UEFI Development Exploration 43 – Protocol Usage 2

原创 luobing4365 Posted on 2020-02-28 11:33:02 Read 1.8k Collection 10 Likes 4 copyright

Category Column: UEFI Development Article Tags: UEFI Programming UEFI protocol Low-level programming BIOS Programming

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

Today, let's explore the third question raised last time: How to generate a Protocol?

In the often-read book "UEFI Principles and Programming", how to develop UEFI services has actually been introduced. He takes video decoding as an example and provides a complete decoding library.

I have no interest in video decoding at the moment, so this article has too many unnecessary details for me. I plan to build a relatively simple framework code that can be used to draw geometric figures on the screen to familiarize myself with how to develop Protocol.

1 UEFI Driver

Compared with Windows driver, UEFI driver is much simpler and can be roughly divided into two categories: drivers that conform to the UEFI driver model and drivers that do not conform to the UEFI driver model, as shown in the figure:

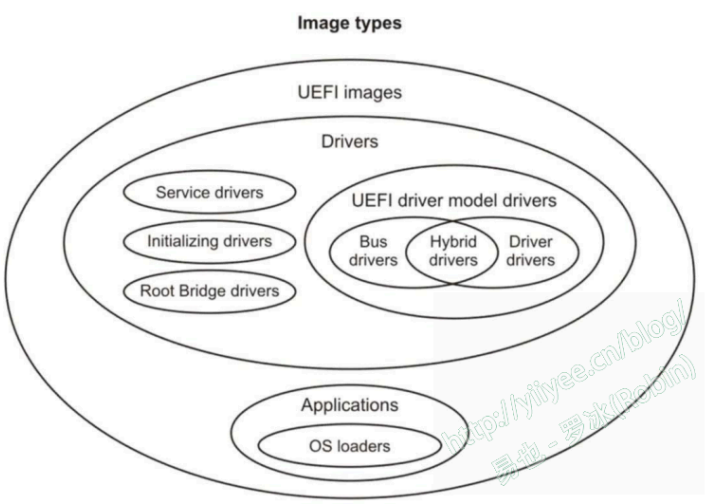

Figure 1 UEFI Images – EDKII Driver Writer's Guide section3.7

1) Service Drivers

A service driver does not manage any device, nor does it generate any EFI_DRIVER_BINDING_PROTOCOL interface, and it does not have a Device Path Protocol. This type of driver will load one or more Protocols on one or more Service Handles and return EFI_SUCCESS in its Entry Point.

2) Initializing Drivers

It will not generate any Handle, nor will it add any Protocol to the Handle Database. It will only perform some initialization operations and return an error code. Therefore, this type of Driver will be unloaded from the system memory after execution.

3) Root Bridge Drivers

One or more Control Handles will be generated, and contain a Device Path Protocol and a Protocol that abstracts the I/O resources provided by the chip and bus in a software manner. The most common is a Driver that generates handles for the PCI Root Bridge, and the generated Handle supports the Device Path Protocol and the PCI Root Bridge I/O Protocol.

4) UEFI Driver Model Drivers

4-A Bus Drivers

One or more Driver Handles or Driver Image handles will be generated in the Handle Database , and one or more instances of the Driver Binding Protocol will be installed on the Handle. This type of driver will generate new Child Handles when calling the Start() function of the Driver Binding Protocol, and will add additional I/O Protocols to the new Child Handles.

4-B Device Drivers

The difference from Bus Drivers is that no new Child Handles will be generated, only additional I/O Protocols will be added to the existing Child Handles.

4-C Hybrid Drivers

It has the characteristics of both Bus Drivers and Device Drivers, which not only adds I/O Protocol to the existing Handle, but also generates new Child Handles.

The UEFI Option ROM I wrote before is actually a PCI Device Driver designed for a specified test card (PID and VID are 0x9999 and 0x8000 respectively).

Our goal is to develop UEFI Protocol, which can be achieved using a service driver. The details of this driver are discussed in detail below.

2Service Drivers

The service-oriented driver is relatively simple and is mainly used to produce Protocol. The following examples are provided for reference:

MdeModulePkg/Universal/Acpi/AcpiTableDxe
MdeModulePkg/Universal/DebugSupportDxe MdeModulePkg
/Universal/DevicePathDxe
MdeModulePkg/Universal/EbcDxe
MdeModulePkg/Universal/HiiDatabaseDxe
MdeModulePkg/Universal/PrintDxe
MdeModulePkg/Universal/SetupBrowserDxe
MdeModulePkg/Universal/SmbiosDxe
MdeModulePkg/Universal/HiiResourcesSampleDxe

Service drivers do not need to implement the functions required by UEFI drivers such as Start, Stop, and Support. When the Image is initialized (that is, the module entry function), it is sufficient to install the required Protocol.

