White Paper

A Tour Beyond BIOS Launching a VMM in EFI Developer Kit II

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

September 2015

Executive Summary

In the current UEFI PI infrastructure, execution on a bare metal machine exposes the rich set of hardware resources, including the memory management unit (MMU) and protection hardware. One set of hardware controls on Intel ® Architecture includes the Virtualization Extensions. As noted in the work on the STM, there needs to be a host virtualization agent to activate the peer monitor. One example of this is the Firmware Resource Monitor (FRM), which is a simple virtualization agent built using EDK II technology and launched from a UEFI PI environment. This paper will provide more background and details on the construction of the FRM.

Prerequisite

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

Table of Contents

Overview

Introduction to VMM

A Virtual Machine Monitor (VMM) acts as a host and has full control of the processor(s) and other platform hardware. A VMM presents guest software with an abstraction of a virtual processor and allows it to execute directly on a logical processor. A VMM is able to retain selective control of processor resources, physical memory, interrupt management, and I/O. Each virtual machine (VM) is a guest software environment that supports a stack consisting of operating system (OS) and application software. [IA32 Manual]

Introduction to EDKII

EDKII is open source implementation of UEFI PI-based firmware which can boot multiple UEFI-aware operating systems. The main responsibility of UEFI BIOS is to launch operating system. Since UEFI provides a rich execution environment, it is also possible to launch a VMM in UEFI BIOS. The early VMM launch capability may provide new usage, for example, initialize SMI Transfer Monitor (STM), prepare resource isolation or resource partitioning for each guest.

In [STM2], we introduced a full open source test VMM – Firmware Resource Monitor (FRM) to launch STM. In this paper, we will use FRM as example to demonstrate how to write VMM in UEFI.

Summary
This section provided an overview of VMM and EDKII.

VMM type in firmware

There are many papers that describe a VMM [VMM1][VMM2], also known as a Hypervisor. There are 2 types of hypervisors:

- Type-1: native or bare-metal hypervisors
- Type-2: hosted hypervisors.

Type 2 VMM

A Type 2 VMM runs on host operating system and provides for a an entireenvironment emulation to guest software. The host and guest systems do not have a common ISA.

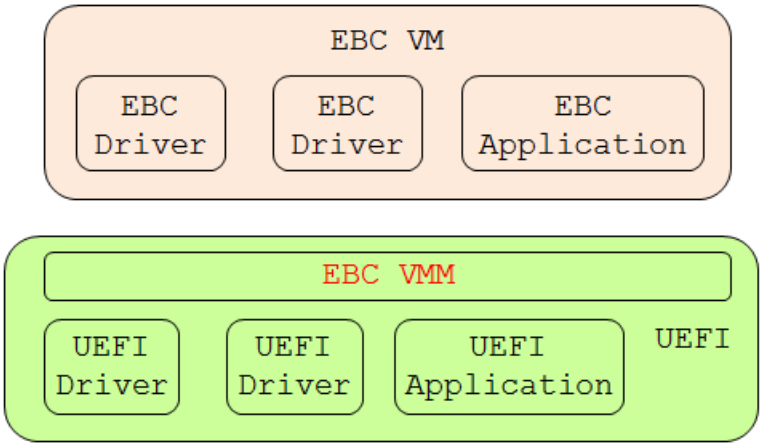

Figure 1 EBC VMM

In UEFI, the EFI Byte Code (EBC) is an example of a type 2 VMM. See [UEFI] Chapter 21, EFI Byte Code Virtual Machine. The EBC VMM runs inside UEFI environment, and EBC driver runs as EBC VM. EBC VMM is standard UEFI driver, using X86 instruction set. EBC drivers must be compiled by using the EBC compiler, and using the EBC ISA as defined in the UEFI specification.

Other Type 2 VMM's include KVM, wherein the whole OS like Linux hosts an alternate OS.

Type 1 VMM

A Type 1 VMM runs on the host machine directly in order to control the hardware, and virtual machine instances fit on top. The VMM runs in the most highly privileged mode, while all the guests run with lesser privileges. Both guests and the host use the same Instruction Set Architecture (ISA) as the underlying hardware.

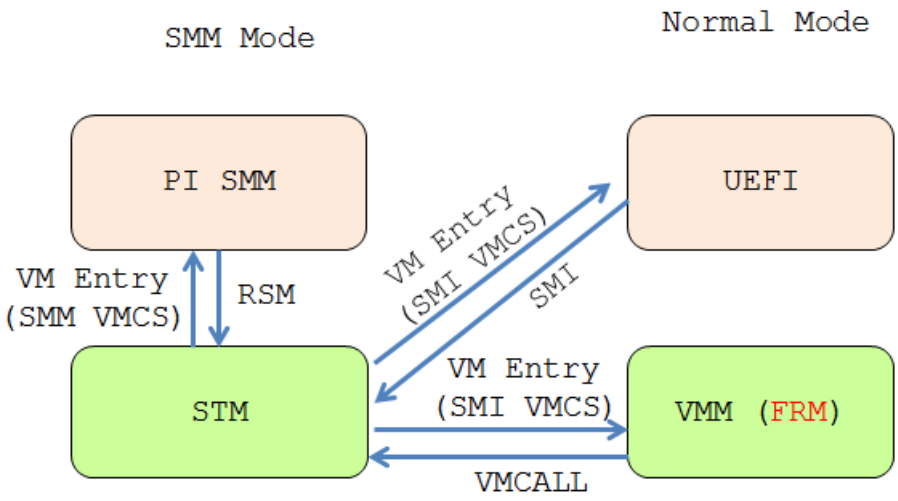

Figure 2 FRM VMM

In UEFI, the FRM (@ <STM_SOURCE>\Test\FrmPkg) is an example of type 1 VMM. The STM is an example of type 1 VMM for PI SMM. Both of them use X86 ISA.

