

UEFI Development Exploration 36 – UEFI Option ROM

原创

luobing4365

Posted on 2019-10-16 23:48:29


Read 4.8k

Collection 40

Likes 13

copyright

Category columns: [UEFI Development](#) Article Tags: [UEFI Programming](#) [UEFI OptionROM](#) [Low-level programming](#) [PCI/PCIE](#) [UEFI Driver](#)

UEFI Development This column includes this content

503 Subscribe 104 articles [Subscribe to our column](#)

(Please keep it-> Author: Luo Bing <https://blog.csdn.net/luobing4365>)

Option ROM development is not a well-known subject, and there is very little information on it. If you work for an ODM manufacturer or do BIOS-related work, you can get access to a lot of relevant materials. However, it is very rare for me to have to develop Option ROM and my company is not in the related industry. This also led to the fact that when developing OproM for Legacy BIOS, I could only explore little by little, and encountered a lot of problems.

Developing Option ROM under UEFI is much easier. On the one hand, the information is more comprehensive and the organization of EDK documents is relatively clear; on the other hand, it also provides standard tools, which is very convenient. My own test card is a [PCIE](#) interface, and the Option ROM discussed below is for PCI expansion ROM (ISA ROM does not seem to be supported in UEFI spec).

The UEFI Option ROM structure is similar to the Legacy BIOS OproM described earlier:

Offset	Byte Length	Value	Description
0x00	1	0x55	ROM Signature, byte 1
0x01	1	0xAA	ROM Signature, byte 2
0x02	2	XXXX	Initialization Size – size of this image in units of 512 bytes. The size includes this header.
0x04	4	0x0EF1	Signature from EFI image header
0x08	2	XX	Subsystem value for EFI image header
0x0a	2	XX	Machine type from EFI image header
0x0c	2	XX	Compression type 0x0000 - The image is uncompressed 0x0001 - The image is compressed. See the UEFI Compression Algorithm and Appendix H . 0x0002 - 0xFFFF - Reserved
0x0e	8	0x00	Reserved
0x16	2	XX	Offset to EFI Image
0x18	2	XX	Offset to PCIR Data Structure

Figure 1 UEFI Option ROM structure (From UEFI Spec 2.8 page 723)

After all, they are all derived from the PCI/PCIE specification and have the same structure as the Legacy Option ROM. The UEFI Option ROM uses the previously reserved byte (offset 0x04) to identify itself.

1 UEFI Option ROM loading process

In the source file MdeModulePkg\Bus\Pci\PciBusDxe\PciOptionRomSupport.c of MdeModulePkg, you can get a glimpse of the UEFI Option ROM processing process. The source file contains 8 functions for processing Option ROM:

The image shows a C++ IDE with the following components:

- EXPLORER:** Lists open files including `PciOptionRomSupport.c`, `Luo2.c`, and `PciOptionRomSupport.c`.
- MYWORKSPACE:** Shows the project structure with folders like `PCiHotPlugSupport.c`, `PCiHotPlugSupport.h`, `PCiIo.c`, `PCiIo.h`, `PCiLib.c`, and `PCiLib.h`.
- OUTLINE:** A list of functions and variables, with a red box highlighting the `OUTLINE` section. The items listed are:
 - `ContainEfiImage(IN VOID *, IN ...)`
 - `GetOpRomInfo(IN OUT PCI_IO_...`
 - `InitializePciLoadFile2(IN PCI_IO_...`
 - `LoadFile2(IN EFI_LOAD_FILE2_P...`
 - `LoadOpRomImage(IN PCI_IO_DEV...`
 - `LocalLoadFile2(IN PCI_IO_DEVIC...`
 - `ProcessOpRomImage(IN PCI_IO...`
 - `RomDecode(IN PCI_IO_DEVICE ...)`
- Code Editor:** Displays the source code for `PciOptionRomSupport.c`. The code includes comments in Chinese and C++ code for loading and processing the Option Rom image. The code is as follows:


```

/**
 * Load and start the Option Rom image.
 */
@param PciDevice      Pci device instance.

@return EFI_SUCCESS    Successfully loaded and started PCI Option R
@return EFI_NOT_FOUND  Failed to process PCI Option Rom image.

**/
EFI_STATUS
ProcessOpRomImage (
    IN  PCI_IO_DEVICE  *PciDevice
)
{
    UINT8                Indicator;
    UINT32               ImageSize;
    VOID                 *RomBar;
    UINT8                *RomBarOffset;
    EFI_HANDLE           ImageHandle;
    EFI_STATUS            RetStatus;
    EFI_PCI_EXPANSION_ROM_HEADER *EfiRomHeader;
    PCI_DATA_STRUCTURE    *PcIr;
    EFI_DEVICE_PATH_PROTOCOL *PciOptionRomImageDevicePa
    MEDIA_RELATIVE_OFFSET_RANGE_DEVICE_PATH
    VOID                 *EfiOpRomImageNode;
    VOID                 *Buffer;
    UINTN                BufferSize;

    Indicator = 0;

    //
    // Get the Address of the Option Rom image
    //
    RomBar      = PciDevice->PciIo.RomImage;
    RomBarOffset = (UINT8 *) RomBar;
    RetStatus    = EFI_NOT_FOUND;

    if (RomBar == NULL) {
        return RetStatus;
    }
    ASSERT (((EFI_PCI_EXPANSION_ROM_HEADER *) RomBarOffset)->Signature

