White Paper

A Tour Beyond BIOS Memory Map and Practices in UEFI BIOS

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

Matt Fleming

Jan 31, 2016

Executive Summary

This paper introduces the memory map security practices in UEFI BIOS.

Prerequisite
This paper assumes that audience has basic EDKII/UEFI firmware development experience.

Table of Contents

Overview

The main job of BIOS is to initialize the platform hardware and report information to a generic operating system (OS). The memory map is one of the most important pieces of information. The operating system can only load a kernel, driver or application in the right place if it knows how memory is allocated.
In [UEFI Memory Map], we introduced the memory map design in UEFI BIOS, and saw how a typical platform reports the memory map to an OS. In this paper we will discuss how to enhance the memory map reporting and provide security practice for memory protection to harden platforms.

Introduction to memory map

"Memory map" here means to a structure of data (which usually resides in memory itself) that indicates how memory is laid out. "Memory" here means the storage which can access by processor directly.
Typical memory map includes the storage accessed by processor directly.

1) Physical memory. E.g. main memory, SMRAM (SMM stolen memory), integrated graphic stolen memory.
2) Memory Mapped IO. E.g. PCI-Express Memory Mapped Configruation Space, PCI device MMIO BAR, CPU Local APIC, legacy video buffer, memory mapped flash device, TPM memory map configuration space.

Memory map does not includes:

1) Cache. E.g. CPU internal cache
2) Disk. E.g. ATA hard driver, SCSI hard drive, CDROM/DVDROM, USB storage device.

Summary
This section provided an overview of the UEFI memory map.

Memory Map – Hardware Perspective

In this chapter, we will discuss the memory map from hardware perspective. We will use typical Intel x86 platform as example.

System Memory

The system memory means the main dynamic random access memory (DRAM). It can be classified as below:

1) Legacy region less-than 1MiB
2) Main memory between 1MiB and 4GiB
3) Main memory great-than 4GiB

Legacy Region is for legacy OS or device consideration. (See figure 1) It is divided into following area:

- 0–640KiB (0-0xA0000): DOS Area. The normal DRAM is for legacy OS (DOS) or boot loader usage.
- 640–768KiB (0xA0000-0xC0000): SMRAM/Legacy Video Buffer Area. This region can be configured as SMM memory, or mapped IO for legacy video buffer.
- 768–896KiB (0xC0000-0xE0000): Expansion Area for legacy option ROM. This region is DRAM and could be locked to be read-only.
- 896KiB–1MiB (0xE0000-0x100000): System BIOS Area. This region could be DRAM or could be configured as memory IO to flash region.

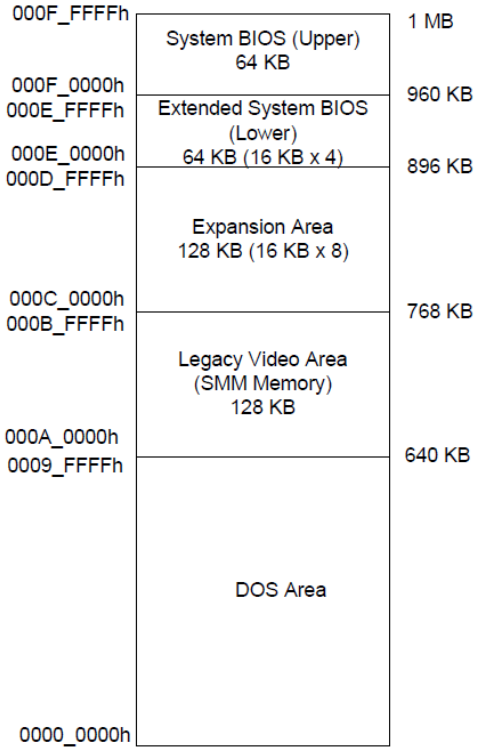

Figure 1

The main memory between 1MiB and 4GiB is for traditional OS. (See figure 2) However, memory region just below 4GiB is occupied by memory-mapped IO, like flash, CPU APIC, TPM. There is a dedicated register Top of Low Usable DRAM (TOLUD) to indicate the bar. Memory between 1MiB and TOLUD is divided into following area:

- Normal DRAM: Used by OS.
- ISA Hole (15MiB–16MiB): Optional feature. If enabled, the normal DRAM is disabled by opening the hole.
- Protected Memory Range (PMR) below 4GiB: Programmable optional feature. If enabled, this DRAM cannot be accessed by DMA.
- DRAM Protected Range (DPR): Optional feature. If enabled, this DRAM cannot be accessed by DMA.
- Top memory for SMM: TSEG (Top Segment) SMRAM. If enabled, this DRAM can be access if and only if CPU in SMM mode.
- Top memory for SMM: TSEG (Top Segment) SMRAM. If enabled, this DRAM can be access if and only if CPU in SMM mode.
- Top memory for integrated graphic device (IGD): IGD stolen memory. If enabled, this DRAM is served for IGD.
- Top memory for Intel Manageability Engine (ME): ME stolen memory. If enabled, this DRAM is served for ME. If the total memory is greater-than-4GiB, the ME stolen memory will be greater-than-4Gib. So this region is always on top of physical memory and it is not covered by TOLUD.

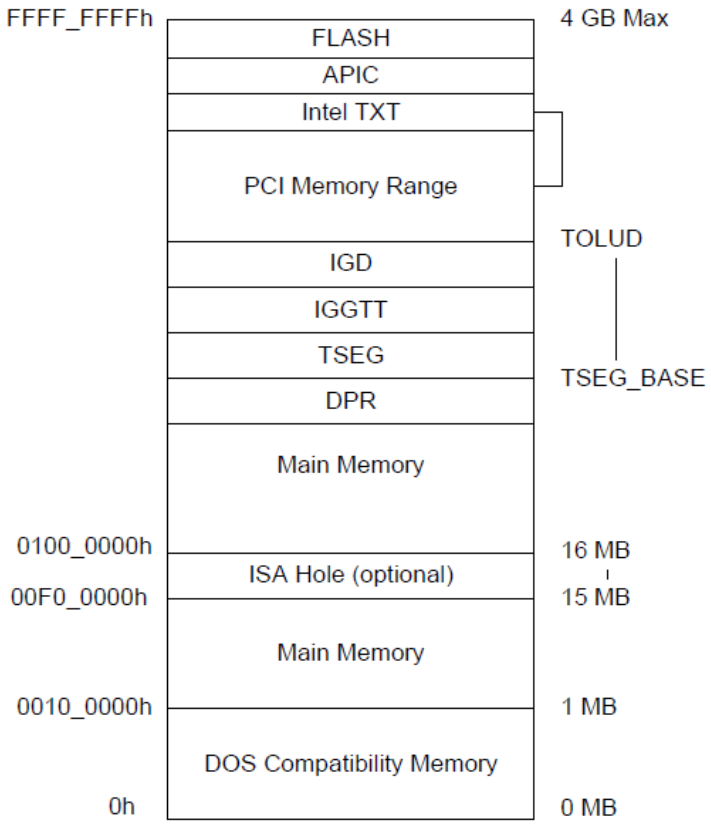

Figure 2

The main memory greater-than 4GiB is also for traditional OS. (See figure 3) If there is DRAM between TOLUD and 4GiB, this memory can be reclaimed by chipset, to map to greater-than 4GiB. There is a set of dedicated register Reclaim BASE/SIZE to indicate the remap action. As a result, the physical memory view (from DRAM controller) might be less-than-4GiB, while the system memory view (from Host CPU) could be greater-than-4GiB. There is a dedicated register Top of Upper Usable DRAM (TOUUD) to indicate the bar for highest system memory. Memory greater-than-4GiB has below area:

- Normal DRAM: Used by OS.
- Protected Memory Range (PMR) above 4GiB: Programmable optional feature. If enabled, this DRAM cannot be accessed by DMA.
- Top memory for Intel Manageability Engine (ME): ME stolen memory. If enabled, this DRAM is served for ME. This region is always on top of physical memory and it is not covered by TOUUD.