Type 0 VMM

Today's Type 1 hypervisors are typically integrated with a special-purpose host OS and additional service applications. The Type Zero hypervisor is a new concept that is even smaller than Type 1. Type Zero is a bare-metal architecture that removes the need for a supporting OS. [VMM3]

If we adopt the above definition, the FRM can also be considered as a type 0 VMM because it is a tiny VMM that only provides minimal functions, and it does not have an OS inside.

Summary

This section introduces the VMM classification in UEFI. In next chapters, we only focus on the FRM – a type 1 VMM, or a type 0 VMM if we use the new definition.

Launch of the FRM

FRM

FRM is a tiny bare-metal VMM launched in the UEFI environment. It exists until the OS boot and shutdown. We used FRM to test the STM capability because the STM needs to be started or shut down by a VMM.

The FRM is self-contained execution environment. The FRM should not use any UEFI services because the UEFI services disappear in OS runtime environment.

The FRM source is located at @<STM_SOURCE>\Test\FrmPkg\Core. It uses the EDKII library, and it is built in the EDKII build environment. The FRM does not use any UEFI services, such as UEFI boot services, runtime services, or any UEFI protocol. If the FRM needs any information, the FRM loader collects this infromation and passes it into the FRM entrypoint.

FRM loader

The FRM loader is a UEFI driver @<STM_SOURCE>\Test\FrmPkg\LoaderDriver. It finds the FRM bin from flash, allocates reserved memory space, and finally calls the FRM entry point.

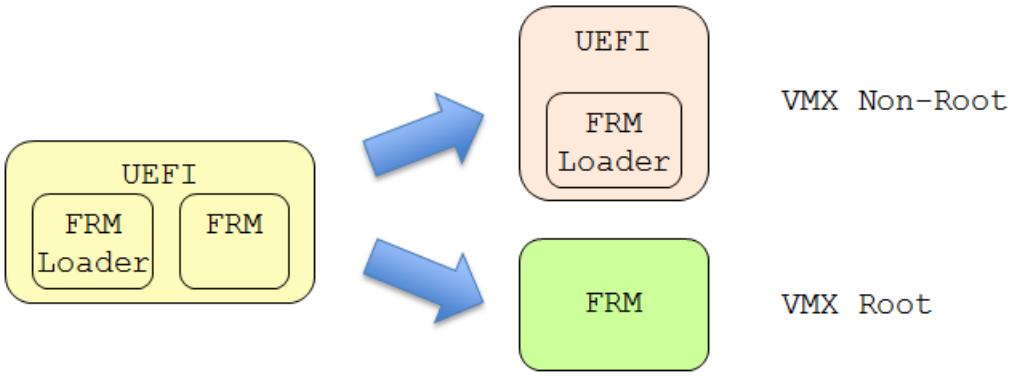

Figure 3 FRM and FRM loader

During BIOS build, both the FRM and FRM loader are included in the UEFI BIOS. During boot, the FRM loader is loaded by the PI DXE core into UEFI Boot services memory, and the FRM loader executes in VMX Non-Root mode. Then the FRM is loaded by the FRM loader into reserved memory, and the FRM executes in VMX root mode.

Figure 4 shows final memory location of FRM and FRM loader. The GREEN box is memory used by the hypervisor. The RED box is memory used by the normal OS, driver, or application.

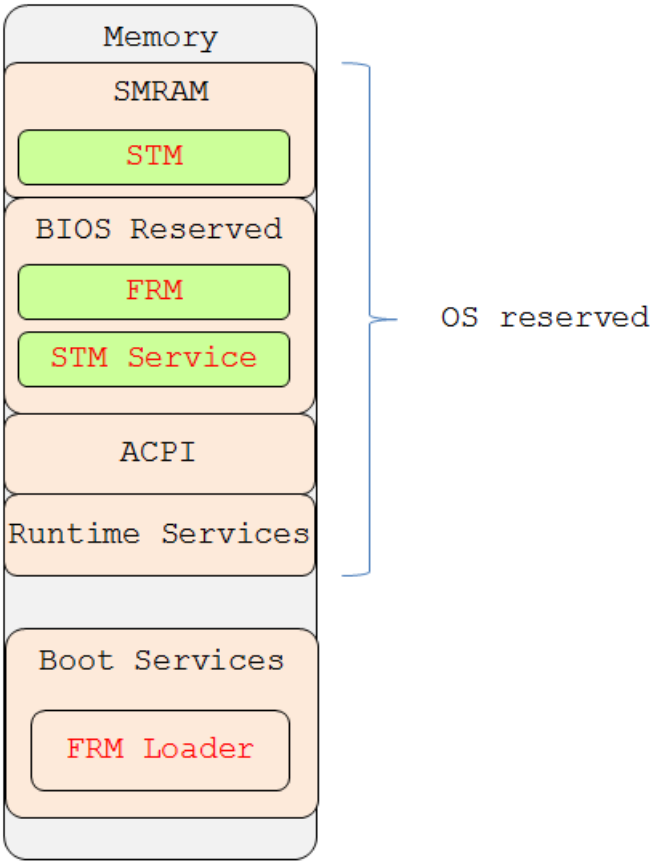

Figure 4 System memory layout

FRM - FRM loader interface

The FRM_COMMUNICATION_DATA (@<STM_SOURCE>\Test\FrmPkg\Include\FrmCommon.h) is the interface between the FRM and the FRM loader.

- **HighMemoryBase/HighMemorySize**: FRM need to know which BIOS reserved region can be used as FRM Heap. NOTE: the FRM Stack is also allocated inside of FRM heap.
- **LowMemoryBase/LowMemorySize**: the FRM needs to know which <1M region can be used to wake up Application Processors (Aps). After APs are waken up, the APs will be in high memory, and low memory is never used any more.
- **ImageBase/ImageSize**: the FRM needs to know the FRM image location because the FRM needs protect itself via Extended Page Tables (EPT).
- **AcpiRsdp**: The FRM loader needs to find the ACPI table from a UEFI configuration table and pass the ACPI table address into the FRM because FRM does not know where ACPI table is located.
- **SmMonitorServiceProtocol**: The FRM locates the SmMonitorServiceProtocol for the FRM because the FRM needs to consume these services to manage the STM, including how to start the STM, stop the STM, and protect resources.

