# UEFI Runtime Drivers

原创 Pedro Posted on 2020-08-07 10:54:24 Read 2.1k Collection 7 Likes 2
Category Column: UEFI-DxeCore Article Tags: uefi

摘要 This article takes a deep look at how UEFI Runtime Drivers work, including how they continue to provide services after the OS calls ExitBootServices, the role of the UNDI driver, and the challenges of address translation in the UEFI environment. It also introduces the two core functions of RuntimeCore: RuntimeDriverSetVirtualAddressMap and RuntimeDriverConvertPointer.

The summary is generated in C Know , supported by DeepSeek-R1 full version, go to experience>

## UEFI Runtime Drivers

UEFI runtime driver is the driver that continues to provide services to OS after OS calls ExitBootServices. UEFI Driver Writer Guide also mentions that a better UEFI runtime driver is UNDI driver. In fact, students who have written network card drivers know that in UEFI environment, we only need to provide SNP driver (refer to VirtNet). UNDI driver is provided to support some OS to directly call the interface provided by UNDI to send and receive network packets when starting. (If you are interested, you can refer to Network Driver Design Guidelines)

UEFI runtime driver is a little more troublesome than general drivers. Please note that after OS starts, it uses virtual address, and there is a conversion relationship between its physical address and logical address. However, in UEFI environment, such as X86_64, it usually enters 64Bit Mode after DXE. Although paging mode is turned on, the logical address and physical address are in a 1:1 relationship. Therefore, some addresses driven by our UEFI runtime driver need to be converted, such as global variable address, address returned by allocate, register mmio address, etc. Of course, UEFI core will help convert the code address of UEFI runtime driver itself.

Take a simple example, the conversion relationship between virtual address and physical address under OS is f(x) = phyaddr + 0x1000;

AI generated projects 登录复制

```
// 变量A 地址是0x2000 存的值是100
.data
   0x2000 : 100
.text
    movl  (0x2000), %eax     ;  把A的值赋给寄存器eax
```

If in UEFI environment, the logical address and physical address of variable A are both 0x2000 and the value stored in the address is 100. However, if in OS, since MMU is enabled in OS, the addressing will be changed when movl (0x2000), %eax is used. As shown in the above relationship, phyaddr = f(x) - 0x1000 = 0x2000 - 0x1000 = 0x1000, so this line of code will give the value of physical address 0x1000 to eax, which is of course not the result we want. Therefore, we will call the EfiConvertPointer function to convert the address of A. The conversion result of A will become 0x3000. Therefore, executing movl (0x3000), %eax will give the value of physical address 0x2000 to eax.

The following is a detailed explanation of the UEFI runtime driver from UEFI to OS

**UEFI section:**

The uefi runtime core related code is a little scattered

- DxeMain.c
- DxeProtocolNotify.c
- Image.c
- Event.c
- Runtime.c

The connection between them is that DXE will temporarily declare an EFI_RUNTIME_ARCH_PROTOCOL. This Protocol will collect all drivers of type RUNTIME_DRIVER reported by Image.c into the ImageHead List, and will also collect all registered EVT_RUNTIME (EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE) Events reported by Event.c and put them into the EventHead List for use. When the real Runtime runs, the NotifyProtocol registered by DxeProtcolNotify.c will copy the content of the temporary EFI_RUNTIME_ARCH_PROTOCOL to the EFI_RUNTIME_ARCH_PROTOCOL registered by Runtime.c. Runtime.c will redirect all runtime driver images and the registered EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE variable.

c AI generated projects 登录复制 run

```c
//DxeMain.c
EFI_RUNTIME_ARCH_PROTOCOL gRuntimeTemplate = {
  INITIALIZE_LIST_HEAD_VARIABLE (gRuntimeTemplate.ImageHead),
  INITIALIZE_LIST_HEAD_VARIABLE (gRuntimeTemplate.EventHead),

  //
  // Make sure Size != sizeof (EFI_MEMORY_DESCRIPTOR). This will
  // prevent people from having pointer math bugs in their code.
  // now you have to use *DescriptorSize to make things work.
  //
  sizeof (EFI_MEMORY_DESCRIPTOR) + sizeof (UINT64) - (sizeof (EFI_MEMORY_DESCRIPTOR) % sizeof (UINT64)),
  EFI_MEMORY_DESCRIPTOR_VERSION,
  0,
  NULL,
  NULL,
  FALSE,
  FALSE
};

EFI_RUNTIME_ARCH_PROTOCOL *gRuntime = &gRuntimeTemplate;
```

收起 ^

c　　AI generated projects　登录复制　run

```c
// Image.c   每当调用LoadImage 就会把runtime driver 注册到链表Imagehead中

    if (Image->ImageContext.ImageType == EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER) {
      //
      // Make a list off all the RT images so we can let the RT AP know about them.
      //
      Image->RuntimeData = AllocateRuntimePool (sizeof(EFI_RUNTIME_IMAGE_ENTRY));
      if (Image->RuntimeData == NULL) {
        goto Done;
      }
      Image->RuntimeData->ImageBase      = Image->Info.ImageBase;
      Image->RuntimeData->ImageSize      = (UINT64) (Image->Info.ImageSize);
      Image->RuntimeData->RelocationData = Image->ImageContext.FixupData;
      Image->RuntimeData->Handle         = Image->Handle;
      InsertTailList (&gRuntime->ImageHead, &Image->RuntimeData->Link);
      InsertImageRecord (Image->RuntimeData);
    }
```

