UEFI Development Exploration 68- YIE001PCIe Development Board (04 UEFI Driver)

原创 luobing4365 Posted on 2021-01-15 13:12:00 Read 787 Collection 1 Likes

Category Column: UEFI Development Article Tags: UEFI BIOS Low-level application development UEFI Drivers YIE001 Development Board

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

A new announcement: The idea of the YIE001 development board is relatively clear - use the PCIE chip, write the Option ROM, control the hardware on this basis, and do some interesting experiments. Since it is mainly in my spare time, it is estimated that the experiments to be done can be completed within two months.

Therefore, I have a new idea - to make a USB development board YIE002, which can realize full-channel USB HID device accessed from UEFI and Windows/Linux. Of course, it can be used to collect temperature, generate random numbers, etc. It is prepared to be made into a USB flash drive size, which is convenient for directly plugging into the computer for experiments. I am collecting ideas from netizens. If you have anything you want to realize, just leave a message. If resources allow, I will realize it on the board.

Back to the introduction of UEFI driver.

UEFI Development Exploration 66 and 67 introduced the construction and implementation of service-oriented drivers. The following articles are used to introduce drivers that conform to the UEFI driver model. This article mainly introduces its framework structure and program interface. The content is a bit boring, so it is best to read it with reference to the source code.

A complete driver that complies with the UEFI driver model can be roughly divided into two parts: EFI Driver Binding Protocol and the services provided by the driver itself. The former is used to manage the driver, and the latter is the part that users need to use, generally including one or more protocols.

For example, the UEFI USB host controller driver provides two protocols for users to use, including EFI_USB_HC_PROOCOL and EFI_USB2_HC_PROTOCOL, which are used to access USB devices. In addition, for the convenience of users, the driver program generally also includes the EFI Component Name Protocol, which is used to display driver information.

On the webpage https://sourceforge.net/projects/edk2/files/EDK%20II%20Releases/Demo%20apps/ , you can download the driver framework example BlankDrv provided by Intel earlier to learn the UEFI driver model.

1 EFI Driver Binding Protocol

The sample project BlankDrv contains four files:

1. BlankDrv.c implements EFI_DRIVER_BINDING_PROTOCOL and its interface functions, and installs EFI_DRIVER_BINDING_PROTOCOL and EFI_COMPONENT_NAME_PROTOCOL.
2. BlankDrv.h. Defines the data structure required by the driver and the prototype of the Protocol interface function;
3. ComponentName.c. Implements the interface function of EFI_COMPONENT_NAME_PROTOCOL;
4. BlankDrv.inf, INF file for compiling UEFI driver.

The EFI_DRIVER_BINDING_PROTOCOL interface function implemented in the BlankDrv.c file is the core of the UEFI driver being managed by the system. The structure of EFI_DRIVER_BINDING_PROTOCOL is shown in Listing 1.

Code Listing 1 EFI_DRIVER_BINDING_PROTOCOL structure

```
typedef struct _EFI_DRIVER_BINDING_PROTOCOL {  EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED Supported; //Check if the device controller supports the driver  EFI_DRIVER_BINDING_PROTOCOL_START Start; //Install the driver and start the device  EFI_DRIVER_BINDING_PROTOCOL_STOP Stop; //Stop the device and uninstall the driver  UINT32 Version; //Version  EFI_HANDLE ImageHandle; //Image handle  EFI_HANDLE DriverBindingHandle; //Protocol instance installed on it } EFI_DRIVER_BINDING_PROTOCOL;
```

EFI_DRIVER_BINDING_PROTOCOL has 3 interface functions and 3 interface variables. Among them, the interface variable ImageHandle is the image handle that generates this Protocol instance, and DriverBindingHandle is the handle of the installed Protocol instance. In most cases, the two are the same. Of course, if the driver generates multiple Protocol instances, the two are not the same.

The interface variable Version indicates the version number of the driver. The ConnectController() function that starts the service uses it to determine which driver service to use. The higher version is installed on the device first. 0x0~0x0f and 0xfffffff0~0xffffffff are reserved for platform and OEM drivers, and 0x10~0xfffffef is reserved for drivers developed by third-party independent hardware vendors (IHVs).

The three interface functions of EFI_DRIVER_BINDING_PROTOCOL, Supported(), Start() and Stop(), are the core of this Protocol, which are used to detect drivers, install drivers and uninstall drivers respectively.

1.1 Supproted() function

The Supported() function is used to detect whether a given device controller supports the driver. Its function prototype is shown in Code Listing 2.

Code Listing 2 Supported() function prototype

```
typedef EFI_STATUS (EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED) (
  IN EFI_DRIVER_BINDING_PROTOCOL *This, //Protocol instance
  IN EFI_HANDLE ControllerHandle, //Device controller handle
  //If non-NULL, the bus driver uses this parameter; this parameter is invalid for the device driver
  IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
);
```

This function may be called multiple times when the platform is initialized. Therefore, in order to shorten the startup time, the detection process and the usage time should be as short as possible. During the execution of this function, the status of the hardware device cannot be modified. In addition, when implementing this function code, you should always be aware that the device controller handle may have been used by other drivers or this driver.

When calling the interface functions provided by the startup service, you need to pay attention to the matching. If you use AllocatePages(), you must use FreePages() in conjunction with it, AllocatePool() and FreePool() in conjunction with it, and OpenProtocol() and CloseProtocol() in conjunction with it. The sample project BlankDrv has complete code, please check it yourself.