- **SmMonitorServiceImageBase/SmMonitorServiceImageSize**: The FRM also needs to know the SmMonitorService image location so that the FRM can protect such services via EPT. These services will be called by the FRM at runtime.

FRM initialization

The entry point of FRM is at _ModuleEntryPoint() (@<STM_SOURCE>\Test\FrmPkg\Core\Init\FrmInit.c) The initialization code is @<STM_SOURCE>\Test\FrmPkg\Core\Init directory.

In the entry point, the FRM will perform the following items step-by-step:

1) **InitHeap**() – This function sets up the FRM heap.

2) **InitBasicContext**() – This function collects general information – the CPU number from ACPI MADT table; the PCI Express memory information from ACPI MCFG table; ACPI Timer, ACPI Reset Port, and ACPI Power Management Control information from ACPI FADT table. It also records current information, such as CR0, CR3, CR4, GDT, IDT.

3) **InitHostContext**() – This function calls InitHostContextCommon() and InitHostContextPerCpu() for the Boot Strap Processor (BSP). Then it wakes up the APs and waits for the Aps to finish.
    a. InitHostContextCommon() – This function creates host page tables, host GDT, host IDT, and initializes the VmExit handlers. It also creates the host stack and allocates a VMCS for each CPU.
    b. InitHostContextPerCpu() – This function writes CR3, CR0, CR4 and executes VMXON on the current CPU.

4) **InitGuestContext**() – This function calls InitGuestContextCommon() and InitGuestContextPerCpu() for the BSP.
    a. InitGuestContextCommon() – This function prepares the guest VMCS context, MsrBitmap, EPT page table, IoBitmap, VMX timer. Then it creates a temporary guest stack and allocates a VMCS for each guest CPU.
    b. InitGuestContextPerCpu() – This function loads the guest VMCS and sets up the VMCS host field, guest field, and control field.

5) **LaunchGuest**() – This function is the last step. It calls LaunchGuestAp() and LaunchGuestBsp().
    a. LaunchGuestAp() – The BSP just sets up a flag to tell all the APs to launch. Once this flag is set, ApWakeupC() continues running. ApWakeupC() calls InitGuestContextPerCpu() for the AP to set up the guest context, and then it launches control into the GuestApEntrypoint().
    b. LaunchGuestBsp() – The BSP sets up a jump buffer jump point and launches control into GuestEntrypoint(). Then GuestEntrypoint() will long jump back into the original LaunchGuestBsp() function and return to original function from the stack.

Figure 5 shows the final memory layout of FRM.

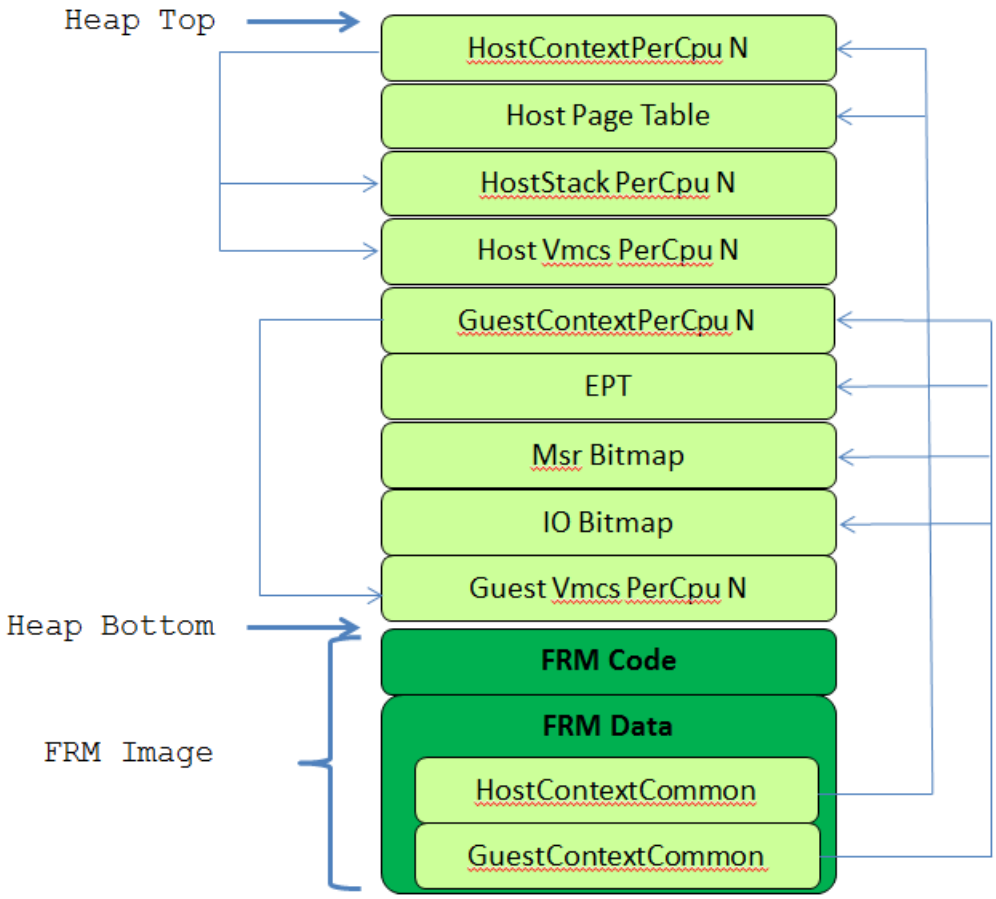

Figure 5 FRM memory layout

Summary

This section introduces the FRM memory layout and how to initialize the FRM.