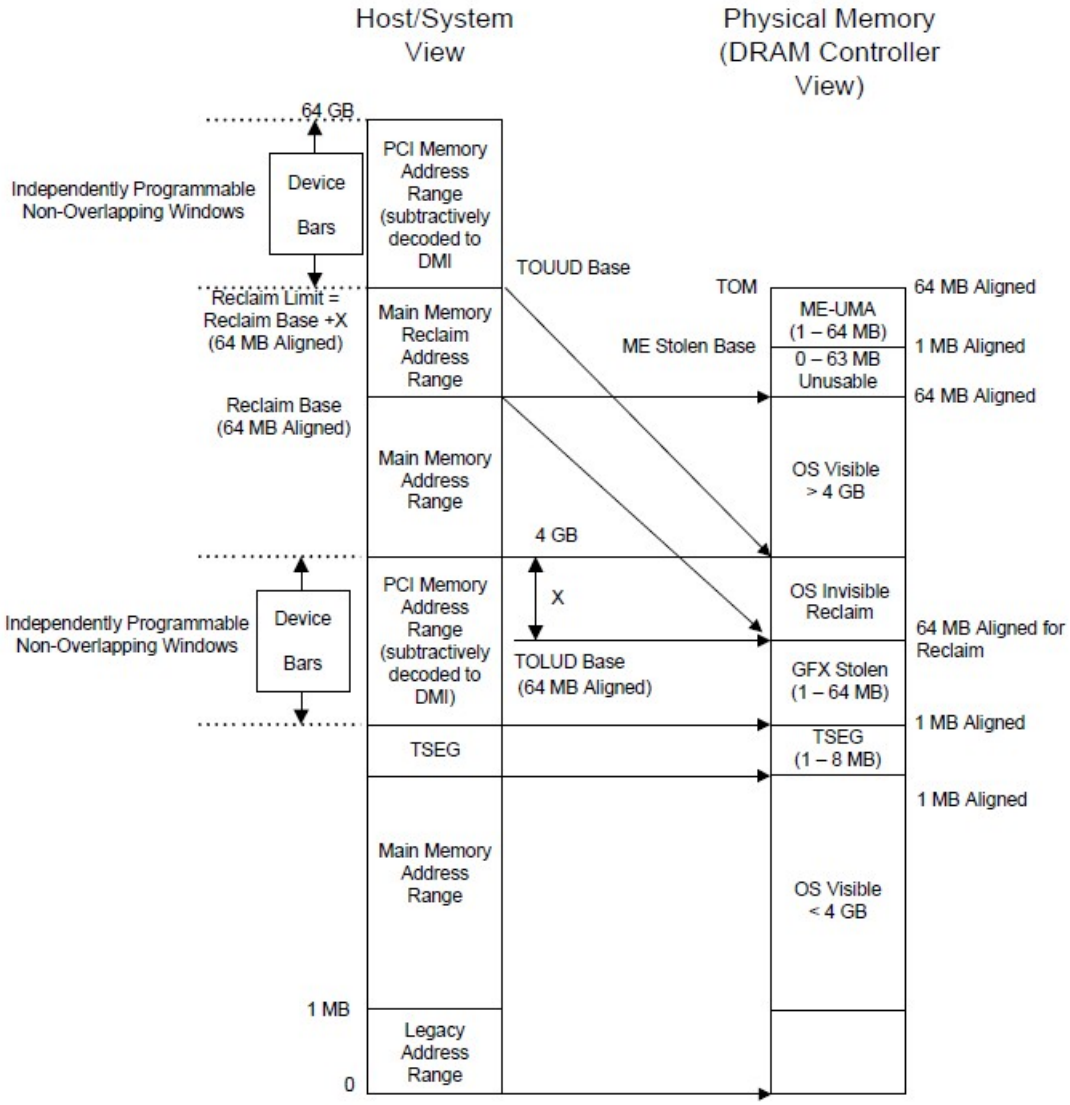

Figure 3

A typical hardware memory map on an X86 platform is like below (2GiB):

```
============================================================================
Found 00000000000A0000h bytes at 0000000000000000h (DOS region)
Found 0000000000020000h bytes at 00000000000A0000h (SMRAM AB-SEG)
Found 000000007A6FF000h bytes at 0000000000100000h (main memory less-than-4GiB)
Found 0000000000001000h bytes at 000000007A7FF000h (Reserved for Intel PTT)
Found 0000000000800000h bytes at 000000007A800000h (DPR)
Found 0000000001000000h bytes at 000000007B000000h (SMRAM T-SEG)
Found 0000000002800000h bytes at 000000007C000000h (IGD Stolen)
      0000000000800000h bytes at 000000007E800000h (Hole for alignment - reclaimed)
      0000000001000000h bytes at 000000007F000000h (Intel ME Stolen - not reported)
Found 0000000000800000h bytes at 0000000100000000h (main memory greater-than-4GiB)
Total: 2048MiB
============================================================================
```

Memory Mapped IO

The memory mapped IO provides methods of performing input/output between the CPU and peripheral devices. (See figure 4) In a typical x86 platform, there are BIOS (flash area), MSI interrupts, CPU local APIC, I/O APIC, PCI Express configuration space, TPM device space.

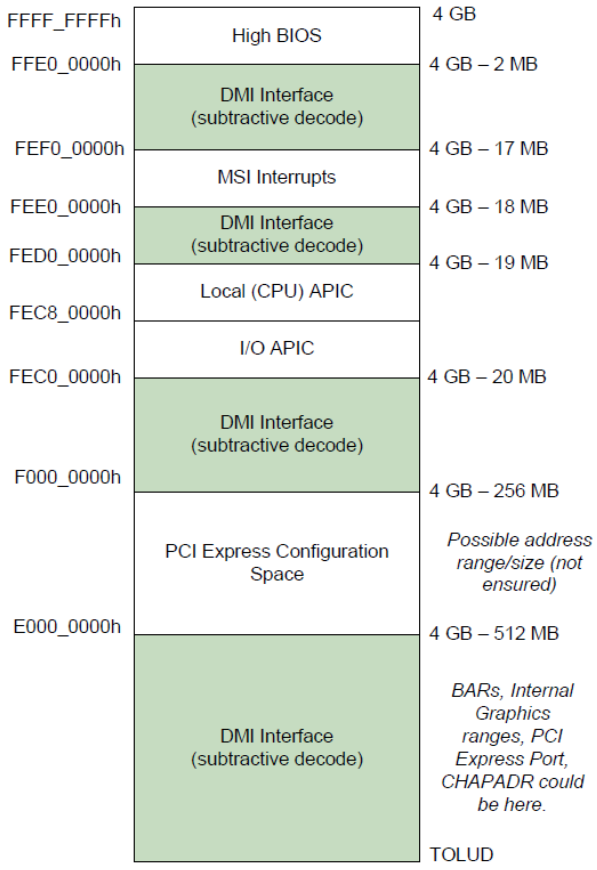

Figure 4

This paper will focus more on system memory. Since the memory-mapped IO is platform specific, please refer to chipset document for detail.

A typical hardware memory map on an X86 platform is like below:

```
================================================================================
PCI Express: 0xE0000000 - 0x10000000
IO APIC:     0xFEC00000 - 0x1000
HPET:        0xFED00000 - 0x400
MCH BAR:     0xFED10000 - 0x8000
DMI BAR:     0xFED18000 - 0x1000
EGP BAR:     0xFED19000 - 0x1000
RCBA:        0xFED1C000 - 0x4000
Intel TXT:   0xFED20000 - 0x20000
TPM:         0xFED40000 - 0x5000
Intel PTT:   0xFED70000 - 0x1000
Intel VTd:   0xFED90000 - 0x4000
Local APIC:  0xFEE00000 - 0x100000
Flash:       0xFF000000 - 0x1000000
================================================================================
```

Summary

This section gives introduction on memory map from hardware perspective. We discussed how system memory and memory mapped IO look like by using an Intel x86 platform as example.

Memory Map – Firmware Perspective

In this chapter, we will discuss the memory map from firmware perspective. We will use Platform Initialization (PI) specification as reference.

Memory Map in PI specification (PEI Phase)

There is no memory map term in PI Pre-EFI-initialization (PEI) phase. The “memory map” concept is reported by in PEI Hand-of-Block (HOB).

The resource HOB is defined in PI specification, vol 3, 5.5 Resource Descriptor HOB. The **resource type** of the resource description HOB is below: (It is similar as GCD resource type, which will be discussed later.)

- **EFI_RESOURCE_SYSTEM_MEMORY**: Memory that persists out of the HOB producer phase.
- **EFI_RESOURCE_MEMORY_MAPPED_IO**: Memory-mapped I/O that is programmed in the HOB producer phase.
- **EFI_RESOURCE_FIRMWARE_DEVICE**: Memory-mapped firmware devices.
- **EFI_RESOURCE_MEMORY_MAPPED_IO_PORT**: Memory that is decoded to produce I/O cycles.
- **EFI_RESOURCE_MEMORY_RESERVED**: Reserved memory address space.

The **resource attribute** of the resource description HOB is below: (It is similar as UEFI memory map attribute, which will be discussed later.)

- **Physical memory attribute**: EFI_RESOURCE_ATTRIBUTE_PRESENT, EFI_RESOURCE_ATTRIBUTE_INITIALIZED, EFI_RESOURCE_ATTRIBUTE_TESTED: The memory region exists, has been initialized, or has been tested.
- **Physical memory protection attribute:** **EFI_RESOURCE_ATTRIBUTE_READ_PROTECTED**, **EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTED**, **EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTED**: The memory region is read protected, write protected, or execution protected.
- **Memory capability attribute:** **EFI_RESOURCE_ATTRIBUTE_READ_PROTECTABLE**, **EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTABLE**, **EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTABLE**: The memory supports being protected from processor read, write, or execution.
- **Memory cache ability attribute**: EFI_RESOURCE_ATTRIBUTE_UNCACHEABLE, EFI_RESOURCE_ATTRIBUTE_WRITE_COMBINEABLE, EFI_RESOURCE_ATTRIBUTE_WRITE_THROUGH_CACHEABLE，EFI_RESOURCE_ATTRIBUTE_WRITE_BACK_CACHEABLE, EFI_RESOURCE_ATTRIBUTE_UNCACHED_EXPORTED, EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTABLE. On x86 system, those attributes can be set to CPU MSR. (see [IA32SDM] for detail).

- **Other Physical Memory attribute**: EFI_RESOURCE_ATTRIBUTE_PERSISTENT: The memory region is configured for byte-addressable nonvolatility. EFI_RESOURCE_ATTRIBUTE_MORE_RELIABLE: The memory region provides higher reliability relative to other memory in the system. If all memory has the same reliability, then this bit is not used.

The resource descriptor HOB describes the resource properties of all fixed, non-relocatable resource ranges found on the processor host bus during the HOB producer phase. This HOB type does not describe how memory is used but instead describes the attributes of the physical memory present.

The HOB consumer phase reads all resource descriptor HOBs when it established the initial Global Coherency Domain (GCD) map. The minimum requirement for the HOB producer phase is that executable content in the HOB producer phase report one of the following:

1) The resources that are necessary to start the HOB consumer phase
2) The fixed resources that are not captured by HOB consumer phase driver components that were started prior to the dynamic system configuration performed by the platform boot-policy phase.

For example, executable content in the HOB producer phase should report any physical memory found during the HOB producer phase. Executable content in the HOB producer phase does not need to report fixed system resources like IO or MMIO, because these fixed resources can be allocated from the GCD by a platform-specific chipset driver loading in the HOB consumer phase prior to the platform boot-policy phase, for example.

In most platforms, once memory initialization is done, the memory reference code (MRC) wrapper will report all system memory. The platform initialization will report all fixed location MMIO instead.

A typical resource description HOB on an X86 platform is like below:

```
============================================================================
Resource Descriptor HOBs
  BA=0000000077161000  L=000000000369E000  Attr=0000000000003C07  Mem (Tested)
  BA=0000000000000000  L=00000000000A0000  Attr=0000000000003C07  Mem (Tested)
  BA=00000000000A0000  L=0000000000020000  Attr=0000000000000000  Reserved Mem
  BA=0000000000100000  L=0000000077061000  Attr=0000000000003C07  Mem (Tested)
  BA=000000007C000000  L=0000000002800000  Attr=0000000000000000  Reserved Mem
  BA=000000007B000000  L=0000000001000000  Attr=0000000000000000  Reserved Mem
  BA=000000007A800000  L=0000000000800000  Attr=0000000000000000  Reserved Mem
  BA=000000007A7FF000  L=0000000000001000  Attr=0000000000000000  Reserved Mem
  BA=0000000100000000  L=0000000000800000  Attr=0000000000003C03  Mem (Init)
  BA=00000000FED10000  L=0000000000008000  Attr=0000000000000403  MMIO
  BA=00000000FED18000  L=0000000000001000  Attr=0000000000000403  MMIO
  BA=00000000FED19000  L=0000000000001000  Attr=0000000000000403  MMIO
  BA=00000000FED84000  L=0000000000001000  Attr=0000000000000403  MMIO
  BA=00000000E0000000  L=0000000010000000  Attr=0000000000000403  MMIO
  BA=00000000FEC00000  L=0000000000001000  Attr=0000000000000403  MMIO
  BA=00000000FED00000  L=0000000000004000  Attr=0000000000000403  MMIO
  BA=00000000FED1C000  L=0000000000004000  Attr=0000000000000403  MMIO
  BA=00000000FEE00000  L=0000000000001000  Attr=0000000000000403  MMIO
  BA=00000000FFA00000  L=0000000000600000  Attr=0000000000000403  MMIO
============================================================================
```

Besides resource descriptor HOB, firmware volume HOB details the location of firmware volumes that contain firmware files, and firmware volume2 HOB details the location of a firmware volume which was extracted from a file within another firmware volume. By recording the volume and file name, the HOB consumer phase can avoid processing the same file again.

