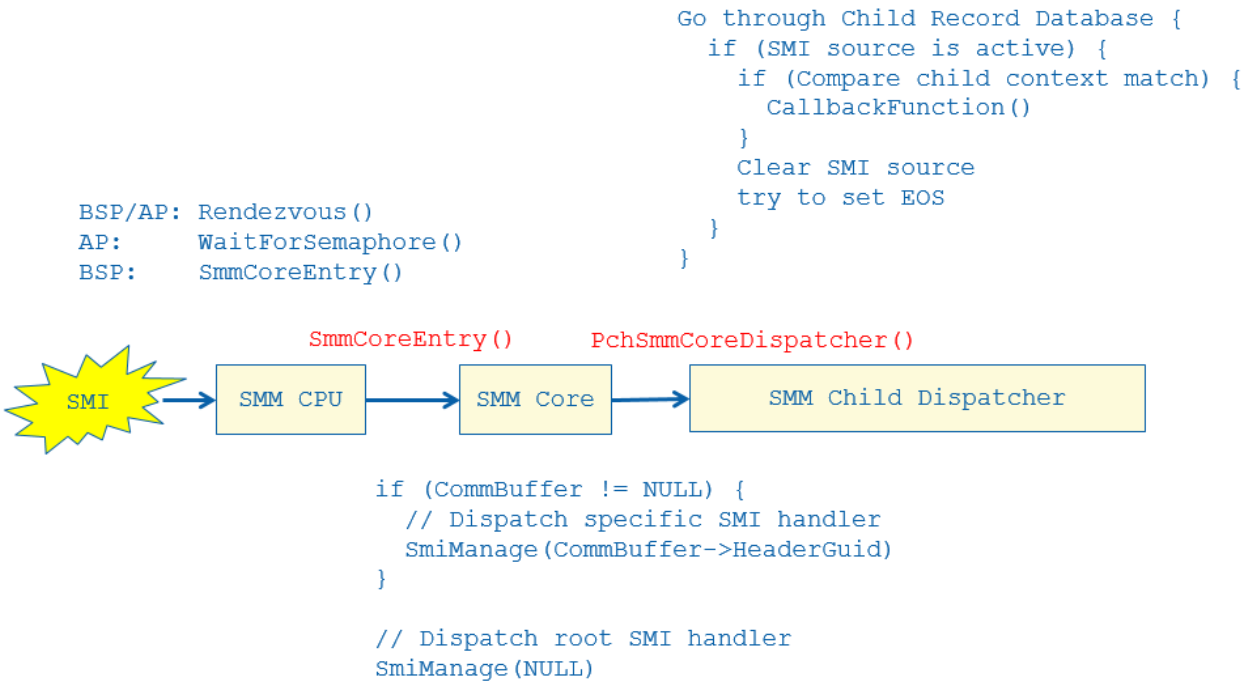

Figure 27: SMM handler dispatch

When an SMI is triggered, the processor invokes the SMI entry in the `PiSmmCpu` driver (https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/X64/SmiEntry.asm). The `PiSmmCpu` driver does BSP/AP rendezvous to make sure that all processors enter SMM mode. Then the APs

call `WaitForSemaphore()` and BSP calls `SmmCoreEntry()` to transfer the control to the `PiSmmCore` module (https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/MpService.c). In `SmmEntryPoint`, the `PiSmmCore` checks the communication buffer and calls `SmiManage()` with the header GUID in the communication buffer (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/PiSmmCore/PiSmmCore.c). This is the first dispatch for the GUID SMI handlers. Then the `PiSmmCore` call `SmiManage()` again with NULL GUID. This is the second dispatch for all root SMI handlers. Usually, a silicon specific SMM module should register a root SMI handlers, such as `PchSmmCoreDispatch()`. Once it is called, the `PchSmmCoreDispatch()` goes through the `SmmChild` record database. If it finds an SMI source is active and the registered context is matched, the registered child SMI callback function is called. This is the third dispatch for all hardware specific SMI handlers. See figure 27.