FRM runtime

The FRM runtime code is @<STM_SOURCE>\Test\FrmPkg\Core\Runtime directory.
The FRM tries to eliminate unnecessary VmExits, although the IA32 SDM defines some mandatory VmExit events.

FRM VmExit entry point

The host entry point of FRM is HostEntrypoint() (@<STM_SOURCE>\Test\FrmPkg\Core\Runtime\x64\VmExit.asm), which is the first instruction executed when a VmExit event happens. It saves the register context and jumps to the FrmHandler() (@<STM_SOURCE>\Test\FrmPkg\Core\Runtime\FrmHandler.c)

The FrmHandler() reads the VMCS_32_RO_EXIT_REASON_INDEX and dispatches control to the VmExit handler (mFrmHandler) based upon the VmExit reason. Then it calls VmResume in order to pass control back into the VM.

FRM VmExit handler

Each VmExit handler is registered in InitFrmHandler(), which is called by InitHostContextCommon(). Here we only introduce some important handlers:

- **VmExitReasonCrAccess**: This handler is intended to handle CR register access. The FRM assumes a CPU has UnrestrictedGuest feature, so that real mode guest execution is supported. The FRM passes through most CR access. For IA32 PAE mode guest, the FRM needs to run a special function Ia32PAESync() to sync VMCS_64_GUEST_PDPTEx_INDEX.
- **VmExitReasonEptViolation/VmExitReasonEptMisConfiguration**: These 2 handlers are for EPT. The FRM can configure EPT page table in EptInit() to prevent guest accessing FRM memory. If the EPT page table itself is wrong, EptMisConfiguration VmExit will be triggered. This should not happen, so FRM just DeadLoop()s for this case. If the guest violates the EPT rule, EptViolation VmExit will be triggered. This might happen because the FRM cannot predict guest OS behavior. In this case the FRM moves the RIP forward to skip the instruction.
- **VmExitReasonIoInstruction**: This handler is for IO access. The FRM allocates the IO bitmap and registers an ACPI PM Control port handler and an ACPI reset port access handler so that the FRM knows when the OS wants to shut down or reset the system, and it prepares the FRM tear down before that.
- **VmExitReasonCpuid**: This handler is for the CPUID instruction. CPUID is a mandatory VmExit handler. The FRM passes through the instruction.
- **VmExitReasonRdmsr/VmExitReasonWrmsr**: These 2 handlers are for MSR access. The FRM may need to observe OS behavior on how to change system state by MSRs and change the VMCS area correspondingly. Examples of these MSR's include IA32_EFER_MSR, IA32_SYSENTER_CS_MSR, IA32_SYSENTER_ESP_MSR, IA32_SYSENTER_EIP_MSR, IA32_FS_BASE_MSR, and IA32_GS_BASE_MSR.

- **VmExitReasonInit/VmExitReasonSipi**: These 2 handlers are for AP wakeup. Once the FRM host receives an Init VmExit, the FRM puts the guest into wait for SIPI guest active state. When the FRM host receives a Sipi VmExit, the FRM initializes the AP state defined according to the IA32 SDM in SetVmcsGuestApWakeupField(), @<STM_SOURCE>\Test\FrmPkg\Core\Runtime\ApHandler.c.
- **VmExitReasonInvd/VmExitReasonWbinvd**: These 2 handlers are for cache management. The FRM passes through these instructions.
- **VmExitReasonVmxPreEmptionTimerExpired**: This handler is for the VMX pre-emptive timer. The FRM uses the VMX timer for AP sync up on shutdown. We will discuss this capability in more detail in the next section.
- **VmExitReasonExternalInterrupt/VmExitReasonInterruptWindow**: These 2 handlers are for interrupts. The FRM passes through all guest interrupts. As such, these handlers should not be invoked.
- **VmExitReasonVmCall**: This handler is for VMCALL support. The current FRM does not support any VMCALL, but developers can add such support in VmcallHandler(), @<STM_SOURCE>\Test\FrmPkg\Core\Runtime\VmcallHandler.c.