1.2 Start() function

The Start() interface function is used to install the driver on the device and start the hardware device. The Protocol provided by the UEFI driver is generally installed in this function using the InstallProtocolInterface() or InstallMultipleProtocolInterfaces() function. Its function prototype is shown in Code Listing 3.

Code Listing 3 Start() function prototype

```
typedef EFI_STATUS (EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_START) (
  IN EFI_DRIVER_BINDING_PROTOCOL *This, //Protocol instance
  IN EFI_HANDLE ControllerHandle, //Controller handle installed by the driver
  IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
);
```

The third parameter, RemainingDevicePath, is used by the bus driver to specify how to create a child device handle; for the device driver, this parameter can be ignored.

It should be noted that the resources requested in the Start() function must be released in the Stop() function. That is, when AllocatePool(), AllocatePages(), OpenProtocol(), and InstallProtocolInterface() are used in the Start() function, FreePool(), FreePages(), CloseProtocol(), and UninstallProtocolInterface() should be used in the Stop() function.

In the sample project BlankDrv, the Start() function does not implement any specific functionality. It just prints a string indicating that the function is running and returns EFI_UNSUPPORTED.

1.3 Stop() function

The Stop() function is used to stop the hardware device and uninstall the driver. It is a mirror image of the Start() function. Its function prototype is shown in Listing 4.

Code Listing 4 Stop() function prototype

```
typedef EFI_STATUS (EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_STOP) (
  IN EFI_DRIVER_BINDING_PROTOCOL *This, //Protocol instance
  IN EFI_HANDLE ControllerHandle, //Stop the driver corresponding to this controller handle
  IN UINTN NumberOfChildren, //Number of child controllers
  IN EFI_HANDLE *ChildHandleBuffer OPTIONAL //Child controller array
);
```

This function has different operations depending on the number of subcontrollers. For device drivers, if the number of subcontrollers is 0, the subcontroller array is NULL; for bus drivers, if the number of subcontrollers is not 0, all child node handles saved in the subcontroller array will be released.

Since the Start() function of the sample project BlankDrv does not implement any specific functions, the corresponding Stop() function does not need to release any resources.

2 EFI Component Name Protocol

For user convenience, UEFI drivers usually provide names to facilitate displaying driver information to users. This function can be provided by EFI_COMPONENT_NAME_PROTOCOL or EFI_COMPONENT_NAME2_PROTOCOL.

The two protocols have the same functions and the same structure, the only difference is the format of the language code. The former uses ISO 639-2 language codes, and the latter uses RFC 4646 language codes.

Listing 5 shows the structure of EFI_COMPONENT_NAME2_PROTOCOL.

Code Listing 5 EFI_COMPONENT_NAME2_PROTOCOL structure

```
typedef struct _EFI_COMPONENT_NAME2_PROTOCOL {  EFI_COMPONENT_NAME_GET_DRIVER_NAME GetDriverName; //Get the driver name EFI_COMPONENT_NAME_GET_CONTROLLER_NAME GetControllerName; //Get the controller name  CHAR8 *SupportedLanguages; //List of supported languages, this protocol is RFC 4646 } EFI_COMPONENT_NAME2_PROTOCOL; //Return the driver name according to the specified language code typedef EFI_STATUS (EFIAPI *EFI_COMPONENT_NAME_GET_DRIVER_NAME) (  IN EFI_COMPONENT_NAME2_PROTOCOL *This, //Protocol instance  IN CHAR8 *Language, //Language code  OUT CHAR16 **DriverName //Return the driver name ); //Return the controller or sub-controller name according to the specified language code typedef EFI_STATUS (EFIAPI *EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) (  IN
```

*EFI_COMPONENT_NAME2_PROTOCOL *This, //Protocol instance IN EFI_HANDLE ControllerHandle, //Controller handle IN EFI_HANDLE ChildHandle OPTIONAL, //Child controller handle IN CHAR8 *Language, //Language code OUT CHAR16 **ControllerName //Controller or child controller name);*

The structure and interface functions of EFI_COMPONENT_NAME_PROTOCOL are exactly the same as those given in Listing 5, except that the list of supported languages is ISO 639-2.

In the sample project BlankDrv, the variables defined for these two protocols are shown in Example 1.

[Example 1] Define two Protocol variables

EFI_COMPONENT_NAME_PROTOCOL gBlankDrvComponentName = { BlankDrvComponentNameGetDriverName, BlankDrvComponentNameGetControllerName, "eng" //List of supported languages }; EFI_COMPONENT_NAME2_PROTOCOL gBlankDrvComponentName2 = { (EFI_COMPONENT_NAME2_GET_DRIVER_NAME)BlankDrvComponentNameGetDriverName, (EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME) \ BlankDrvComponentNameGetControllerName, "en" //List of supported languages };

As can be seen from Example 1, the two interface functions are identical except for the list of supported languages (which are actually the same, except for the language codes used).

For the implementation of the two interface functions, you can directly view the source code in BlankDrv, which is not listed here.

At this point, the basic construction of the UEFI driver model has been completed. In the driver framework, there are still functions for installing the Protocol and uninstalling the driver that have not been implemented. In the next article, the construction of the entire driver framework will be completed, and how to test such a driver under the UEFI Shell will be introduced.