A typical firmware volume HOB on an X86 platform is like below:

```
============================================================================
FV HOBs
  BA=00000000FFA00000  L=0000000000020000
  BA=00000000FFA00000  L=0000000000600000
  BA=000000007847A000  L=0000000000CF0000
FV2 HOBs
  BA=000000007847A000   L=0000000000CF0000   Fv={00000000-0000-0000-0000-000000000000}
File={9E21FD93-9C72-4C15-8C4B-E77F1DB2D792}
============================================================================
```

The resource descriptor HOB does not describe how memory is used. This work is done by memory allocation HOB. It describes all memory ranges used during the HOB producer phase that exist outside the HOB list. This HOB type describes how memory is used, not the physical attributes of memory.

The HOB consumer phase does not make assumptions about the contents of the memory that is allocated by the memory allocation HOB, and it will not move the data unless it has explicit knowledge of the memory allocation HOB's Name (EFI_GUID). Memory may be allocated in either the HOB producer phase memory area or other areas of present and initialized system memory.

A HOB consumer phase driver that corresponds to the specific Name GUIDed memory allocation HOB can parse the HOB list to find the specifically named memory allocation HOB and then manipulate the memory space as defined by the usage model for that GUID. For example:

1) BSP stack memory allocation HOB
2) BSP store memory allocation HOB
3) Memory allocation module HOB

A typical memory allocation HOB on an X86 platform is like below:

```
============================================================================
Memory Allocation HOBs
  BA=00000000783D0000  L=0000000000020000  BS Data (Stack)
  BA=000000007A150000  L=0000000000001000  BS Code
  BA=000000007A14F000  L=0000000000001000  BS Data
  BA=000000007A139000  L=0000000000016000  BS Data
  BA=00000000FED10000  L=0000000000008000  Loader Code
  BA=00000000FED18000  L=0000000000001000  Loader Code
  BA=00000000FED19000  L=0000000000001000  Loader Code
  BA=00000000FED84000  L=0000000000001000  Loader Code
  BA=00000000E0000000  L=0000000010000000  Loader Code
  BA=00000000FEC00000  L=0000000000001000  Loader Code
  BA=00000000FED00000  L=0000000000004000  Loader Code
  BA=00000000FED1C000  L=0000000000004000  Loader Code
  BA=00000000FEE00000  L=0000000000001000  Loader Code
  BA=00000000FFA00000  L=0000000000600000  Loader Code
  BA=000000007A137000  L=0000000000002000  BS Data
  BA=000000007A12E000  L=0000000000009000  BS Data
...... (Skip lots of BS_Data entries)
```

```
  BA=000000007847A000  L=0000000000CF0000  BS Data
  BA=00000000783F0000  L=000000000008A000  BS Data
  BA=00000000783F0000  L=000000000008A000  BS Code {Module=D6A2CB7F-6A18-4E2F-B43B-
9920A733700A}
  BA=00000000783D0000  L=0000000000020000  BS Data
  BA=00000000781CE000  L=0000000000202000  BS Data
  BA=0000000077161000  L=0000000000020000  BS Data
  BA=00000000781CD000  L=0000000000001000  BS Data
================================================================================
```

Memory Map in PI specification (DXE Phase)

The firmware need to have a way to manage all hardware memory map. This is defined by PI specification Volume 2, 7.2 Global Coherency Domain Services (GCD). The GCD Services are used to manage the memory and I/O resources visible to the boot processor, including GCD memory space map, and GCD IO space map. GCD memory services include following functions to manage memory resource: AddMemorySpace(), AllocateMemorySpace(), FreeMemorySpace(), RemoveMemorySpace(), SetMemorySpaceAttributes() and SetMemorySpaceCapabilities(). GCD memory services include following functions to retrace GCD memory space map: GetMemorySpaceDescriptor(), GetMemorySpaceMap().

The **GCD memory map** defines below 4 types. (It is different with UEFI memory map, which will be discussed later)

- **EfiGcdMemoryTypeNonExistent**: A memory region that is visible to the boot processor. However, there are no system components that are currently decoding this memory region.
- **EfiGcdMemoryTypeReserved**: A memory region that is visible to the boot processor. This memory region is being decoded by a system component, but the memory region is not considered to be either system memory or memory-mapped I/O.
- **EfiGcdMemoryTypeSystemMemory**: A memory region that is visible to the boot processor. A memory controller is currently decoding this memory region and the memory controller is producing a tested system memory region that is available to the memory services.
- **EfiGcdMemoryTypeMemoryMappedIo**: A memory region that is visible to the boot processor. This memory region is currently being decoded by a component as memory-mapped I/O that can be used to access I/O devices in the platform.
- **EfiGcdMemoryTypePersistent**: A memory region that is visible to the boot processor. This memory supports byte addressable non-volatility.
- **EfiGcdMemoryTypeMoreReliable**: A memory region that provides higher reliability relative to other memory in the system. If all memory has the same reliability, then this bit is not used.

GCD memory map does not describe how system memory is used, but instead describes the location of each memory region including system memory and memory mapped IO. GCD memory map is invisible to OS. It is for firmware internal usage only.

The GCD memory space map is initialized from the HOB list that is passed to the entry point of the DXE Foundation. GCD memory space map must reflect the memory regions described in the

HOB list. A platform may have driver to add more memory regions which is not needed to be reported in PEI phase. For example, PciHostBridge driver may add MMIO for PCI express configuration space and add MMIO for PCI BAR.

A typical GCD memory map on an X86 platform is like below:

```
============================================================================
                                        U
                                   RRMNXRWCWWWU
Base Address     End Address      Type TORVPPPEBTCC Image            Device
================ ================ ==== ============ ================ ================
0000000000000000-000000000009FFFF SYS  0-------1000 000000007A15EE18
00000000000A0000-00000000000BFFFF RSVD 0-----------
00000000000C0000-00000000000FFFFF NE
0000000000100000-000000007A7FEFFF SYS  0-------1000 000000007A15EE18
000000007A7FF000-000000007E7FFFFF RSVD 0-----------
000000007E800000-000000007E80FFFF MMIO 0----------- 0000000076A31318
000000007E810000-000000007E81FFFF MMIO 0----------- 00000000769D6918
000000007E820000-000000007FFFFFFF MMIO 0-----------
0000000080000000-00000000911FFFFF MMIO 0----------- 0000000077697798
0000000091200000-00000000DFFFFFFF MMIO 0-----------
00000000E0000000-00000000E00F7FFF MMIO 0----------1 000000007A15EE18
00000000E00F8000-00000000E00F8FFF MMIO 1----------1 000000007A15EE18
00000000E00F9000-00000000EFFFFFFF MMIO 0----------1 000000007A15EE18
00000000F0000000-00000000FE100FFF NE
00000000FE101000-00000000FE112FFF RSVD 0----------- 0000000077240318
00000000FE113000-00000000FEBFFFFF NE
00000000FEC00000-00000000FEC00FFF MMIO 0----------1 000000007A15EE18
00000000FEC01000-00000000FECFFFFF NE
00000000FED00000-00000000FED03FFF MMIO 0----------1 000000007A15EE18
00000000FED04000-00000000FED0FFFF NE
00000000FED10000-00000000FED19FFF MMIO 0----------1 000000007A15EE18
00000000FED1A000-00000000FED1BFFF NE
00000000FED1C000-00000000FED1FFFF MMIO 1----------1 000000007A15EE18
00000000FED20000-00000000FED83FFF NE
00000000FED84000-00000000FED84FFF MMIO 0----------1 000000007A15EE18
00000000FED85000-00000000FEDFFFFF NE
00000000FEE00000-00000000FEE00FFF MMIO 0----------1 000000007A15EE18
00000000FEE01000-00000000FF9FFFFF NE
00000000FFA00000-00000000FFA1FFFF MMIO 1----------1 000000007A15EE18
00000000FFA20000-00000000FFFFFFFF MMIO 1----------1
0000000100000000-00000001007FFFFF SYS  0-------1000 000000007A15EE18
0000000100800000-0000007FFFFFFFFF NE
============================================================================
```

Memory Map in UEFI specification

As GCD only describes the location of each memory region, we need services to allocate/free system memory. There are defined in UEFI specification 6.2 Memory Allocation Services. During preboot, all components (including executing EFI images) must cooperate with the firmware by allocating and freeing memory from the system with the functions AllocatePages(), AllocatePool(), FreePages(), and FreePool(). The firmware dynamically maintains the memory map as these functions are called.

When memory is allocated, it is “typed” according to the values in EFI_MEMORY_TYPE. The meaning of memory type is defined in UEFI specification Table 25. Memory Type Usage before ExitBootServices(). Some important types are below:

- **EfiReservedMemoryType**: no used by OS. According to ACPI specification, one of possible reasons a BIOS need this memory type is: the address range contains RAM in use by the ROM.
- **EfiLoaderCode/EfiLoaderData**: Used for UEFI application.
- **EfiBootServicesCode/EfiBootServicesData**: Used for UEFI boot services driver.
- **EfiRuntimeServicesCode/EfiRuntimeServicesData**: Used for UEFI runtime services driver.
- **EfiACPIReclaimMemory**: Used for most ACPI tables. The memory is to be preserved by the loader and OS until ACPI in enabled. Once ACPI is enabled, the memory in this range is available for general use.
- **EfiACPIMemoryNVS**: Address space reserved for use by the firmware (for example ACPI FACS). The OS and loader must preserve this memory range in the working and ACPI S1–S3 states.
- **EfiMemoryMappedIO**: Used by system firmware to request that a memory-mapped IO region be mapped by the OS to a virtual address so it can be accessed by EFI runtime services. The OS does not use this memory. All system memory-mapped I/O port space information should come from ACPI tables.
- **EfiPersistentMemory**: Used by system firmware to report a region supporting byte-addressable non-volatility. This memory region that operates as EfiConventionalMemory.
- **EfiConventionalMemory**: Memory available for general use.

NOTE: EfiMemoryMappedIO cannot be allocated by UEFI memory allocation services. It is only used in GetMemoryMap() interface, which will be discussed in next chapter.

Summary

This section gives introduction on memory map from firmware perspective. We discussed resource description HOB in PEI phase and GCD memory map in DXE phase.

Memory Map – OS Perspective

In this chapter, we will discuss the memory map from OS perspective. OS may use UEFI specification and ACPI specification defined method to get memory map information.

Memory Map in UEFI specification

EFI enabled systems use the UEFI GetMemoryMap() boot services function to convey memory resources to the OS loader. These resources must then be conveyed by the OS loader to OSPM.