收起 ^

c　　AI generated projects　登录复制　run

```c
// 我们代码每次注册 gEfiEventVirtualAddressChangeGuid 的时候就会把相关信息收集到链表Eventhead中
  if ((Type & EVT_RUNTIME) != 0) {
    //
    // Keep a list of all RT events so we can tell the RT AP.
    //
    IEvent->RuntimeData.Type           = Type;
    IEvent->RuntimeData.NotifyTpl      = NotifyTpl;
    IEvent->RuntimeData.NotifyFunction = NotifyFunction;
    IEvent->RuntimeData.NotifyContext  = (VOID *) NotifyContext;
    IEvent->RuntimeData.Event          = (EFI_EVENT *) IEvent;
    InsertTailList (&gRuntime->EventHead, &IEvent->RuntimeData.Link);
  }
```

收起 ^

c　　AI generated projects　登录复制　run

```c
// DxeProtocolNotify.c 当Runtime.c 跑起来并且注册EFI_RUNTIME_ARCH_PROTOCOL 就会把临时数据拷贝到新的Runtime_Arch_Protocol上

    // Copy all the registered Image to new gRuntime protocol
    //
    for (Link = gRuntimeTemplate.ImageHead.ForwardLink; Link != &gRuntimeTemplate.ImageHead; Link = TempLinkNode.ForwardLink) {
      CopyMem (&TempLinkNode, Link, sizeof(LIST_ENTRY));
      InsertTailList (&gRuntime->ImageHead, Link);
    }
    //
    // Copy all the registered Event to new gRuntime protocol
    //
    for (Link = gRuntimeTemplate.EventHead.ForwardLink; Link != &gRuntimeTemplate.EventHead; Link = TempLinkNode.ForwardLink) {
      CopyMem (&TempLinkNode, Link, sizeof(LIST_ENTRY));
      InsertTailList (&gRuntime->EventHead, Link);
    }
```

收起 ^

Next is the UEFI Runtime Core entry function. It will first install EfiRuntimeArchProtocol. According to the above introduction of mRuntime's ImageHead, EventHead will copy the runtime type driver and related runtime events that run earlier than RuntimeDxe Driver to the mRuntime structure in Runtime.c in DxeProtocolNotify.c. In the future, mRuntime in Runtime.c will always be used. In this driver, we can see two core functions

1. RuntimeDriverSetVirtualAddressMap
2. RuntimeDriverConvertPointer

Before RuntimeCore, we should know what state our processor is in. Taking X86-64 as an example, we are currently in long mode (PE, PAE, PG, LME are all enabled), the page table is in 1:1 mapping state, and the page is 1G/2M/4K, which depends on the code implementation. If you want to check, the method is simple as follows PDPTE.PS ? 1G: (PDE.PS ? 2M : 4K) To establish a 1:1 page table, you can refer to CreateIdentityMappingPageTables;

RuntimeCore has two core functions: RuntimeDriverSetVirtualAddressMap and RuntimeDriverConvertPointer.
RuntimeDriverSetVirtualAddressMap is called by OS. Before calling this function, OS will call GetMemoryMap. We know that MemoryMap will get EFI_MEMORY_DESCRIPTOR as shown below. OS will fill in VirtualStart in all EFI_MEMORY_DESCRIPTORs with Runtime attributes, and then pass these MemoryMap descriptors to RuntimeDriverSetVirtualAddressMap together. Therefore, RuntimeDriverConvertPointer knows the conversion relationship between virtual address and physical address. What RuntimeDriverSetVirtualAddressMap has to do is to relocate all Runtime Drivers and convert the addresses of registered gEfiEventVirtualAddressChangeGuid Events. It is quite simple to relocate PE, you can refer to

UEFI File Types

c AI generated projects 登录复制 run

```c
typedef struct {
  UINT32                Type;
  EFI_PHYSICAL_ADDRESS  PhysicalStart;
  EFI_VIRTUAL_ADDRESS   VirtualStart;
  UINT64                NumberOfPages;
  UINT64                Attribute;
} EFI_MEMORY_DESCRIPTOR;
```

These are the logic of the entire UEFI Runtime Core. It is relatively simple and easy to understand the UEFI part alone. However, there are some difficulties from UEFI to OS as a whole, such as:

1. We know that RuntimeDriver is accessed by OS only after being relocated and translated by RuntimeDriverSetVirtualAddressMap in RuntimeCore . However, RuntimeDriverSetVirtualAddressMap is directly called by OS. So is the OS page table the same as UEFI? Will there be any problem with direct calling?
2. The OS is multi-tasking, so is it possible for the RuntimeDriver and other processes to access device resources at the same time? Will there be any problems?
3. OS is more concerned about performance and concurrency. So if RuntimeDriver is time-consuming, will it affect the performance of the operating system? Can RuntimeDriver be accessed by multiple processes at the same time? What kind of synchronization mechanism is used to access RuntimeDriver?
   We need to study other issues. The blog will be updated later on OS-related issues.

Regarding the second question, I would like to quote a passage from EBBR:

> 2.5.1. Runtime Device Mappings
> Firmware shall not create runtime mappings, or perform any runtime IO that will conflict with device access by the OS. Normally this means a device may be controlled by firmware, or controlled by the OS, but not both. e.g. If firmware attempts to access an eMMC device at runtime then it will conflict with transactions being performed by the OS.
> Devices that are provided to the OS (i.e., via PCIe discovery or ACPI/DT description) shall not be accessed by firmware at runtime. Similarly, devices retained by firmware (i.e., not discoverable by the OS) shall not be accessed by the OS.
> Only devices that explicitly support concurrent access by both firmware and an OS may be mapped at runtime by both firmware and the OS.