The sample output for `SmmCoreDump` on SMM enabled Nt32Pkg (https://github.com/jyao1/Nt32Ex/tree/master/Nt32Pkg) is below:

```
====================================
##################
# IMAGE DATABASE #
##################
Image - PiSmmCore (0x6927000 - 0x18000, EntryPoint:0xA9C1040, Base:0xA9C0000)
       (FvFile(E94F54CD-81EB-47ED-AEC3-856F5DC157A9))
Image - FvbServicesRuntimeSmm (0x6914000 - 0x9000, EntryPoint:0xA9E1040, Base:0xA9E0000)
       (FvFile(B23ED876-80E5-4418-B7F1-D2849ECF3DEF))
Image - VariableSmm (0x688C000 - 0x87000, EntryPoint:0xA9F1040, Base:0xA9F0000)
       (FvFile(36A67CDE-13EC-46BA-ACDC-8C7944E731FB))
Image - SmmFaultTolerantWriteDxe (0x6843000 - 0xE000, EntryPoint:0x4A1040, Base:0x4A0000)
       (FvFile(470CB248-E8AC-473C-BB4F-81069A1FE6FD))
Image - StatusCodeHandlerSmm (0x6831000 - 0xA000, EntryPoint:0x101040, Base:0x100000)
       (FvFile(79CD78D8-6EDC-4978-BD02-3299C387AB17))
Image - WinNtOemHookStatusCodeHandlerSmm (0x6826000 - 0xA000, EntryPoint:0x51040, Base:0x50000)
       (FvFile(B1E5A507-CD1B-4A44-85FA-AFAAF1E55D09))
Image - WinNtSmmCpu (0x681C000 - 0x8000, EntryPoint:0xAAA1040, Base:0xAAA0000)
       (FvFile(84393810-8C42-4895-ABE9-AA4C7560AB81))
Image - WinNtSmmChildDispatcherSmm (0x6813000 - 0x8000, EntryPoint:0xD1040, Base:0xD0000)
       (FvFile(18B05910-8DAA-445D-ACF7-E8F953D073BE))
Image - PiSmmCommunicationSmm (0x6808000 - 0xA000, EntryPoint:0x511040, Base:0x510000)
       (FvFile(E21F35A8-42FF-4050-82D6-93F7CDFA7073))

####################
# HANDLE DATABASE #
####################
Handle - 0x6926A08
  Protocol - gEfiLoadedImageProtocolGuid, Interface - 0x6926978
Handle - 0x6920208
  Protocol - gEdkiiSmmVarCheckProtocolGuid, Interface - 0xAA6F264 (VariableSmm)
  Protocol - gEdkiiSmmVariableExProtocolGuid, Interface - 0xAA6F254 (VariableSmm)
  Protocol - gEfiSmmVariableProtocolGuid, Interface - 0xAA6F244 (VariableSmm)
......

######################
# PROTOCOL DATABASE #
######################
Protocol - gEfiSmmEndOfDxeProtocolGuid
  Interface - 0x0
  Notify - 0xA9CC750 (PiSmmCore)
  Notify - 0xA9FCD70 (VariableSmm)
  Notify - 0xA9F1990 (VariableSmm)
  Notify - 0x4A80C0 (SmmFaultTolerantWriteDxe)
  Notify - 0x4A1BA0 (SmmFaultTolerantWriteDxe)
```

```
  Notify - 0x5139F0 (PiSmmCommunicationSmm)
Protocol - gEfiSmmReadyToLockProtocolGuid
  Interface - 0x0
  Notify - 0xA9CC9D0 (PiSmmCore)
  Notify - 0xA9FCFF0 (VariableSmm)
  Notify - 0x4A8340 (SmmFaultTolerantWriteDxe)
  Notify - 0xAAA2140 (WinNtSmmCpu)
  Notify - 0x513C70 (PiSmmCommunicationSmm)
......
Protocol - gEfiSmmFirmwareVolumeBlockProtocolGuid
  Interface - 0x69207D4
  Interface - 0x6920614
  Notify - 0x4A1790 (SmmFaultTolerantWriteDxe)
Protocol - gEfiSmmFaultTolerantWriteProtocolGuid
  Interface - 0x6840010
  Notify - 0xA9F1A00 (VariableSmm)

########################
# SMI Handler DATABASE #
########################
# 1. GUID SMI Handler #
SmiEntry - gEfiEventLegacyBootGuid
  SmiHandle - 0x6926C88
    Handler - 0xA9C1690 (PiSmmCore)
SmiEntry - gEfiEventExitBootServicesGuid
  SmiHandle - 0x6926C08
    Handler - 0xA9C18C0 (PiSmmCore)
SmiEntry - gEfiEventReadyToBootGuid
  SmiHandle - 0x6926B88
    Handler - 0xA9C1900 (PiSmmCore)
SmiEntry - F866226A-EAA5-4F5A-A90A-6CFBA57C588E
  SmiHandle - 0x6926788
    Handler - 0xA9CF990 (PiSmmCore)
SmiEntry - 931FC048-C71D-4455-8930-470630E30EE5
  SmiHandle - 0x6926708
    Handler - 0xA9CFBA0 (PiSmmCore)
SmiEntry - gEfiSmmVariableProtocolGuid
  SmiHandle - 0x69200C8
    Handler - 0xA9F1D90 (VariableSmm)
SmiEntry - gEfiSmmFaultTolerantWriteProtocolGuid
  SmiHandle - 0x6864208
    Handler - 0x4A1BB0 (SmmFaultTolerantWriteDxe)
# 2. ROOT SMI Handler #
RootSmiHandle - 0x6825B48
  Handler - 0xD1860 (WinNtSmmChildDispatcherSmm)

# 3. Hardware SMI Handler #
ChildHandle - gEfiSmmSwDispatch2ProtocolGuid
  SmiHandle - 0x682588C, SwSmi - 0x1
    Handler - 0x511820 (PiSmmCommunicationSmm)
====================================
```

