UEFI Development Exploration 33 – Serial Port Communication Again

原创 luobing4365 Posted on 2019-09-22 13:06:25 Read 1.2k Collection 4 Likes 1 copyright

Category columns: UEFI Development Article Tags: UEFI Programming UEFI Serial Port Serial Port Programming Low-level programming UEFI serial port read and write

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)

In the 20th article of the UEFI series, I tried to build the serial communication code. Sending serial port data was already implemented at that time, but I didn't find a way to determine whether there was readable data, and reading the serial port was unsuccessful. Occasionally, I could read serial port data, which only proved that the read function worked.

In my years of development experience, I have developed serial port codes for various MCUs, and also developed them under DOS and Legacy BIOS. To read the serial port, we basically use interrupts (or cooperate with DMA). Although the serial port under Windows is a Windows message mechanism on the surface, the driver layer still uses interrupts.

Having developed an isolation card for serial communication, I built the serial port program of the underlying serial port and microcontroller under Legacy BIOS:

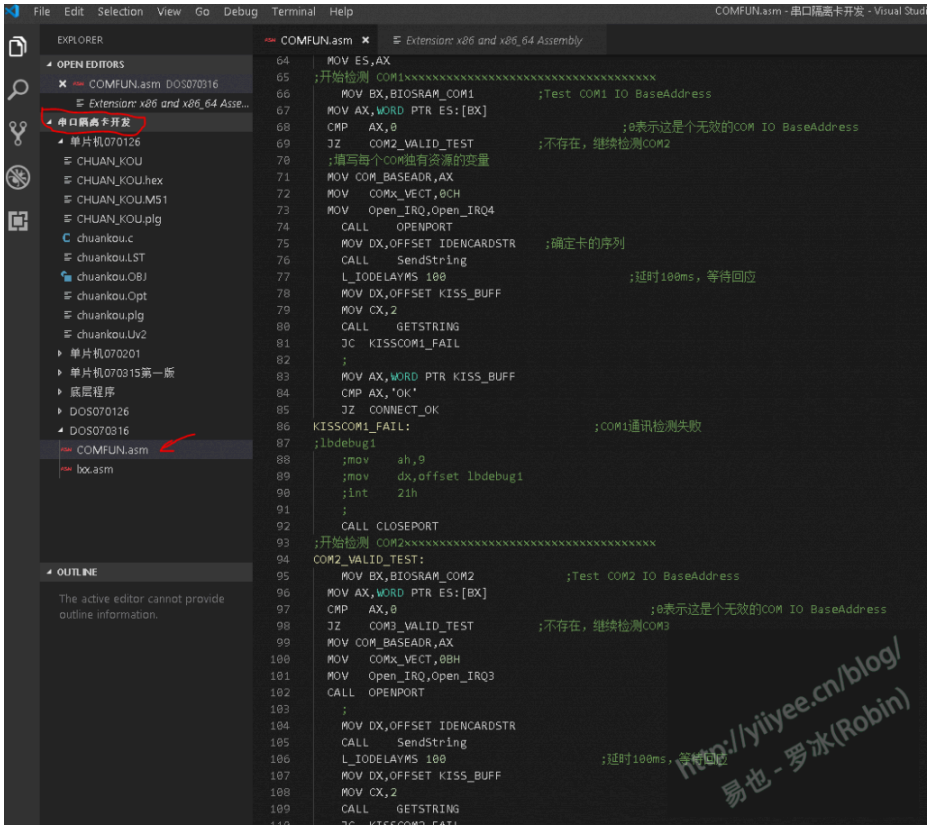

Figure 1 Legacy BIOS: Serial port communication

Although there are many registers, the core idea remains the same: set the serial port parameters, set the interrupt handler, and then create the read and write functions.

However, UEFI only provides the Event method, and there is no corresponding event function in the SerialIO Protocol (similar to the WaitForKey of the keyboard). How to solve this problem?

I've been on a business trip this week and have been thinking about this question over and over again.

