UEFI Development Exploration 21 – SMBUS Communication

原创 luobing4365 Posted on 2019-09-19 11:44:00 Read 4.4k Collection 15 Likes 8 Copyright CC 4.0 BY-SA

Category columns: UEFI Development Article Tags: UEFI Programming UEFI smbus programming SMBus Programming Low-level programming UEFI SmHc

摘要 This article explores the SMBus communication protocol and introduces its application in computer system management. It covers the historical background, technical characteristics, read and write operation procedures and UEFI support of SMBus. It also analyzes the role of SMBus in the development of dual-network isolated computers through actual cases.

The summary is generated in C Know , supported by DeepSeek-R1 full version, go to experience>

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

SMBus is a two-wire communication patent technology proposed by Intel in 1995. It fully complies with the System Management Bus Specification Version 1.1 and is compatible with the I2C serial bus. Compared with the currently popular high-speed serial protocols, SMBus is slower, but because it uses less hardware and many products support this protocol, it still has a large application in the current computer industry.

In the process of developing a dual-network isolated computer, it is necessary to solve the three-party communication problem. That is, the communication between the Windows/Linux layer software, the control card firmware and the BIOS (including the BIOS oprom).

I first used PCI/PCIE chips. The problem was that the drivers provided by the manufacturer were not perfect and always had various problems. In addition, there were not enough control pins on the chip, so the product goals could not be achieved.

Later, it was replaced with C8051F320, which was more affordable. The underlying communication (control card firmware and BIOS) was set to SMBUS bus communication mode; the upper layer communication used USB HID protocol.

This marked the beginning of my journey of developing SMBUS communication codes on various platforms.

1 Introduction to SMBUS Protocol

The SMBus standard is based on Philips' I2C bus and is aimed at communication between different systems. With the continuous improvement and update of its standards, it has been widely used in IT products. Starting from 2.0, SMBus has been standardized into the first three layers of the 7-layer OSI network model, the physical layer, the data link layer, and the network layer.

The SMBUS data line SDA and clock line SCL are both bidirectional. The operating voltage of the SMBus interface can be between 3.0V and 5.0V, and the operating voltage of different devices on the bus can be different. The SCL (serial clock) and SDA (serial data) lines are bidirectional and must be connected to the power supply voltage through a pull-up resistor or equivalent circuit. open-collector, so when the bus is idle, both lines are pulled high.

The standard transmission rate of SMBus is 100KHz to 200KHz. But in fact, it can reach up to one tenth of the system clock frequency. This depends on the user's settings. When there are devices of different speeds connected to the bus, the method of extending the SCL low level time can be used to synchronize the communication between them.

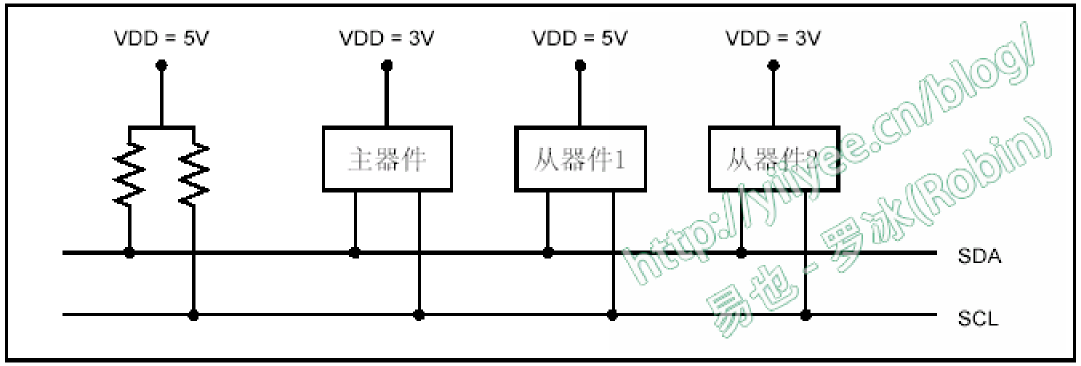


Figure 1 Typical SMBus configuration

There are two possible types of data transfer: from a master-transmitter to an addressed slave-receiver (write) and from an addressed slave-transmitter to a master-receiver (read).