It also shows 3 category SMI handlers.

1) GUID SMI handler – the `PiSmmCore` register SMM Communication handler with `gEfiEventLegacyBootGuid, gEfiEventExitBootServicesGuid, gEfiEventReadyToBootGuid` and etc. The `VariableSmm` driver registered `gEfiSmmVariableProtocolGuid`. The `SmmFaultTolerantWriteDxe` driver registered `gEfiSmmFaultTolerantWriteProtocolGuid`.
2) ROOT SMI Handler – there is only one ROOT SMI handler in Nt32Smm. It is registered by `WinNtSmmChildDispatcherSmm` driver.
3) Hardware SMI handler – there is only one `SwSmi` handler registered in `Nt32Smm`. It is registered by `PiSmmCommunicationSmm` driver.

PEI core resource profile

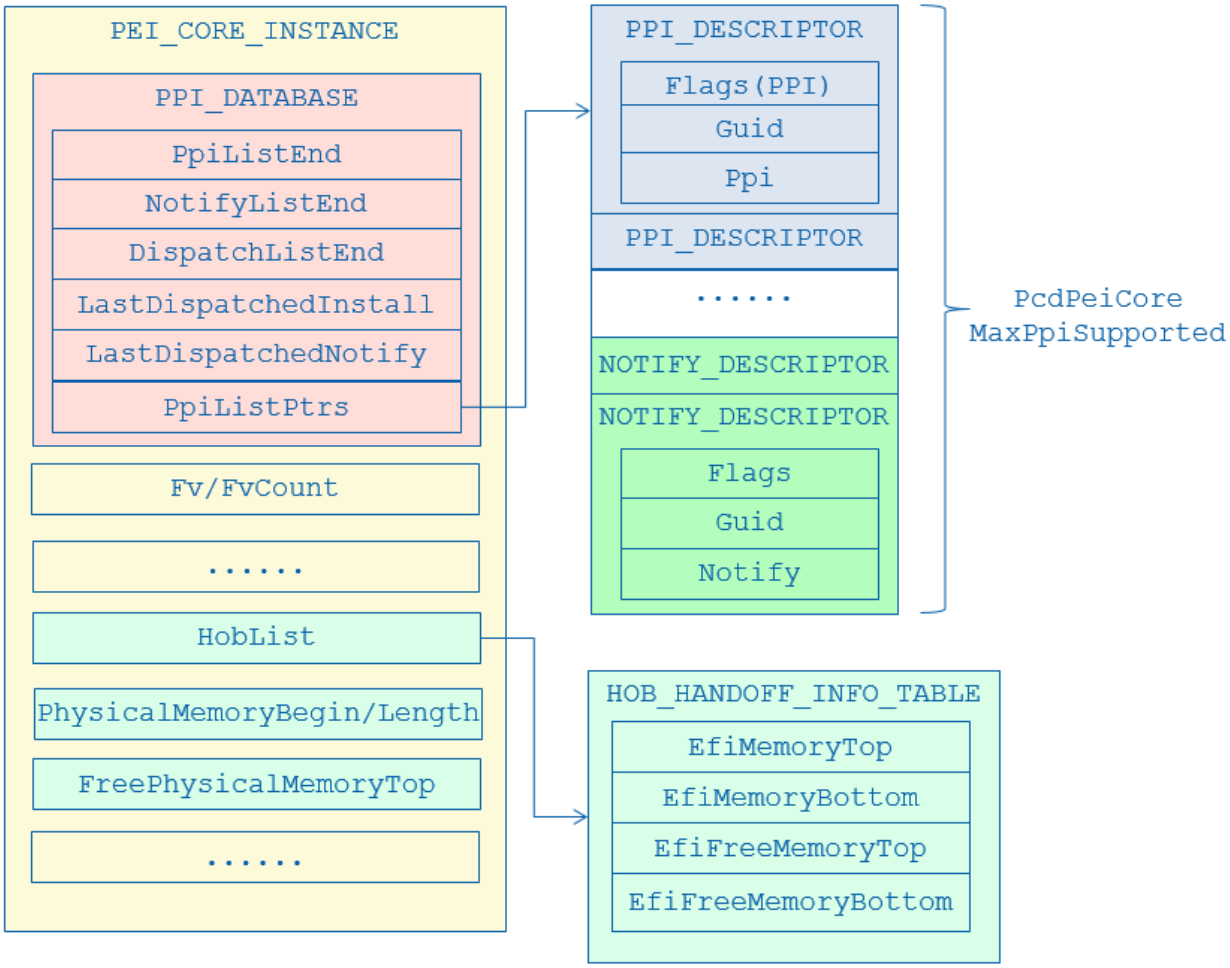

Figure 28: PEI Core internal structure

The EDK II `PeiCore` (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/Pei) exists before the permanent DRAM is ready. It may run in a resource constrained environment. The `PeiCore` PPI database is much simpler. There is a `PpiData` inside of `PEI_CORE_INSTANCE`. The `PpiData` records all PPI and notification information. The number of `PPI/NOTIFY` is controlled by `PcdPeiCoreMaxPpiSupported`, so it is pre-allocated fixed resource buffer.

The memory management in `PeiCore` is also simple. Before permanent DRAM is ready, both `AllocatePages()` and `AllocatePool()` allocate memory between `EfiFreeMemoryBottom` and `EfiFreeMemoryTop` described in PHIT HOB. After permanent DRAM is ready, `AllocatePool()` still allocates memory between `EfiFreeMemoryBottom` and `EfiFreeMemoryTop` described in PHIT HOB and `AllocatePages()` is changed to allocate memory between `PhysicalMemoryBegin` and `FreePhysicalMemoryTop` described in `PEI_CORE_INSTANCE`, so that a caller may allocate a big chunk of memory (greater than 64K) by using `AllocatePages()`.