For demonstration and to facilitate future reuse, I built a more service-oriented driver framework code. The code is modified based on the HiiResourcesSampleDxe example (in UDK2018). The main steps are as follows:

1) Modify the contents of the *.inf file

MODULE_TYPE is changed to UEFI_DRIVER;
UefiDriverEntryPoint is added to the [LibraryClasses] section;

As shown in the figure:

```
[Defines]
  INF_VERSION                    = 0x00010005
  BASE_NAME                      = ServiceDrvSample
  #MODULE_UNI_FILE                = HiiResourcesSample.uni
  FILE_GUID                      = D49A2EB0-44D5-4E21-9FB6-1A9AC9109B99
  MODULE_TYPE                    = UEFI_DRIVER
  VERSION_STRING                 = 1.0
  ENTRY_POINT                    = MyProtocolEntry
  #UNLOAD_IMAGE                   = HiiResourcesSampleUnload
#
#  This flag specifies whether HII resource section is generated into PE image.
#
  #UEFI_HII_RESOURCE_SECTION      = TRUE

#
# The following information is for reference only and not required by the build tools.
#
#  VALID_ARCHITECTURES            = IA32 X64 IPF EBC
#

[Sources]
  ServiceDrvSample.c
  MyProtocol.c
  MyProtocol.h

[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec

[LibraryClasses]
  #UefiBootServicesTableLib
  UefiDriverEntryPoint
  UefiLib
  #UefiHiiServicesLib
  #HiiLib
```


Figure 2 Modify the inf file

2) Add entry function code

After building the protocol (the next section describes how to build your own protocol), you need to install it in the driver entry function. You can use InstallProtocolInterface() or InstallMultipleProtocolInterfaces() to install it. The UEFI spec believes that InstallMultipleProtocolInterfaces() will perform more error checks, so it is recommended to use this function.

However, InstallProtocolInterface() is easier to use, and I use it in my example. I didn't explain these two functions in the previous blog, so I'll add them here:

```
typedef EFI_STATUS (EFIAPI *EFI_INSTALL_PROTOCOL_INTERFACE) (
  IN OUT EFI_HANDLE  *Handle ,  //Protocol is installed here
  IN EFI_GUID  *Protocol ,  // GUID of the Protocol to be installed   IN EFI_INTERFACE_TYPE  InterfaceType ,  // Generally EFI_NATIVE_INTERFACE
IN VOID  *Interface //Protocol instance );
```

```
typedef EFI_STATUS EFIAPI *EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES) (
  IN OUT EFI_HANDLE  *Handle, //Protocol is installed here
  ...  // Protocol GUID and Protocol instance appear in pairs );
```

According to the description in UEFI Spec, InstallMultipleInterface() still calls InstallProtocolInterface() internally for processing. There is an example of using this function in UDWG (driver writing manual), which you can take a look at.

Finally, take a look at the driver entry function I wrote:

```
**/
EFI_STATUS
EFIAPI
MyProtocolEntry (
  IN EFI_HANDLE                  ImageHandle,
  IN EFI_SYSTEM_TABLE            *SystemTable
  )
{
  EFI_STATUS                     Status;
  MySampleProtocolInit();

  Print(L"Install RobinSampleProtocol...\n");
  Status = gBS->InstallProtocolInterface (
            &ImageHandle,
            &gEfiMYSampleProtocolGUID ,
            EFI_NATIVE_INTERFACE,
            &RobinSampleProtocol
            );
  if(Status == EFI_SUCCESS)
    Print(L"Success!\n");
  else
    Print(L"Fail!\n");

  return Status;
}
```


Figure 3 Example driver entry function

3. Build your own protocol

Building a protocol requires several parts:

1) Protocol GUID, which can be generated using Microsoft tools or online, such as https://www.guidgen.com/ . Of course, you can just copy one and change a few characters;

2) Construct the member functions and structures of Protocol ;

3) Use the Protocol structure to instantiate a required Protocol;

The member function of Protocol must be of EFIAPI modified type, and the first function parameter must be the This pointer. The constructed Protocol source code is as follows:

```
/**   Sample of protocol,

  @param  This       Indicates a pointer to the calling context.
  @retval EFI_SUCCESS           The video is opened successfully.
  @retval EFI_NOT_FOUND         There is no such file.
**/
typedef
EFI_STATUS
(EFIAPI* EFI_MYSAMPLE_OUT)(
        IN EFI_MYSAMPLE_PROTOCOL* This
        );
/**
 * **/
struct _EFI_MYSAMPLE_PROTOCOL{
 UINT64          Revision;
 EFI_MYSAMPLE_IN  MySample_In;
 EFI_MYSAMPLE_OUT MySample_Out;
 EFI_MYSAMPLE_DOSTH MySample_DoSth;
};

EFI_MYSAMPLE_PROTOCOL RobinSampleProtocol;
//------------------------------------------------  GUID data  ------------------------------------------------
#define EFI_MYSAMPLE_PROTOCOL_GUID \
{ \
    0xce345181, 0xabad, 0x11e2, {0x8e, 0x5f, 0x0, 0xa0, 0xc9, 0x69, 0x72, 0x3b } \
}

EFI_GUID gEfiMYSampleProtocolGUID = EFI_MYSAMPLE_PROTOCOL_GUID ;
```


Figure 4 Building your own Protocol

Of course, you can also use the This pointer to transfer internal private data to facilitate sharing among functions. The video decoding code in "UEFI Principles and Programming" provides relevant processing methods.

4 Test Code

To facilitate testing, I wrote two programs, one is a service driver that provides the protocol, and the other is a test program to test the protocol. As usual, the code download address is provided at the end of the article.

I won't explain the code one by one, the function is relatively simple: the Protocol provided in the service driver has three member functions, as can be seen in Figure 4 above. These three functions simply print some information for demonstration.

The test results are as follows:

Figure 5 Test results

As can be seen from the animated picture, before installation, when running TestSDSample.efi, it was found that the required Protocol could not be found.

Use the command Load -nc ServiceDrvSample.efi to install the self-made Protocol. Then run the test program TestSDSample.efi and find that the protocol we built ourselves can be found and each member function can be used normally.

Gitee address: https://gitee.com/luobing4365/uefi-explorer
The project code is located in: / FF RobinPkg/RobinPkg/Applications/TestServiceDrvSample and /FF RobinPkg/RobinPkg/Drivers/ServiceDrvSample