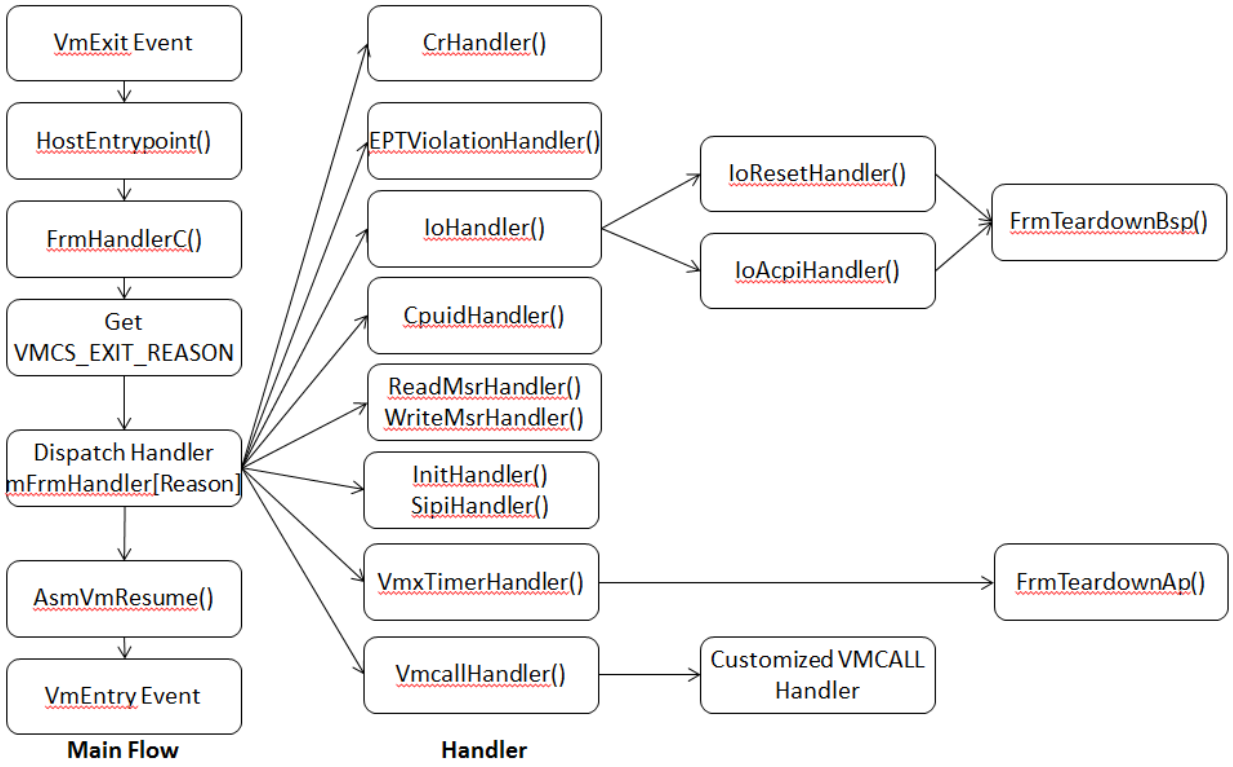

Figure 6 FRM runtime handler

FRM tear down

The FRM registers ACPI PM Control port and ACPI reset port IO access handlers so that the FRM knows when the OS wants to shut down or reset the system, and prepares FRM tear down before that. Once one CPU receives such an access request in IoResetHandler()

(@<STM_SOURCE>\Test\FrmPkg\Core\Runtime\IoResetHandler.c) or IoAcpiHandler() (@<STM_SOURCE>\Test\FrmPkg\Core\Runtime\IoAcpiHandler.c), it calls FrmTeardownBsp() (@<STM_SOURCE>\Test\FrmPkg\Core\Runtime\FrmTeardown.c).

FrmTeardownBsp() sets up a global flag (mReadyForTeardown) and VMXOFF for this CPU. In the VMX timer handler, VmxTimerHandler() calls FrmTeardownAp() to check if there is a tear down request (mReadyForTeardown). If there is a pending tear down request, FrmTeardownAp() issues VMXOFF for this CPU and sets the finished flag (mTeardownFinished). Only after FrmTeardownBsp() receives mTeardownFinished from all CPUs, the FRM will forward the IO request to actual hardware to perform the shut down or reset system action.

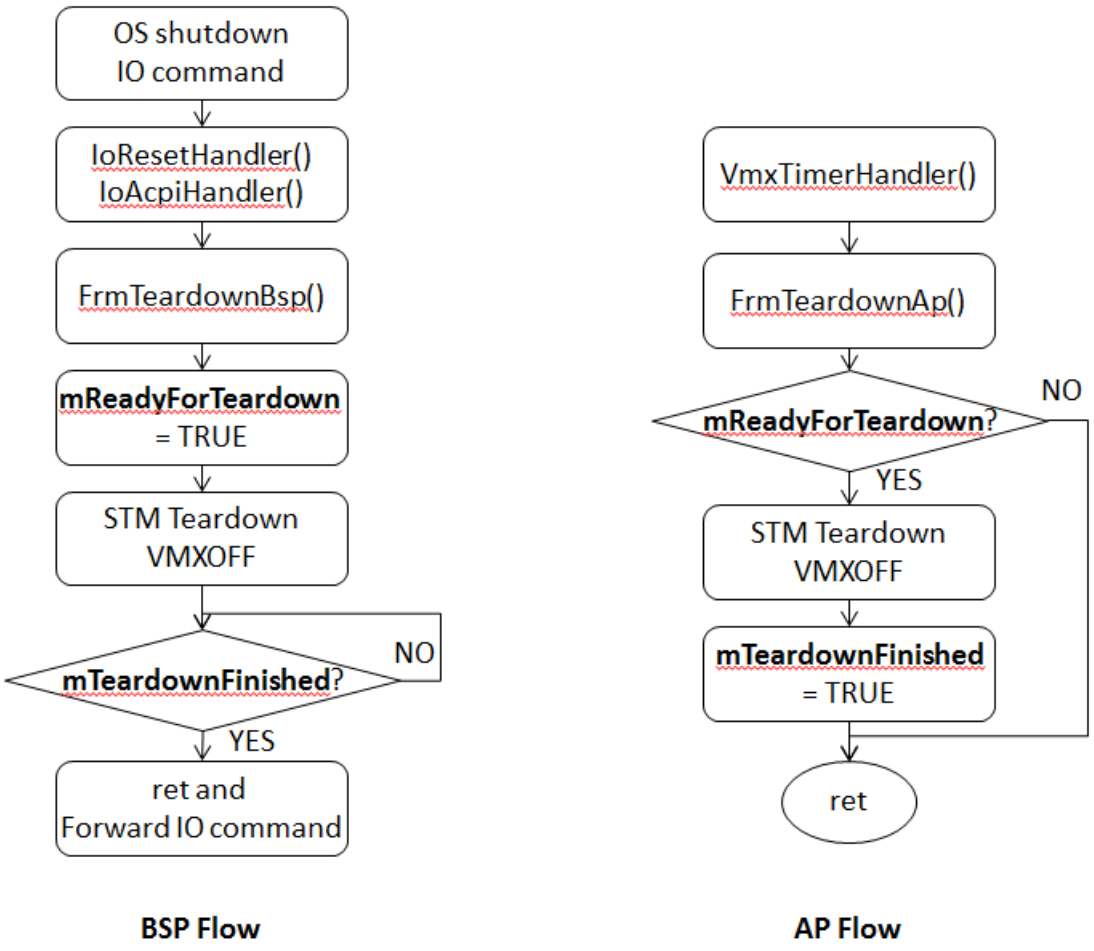

Figure 7 FRM tear down

Summary

This section introduces the FRM runtime flow and how to tear down the FRM.

