UEFI Development Exploration 52 – UEFI and Network 3 (UEFI TCP4)

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

After setting up the network testing environment, you can start network programming.

UEFI provides the corresponding Protocol, which can be used for TCP and UDP programming, and provides corresponding support for IPv4 and IPv6. In addition, programming can also be performed through the Socket interface encapsulated in StdLib.

If all programming methods are implemented, the blog will be too long. I originally planned to use about 5 blogs to explore network programming. Therefore, I plan to use UEFI Protocol to write TCP (IPv4) examples and UDP (IPv4) examples, and StdLib interface to re-implement the above examples.

1 Use of EFI_TCP4_PROTOCOL

Most UEFI protocols can be found through the device GUID and accessed directly. Unlike other UEFI protocols, the network needs to frequently generate new sockets . In the specification, two protocols are provided for TCP4.

One is EFI_TCP4_PROTOCOL, which can perform TCP network configuration and communication; the other is EFI_TCP4_SERVICE_BINDING_PROTOCOL, which is used to generate EFI_TCP4_PROTOCOL instances.

However, in the implementation of EDK2, EFI_TCP4_SERVICE_BINDING_PROTOCOL is not provided, but EFI_SERVICE_BINDING_PROTOCOL is provided for all network protocols. In other words, although various protocols named EFI_XXXX_SERVICE_BINDING_PROTOCOL are provided in the Spec, they all use EFI_SERVICE_BINDING_PROTOCOL.

In contrast, EFI_SERVICE_BINDING_PROTOCOL itself does not have a GUID. Protocols of other network protocols have their own GUIDs, such as TCP4, UDP4, TCP6, etc. They share the same EFI_SERVICE_BINDING_PROTOCOL interface to generate their own instances.

1) EFI_SERVICE_BINDING_PROTOCOL interface description

typedef struct **_EFI_SERVICE_BINDING_PROTOCOL** { **EFI_SERVICE_BINDING_CREATE_CHILD** *CreateChild; //Generate a child device and install the corresponding Protocol* **EFI_SERVICE_BINDING_DESTROY_CHILD** *DestroyChild; //Destroy the generated child device* } **EFI_SERVICE_BINDING_PROTOCOL** ;

typedef EFI_STATUS (EFIAPI *EFI_SERVICE_BINDING_CREATE_CHILD) (
IN EFI_SERVICE_BINDING_PROTOCOL **This , //EFI_SERVICE_BINDING_PROTOCOL instance*
IN OUT EFI_HANDLE **ChildHandle // Created child device handle*
);

typedef EFI_STATUS (EFIAPI *EFI_SERVICE_BINDING_DESTROY_CHILD) (
IN EFI_SERVICE_BINDING_PROTOCOL **This, //EFI_SERVICE_BINDING_PROTOCOL instance*
IN EFI_HANDLE *ChildHandle // Created child device handle*
);

The basic operation process is as follows:

1-A) Open the EFI_SERVICE_BINDING_PROTOCOL instance through gEfiArpServiceBindingProtocolGuid;
1-B) Use CreateChild of this instance to create a child device;
1-C) Use the GUID of each network protocol to install the corresponding Protocol on the child device. For specific examples, please refer to UEFI sepc 2.8 P390.

2) EFI_TCP4_PROTOCOL interface description

typedef struct _EFI_TCP4_PROTOCOL { EFI_TCP4_GET_MODE_DATA
GetModeData;// Get the current protocol stack statusEFI_TCP4_CONFIGURE
Configure *; // Configure TCP address , port and other propertiesEFI_TCP4_ROUTES*
Routes *; // Add or delete routes for this TCP* **instanceEFI_TCP4_CONNECT** *Connect;// Initialize TCP three-way handshake and establish a TCP connectionEFI_TCP4_ACCEPT Accept ;// Listen for TCP connection requestsEFI_TCP4_TRANSMIT* **Transmit** *;// Send dataEFI_TCP4_RECEIVE* **Receive** *; // Receive* **dataEFI_TCP4_CLOSE** *Close;// Close the* **connectionEFI_TCP4_CANCEL** *Cancel;// Cancel the asynchronous operation on the current connectionEFI_TCP4_POLL* **Poll** *; // Complete the send or receive operation on the current connection* **} EFI_TCP4_PROTOCOL;**

typedef EFI_STATUS (EFIAPI *EFI_TCP4_TRANSMIT) (
IN EFI_TCP4_PROTOCOL **This, // Instance*
IN EFI_TCP4_IO_TOKEN **Token // Points to the queue of completion tokens*
);

typedef EFI_STATUS (EFIAPI *EFI_TCP4_RECEIVE) (
IN EFI_TCP4_PROTOCOL **This, // Instance*
IN EFI_TCP4_IO_TOKEN **Token // Points to the queue of completion tokens*
);