1. Debugging

The initial idea was to start with OVMFPkg.

In the 20th article, I used a self-made tool called pipetool to establish serial communication with SecMain.exe running in the virtual machine. The host machine used named pipes and the virtual machine used serial ports .

This proves that Nt32Pkg supports serial port reading and writing, and is directly used to output debugging information. In addition, the Nt32Pkg code is open source and can be traced section by section. OvpmfPkg should be the same in theory.

Things in the world are not always as we wish. As shown in the picture:

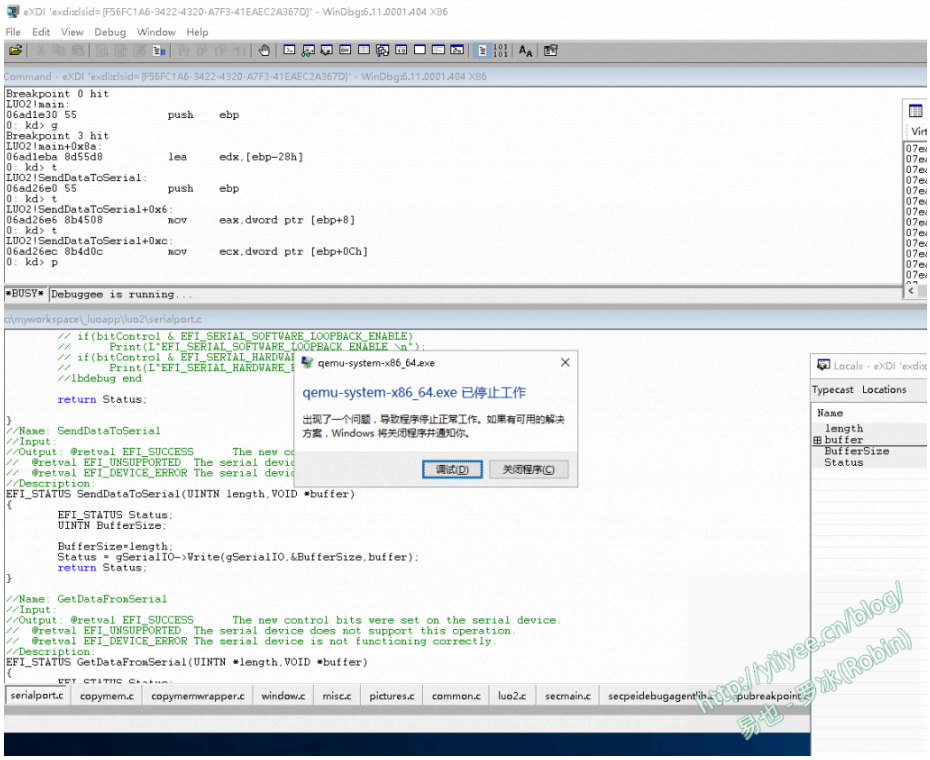

Figure 2: Trying to track OvmfPkg

The debugging environment built by Qemu directly reports an error. Maybe it is caused by the serial port simulation not being done well. In short, I can't continue debugging.

I haven't figured out the structure of OvmfPkg yet, and I need to learn a lot more to locate the code I want.

This means that the reverse learning path is not feasible at present. It is better to take the positive path, study the UEFI Spec carefully, and build it yourself.

2 GetControl of Serial IO Protocol

I read the UEFI Spec carefully, and the only thing I can use is a few flags provided in GetControl.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SERIAL_GET_CONTROL_BITS) (
 IN EFI_SERIAL_IO_PROTOCOL *This,
 OUT UINT32         *Control
 );
```

Parameters

This	A pointer to the **EFI_SERIAL_IO_PROTOCOL** i **EFI_SERIAL_IO_PROTOCOL** is defined in Sect
Control	A pointer to return the current control signals See “Related Definitions” below.

Related Definitions

```
//*******************************************************
// CONTROL BITS
//*******************************************************