The GetMemoryMap() interface returns an array of UEFI memory descriptors. These memory descriptors define a system memory map of all the installed RAM, and of physical memory ranges reserved by the firmware. Each descriptor contains memory base and size at page level, as well as memory type and attributes.

The meaning of memory type is introduced in last chapter. The meaning of memory attributes is also defined in UEFI specification - Memory Attribute Definitions. The attributes are classified below:

- **Memory cache ability attribute**: EFI_MEMORY_UC, EFI_MEMORY_WC, EFI_MEMORY_WT, EFI_MEMORY_WB, EFI_MEMORY_UCE, EFI_MEMORY_WP. It means the memory region supports to be configured as cache attributes. On x86 system, those attributes can be set to CPU MSR. (see [IA32SDM] for detail).
- **Physical memory protection attribute**: **EFI_MEMORY_RO**, **EFI_MEMORY_RP**, **EFI_MEMORY_XP**. It means the memory region supports to be configured as write-protected, read-protected, or execution-protected by system hardware. On x86 system, those attributes can be set to CPU page table. (see [IA32SDM] for detail)
- **Other Physical Memory attribute**: EFI_MEMORY_NV: The memory region is configured for byte-addressable nonvolatility. EFI_MEMORY_MORE_RELIABLE: The memory region provides higher reliability relative to other memory in the system. If all memory has the same reliability, then this bit is not used.
- **Runtime memory attribute**: EFI_MEMORY_RUNTIME. It means the memory region needs to be given a virtual mapping by the operating system when SetVirtualAddressMap() is called.

According to ACPI specification, 15.4 UEFI Assumptions and Limitations, below memory range need to be reported:

- Current system memory configuration
- Chipset-defined address holes that are not being used by devices as reserved.
- Baseboard memory-mapped I/O devices, such as APICs, are returned as reserved.
- All occurrences of the system firmware are mapped as reserved. Including the areas below 1 MB, at 16 MB (if present), and at end of the 4-GB address space.

- All of lower memory is reported as normal memory. The OS must handle standard RAM locations that are reserved for specific uses, such as the interrupt vector table (0:0) and the BIOS data area (40:0).
- Memory mapped I/O and memory mapped I/O port space allowed for virtual mode calls to UEFI run-time functions.

Below memory range does not need to be reported:

- Memory mapping of PCI devices, ISA Option ROMs, and ISA Plug and Play cards. Because the OS has mechanisms available to detect them. For example, PCI BAR MMIO can be got by standard PCI bus enumeration. On-board device (e.g. TPM) MMIO could be got by ACPI _CRS method.
- Standard PC address ranges are not reported. For example, video memory at 0xA0000 to 0xBFFFF physical addresses are not described by this function.

A typical UEFI memory map on an X86 platform is like below:

```
============================================================================
Type        Start             End               # Pages          Attributes
BS_Code     0000000000000000-0000000000000FFF 0000000000000001 000000000000000F
Available   0000000000001000-000000000003CFFF 000000000000003C 000000000000000F
BS_Code     000000000003D000-0000000000057FFF 000000000000001B 000000000000000F
Reserved    0000000000058000-0000000000058FFF 0000000000000001 000000000000000F
Available   0000000000059000-000000000005FFFF 0000000000000007 000000000000000F
BS_Code     0000000000060000-0000000000087FFF 0000000000000028 000000000000000F
BS_Data     0000000000088000-0000000000088FFF 0000000000000001 000000000000000F
BS_Code     0000000000089000-000000000009EFFF 0000000000000016 000000000000000F
Reserved    000000000009F000-000000000009FFFF 0000000000000001 000000000000000F
Available   0000000000100000-000000000FFFFFFF 0000000000000FF00 000000000000000F
BS_Code     0000000010000000-000000001000AFFF 000000000000000B 000000000000000F
Available   000000001000B000-000000006AAFCFFF 0000000000005AAF2 000000000000000F
BS_Data     000000006AAFD000-000000006AFAAFFF 00000000000004AE 000000000000000F
Available   000000006AFAB000-000000006AFDBFFF 0000000000000031 000000000000000F
BS_Data     000000006AFDC000-000000006B1BDFFF 00000000000001E2 000000000000000F
ACPI_Recl   000000006B1BE000-000000006B1DCFFF 000000000000001F 000000000000000F
BS_Data     000000006B1DD000-000000006B26DFFF 0000000000000091 000000000000000F
Available   000000006B26E000-000000006B2ABFFF 000000000000003E 000000000000000F
BS_Data     000000006B2AC000-000000006B2BCFFF 0000000000000011 000000000000000F
Available   000000006B2BD000-000000006C27BFFF 0000000000000FBF 000000000000000F
BS_Data     000000006C27C000-000000006C2CEFFF 0000000000000053 000000000000000F
Available   000000006C2CF000-000000006C2D2FFF 0000000000000004 000000000000000F
BS_Data     000000006C2D3000-000000006C431FFF 000000000000015F 000000000000000F
LoaderCode  000000006C432000-000000006C50DFFF 00000000000000DC 000000000000000F
BS_Data     000000006C50E000-000000006C933FFF 0000000000000426 000000000000000F
BS Code     000000006C934000-0000000075F49FFF 0000000000009616 000000000000000F
...... (Skip lots of BS_Data/BS_Code entries)
BS_Data     00000000776E5000-00000000776E8FFF 0000000000000004 000000000000000F
BS_Code     00000000776E9000-000000007770AFFF 0000000000000022 000000000000000F
BS_Data     000000007770B000-000000007770BFFF 0000000000000001 000000000000000F
BS_Code     000000007770C000-0000000077713FFF 0000000000000008 000000000000000F
BS_Data     0000000077714000-0000000077718FFF 0000000000000005 000000000000000F
BS_Code     0000000077719000-0000000077726FFF 000000000000000E 000000000000000F
BS_Data     0000000077727000-000000007772BFFF 0000000000000005 000000000000000F
BS_Code     000000007772C000-0000000077733FFF 0000000000000008 000000000000000F
BS_Data     0000000077734000-0000000077736FFF 0000000000000003 000000000000000F
BS_Code     0000000077737000-0000000077744FFF 000000000000000E 000000000000000F
BS_Data     0000000077745000-0000000077745FFF 0000000000000001 000000000000000F
BS_Code     0000000077746000-0000000077748FFF 0000000000000003 000000000000000F
BS_Data     0000000077749000-000000007A14FFFF 0000000000002A07 000000000000000F
BS_Code     000000007A150000-000000007A150FFF 0000000000000001 000000000000000F
BS_Data     000000007A151000-000000007A15FFFF 000000000000000F 000000000000000F
```

```
RT_Code   000000007A160000-000000007A22FFFF  00000000000000D0  800000000000000F
RT_Data   000000007A230000-000000007A24FFFF  0000000000000020  800000000000000F
Reserved  000000007A250000-000000007A74FFFF  0000000000000500  800000000000000F
ACPI_NVS  000000007A750000-000000007A767FFF  0000000000000018  800000000000000F
Reserved  000000007A768000-000000007A768FFF  0000000000000001  800000000000000F
ACPI_NVS  000000007A769000-000000007A7B5FFF  000000000000004D  800000000000000F
ACPI_Recl  000000007A7B6000-000000007A7E2FFF 000000000000002D  800000000000000F
BS_Data   000000007A7E3000-000000007A7FEFFF  000000000000001C  800000000000000F
Available  0000000100000000-00000001007FFFFF 0000000000000800  800000000000000F
MMIO       00000000E00F8000-00000000E00F8FFF 0000000000000001 8000000000000001
MMIO       00000000FED1C000-00000000FED1FFFF 0000000000000004 8000000000000001
MMIO       00000000FFA00000-00000000FFFFFFFF 0000000000000600 8000000000000001

  Reserved  :           1,283 Pages (5,255,168 Bytes)
  LoaderCode:             220 Pages (901,120 Bytes)
  LoaderData:               0 Pages (0 Bytes)
  BS_Code   :          40,830 Pages (167,239,680 Bytes)
  BS_Data   :          17,978 Pages (73,637,888 Bytes)
  RT_Code   :             208 Pages (851,968 Bytes)
  RT_Data   :              32 Pages (131,072 Bytes)
  ACPI_Recl :              76 Pages (311,296 Bytes)
  ACPI_NVS  :             101 Pages (413,696 Bytes)
  MMIO      :           1,541 Pages (6,311,936 Bytes)
  MMIO_Port :               0 Pages (0 Bytes)
  PalCode   :               0 Pages (0 Bytes)
  Available :         442,983 Pages (1,814,458,368 Bytes)
              --------------
Total Memory:           1,962 MB (2,057,945,088 Bytes)
================================================================================
```

Shell command “memmap” can dump above UEFI memory map information.

Memory Attribute Table in UEFI specification

When published by the system firmware, the **EFI_MEMORY_ATTRIBUTES_TABLE** provides additional information about regions within the run-time memory blocks defined in the EFI_MEMORY_DESCRIPTOR entries returned from FI_BOOT_SERVICES.GetMemoryMap() function. The Memory Attributes Table is currently used to describe memory protections that may be applied to the EFI Runtime code and data by an operating system or hypervisor. The Memory Attributes Table may define multiple entries to describe sub-regions that comprise a single entry returned by GetMemoryMap() however the sub-regions must total to completely describe the larger region and may not cross boundaries between entries reported by GetMemoryMap(). If a run-time region returned in GetMemoryMap() entry is not described within the Memory Attributes Table, this region is assumed to not be compatible with any memory protections.

A typical UEFI memory map on an X86 platform is like below:

```
================================================================================
Type        Start             End               # Pages          Attributes
RT_Code     000000007A160000-000000007A1ABFFF  000000000000004A  800000000000400F
RT_Code     000000007A1AA000-000000007A1ACFFF  0000000000000003  800000000002000F
RT_Code     000000007A1AD000-000000007A1B1FFF  0000000000000005  800000000000400F
......
RT_Code     000000007A226000-000000007A226FFF  0000000000000001  800000000000400F
RT_Code     000000007A227000-000000007A22AFFF  0000000000000004  800000000002000F
```

```
RT_Code     000000007A22B000-000000007A22FFFF 0000000000000005 800000000000400F
RT_Data     000000007A230000-000000007A24FFFF 0000000000000020 800000000000000F
================================================================================
```

Memory Map in ACPI specification

Besides UEFI GetMemoryMap(), ACPI specification defines a legacy way to report memory map on IA-PC-based system, it is named as INT15 E820H function call, or E820 table. E820 table is only needed to support legacy OS, so it can be derived from UEFI GetMemoryMap(). In EDKII, this is done by compatibility support module (CSM) in LegacyBiosBuildE820() in https://github.com/tianocore/edk2/blob/master/IntelFrameworkModulePkg/Csm/LegacyBiosDxe/LegacyBootSupport.c

According to ACPI specification, 15.2 E820 Assumptions and Limitations, rule of E820 table is similar as UEFI GetMemoryMap. The difference is that E820 table does not have runtime concept, so memory mapped I/O and memory mapped I/O port space allowed for virtual mode calls to UEFI run-time functions does not exist.