FRM as a monitor

The main purpose of the FRM is to demonstrate how to launch the SMI Transfer Monitor (STM) by having the FRM act as a tiny VMM. The FRM can also provide some additional features on resource protection or resource isolation.

Resource Protection

The FRM can be used to protect system resources including but not limit to physical memory, memory mapped IO (MMIO), IO, PCI configuration space, CPU Machine Specific Register (MSR).

- Memory protection (physical memory, or MMIO) can be done via EPT. If a guest accesses a protected memory resource, an EptViolation VmExit will be triggered.
- IO protection can be done via the IoBitmap. If a guest accesses a protected IO resource, an Io VmExit will be triggered.
- PCI configuration space protection can be done by IO protection (0xCF8/0xCFC IO port access), and memory protection (PCI express MMIO access). If a guest accesses a protected PCI resource, an EptViolation VmExit or an Io VmExit will be trigged.
- MSR protection can be done via the MsrBitmap. If a guest accesses a protected MSR, a ReadMsr VmExit or a WriteMsr VmExit will be triggered.

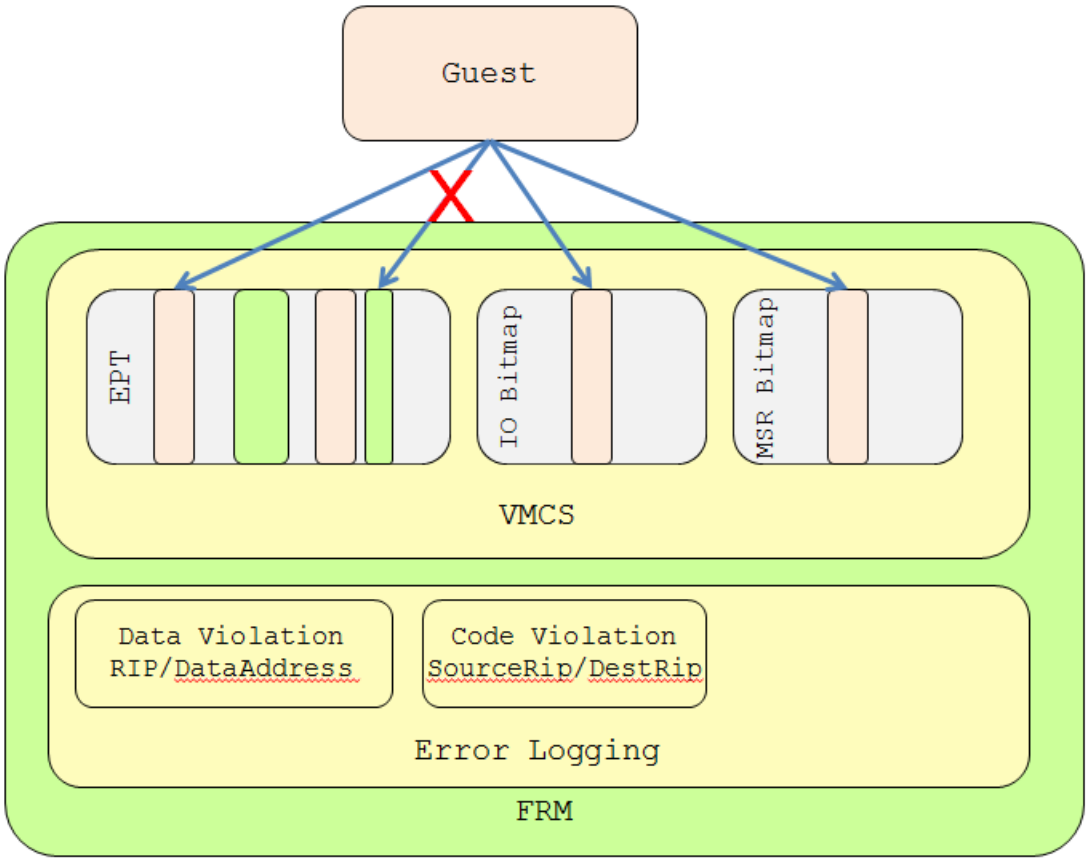

Figure 8 FRM as Monitor

In order to support audit and error analysis, the FRM can also record the violation in the event log area.

Please refer to [STM] to learn more about how the event log is supported.

Resource Partitioning

Besides resource protection, the FRM can also support resource partitioning.
Take the figure below as example. Therein you see 2 guests existing on a system. Each guest may own different resources. The FRM may construct different guest contexts for each guest and assign CPU resources, interrupt resources, and PCI device resources to each guest based on the need.