```

Figure 2 Code for processing UEFI Option ROM in UDK2018 MdeModulePkg

The executable file of Option ROM is not run on Flash, it will be copied to memory and then executed in memory. Legacy BIOS Option ROM is generally loaded to 0xC0000~0xE0000 (ie 0xC000 segment to 0xE000 segment), while UEFI Option ROM does not have such a convention.

Carefully reading the code of `ProcessOpRomImage()` gives you a glimpse into the process of handling Option Rom.

ProcessOpRomImage() has only one entry parameter: PCI_IO_DEVICE *PciDevice. This is a pointer to a PCI device, which contains all the properties of the device and the information of its sibling devices and parent devices. The data structure PCI_IO_DEVICE is defined in PciBus.h and can be understood by comparing it with the PCI protocol.

The function uses a do-while loop to process the device and find the ROM signature of the Option ROM (that is, 0xAA55). Then it analyzes the ROM structure, including whether it is an EFI image, whether the machine type is supported, etc., and creates its device path (DevicePath).

In UEFI, Device Path is used to describe the location information of a device. It is also often used to describe buses, boot items, etc. When entering the UEFI shell, the string description of each device that appears is the Device Path:

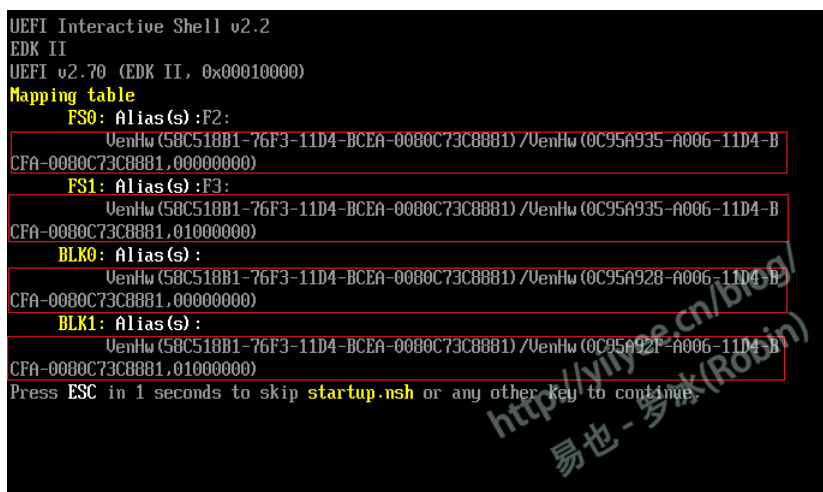


Figure 3 Startup information in TianoCore simulation environment

There are six types of Device Path, which are described in detail in Uefi Spec:

10.3.1 Generic Device Path Structures

A Device Path is a variable-length binary structure that is made up of variable-length generic Device Path nodes. [Table 44](#) defines the structure of a variable-length generic Device Path node and the lengths of its components. The table defines the type and sub-type values corresponding to the Device Paths described in [Section 10.3](#); all other type and sub-type values are *Reserved*.