A typical E820 memory map on an X86 platform is like below:

```
================================================================================
E820[ 0]: 0x               0 ---- 0x           9D800, Type = 0x1
E820[ 1]: 0x           9D800 ---- 0x           A0000, Type = 0x2
E820[ 2]: 0x           E0000 ---- 0x          100000, Type = 0x2
E820[ 3]: 0x          100000 ---- 0x        6B1B9000, Type = 0x1
E820[ 4]: 0x        6B1B9000 ---- 0x        6B1D8000, Type = 0x3
E820[ 5]: 0x        6B1D8000 ---- 0x        7A250000, Type = 0x1
E820[ 6]: 0x        7A250000 ---- 0x        7A750000, Type = 0x2
E820[ 7]: 0x        7A750000 ---- 0x        7A768000, Type = 0x4
E820[ 8]: 0x        7A768000 ---- 0x        7A769000, Type = 0x2
E820[ 9]: 0x        7A769000 ---- 0x        7A7B6000, Type = 0x4
E820[10]: 0x        7A7B6000 ---- 0x        7A7E3000, Type = 0x3
E820[11]: 0x        7A7E3000 ---- 0x        7A7FF000, Type = 0x1
E820[12]: 0x        7A7FF000 ---- 0x        7A800000, Type = 0x2
E820[13]: 0x        7A800000 ---- 0x        7B000000, Type = 0x2
E820[14]: 0x        7B000000 ---- 0x        7C000000, Type = 0x2
E820[15]: 0x        7C000000 ---- 0x        7E800000, Type = 0x2
E820[16]: 0x        E0000000 ---- 0x        F0000000, Type = 0x2
E820[17]: 0x        FEC00000 ---- 0x        FEC01000, Type = 0x2
E820[18]: 0x        FED00000 ---- 0x        FED04000, Type = 0x2
E820[19]: 0x        FED10000 ---- 0x        FED18000, Type = 0x2
E820[20]: 0x        FED18000 ---- 0x        FED19000, Type = 0x2
E820[21]: 0x        FED19000 ---- 0x        FED1A000, Type = 0x2
E820[22]: 0x        FED1C000 ---- 0x        FED20000, Type = 0x2
E820[23]: 0x        FED84000 ---- 0x        FED85000, Type = 0x2
E820[24]: 0x        FEE00000 ---- 0x        FEE01000, Type = 0x2
E820[25]: 0x        FFA00000 ---- 0x       100000000, Type = 0x2
E820[26]: 0x       100000000 ---- 0x       100800000, Type = 0x1
================================================================================
```

Besides E820 table, ACPI specification requires each device node to report current resource by using _CRS method. _CRS can return any resource, including Memory, IO, IRQ, etc. If an MMIO not assigned to standard PCI MMIO bar, it should be reported here.

Sample memory resource returned by ASL _CRS on an X86 platform is below:

```
================================================================================
Device(PDRC): Memory32Fixed (ReadWrite, 0xE0000000, 0x10000000)// PCI Express
Device(HPET): Memory32Fixed (ReadWrite, 0xFED00000, 0x400)     // HPET
Device(PDRC): Memory32Fixed (ReadWrite, 0xFED10000, 0x8000)    // Intel MCH BAR
Device(PDRC): Memory32Fixed (ReadWrite, 0xFED18000, 0x1000)    // Intel DMI BAR
Device(PDRC): Memory32Fixed (ReadWrite, 0xFED19000, 0x1000)    // Intel EGP BAR
Device(PDRC): Memory32Fixed (ReadWrite, 0xFED1C000, 0x4000)    // Intel RCBA
Device(PDRC): Memory32Fixed (ReadWrite, 0xFED20000, 0x20000)   // Intel TXT
Device(TPM) : Memory32Fixed (ReadOnly,  0xFED40000, 0x5000)    // TPM
Device(PTT) : Memory32Fixed (ReadOnly,  0xFED70000, 0x1000)    // Intel PTT
Device(PDRC): Memory32Fixed (ReadOnly,  0xFED90000, 0x4000)    // Intel VTd
Device(PDRC): Memory32Fixed (ReadOnly,  0xFEE00000, 0x100000)  // Local APIC
Device(FWHD): Memory32Fixed (ReadOnly,  0xFF000000, 0x1000000) // Flash
Device(PCI0): DWordMemory(ResourceProducer,PosDecode,MinFixed,MaxFixed,Cacheable,
            ReadWrite,0x00,<TOLUD>,0xFEAFFFFF,0x00,<MMIO_LO_LEN>,,,PM01) // Low MMIO
Device(PCI0): QWordMemory(ResourceProducer,PosDecode,MinFixed,MaxFixed,Cacheable,
            ReadWrite,0x00,<TOUUD>,<MMIO_HIGH>,0x00,<MMIO_HI_LEN>,,,PM02) // High MMIO
================================================================================
```

Memory Map in S3 resume

There is no UEFI memory map for S3 resume, because S3 resume only has PEI phase.

Memory Map in S4 resume – Memory Type Information

The memory map should NOT be changed in S4 resume phase, because OS restores the system memory from disk directly. So the best way is to keep the ReservedMemory, ACPINvs, ACPIReclaim, RuntimeCode, RuntimeData region be same. The easiest way is to have a big enough range to put memory with same type together. It means there is only one entry in final memory map, so there is no fragmentation.

In EDKII, DXE Core has capability to pre-allocate a BIN for each above type. See below figure 5. The BIN location is reserved in PEI phase, so that in most x86 platform BIN is in highest system memory below 4G if PEI is 32bit.

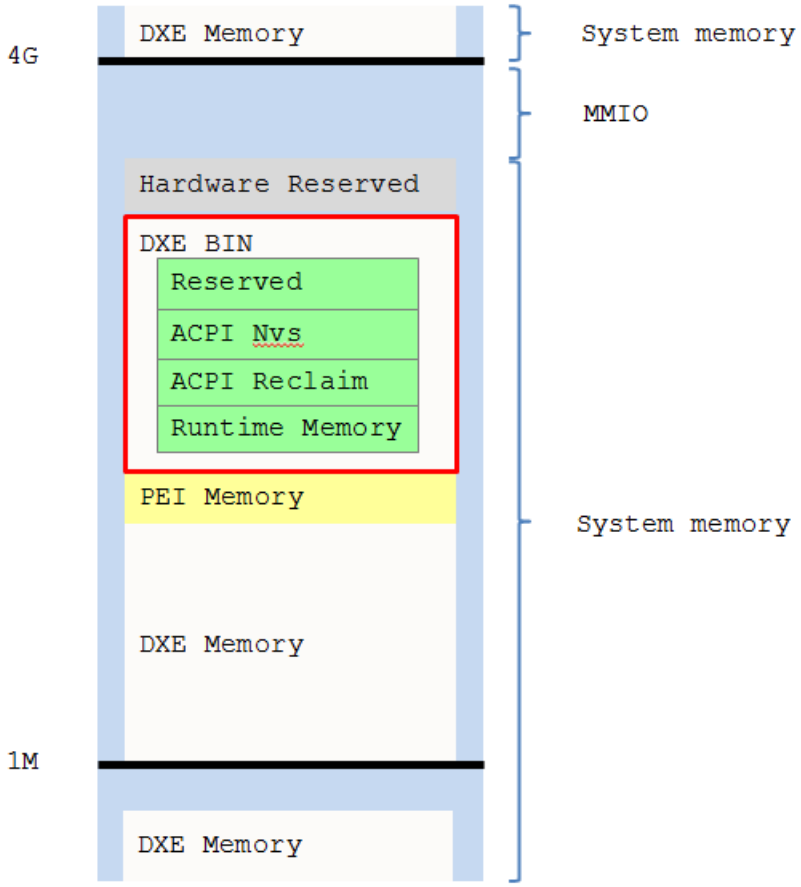

Figure 5

The BIN data structure is stored in gMemoryTypeInformation at
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/Page.c
During first boot, platform need report MemoryTypeInformation HOB to try to allocate BIN for each of those types. MRC wrapper may get this information to reserve BIN range, and allocate memory for PEI core below BIN. DXE core gets this HOB and pre-allocate BIN for each type. Then if DXE core gets request to allocate memory for this type, DXE core will find the free memory in BIN first. Only if there is no enough memory in BIN, DXE core will allocate free memory outside BIN. DXE core also create UEFI configuration table for the MemoryTypeInformation to record the memory usage in current boot.

Then in BDS after ReadyToBoot event, BdsSetMemoryTypeInformationVariable() will check if BIN can hold the current allocated pages, by comparing the memory information in variable (BIN usage) and in UEFI configuration table (current usage).
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Library/UefiBootManagerLib/BmMisc.c
If all BIN can hold current pages, that means there is no fragmentation in memory page. If some BIN is smaller than current allocated pages, that means there is fragmentation. BdsSetMemoryTypeInformationVariable() will set MemoryTypeInformation variable to save the expected BIN size, then reset system. During next boot, platform can get MemoryTypeInformation variable and report updated MemoryTypeInformation HOB. Then the BIN should be big enough. See below figure 6 for the flow.