The sample output for PeiCoreDump on Nt32Pkg is below:

```
====================================
##################
# IMAGE DATABASE #
##################
Image: PcdPeim (0x23BB000)
       (Fv(9B3ADA4F-AE56-4C24-8DEA-F03B7558AE50))
Image: ReportStatusCodeRouterPei (0x23C9000)
       (Fv(A3610442-E69F-4DF3-82CA-2360C4031A23))
......

################
# PPI DATABASE #
################
PpiListEnd            - 24
NotifyListEnd         - 59
DispatchListEnd       - 62
LastDispatchedInstall - 23
LastDispatchedNotify  - 62
PpiCount              - 64
[0] - Flags (0x80000010)
  Ppi - gEfiFirmwareFileSystem2Guid, Interface - 0x51434280
[1] - Flags (0x80000010)
  Ppi - gEfiFirmwareFileSystem3Guid, Interface - 0x514342B0
[2] - Flags (0x80000010)
  Ppi - gEfiPeiLoadFilePpiGuid, Interface - 0x51434304
[3] - Flags (0x10)
  Ppi - gNtPeiLoadFilePpiGuid, Interface - 0x19B018
[4] - Flags (0x10)
  Ppi - gPeiNtAutoScanPpiGuid, Interface - 0x19B01C
[5] - Flags (0x10)
  Ppi - gPeiNtThunkPpiGuid, Interface - 0x19B020
[6] - Flags (0x80000010)
  Ppi - gEfiPeiStatusCodePpiGuid, Interface - 0x676B607C
[7] - Flags (0x10)
  Ppi - gEfiTemporaryRamSupportPpiGuid, Interface - 0x19B02C
[8] - Flags (0x80000010)
  Ppi - gNtFwhPpiGuid, Interface - 0x19B028
[9] - Flags (0x10)
  Ppi - gPcdPpiGuid, Interface - 0x679990B8
[10] - Flags (0x80000010)
  Ppi - gEfiPeiPcdPpiGuid, Interface - 0x67999140
[11] - Flags (0x10)
  Ppi - gGetPcdInfoPpiGuid, Interface - 0x67999134
[12] - Flags (0x80000010)
  Ppi - gEfiGetPcdInfoPpiGuid, Interface - 0x67999188
[13] - Flags (0x80000010)
  Ppi - gEfiPeiRscHandlerPpiGuid, Interface - 0x676B6074
[14] - Flags (0x80000010)
  Ppi - gEfiPeiMasterBootModePpiGuid, Interface - 0x0
[15] - Flags (0x80000010)
  Ppi - gEfiPeiStallPpiGuid, Interface - 0x3A5054
[16] - Flags (0x80000010)
  Ppi - gEdkiiFaultTolerantWriteGuid, Interface - 0x0
[17] - Flags (0x10)
  Ppi - gEfiPeiReadOnlyVariable2PpiGuid, Interface - 0x9A30C4
[18] - Flags (0x80000010)
  Ppi - gEdkiiPeiReadOnlyVariable2ExPpiGuid, Interface - 0x9A30CC
[19] - Flags (0x80000010)
  Ppi - gEfiPeiMemoryDiscoveredPpiGuid, Interface - 0x0