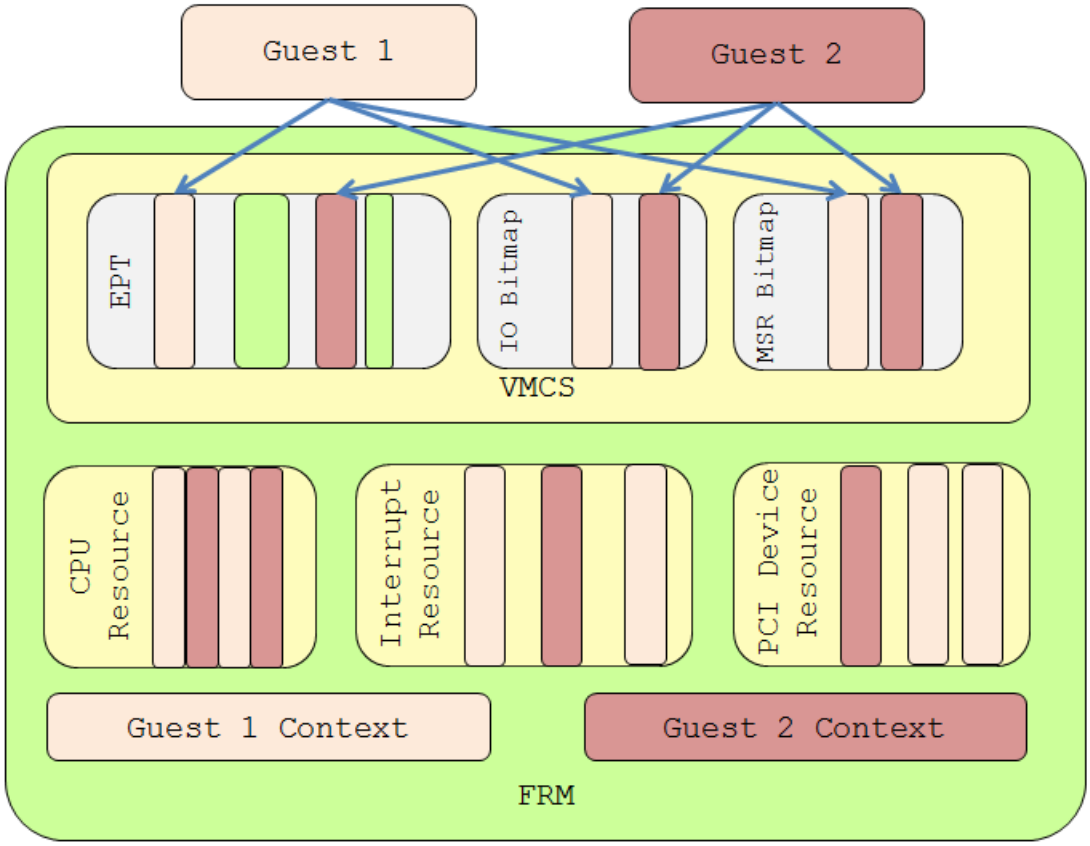

Figure 9 Resource Partitioning

The configuration information can be stored as file on UEFI file system or as standard UEFI variable. During boot, the FRM can read the configuration and assign resources to a different guest.

Summary
This section introduces the FRM as a monitor and typical usages thereof.

Other Considerations

S3 support

During normal S3 suspend, the OS writes a waking vector in the ACPI FACS table, and then the OS writes S3 to the PM Control IO port. During a normal S3 resume, the BIOS performs minimal silicon configuration restoration, and then jumps to the OS waking vector in the FACS.

If the FRM exists, the FRM records the OS waking vector and replaces it with the FRM waking vector in the S3 suspend path. During an S3 resume, the BIOS jumps to the FRM waking vector. Then the FRM can do initialization for S3. After this, the FRM can jump to the OS waking vector to continue the S3 resume.

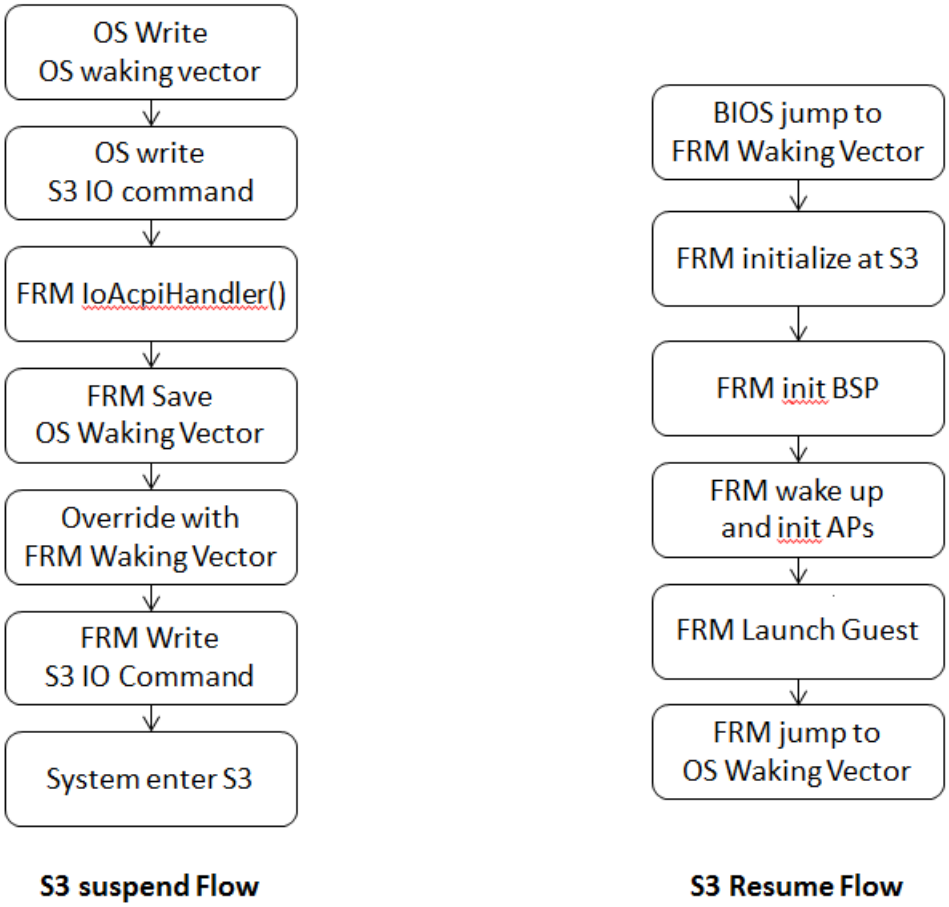

Figure 10 S3 Consideration

VT-d support

The open source FRM does not support VT-d yet. However, VT-d must be enabled if the FRM is used for resource protection because FRM must resist DMA attack. [VTd] provides detailed

information on VT-d hardware/software design. [VT-d in UEFI] provides a sample on how to setup VT-d in a UEFI BIOS.