Table 44. Generic Device Path Node Structure

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 0x01 – Hardware Device Path Type 0x02 – ACPI Device Path Type 0x03 – Messaging Device Path Type 0x04 – Media Device Path Type 0x05 – BIOS Boot Specification Device Path Type 0x7F – End of Hardware Device Path
Sub-Type	1	1	Sub-Type – Varies by Type. (See Table 45 .)
Length	2	2	Length of this structure in bytes. Length is 4 + n bytes.
Specific Device Path Data	4	n	Specific Device Path data. Type and Sub-Type define type of data. Size of data is included in Length.

Figure 4 Device Path Classification (UEFI Spec 2.8 page 285)

The Option ROM device path type created in the function is MEDIA_DEVICE_PATH, which is a Device Path that can be used as a startup device.

After creation, call LoadImage() and StartImage() to execute the Option ROM code.

```
// load image and start image
//
BufferSize = 0;
Buffer = NULL;
ImageHandle = NULL;

Status = gBS->LoadImage (
    FALSE,
    gPciBusDriverBinding.DriverBindingHandle,
    PciOptionRomImageDevicePath,
    Buffer,
    BufferSize,
    &ImageHandle
);
if (EFI_ERROR (Status)) {
    //
    // Record the Option ROM Image device path when LoadImage fails.
    // PciOverride.GetDriver() will try to look for the Image Handle using the device path later.
    //
    AddDriver (PciDevice, NULL, PciOptionRomImageDevicePath);
} else {
    Status = gBS->StartImage (ImageHandle, NULL, NULL);
    if (!EFI_ERROR (Status)) {
        //
        // Record the Option ROM Image Handle
        //
        AddDriver (PciDevice, ImageHandle, NULL);
        PciRomAddImageMapping (
            ImageHandle,
            PciDevice->PciRootBridgeIo->SegmentNumber,
            PciDevice->BusNumber,
            PciDevice->DeviceNumber,
            PciDevice->FunctionNumber,
            PciDevice->PciIo.RomImage,
            PciDevice->PciIo.RomSize
        );
        RetStatus = EFI_SUCCESS;
    }
}
FreePool (PciOptionRomImageDevicePath);
```

Figure 5 ProcessOptionRomImage() in PciOptionRomSupport.c

2How to Generate UEFI Option ROM

UEFI Option ROM is actually a type of UEFI driver. EDKII provides the corresponding tools to convert the generated file into Option ROM. As mentioned before, the option ROM we are currently developing is mainly PCI Option ROM. For the content of PCI Option ROM, please refer to "EDKII Driver Writer's Guide for UEFI 2.3.1". The following introduction is also mainly from this document.

There are two ways to generate PCI Option ROM image, use EfiRom conversion tool or directly use EDKII INF/FDF file to compile and generate.

EfiRom provides [source code](#), allowing users to compile on any operating system that supports EDKII. The source code is located in \BaseTools\Source\C\EfiRom. On my development machine (Win10), the compiled executable file is located in \BaseTools\Bin\Win32.

It provides the following functions:

```

C:\MyWorkspace\BseTools\Bin\Win32>EfiRom.exe
Usage: EfiRom -f VendorId -i DeviceId [options] [file name(s)]

Copyright (c) 2007 - 2017, Intel Corporation. All rights reserved.

Options:
-o FileName, --output FileName
    File will be created to store the output content.
-e EfiFileName
    EFI PE32 image files.
-ec EfiFileName
    EFI PE32 image files and will be compressed.
-b BinFileName
    Legacy binary files.
-l ClassCode
    Hex ClassCode in the PCI data structure header.
-r Rev
    Hex Revision in the PCI data structure header.
-n
    Not to automatically set the LAST bit in the last file.
-f VendorId
    Hex PCI Vendor ID for the device OpROM, must be specified
-i DeviceId
    One or more hex PCI Device IDs for the device OpROM, must be specified
-p, --pci23
    Default layout meets PCI 3.0 specifications
    specifying this flag will for a PCI 2.3 layout.
-d, --dump
    Dump the headers of an existing option ROM image.
-v, --verbose
    Turn on verbose output with informational messages.
--version
    Show program's version number and exit.
-h, --help
    Show this help message and exit.
-q, --quiet
    Disable all messages except FATAL ERRORS.
--debug [#]
    Enable debug messages at level #.