Both data transfers are initiated by the master device, which also provides the serial clock on SCL. The SMBus interface can operate in master mode or slave mode, and there can be multiple master devices on the bus.

If two or more master devices start data transmission at the same time, the arbitration mechanism will ensure that one master device will win the bus. The protocol stipulates that any device that sends the start condition (START) and the slave device address becomes the master device for the data transmission.

For more details, please read the relevant Spec.

2 Experimental design

The biggest problem I encountered was how to find a device that supports SMBUS to access it. I used two methods: one is to write code to directly access the SPD in the memory; the other is to make a control board that supports the SMBUS protocol.

In my actual development process, the control card is inserted into the PCI/PCIE slot (not a PCIE device, just for power). The control card firmware implements access support for the SMUBS protocol, and the programs on the PC (mainly BIOS or BIOS Oprom) can communicate with it through SMBUS. This access is also because the PCI protocol supports SMBUS. Among the pins provided by PCI, SMBCLK and SMBDAT are provided.

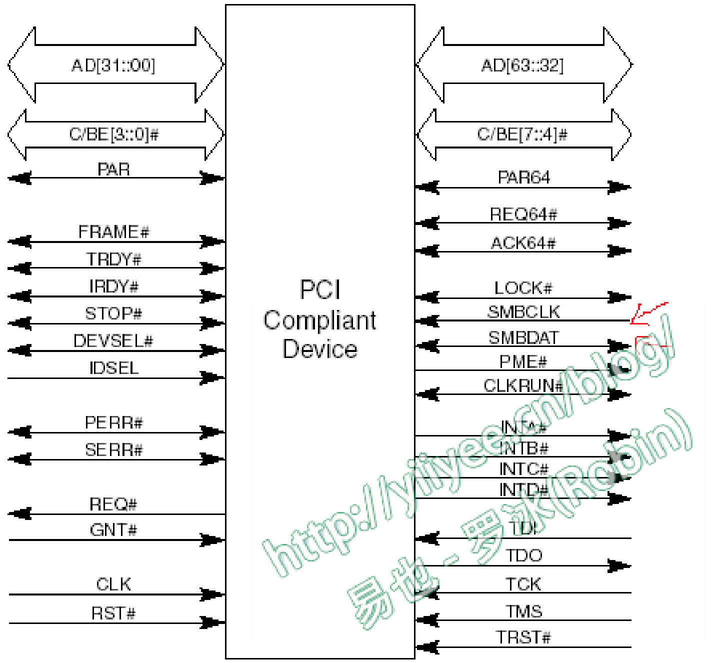


Figure 2 PCI pinout

This opens up the SMBUS communication channel between the PC and the control card from the hardware level. The way I usually use is that the PC works as the master and the control card works as the slave. In the previous blog "UEFI Development Exploration 07 - About the Development Story of SMBUS", I introduced the working method of the control card in more detail.

Of course, when there is no control card, I also use the program to read the memory SPD to determine whether the code is working properly. The SMBUS address of the general memory is A0, A2... You can find it by looking up the Spec of a chipset.

3 Reading and Writing Process

According to the SMBus protocol, the host (PC) generates clock signals and stop signals, and the slave (control card firmware) provides data. From the application point of view, there are two tasks: host "read" and "write", that is, the host reads the slave and the host writes the slave.

The host reads the slave in a composite format, as shown in Figure 3. In the figure, SLA refers to the slave address. By referring to the programming manual of C8051F32x, we can know that the SMBUS address of the control card I use is 0xF0. The program flow can be summarized as follows:

I The host sends the slave address and data byte 1;
II The data transmission process generated on the bus is shown in the figure;
III The host receives data byte 2 and sends a stop command.