#define EFI_SERIAL_CLEAR_TO_SEND                0x0010
#define EFI_SERIAL_DATA_SET_READY               0x0020
#define EFI_SERIAL_RING_INDICATE                0x0040
#define EFI_SERIAL_CARRIER_DETECT               0x0080
#define EFI_SERIAL_REQUEST_TO_SEND              0x0002
#define EFI_SERIAL_DATA_TERMINAL_READY          0x0001
#define EFI_SERIAL_INPUT_BUFFER_EMPTY           0x0100
#define EFI_SERIAL_OUTPUT_BUFFER_EMPTY          0x0200
#define EFI_SERIAL_HARDWARE_LOOPBACK_ENABLE     0x1000
#define EFI_SERIAL_SOFTWARE_LOOPBACK_ENABLE     0x2000
#define EFI_SERIAL_HARDWARE_FLOW_CONTROL_ENABLE 0x4000
```

Figure 3 UEFI Spec: GetControl

From SerialIo.h, we know that there are 6 flags related to reading, namely EFI_SERIAL_CLEAR_TO_SEND, EFI_SERIAL_DATA_SET_READY, EFI_SERIAL_RING_INDICATE, EFI_SERIAL_CARRIER_DETECT, EFI_SERIAL_INPUT_BUFFER_EMPTY, and EFI_SERIAL_OUTPUT_BUFFER_EMPTY.

I wrote a test program to print out the values of all these bits, and finally focused my suspicion on the EFI_SERIAL_INPUT_BUFFER_EMPTY bit, which changes to 1 when there is data to read.

According to the ancient and magical book "PC Technology Insider", the above bits can also correspond to the signals of the serial port. I only have the e-book, which is not very clear. After returning to Nanjing, I will study the paper books one by one, and I think I will have a deeper understanding.

3. Build the code

Because there is no interrupt mechanism, you can only use the loop query method or use Event to establish a timed query. For the convenience of programming, I did not use Event and directly looped the query.

```
// WaitKey();
Status=SendDataToSerial(sizeof(lbtest)/sizeof(UINT8),lbtest);
WaitKey();
Status=SendDataToSerial(sizeof(lbtest)/sizeof(UINT8),lbtest);
WaitKey();



while(readData[0]!='Q')
{
  Delayms(2);
  Status = GetSerialControlBits(&controlBits);
  if (!EFI_ERROR (Status))
  {
    if ((controlBits & EFI_SERIAL_INPUT_BUFFER_EMPTY) != 0)
    {
      Status = EFI_NOT_READY;
    }
    else
    {
      // Status = EFI_SUCCESS;
      //获取数据
      if ((controlBits & EFI_SERIAL_DATA_SET_READY) != EFI_SERIAL_DATA_SET_READY)
        Delayms(2);