TXT support

The open source FRM does not support TXT. TXT is designed to support a dynamic root of trust. If we trust BIOS, which is the static root of trust, the TXT is not needed. [MLE] provides detailed information on TXT hardware/software design.

STM support

The FRM is designed to support the STM. It is a test VMM for the STM. Please refer to [STM] for more details on how to launch the STM.

Summary

This section introduces other parts of the FRM.

Conclusion

FRM is test VMM for STM. The FRM can be one implementation of a tiny VMM. It can be launched in UEFI environment.

Glossary

DEP – Data Execution Protection.

EBC – EFI Byte Code. See [UEFI Specification].

EPT – Extended Page Table. See [IA32 SDM]

IPL – Initial program loader.

MSEG – Monitor Segment. A special SMRAM for STM.

MLE – Measured Launched Environment

NX – No Execution. See DEP.

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.

SMRR – System Management Range Register.

STM – SMI Transfer Monitor.

TXT – Intel Trusted Execution Environment

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

XD – Execution Disable. See DEP.

VMCS – Virtual Machine Control Structure. See [IA32 SDM]

VT – Virtualization Technology. See [IA32 SDM]

WP – Write Protection.

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

[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

[IA VMM] John Scott Robin & Cynthia E. Irvine , "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor", 2000, http://www.usenix.org/events/sec2000/full_papers/robin/robin.pdf

[MEMORY] Yao, Zimmer, Fleming, "A Tour Beyond BIOS Memory Practices in UEFI", June 2015 https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Practices_with_UEFI.pdf

[MLE] Intel® Trusted Execution Technology (Intel® TXT) Software Development Guide - Measured Launched Environment Developer's Guide http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf

[MONITOR] Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, US Air Force Electronic Systems Division, 1973 http://csrc.nist.gov/publications/history/ande72.pdf

[SMM01] Oleksandr Bazhaniuk, et al, "A New Class of Vulnerabilities in SMI Handlers", 2015, https://cansecwest.com/slides/2015/A%20New%20Class%20of%20Vulnin%20SMI%20-%20Andrew%20Furtak.pdf

[SMM02] Rafal Wojtczuk, et al, "Attacks on UEFI Security", 2015, https://bromiumlabs.files.wordpress.com/2015/01/attacksonuefi_slides.pdf

[SMM03] Corey Kallenberg, et al, "How many million BIOSes world you like to infect?", 2015, http://legbacore.com/Research_files/HowManyMillionBIOSWouldYouLikeToInfect_Full2.pdf

[SMM04] Loïc Duflot, et al, "System Management Mode Design and Security Issues", 2010, http://www.ssi.gouv.fr/uploads/IMG/pdf/IT_Defense_2010_final.pdf

[SMM05] Rafal Wojtczuk, et al, "Attacking Intel BIOS", 2009, http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf

[SMM06] "ASUS Eee PC and other series: BIOS SMM Privilege Escalation Vulnerabilities", 2009, http://www.securityfocus.com/archive/1/505590

[SMM07] Dick Wilkins, "UEFI Firmware –Securing SMM", 2015, http://www.uefi.org/sites/default/files/resources/UEFI_Plugfest_May_2015%20Firmware%20-%20Securing%20SMM.pdf

[SMM08] Rafal Wojtczuk, et al, "Attacking Intel® Trusted Execution Technology", 2009, http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf

[SMM09] Jiewen Yao, Zimmer Vincent, "A_Tour_Beyond_BIOS_Launching_Standalone_SMM_Drivers_in_PEI_using_the_EFI_Developer_Kit_II", 2015, http://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Creating_the_Intel_Firmware_Support_Package_Version_1_1_with_the_EFI_Developer_Kit_II.pdf

[SMM10] Jiewen Yao, Zimmer Vincent, "A_Tour_Beyond_BIOS_Supporting_SMM_Resource_Monitor_using_the_EFI_Developer_Kit_II", 2015, https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Supporting_SMM_Resource_Monitor_using_the_EFI_Developer_Kit_II.pdf

[SMM11] Steve Weis, "Protecting Data In Use From Firmware And Physical Attacks", 2014, https://www.blackhat.com/docs/us-14/materials/us-14-Weis-Protecting-Data-In-Use-From-Firmware-And-Physical-Attacks.pdf

[STM] STM User Guide - https://firmware.intel.com/content/smi-transfer-monitor-stm

[STM2] Jiewen Yao, Zimmer Vincent, "A_Tour_Beyond_BIOS_Launching_STM_to_Monitor_SMM_in_EFI_Developer_Kit_II", 2015, https://firmware.intel.com/content/smi-transfer-monitor-stm

[TrustedPlatform] David Grawrock, "Dynamics of a Trusted Platform: A Building Block Approach", 2009, http://www.amazon.com/Dynamics-Trusted-Platform-Building-Approach/dp/1934053171

[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

[VMM1] Gerald J. Popek andRobert P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures", Communications of the ACM, 1974

[VMM2] J. E. Smith and Ravi Nair, "Virtual Machines: Architectures, Implementations and Applications" Morgan Kaufmann Publishers, 2004

[VMM3] Will Keegan, "The Rise of the Type Zero Hypervisor". Embedded Innovator. 2014

[VT-d] Intel Virtualization Technology for Directed I/O specification, Rev 2.3
http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/vt-directed-io-spec.html

[VT-d in UEFI] Jiewen Yao, Vincent Zimmer, "A Tour beyond BIOS Using Intel VT-d for DMA Protection in UEFI BIOS"
https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Using_Intel_VT-d_for_DMA_Protection.pdf

Authors

Jiewen Yao (jiewen.yao@intel.com) is an EDKII BIOS architect, EDKII TPM2 module maintainer, ACPI/S3 module maintainer, and FSP package owner with Software and Services Group (SSG) at Intel Corporation. Jiewen is a BIOS security researcher, co-invented a VMM in BIOS in patent US007827371 using PI DXE infrastructure.

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