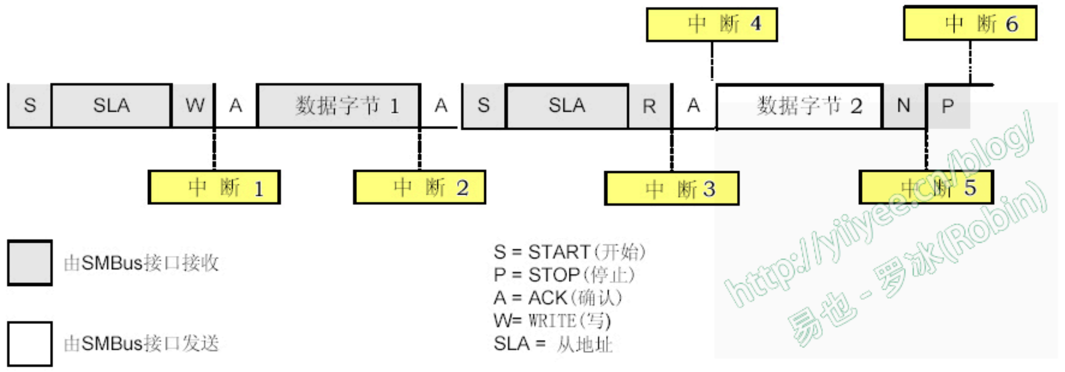


Figure 3 Master reads slave timing diagram

As shown in Figure 4, a detailed master writes to the slave timing diagram. The corresponding program flow is summarized as follows:

I The host sends the Slave address, Register Index, the data byte to be written, and sends a write command;
II The data transmission process generated on the bus is shown in the figure;
III The host sends a stop command.