Config, Connect and Accept can be found in the Spec. Here we will focus on sending and transmitting. Both sending and transmitting use the EFI_TCP_IO_TOKEN pointer, and its prototype is as follows:

typedef struct { EFI_TCP4_COMPLETION_TOKEN
CompletionToken; // Token for completing the operation
union { EFI_TCP4_RECEIVE_DATA
**RxData; // Receive data buffer*
EFI_TCP4_TRANSMIT_DATA **TxData* **;** *// Send data buffer*
} *Packet;*
} EFI_TCP4_IO_TOKEN;

typedef struct { EFI_EVENT
Event; // This event is triggered after the request is completed
EFI_STATUS *Status; // Status flag after the operation is completed*
} EFI_TCP4_COMPLETION_TOKEN;

typedef struct { BOOLEAN
UrgentFlag **;** *//Urgent flag of TCP header*
UINT32 *DataLength; //Total length of data*
UINT32 *FragmentCount; //Number of data segments*
EFI_TCP4_FRAGMENT_DATA *FragmentTable[1];//Array of data segments*
} EFI_TCP4_RECEIVE_DATA;

typedef struct { BOOLEAN
Push **;** *// PSH flag of TCP header* **BOOLEAN** *Urgent* **;** *// URG flag of TCP header* **UINT32** *DataLength; // Total length of data* **UINT32** *FragmentCount; // Number of data segments* **EFI_TCP4_FRAGMENT_DATA** *FragmentTable[1]; // Array of data segments* **} EFI_TCP4_TRANSMIT_DATA;**

There are a lot of data structures , but the structure is clear. Network protocols use a lot of events, so I won't explain them one by one. It's easy to understand by referring to the data structure.

2 TCP4 Programming

In the experiment, I used UEFI as the client for testing. Assuming that the server receives the data from the client, it returns the data as is.

The server code has not been written yet, so we will temporarily use the "Network Assistant" to simulate the operation.

The client code writing process can be roughly divided into the following steps:

1) Use EFI_SERVICE_BINDING_PROTOCOL to generate an EFI_TCP4_PROTOCOL object;
2) Configure the generated object and create various required Event objects;
3) Initiate a connection to the server;
4) Transfer data;
5) Close the connection and destroy the EFI_TCP4_PROTOCOL object.

The following is a brief explanation of the sample code provided.

In order to save the relevant configuration items and buffer addresses of network communication, a data structure named MYTCP4SOCKET is constructed, including various handles, buffers, tokens, etc. used, all in this structure. At the same time, a global array of this type TCP4SocketFd[32] is defined to facilitate the use of various functions.

The function that creates the EFI_TCP4_PROTOCOL object is UINTN CreateTCP4Socket(VOID), which is roughly written according to the example provided in the Spec.

The function that needs to be understood is EFI_STATUS InitTcp4SocketFd (INTN index). This function is called in CreateTCP4Socket (), as shown in the screenshot below:

Figure 1 Code screenshot

The code in the red box in the figure is slightly different from the requirements in the Spec. In the test, if the EVT_NOYIFY_SIGNAL type is used, the data cannot be sent (experiments done in the TianCore simulator), so a little change was made.

The other functions are relatively simple to understand. They are easy to understand by comparing them with the Protocol description in the Spec, so I will not explain them here.

3 Testing

According to the instructions in the previous articles, set up a network test environment. Compile the code using the following command:

C:\MyWorkspace>build -p RobinPkg\RobinPkg.dsc -a IA32 -m RobinPkg\Applications\EchoTCP4\ EchoTCP4.inf

In the built environment, the test results are as follows:

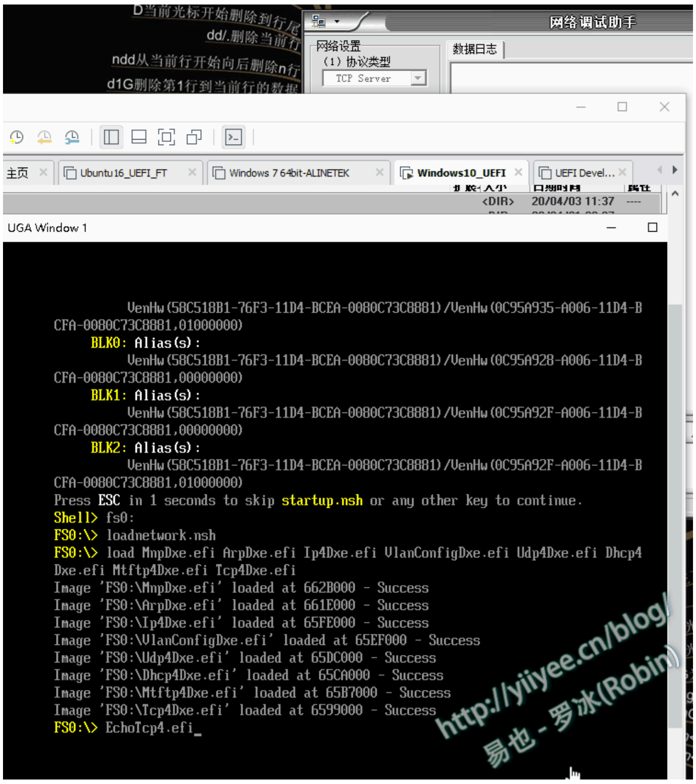

Figure 2 Testing TCP communication

Gitee address: https://gitee.com/luobing4365/uefi-explorer
Project code is located at: / FF RobinPkg/RobinPkg/Applications/EchoTCP4