```

Figure 6 EfiRom function list

UEFI Option ROM is converted from UEFI Driver. As for how to write UEFI driver, that is another topic and will not be discussed here.

EfiRom will verify the incoming efi file (UEFI Driver), such as whether the Rom header is 0xAA55, whether the PCI data structure identifier is "PCIR", etc. If any check fails, EfiRom will exit and the process of creating Option ROM will be terminated.

The generation command is as follows:

```
EfiRom -f 0x9999 -i 0x8000 -e pcidriver.efi
```

Among them, -f specifies the Vendor ID, -i specifies the Device ID, and -e specifies the file to be converted. For more conversion methods, including how to package and convert with Legacy Option ROM, how to compress, etc., please refer to the manual "EDKII Driver Writer's Guide for UEFI 2.3.1" mentioned above, Chapter 18, Section 7.

Another conversion method is to use INF/FDF. When the build command is executed, efirrom is automatically called to convert it to the specified Option ROM. This is the method I often use. The conversion process is completed while compiling. A typical INF example is as follows:

```

17 [Defines]
18   INF_VERSION           = 0x00010005
19   BASE_NAME              = UefiDrv
20   FILE_GUID              = 0A8830B50-5822-4f13-99D8-D0DCAFD631C3
21   MODULE_TYPE            = UEFI_DRIVER
22   VERSION_STRING         = 1.0
23   ENTRY_POINT            = UefiMain
24   UNLOAD_IMAGE          = DefaultUnload
25
26   PCI_VENDOR_ID = 0x9999
27   PCI_DEVICE_ID = 0x8000
28   #PCI_CLASS_CODE = 0x018000
29   PCI_CLASS_CODE = 0x020000
30   PCI_REVISION = 0x0003
31   PCI_COMPRESS = TRUE

```

Figure 7 INF example

The Vendor ID and Device ID can be specified in the Inf file, and other parameters including the PCI class code, PCI version, and whether to compress the Option ROM (PCI_COMPRESS) can also be specified.

Whether you use the EfiRom tool to convert directly or use the Inf file, you can only process one UEFI Driver. If you need to manage multiple UEFI Drivers at the same time and generate multiple types of Option ROMs (IA32, X64, etc.), you can use the FDF file to process. I will not discuss the specific content one by one, and you can also refer to the above programming manual.

3 Software Structure

UEFI Option ROM is essentially UEFI Driver, so it is best to find a relatively "pure" Driver example to start with. I searched for a long time in the EDKII section of github, but still couldn't find a suitable one. EDKII now provides a tool for developing drivers, UEFI Driver Wizard, which I haven't used, but perhaps many examples are integrated into it.

I ended up using BlankDrv which I used before, and I always use it as the basis for building UEFI Option ROM. It can still be downloaded here:

<https://sourceforge.net/projects/edk2/files/EDK%20II%20Releases/Demo%20apps/>

The development of Driver is described in detail in "UEFI Principles and Programming", which is worth studying carefully. However, my goal is to develop UEFI Option ROM, so many details do not need to be studied in depth.

The developed Option ROM complies with the UEFI driver module, which is actually a PCI Driver, so it must implement `EFI_DRIVER_BINDING_PROTOCOL` and instantiate the three services `Supported()`, `Start()`, and `Stop()`.

We have already discussed how to enable Option ROM. Now we are mainly concerned about where to add the actual execution code.

Most of the execution code is added in `Start()`. The main task of `Start()` is to start the **hardware** device. The most important thing in the function is to call `InstallProtocolInterface()` or `InstallMultipleProtocolInterfaces()` to install the driver Protocol on `ControllerHandle`.

The `Stop()` function is used to uninstall the driver and stop the hardware device, and basically there is no need to modify the original code provided by `BlankDrv`.

`Supported()` is used to evaluate whether the PCI controller specified by the device handle passed to the Driver can be driven by the driver, mainly determined by Vendor ID, Device ID, and Class code.