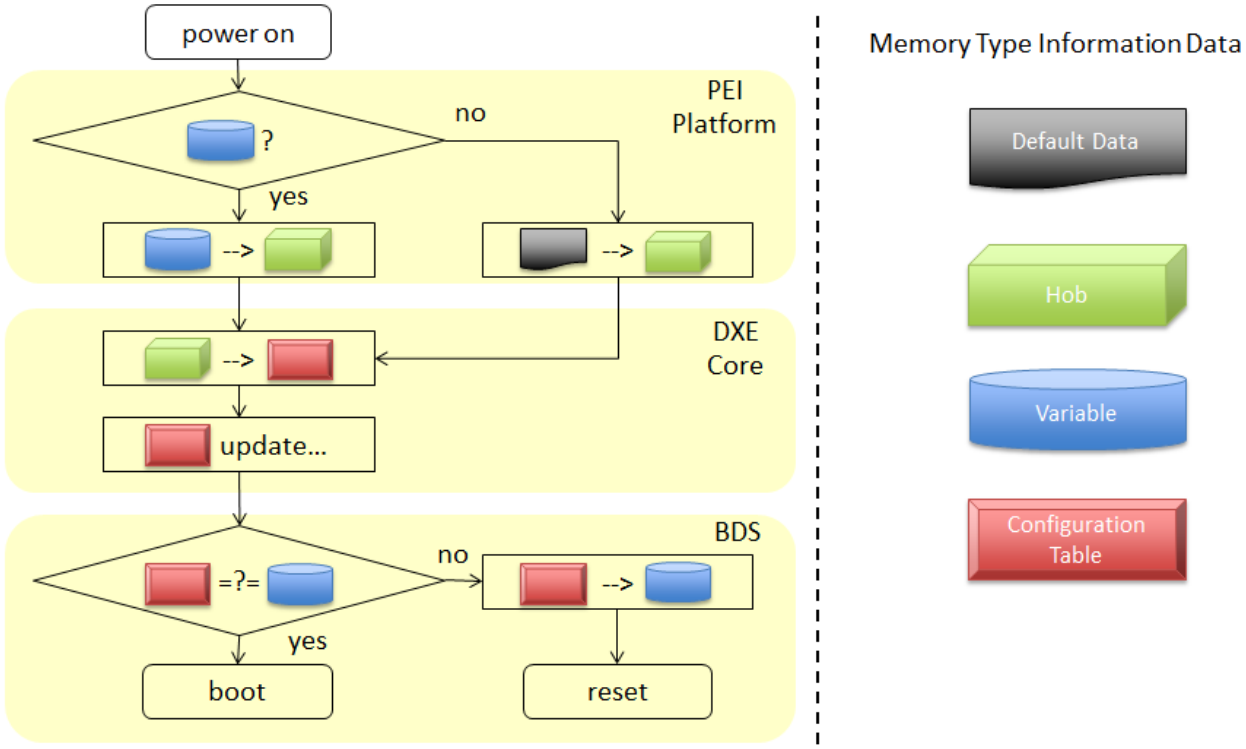

Figure 6

A sample Memory Type Information reporting (BIN not big enough and reset required) on an X86 platform is below:

```
==========================================================================
Memory  Previous  Current    Next
 Type    Pages     Pages     Pages
======  ========  ========  ========
  09    00000048  00000045  00000048
  0A    0000004D  00000052  00000066
  00    00000500  00000307  00000500
  06    00000020  0000001B  00000020
  05    000000D0  0000009E  000000D0
==========================================================================
```

A sample Memory Type Information reporting (BIN big enough and reset not required) on an X86 platform is below:

```
==========================================================================
Memory  Previous  Current    Next
 Type    Pages     Pages     Pages
======  ========  ========  ========
  09    00000048  00000045  00000048
  0A    00000066  00000052  00000066
  00    00000500  00000307  00000500
  06    00000020  0000001B  00000020
  05    000000D0  0000009E  000000D0
==========================================================================
```

NOTE: We recommend a platform only define the ReservedMemory, ACPINvs, ACPIReclaim, RuntimeCode, RuntimeData in Memory Type Information table, because OSes only request these regions to be consistent. There is no need to add BootServicesCode, BootServicesData, LoaderCode, LoaderData in memory type information table, because these regions will not be reserved during S4 resume.

Put it all together

The platform has memory map from hardware perspective. During firmware boot, a silicon driver need initialize system memory and report system memory location by resource description HOB in PEI phase. The PEI driver may also report MMIO information by resource description HOB. In DXE phase, DXE core consumes the resource description HOB, builds GCD memory map, and provide GCD services to manage all system memory and MMIO. A DXE phase platform driver may have capability to add more MMIO resource or system memory resource. DXE core also provides system memory management and builds UEFI memory map to operating system. If a platform boots to a UEFI OS, the UEFI OS loader will call GetMemoryMap() and convey the information to OS kernel. If a platform boots to a legacy OS, the CSM module will create E820 table from UEFI GetMemoryMap(). The legacy OS loader will call INT15 E820 function to get the information and convey to OS kernel.

Summary

This section gives introduction on memory map from OS perspective. We discussed memory type in UEFI/PI/ACPI specification and how platform BIOS report the memory map. We also discussed the S4 requirement for memory map.

Memory Protection - Existing technology

Data Execution Protection (DEP)

DEP is intended to prevent an application or service from executing code from a non-executable memory region. This helps prevent certain exploits that store code via a buffer overflow, for example.

According to UEFI/PI specification, a platform may set a memory range to be read protected, write protected, or execution protected. On x86 systems, those attributes can be set using the CPU page table. For example, if a memory range has the EFI_MEMORY_XP attribute, the consumer (OS loader) may set the IA32_EFER.NXE (No-eXecution Enable) bit in IA32_EFER MSR, then set the XD (eXecution Disable) bit in the CPU PAE page table. DEP protects against some program errors, and helps prevent certain malicious exploits, especially attacks that store executable instructions in a data area via a buffer overflow. Some OS already have Data Execution Protection (DEP) support by setting XD in the page table. Research shows 14 of 19 exploits from popular exploit kits fail with
DEP enabled. See [DEP].

Address space layout randomization (ASLR)

ASLR is intended to prevent an attacker from reliably jumping to a particular exploited function in memory. It involves randomly arranging the positions of key data areas of a program, including the base of the executable and the positions of the stack, heap, and libraries, in a process's address space. ASLR can be used with DEP. See [DEP].

PE/COFF image

UEFI specification requires that the executable image uses the PE/COFF format. The typical PE/COFF image has a DOS header, PE header, section headers and section data.

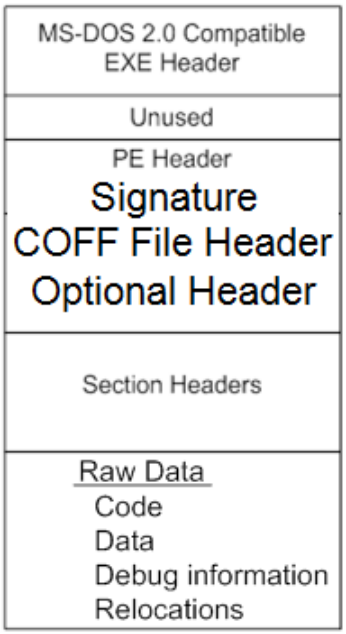

Figure 7

Each section header has a Characteristics field, which may have the SectionFlags below:

- IMAGE_SCN_CNT_CODE: The section contains executable code.
- IMAGE_SCN_MEM_EXECUTE: The section can be executed as code.
- IMAGE_SCN_MEM_READ: The section can be read.
- IMAGE_SCN_MEM_WRITE: The section can be written to.

For DEP, the PE/COFF loader may parse this information and set NX for the data section and set RO for the read-only section.

In addition, the COFF optional header has a DLL Characteristics field, which includes

- IMAGE_DLL_CHARACTERISTICS_NX_COMPAT: Image is NX compatible
- IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE: DLL can be relocated at load time.

A DEP implementation may consult the above field to enable NX or use address space layout randomization (ASLR).

Summary

This section introduces the existing technologies for memory protection.

Memory Protection - Security Practice in UEFI

Using DEP for UEFI/PI and the limitation

In order to user DEP in a UEFI BIOS, we hope to have a memory map like figure 8. All data regions are marked to be non-executable, and all code regions are marked to be write-protected.

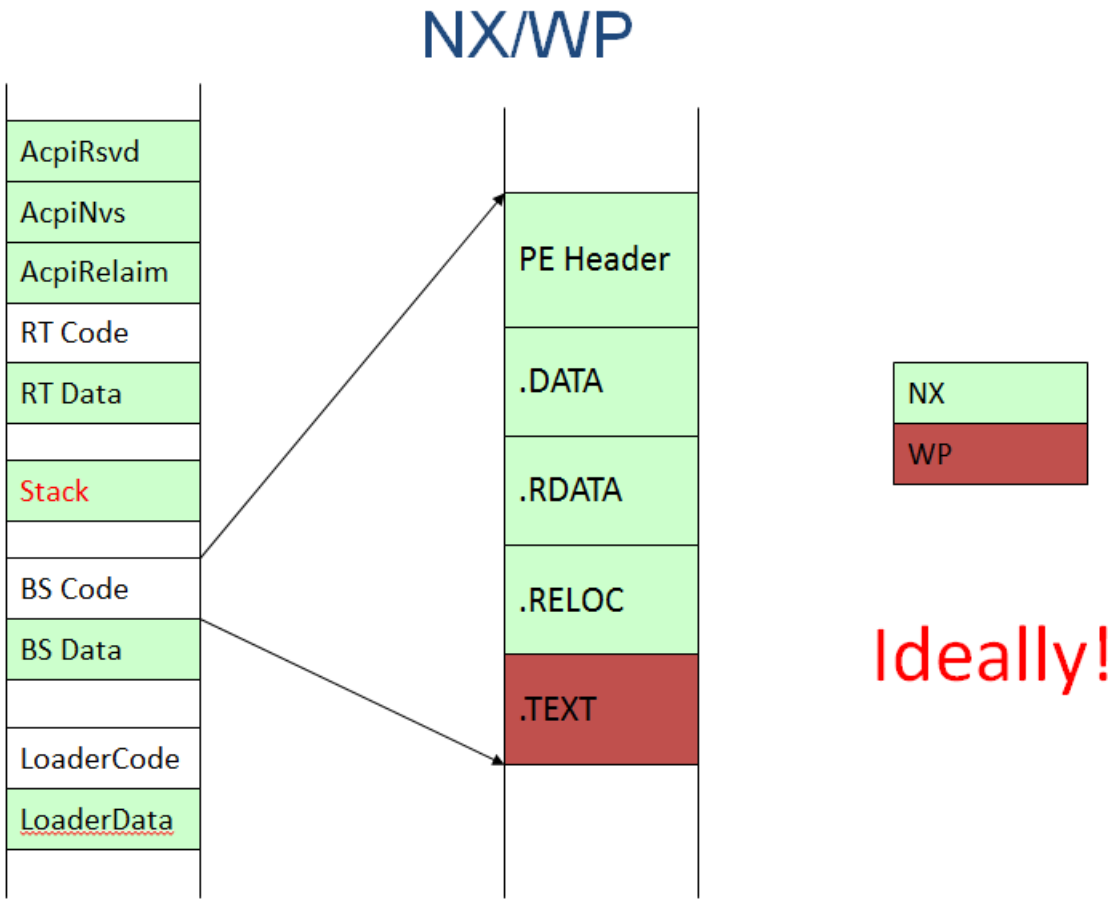

Figure 8

However, the reality is that we can currently only protect the stack, runtime data (RT Data), and ACPI reclaim memory regions. See figure 9. The known limitations are described below.