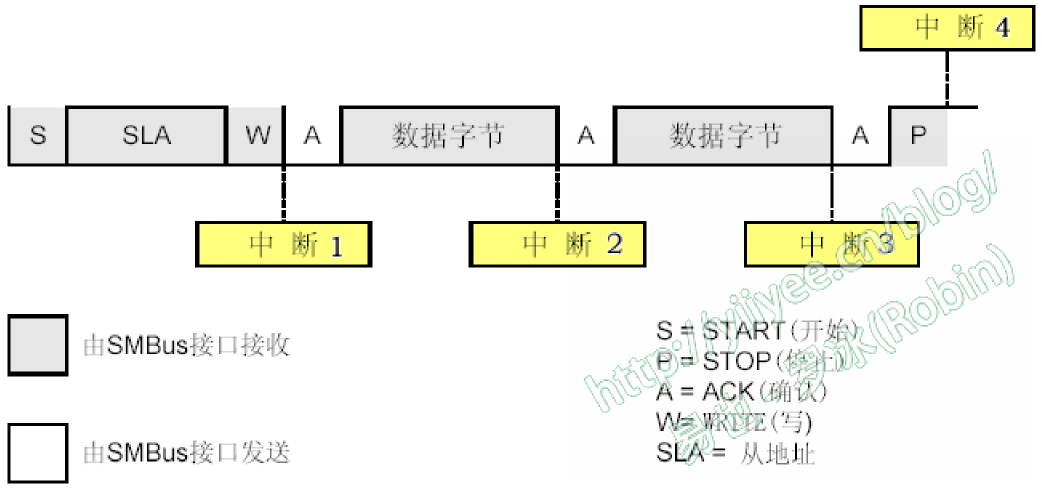


Figure 4 Master writes slave timing diagram

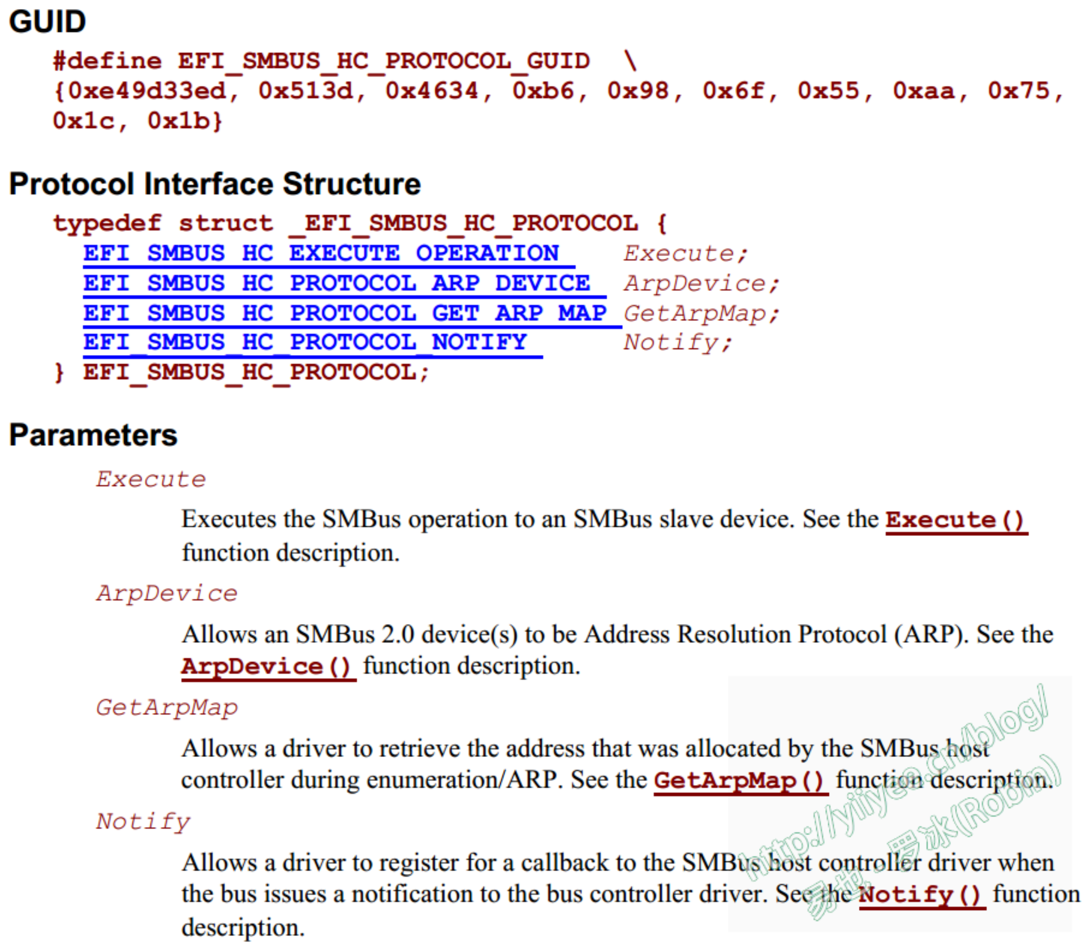
4 UEFI support for SMBUS

I did not find any protocol that supports SMBUS in UEFI Spec, but I can find the corresponding file in UDK. I will start from here because I think practicality comes first. Why doesn't Spec provide support? I have not found the answer yet.

The code you can refer to is \MdePkg\Include\Protocol\SmbusHc.h. In addition, there are two documents: "174_SmbusHostCont.pdf" and "175_SmbusPpi.pdf", which you can also refer to. I don't know if the document has been updated. I think I downloaded it from Sourceforge.net at first. I think it has been moved to github now. If you are interested, you can look for it.

When I saw the Protocol description, tears almost came out. After so many years of being tossed around with assembly language, reading various timing codes, and rewriting them every time a new motherboard was replaced, the days are finally over.

This is the protocol that made me burst into tears: (My dear ones!)



GUID

```
#define EFI_SMBUS_HC_PROTOCOL_GUID  \
{0xe49d33ed, 0x513d, 0x4634, 0xb6, 0x98, 0x6f, 0x55, 0xaa, 0x75,
0x1c, 0x1b}
```

Protocol Interface Structure

```
typedef struct _EFI_SMBUS_HC_PROTOCOL {
  EFI_SMBUS_HC_EXECUTE_OPERATION      Execute;
  EFI_SMBUS_HC_PROTOCOL_ARP_DEVICE    ArpDevice;
  EFI_SMBUS_HC_PROTOCOL_GET_ARP_MAP   GetArpMap;
  EFI_SMBUS_HC_PROTOCOL_NOTIFY        Notify;
} EFI_SMBUS_HC_PROTOCOL;
```

Parameters

Execute

Executes the SMBus operation to an SMBus slave device. See the **Execute()** function description.

ArpDevice

Allows an SMBus 2.0 device(s) to be Address Resolution Protocol (ARP). See the **ArpDevice()** function description.