[20] - Flags (0x80000010)
  Ppi - A31280AD-481E-41B6-95E8-127F4C984779, Interface - 0x9C9150
[21] - Flags (0x10)
  Ppi - gEfiDxeIplPpiGuid, Interface - 0x9C914C
```

```
[22] - Flags (0x80000010)
  Ppi - gEfiPeiDecompressPpiGuid, Interface - 0x9C9154
[23] - Flags (0x80000010)
  Ppi - gEfiEndOfPeiSignalPpiGuid, Interface - 0x0
[60] - Flags (0x80000020)
  Notify - gEfiEndOfPeiSignalPpiGuid, Interface - 0x5C1120
[61] - Flags (0x80000020)
  Notify - gEfiPeiFirmwareVolumeInfo2PpiGuid, Interface - 0x514236B0
[62] - Flags (0x20)
  Notify - gEfiPeiFirmwareVolumeInfoPpiGuid, Interface - 0x514236B0
[63] - Flags (0x80000040)
  Notify - gEfiPeiSecurity2PpiGuid, Interface - 0x51428EE0
====================================
```

How to enable

1) In order to enable EDK II DXE core resource profile, a user can build https://github.com/jyao1/EdkiiShellTool/blob/master/EdkiiShellToolPkg/EdkiiCoreDatabaseDump/DxeCoreDump/DxeCoreDumpApp.inf and run `DxeCoreDumpApp` in the UEFI shell environment.

2) In order to enable EDK II SMM core resource profile, a user need include https://github.com/jyao1/EdkiiShellTool/blob/master/EdkiiShellToolPkg/EdkiiCoreDatabaseDump/SmmCoreDump/SmmCoreDump.inf in the system BIOS and rebuild the BIOS. Then he or she can build https://github.com/jyao1/EdkiiShellTool/blob/master/EdkiiShellToolPkg/EdkiiCoreDatabaseDump/SmmCoreDump/SmmCoreDumpApp.inf and run `SmmCoreDumpApp` in the UEFI shell environment.

   If a user wants to dump `SmmChildDispatch` handler (Hardware SMI handler), he or she need create a BIOS specific `SmmChildDumpLib` (https://github.com/jyao1/EdkiiShellTool/blob/master/EdkiiShellToolPkg/EdkiiCoreDatabaseDump/Include/Library/SmmChildDumpLib.h). The default one is NULL `SmmChildDumpLib` (https://github.com/jyao1/EdkiiShellTool/tree/master/EdkiiShellToolPkg/EdkiiCoreDatabaseDump/Library/SmmChildDumpLibNull) and we have NT32 SMM sample - `SmmChildDumpLibNt32` (https://github.com/jyao1/EdkiiShellTool/tree/master/EdkiiShellToolPkg/EdkiiCoreDatabaseDump/Library/SmmChildDumpLibNt32) to work with SMM enabled NT32 package (https://github.com/jyao1/Nt32Ex/tree/master/Nt32Pkg).