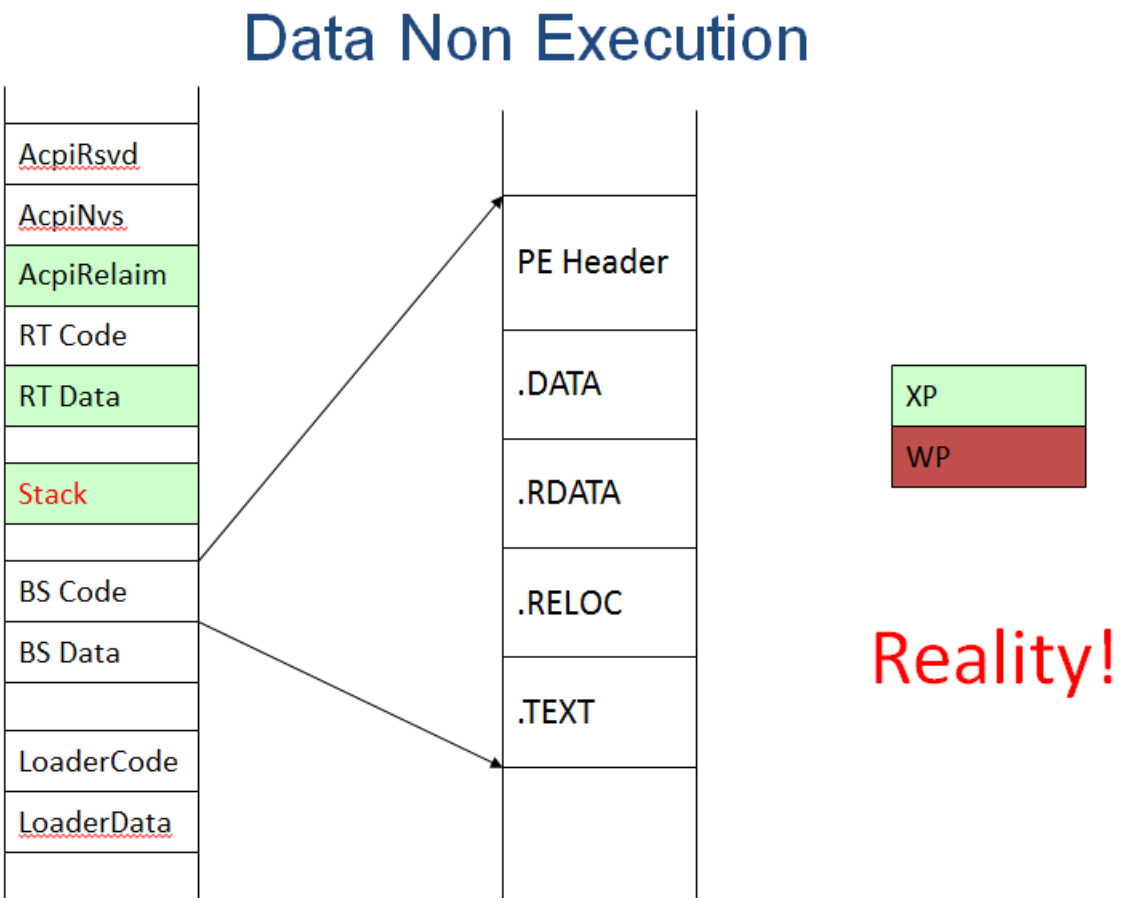

Figure 9

Specification limitation:

1) Some platforms use EFI_MEMORT_WP as a cache attribute, instead of a memory attribute.
   UEFI 2.5 specification clarifies this. It defines EFI_MEMORY_RO as a write protection memory attribute, and uses EFI_MEMORY_WP as a cache attribute.

2) Previous UEFI specification needed paging disabled for IA32 platforms, but NX requires paging.
   UEFI 2.5 specification clarifies this. It allows a 1:1 mapped PAE page table for IA32 platforms.

Firmware limitation:

3) PE/COFF loader uses BootServicesCode/RuntimeServicesCode for both the code and data segment of a PE image.
   This implementation limitation can be resolved by updating the PE/COFF loader.

4) DxeIpl uses BootServicesData for code (DxeCore)
   This implementation limitation can be resolved by updating DxeIpl.

5) Some DXE drivers have self-modified-code.

This implementation limitation can be resolved by adding a new API to enable and disable DEP.

6) Some drivers are relocated to ACPINvs (BootScript).
   This implementation limitation can be resolved by not setting this specific region to be XD.

7) Some drivers are loaded into reserved memory for execution.
   This implementation limitation can be resolved by not setting this specific region to be XD.

8) Some ASM codes use TEXT segment keyword for code section, but it is data section in final PE image.
   This implementation limitation can be resolved by update ASM code to use .CODE as the segment name.

9) Some platforms use /MERGE:.data=.text /MERGE:.rdata=.text (link) to mix PE code section with data section.
   This implementation limitation can be resolved by removing this link option.

10) Most platforms use /ALIGN:32 (link), PE sections are not page aligned
   This implementation limitation can be resolved by overriding /ALIGN:4096. This can be done for compressed DXE driver or PEI driver. For uncompressed PEI driver, we still recommend using /ALIGN:32 to save flash image size. NOTE: There is no need to update /FILEALIGN. It can still be 32.

11) Some drivers assume PE /ALIGN is the same as /FILEALIGN, when they do PE relocation inside the driver by themselves.
   This implementation limitation can be resolved by updating code to remove the assumption.

OS limitation:

12) Some OS loaders use LoadData for code (UEFI Vista64)
   This implementation limitation can be resolved by latest OS loader.

13) Some OS loaders reverse the virtual address in memory map when calling SetVirtualAddressMap(). For example: We observed the below virtual address mapping in UEFI Win7/Win8. (The PhysicalStart is from low to high, while the VirtualStart is from high to low.

```
==============================================================================
Type       PhysicalStart    PhysicalEnd      VirtualStart     Attributes
RT_Code    000000007A157000-000000007A226FFF FFFFFFFFFB30000 800000000000000F
RT_Data    000000007A227000-000000007A246FFF FFFFFFFFFB10000 800000000000000F
MMIO       00000000E00F8000-00000000E00F8FFF FFFFFFFFFB0F000 8000000000000001
MMIO       00000000FED1C000-00000000FED1FFFF FFFFFFFFFB0B000 8000000000000001
MMIO       00000000FFA00000-00000000FFFFFFFF FFFFFFFFF50B000 8000000000000001
==============================================================================
```

This will cause a runtime services error when we separate PE Code and Data for OS, because PE image assumes that PE sections are in sequence order.

This implementation limitation can be resolved by latest OS loader to set virtual address according to physical memory order from lowest to highest, instead of the sequence order in UEFI memory map.

Currently, we only set UEFI stack to be execute-protected. This is controlled by gEfiMdeModulePkgTokenSpaceGuid.PcdSetNxForStack (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/MdeModulePkg.dec). If this PCD is true, the DxeIpl will mark stack to be NX in page page. (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/DxeIplPeim).

Prepare DEP in UEFI for OS runtime

As an alternative, if we think the threat in firmware is low, but the threat in the OS is high, we can compromise by not using DEP in UEFI, and instead preparing the DEP environment for use by the OS runtime.

The assumptions here are:

1) OS/loader only needs to setup DEP at runtime. No protection is needed at UEFI boot time.
2) OS/loader needs to setup DEP to protect 3rd party code if possible. Since UEFI runtime services are considered to be 3rd party code, OS needs to setup DEP for PE non-code segments in UEFI runtime drivers. No boot time images need to be protected because they are gone during UEFI runtime.
3) OS/loader needs to setup DEP to protect unknown memory regions, so OS may setup DEP for reserved memory.

For example, the UEFI memory map might look like this.

```
============================================================================
Type       Start            End                # Pages          Attributes
......
BS_Data    0000000077453000-0000000077453FFF 0000000000000001 000000000000000F
BS_Code    0000000077454000-0000000077456FFF 0000000000000003 000000000000000F
BS_Data    0000000077457000-000000007A1CBFFF 0000000000002D75 000000000000000F
BS_Code    000000007A1CC000-000000007A1CCFFF 0000000000000001 000000000000000F
BS_Data    000000007A1CD000-000000007A1DBFFF 000000000000000F 000000000000000F
RT_Code    000000007A1DC000-000000007A349FFF 000000000000016E 800000000000000F
RT_Data    000000007A34A000-000000007A369FFF 0000000000000020 800000000000000F
Reserved   000000007A36A000-000000007A869FFF 0000000000000500 000000000000000F
ACPI_NVS   000000007A86A000-000000007A8B6FFF 000000000000004D 000000000000000F
ACPI_Recl  000000007A8B7000-000000007A8E2FFF 000000000000002C 000000000000000F
BS_Data    000000007A8E3000-000000007A8FFFFF 000000000000001D 000000000000000F
Available  0000000100000000-0000000100DFFFFF 0000000000000E00 000000000000000F
MMIO       00000000E00F8000-00000000E00F8FFF 0000000000000001 8000000000000001
MMIO       00000000FED1C000-00000000FED1FFFF 0000000000000004 8000000000000001
MMIO       00000000FFA00000-00000000FFFFFFFF 0000000000000600 8000000000000001

  Reserved  :           1,282 Pages (5,251,072 Bytes)
  LoaderCode:             219 Pages (897,024 Bytes)
  LoaderData:               0 Pages (0 Bytes)
  BS_Code   :          40,958 Pages (167,763,968 Bytes)
  BS_Data   :          19,111 Pages (78,278,656 Bytes)
  RT_Code   :             366 Pages (1,499,136 Bytes)
  RT_Data   :              32 Pages (131,072 Bytes)
```

```
  ACPI_Recl :             75 Pages (307,200 Bytes)
  ACPI_NVS  :             77 Pages (315,392 Bytes)
  MMIO      :          1,541 Pages (6,311,936 Bytes)
  MMIO_Port :              0 Pages (0 Bytes)
  PalCode   :              0 Pages (0 Bytes)
  Available :        443,384 Pages (1,816,100,864 Bytes)
              --------------
Total Memory:          1,969 MB (2,065,293,312 Bytes)
================================================================================
```

Then after we adopt the new practice to prepare DEP environment, the UEFI 2.6 specification memory attribute table could provide additional PE/COFF image information, look like this:

```
================================================================================
Type        Start             End                 # Pages          Attributes
RT_Code     000000007A1DC000-000000007A225FFF 000000000000004A 800000000000400F
RT_Code     000000007A226000-000000007A228FFF 0000000000000003 800000000002000F
RT_Code     000000007A229000-000000007A22DFFF 0000000000000005 800000000000400F
......
RT_Code     000000007A340000-000000007A340FFF 0000000000000001 800000000000400F
RT_Code     000000007A341000-000000007A344FFF 0000000000000004 800000000002000F
RT_Code     000000007A345000-000000007A349FFF 0000000000000005 800000000000400F
RT_Data     000000007A34A000-000000007A369FFF 0000000000000020 800000000000400F
================================================================================
```