GetArpMap

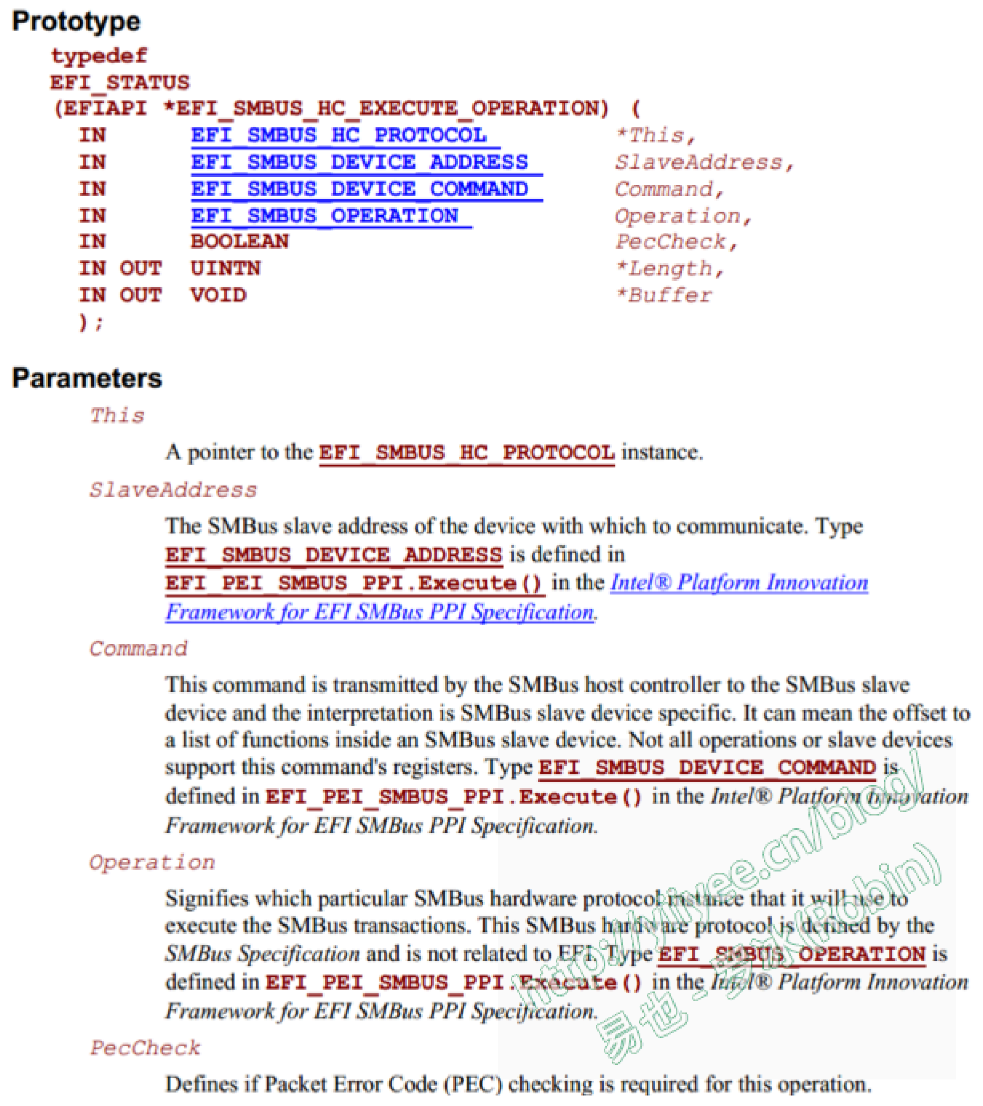
Allows a driver to retrieve the address that was allocated by the SMBus host controller during enumeration/ARP. See the **GetArpMap()** function description.

Notify

Allows a driver to register for a callback to the SMBus host controller driver when the bus issues a notification to the bus controller driver. See the **Notify()** function description.