3) In order to enable EDK II PEI core resource profile, a user need include https://github.com/jyao1/EdkiiShellTool/blob/master/EdkiiShellToolPkg/EdkiiCoreDatabaseDump/PeiCoreDump/PeiCoreDump.inf in the system BIOS and rebuild the BIOS. Then he or she can build https://github.com/jyao1/EdkiiShellTool/blob/master/EdkiiShellToolPkg/EdkiiCoreDatabaseDump/PeiCoreDump/PeiCoreDumpApp.inf and run `PeiCoreDumpApp` in the UEFI shell environment.

Summary

This chapter introduced the DXE/SMM/PEI core resource profile feature, and it also described tool to dump all the core resources to help for the debug and issue analysis.

Conclusion

Profiling activities are important for the firmware execution environment. This paper introduces the memory profiling infrastructure (including memory leak detection), performance profiling and SMM profiling in EDK II to help developers analyze the size, performance and any security issues in their firmware build.

Glossary

ACPI – Advanced Configuration and Power Interface. The specification defines a new interface to the system board that enables the operating system to implement operating system-directed power management and system configuration.

DXE – Driver Execution Environment, see PI specification Volume 2.

PEI – Pre-EFI Initialization, see PI specification Volume 1.

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

SEC – Security Phase, see PI specification Volume 1.

SMM – System Management Mode. x86 CPU operational mode that is isolated from and transparent to the operating system runtime. Also see PI specification Volume 4.

RT – UEFI Runtime, see UEFI specification.

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] ACPI specification, Version 6.1 www.uefi.org

[EDK2] UEFI Developer Kit www.tianocore.org

[Memory Map] Jiewen Yao, Zimmer Vincent, "A Tour Beyond BIOS Memory Map And Practices in UEFI BIOS", https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Memory_Map_And_Practices_in_UEFI_BIOS_V2.pdf

[SMM Attack1] Rafal Wojtczuk, et al, "Attacking Intel® Trusted Execution Technology", 2009, http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf

[SMM Attack2] Rafal Wojtczuk, et al, "Attacking Intel BIOS", 2009, http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf

[SMM Attack3] Oleksandr Bazhaniuk,Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alex Matrosov, Mickey Shkatov, "Attacking and Defending BIOS" in 2015 http://www.intelsecurity.com/advanced-threat-research/content/AttackingAndDefendingBIOS-RECon2015.pdf

[SMM Attack4] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, Mickey Shkatov, "A New Class of Vulnerabilities in SMI Handlers" 2015, http://www.intelsecurity.com/advanced-threat-research/content/ANewClassOfVulnInSMIHandlers_csw2015.pdf

[SMM Attack5] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, Yuriy Bulygin, "Attacking Hypervisors via Firmware and Hardware" 2015 http://www.intelsecurity.com/advanced-threat-research/content/AttackingHypervisorsViaFirmware_bhusa15_dc23.pdf

[SMM Comm Buffer] Jiewen Yao, Zimmer Vincent, Star Zeng, "A Tour Beyond BIOS Secure SMM Communication", https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Open_Source_IA_Firmware_Platform_Design_Guide_in_EFI_Developer_Kit_II.pdf

[SMM Monitor] Jiewen Yao, Vincent Zimmer, A Tour Beyond BIOS Supporting an SMM Resource Monitor using the EFI Developer Kit II, https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Supporting_SMM_Resource_Monitor_using_the_EFI_Developer_Kit_II.pdf

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.6 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

[WHCK requirement] Windows Hardware Certification Requirements for Client and Server Systems, https://msdn.microsoft.com/en-us/library/windows/hardware/jj128256

[WSMT] Windows SMM Security Table, https://msdn.microsoft.com/en-us/library/windows/hardware/dn495660(v=vs.85).aspx#wsmt
http://download.microsoft.com/download/1/8/A/18A21244-EB67-4538-BAA2-1A54E0E490B6/WSMT.docx

[WSMT2] Microsoft Hypervisor Requirements, https://msdn.microsoft.com/en-us/library/windows/hardware/dn614617

Authors

Jiewen Yao (jiewen.yao@intel.com) is EDK II BIOS architect, EDK II FSP package maintainer, EDK II TPM2 module maintainer, EDK II 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 EDK II BIOS engineer, MdeModulePkg maintainer, PEI/DXE core module owner with Software and Services Group at Intel Corporation.

Jeff Fan (fan.jeff@intel.com) is EDK II BIOS engineer, UefiCpuPkg maintainer, PI SMM core and PI SMM CPU module owner with Software and Services Group at Intel Corporation.