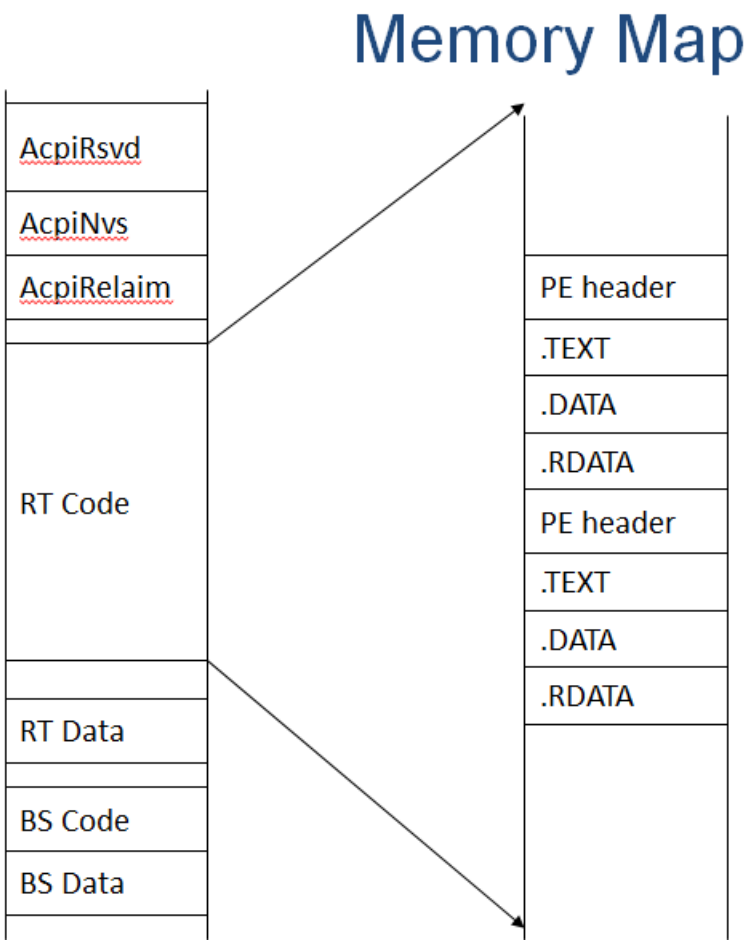

Figure 10

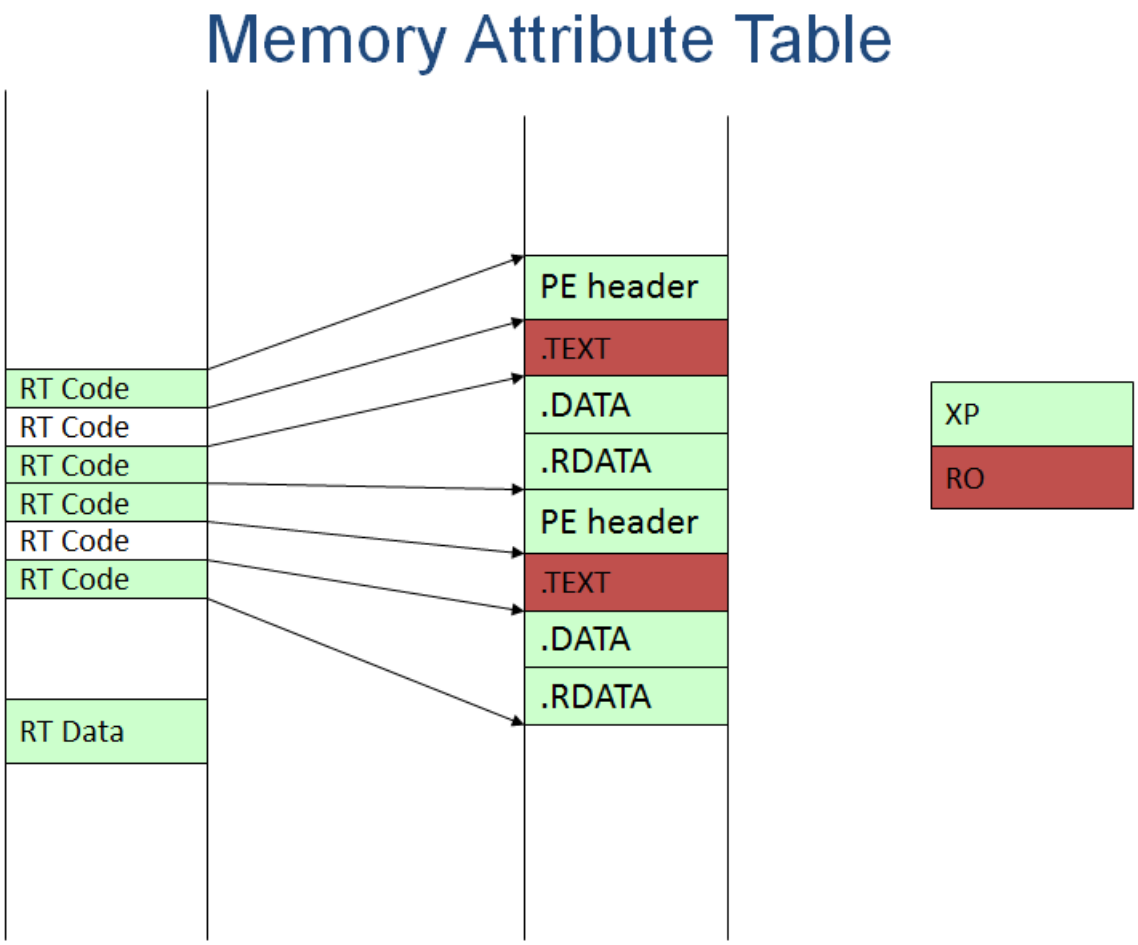

Figure 11

NOTE: UEFI 2.5 specification defines properties table to modify UEFI memory map. This is another way to return similar information to operating system. However, it is recommended not to use this attribute, especially for implementations that broke the runtime code memory map descriptors into the underlying code and data sections within UEFI modules. This splitting causes interoperability issues with operating systems that invoke SetVirtualAddress() without realizing that there is a relationship between these runtime descriptors.

Using DEP at OS runtime

If OS gets the UEFI 2.6 specification memory attribute table, it implies that the UEFI runtime code and data sections of the executable image are separate and aligned as processor page alignment requirement (4KiB for IA32/X64/AArch32, 64KiB for AArch64). It also implies that the data pages do not have any executable code.

The result of this table publication is that the platform provider needs to ensure that there is no executable code in the EFI data section and that the code isn't self-modifying, so that data can be

made execute-protectable and code made read-only.

The valid attribute bit in memory attribute table is **EFI_MEMORY_RO**, **EFI_MEMORY_XP**, plus **EFI_MEMORY_RUNTIME**. A typical setting for write protected code is EFI_MEMORY_RO, read/write data is EFI_MEMORY_XP, read only data is EFI_MEMORY_XP|EFI_MEMORY_RO. Then the OS can adopt the protection policy in its memory management unit (MMU), such as page tables.

EDKII support

Current EDKII implementation already supports UEFI 2.6 memory attribute table.

1) DXE Core MemoryAttributesTable.c and PropertiesTable.c will create memory attribute table.
   (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Misc/MemoryAttributesTable.c and https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Misc/PropertiesTable.c) It will force policy for RuntimeCode(PE Code: EFI_MEMORY_RO, PE Data: EFI_MEMORY_XP) and RuntimeData(EFI_MEMORY_XP).

   PropertiesTable.c parses PE/COFF images in RuntimeCode regions. If the runtime code is page aligned, RuntimeCode entries are split into multiple RuntimeCode entries according to PE Section Table Headers Characteristics , IMAGE_SCN_CNT_CODE SectionFlags (EFI_MEMORY_RO for PE Code, EFI_MEMORY_XP for PE Data). The benefit is that no PE/COFF loader needs to be updated, and no DXE Core BIN algorithm needs to be updated.

   MemoryAttributesTable.c calls the function in PropertiesTable.c and install memory attribute table to UEFI configuration table.

2) For UEFI 2.5 properties table, it is disabled by default by gEfiMdeModulePkgTokenSpaceGuid.PropertiesTableEnable (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/MdeModulePkg.dec).

Call for action

In order to support entry splitting in the new enhanced memory map, the firmware must do the following:

1) Override link flags by using /ALIGN:4096 for runtime drivers, so that the PE Sections are page aligned. /FILEALIGN can still be 32.

   ```
   [BuildOptions.X64.EDKII.DXE_RUNTIME_DRIVER]
   MSFT:*_*_*_DLINK_FLAGS = /ALIGN:4096
   ```

2) A platform may set proper memory attribute in ACPI NVS or Reserved memory regions.

Summary
This section introduces the security practice for memory protection.

Linux Usage

IA32, X64 and Aarch64 Linux implementations map EFI runtime data regions with the non-executable (NX) page table bit set if it is supported by the hardware platform, irrespective of any of the memory protection attributes for the EFI memory descriptor. EFI runtime code regions are marked executable.

Itanium maps both the EFI runtime data regions and EFI runtime code regions as executable irrespective of any of the memory protection attributes in the EFI memory descriptor.

Cacheability EFI descriptor attributes are honored for all architectures on Linux.

Summary
This section introduces the protection usages for Linux.

Conclusion

Memory map is important information from firmware to OS. This paper describes how memory map is constructed from firmware and how firmware conveys this message to OS. This paper also describes how to enable page level execution protection to harden platforms.

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.

ASLR – Address Space Layout Randomization.

DEP – Data Execution Protection.

MMIO – Memory Mapped I/O.

NX – No Execution. See DEP.

PE/COFF – Portable Executable and Common Object File Format. The executable file format for UEFI.

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

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.

XD – Execution Disable. See DEP.

References

[ACPI] ACPI specification, Version 6.1 www.uefi.org

[DEP] Exploit Mitigation Improvements in Windows 8, http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

[EDK2] UEFI Developer Kit www.tianocore.org

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

[PE/COFF] Microsoft Portable Executable and Common Object File Format Specification, Revision 8.3 https://msdn.microsoft.com/library/windows/hardware/gg463119.aspx

[SB] Nystrom, Nicoles, Zimmer, "UEFI Networking and Pre-OS Security," Intel Technology Journal, Volume 15, Issue 1, October 2011
http://www.intel.com/content/www/us/en/research/intel-technology-journal/2011-volume-15-issue-01-intel-technology-journal.html

[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 Memory Map] Yao, Zimmer, "A Tour Beyond BIOS Memory Map in UEFI_BIOS", February 2015
http://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Map_in%20UEFI_BIOS.pdf

[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 System] Windows Hardware Certification Requirements for Client and Server Systems
http://msdn.microsoft.com/en-us/library/windows/hardware/jj128256.aspx

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 the Software and Services Group at Intel Corporation.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer at Intel and chairs the UEFI networking and security sub-team with the Software and Services Group at Intel Corporation.

Matt Fleming (mfleming@suse.de) is the upstream Linux kernel maintainer for UEFI with SUSE. He is also the creator of the Linux UEFI Validation OS.