As mentioned before, the specific driver details can be referred to other documents. After the architecture is built, you can focus on the implementation of Option ROM functions. The topics discussed in the previous 35 blogs and the codes written can be completely ported to Option ROM as long as the expansion ROM size of the hardware device is sufficient.

```

    Status_Command|=0x07;
    Status = PciIo->Pci.Write (
        PciIo,
        EfiPciIoWidthUint8,
        4, // word 0,1 = VendorID
        1,
        &Status_Command
    );
    if (EFI_ERROR (Status)) {
        goto Done;
    }
    //IoBaseAddr= (IoBaseAddr & 0xffff); //clear bit0
    // if (IoBaseAddr < 0x0000) {
    //     goto Done;
    // }
    //gST->ConOut->OutputString(gST->ConOut,L"Try!\n");
    //gST->ConIn->GetKey();
    //InstallMultipleProtocolInterfaces(S_TEXT_INPUT_EX|GRAPHICS_OUTPUT|PCI_ROOTBRIDGE_IO|PCI_IO);
    //GetKey();
    Status = GetGraphicsModeNumber(gGraphicsOutput, HorRes, VerRes, &DualNetModeNum, TRUE, NULL, NULL);

```

BlankDrvDriverBindingStart()

此处开始编写实际执行代码

Figure 8 UEFI Option ROM code

4 Compilation and Demonstration

Due to the company's business reasons, the original code cannot be posted. I am considering using another PCI-E chip as a test card with a larger ROM space to do some relatively complex functions, and then I will make the code public.

For the process of compiling and writing to the test card, please refer to the blog: "UEFI Development Exploration 12 – OproM Test Board".

Here is the test result again:

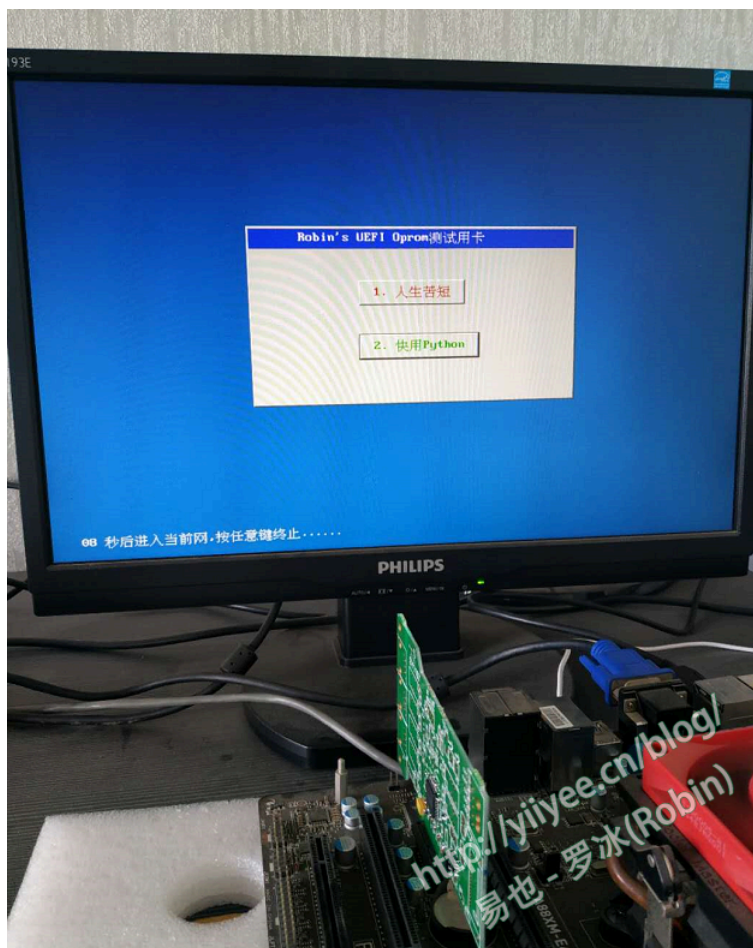


Figure 9 UEFI Option ROM demonstration

At this point, the development of UEFI Option ROM, the main line of the UEFI exploration series, has been completed.

It was a great experience for me. It has been more than half a year since I decided to write this series of blogs. I force myself to sit down at my desk and start typing every week. It has become a must-do for me every week.

In this process, I have completed a lot of knowledge. However, it seems that the more I understand, the more I don't understand.

There are 14 more articles planned, which will be organized according to the topics I am interested in. **This journey of exploration is far from over.**