      readLength = 1000;
      SetMem(readData, 1024, 0);
      Status=GetDataFromSerial(&readLength,readData);
      Print(L"First char:0x%x(hex), %c(ascii)\n",readData[0],readData[0]);
      Print(L"All data: ");
      for (i = 0; i < readLength; i++)
        Print(L"%c", readData[i]);
      Print(L"\n");
    }
  }
```


Figure 4 Building code

The logic of the code is very direct. It sends two strings to test whether the serial communication is normal. Then it enters the loop. When the first received data is 'Q', it exits the loop; otherwise, it keeps checking whether EFI_SERIAL_INPUT_BUFFER_EMPTY is 0, that is, whether there is data to be read.

After that, just receive the data.

It should be noted that I used a virtual machine with PipeTool for testing, so I did not set the serial port parameters and used the default parameters directly. In actual programming, you still need to set parameters such as baud rate and stop bit.

Another thing to note is that you must consider the case of multiple serial ports. For example, in my test, the virtual machine actually uses serial port 2 for communication. I set an array of length 256 in the code to save the enumerated serial ports, and point the working serial port pointer to serial port 2.

4 Run

After compiling, build the test environment according to the introduction of UEFI Development Exploration 20.

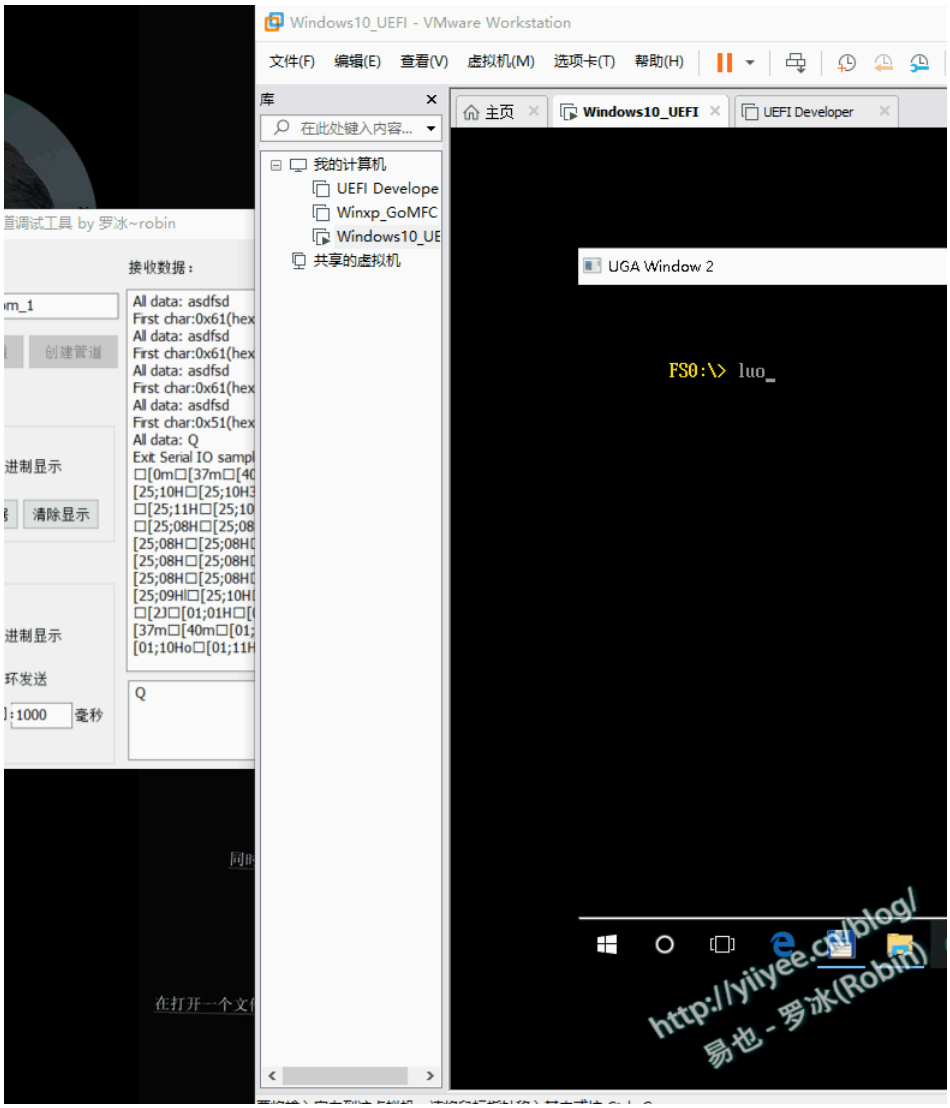

Figure 5: Operation

The program runs well and can capture and print out the information sent from the host machine.

There is also a small problem, sometimes the sent data is not received. When exiting the program, the data is printed out on the TianoCore simulator (the simulation environment also uses this serial port to output information).

From the test information, I guess the serial port in UDK should be operating in FIFO mode. As for whether this problem still exists in the actual environment, I doubt it.

Let's take a look at it later after we learn more about OvmfPkg and find the location of the corresponding code.

Gitee address: https://gitee.com/luobing4365/uefi-explorer
Project code is located at: / 23 SerialPort-RW