Figure 5 SmbusHc protocol (174_SmbusHostCont.pdf page 14)

The SMBus protocol itself is relatively simple, and the functions it provides are also easy to understand. We will focus on the Execute() function, whose prototype is as follows:



Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_HC_EXECUTE_OPERATION) (
  IN       EFI_SMBUS_HC_PROTOCOL          *This,
  IN       EFI_SMBUS_DEVICE_ADDRESS       SlaveAddress,
  IN       EFI_SMBUS_DEVICE_COMMAND       Command,
  IN       EFI_SMBUS_OPERATION            Operation,
  IN       BOOLEAN                        PecCheck,
  IN OUT   UINTN                          *Length,
  IN OUT   VOID                           *Buffer
  );
```

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

SlaveAddress

The SMBus slave address of the device with which to communicate. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification*.

Command

This command is transmitted by the SMBus host controller to the SMBus slave device and the interpretation is SMBus slave device specific. It can mean the offset to a list of functions inside an SMBus slave device. Not all operations or slave devices support this command's registers. Type **EFI_SMBUS_DEVICE_COMMAND** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification.*

Operation

Signifies which particular SMBus hardware protocol instance that it will use to execute the SMBus transactions. This SMBus hardware protocol is defined by the *SMBus Specification* and is not related to EFI. Type **EFI_SMBUS_OPERATION** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Intel® Platform Innovation Framework for EFI SMBus PPI Specification.*

PecCheck

Defines if Packet Error Code (PEC) checking is required for this operation.

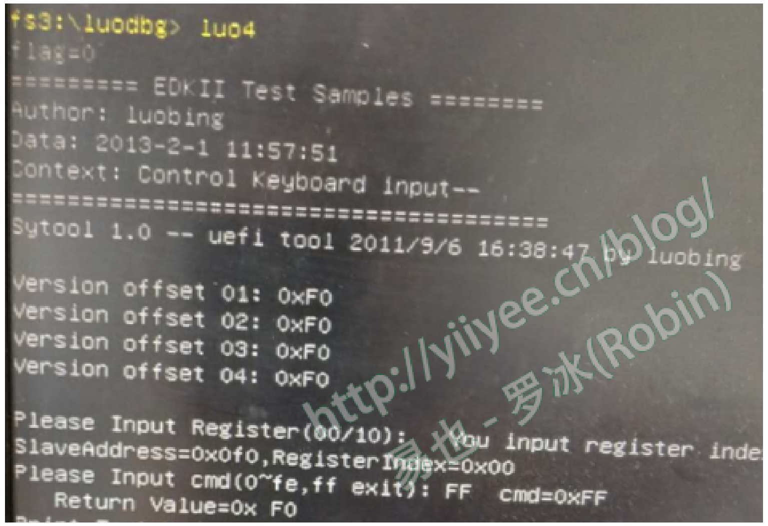
Figure 6 Execute() function description (174_SmbusHostCont.pdf page 16)

The control card I use has SlaveAddress=0xF0. The Command and Operation parameters can be found in "175_SmbusPpi.pdf", which is the Intel Platform Innovation Framework for EFI SMBus PPI Specification mentioned in the figure.

According to the instructions, the code is relatively easy to write. I mainly implemented the host read and host write functions.

5. Compile and run

It cannot be run in the TianoCore simulation environment because the hardware does not exist. I don't have the control card I used for debugging, so I can only find a random machine to demonstrate the running effect.



```
fs3:\luodbg> luo4
flag=0
========= EDKII Test Samples ========
Author: luobing
Data: 2013-2-1 11:57:51
Context: Control Keyboard input--
=====================================
Sytool 1.0 -- uefi tool 2011/9/6 16:38:47 by luobing

Version offset 01: 0xF0
Version offset 02: 0xF0
Version offset 03: 0xF0
Version offset 04: 0xF0

Please Input Register(00/10):  You input register inde
SlaveAddress=0x0f0,RegisterIndex=0x00
Please Input cmd(0~fe,ff exit): FF  cmd=0xFF
   Return Value=0x F0
```

Figure 7: Running results on the actual machine

From the information printed by the program, it can be seen that this is a development tool I made for a project a long time ago. The code for interacting with the user has not been changed, it was directly ported over. It was originally developed using AMI Aptio, and then ported to UDK.

The program has a fixed address of 0xf0 for accessing the control card. If you need to access the memory SPD, change it to 0xA0, 0xA2, etc. When I get a control card from the company and actually run it, I will post photos.

6 One More Thing

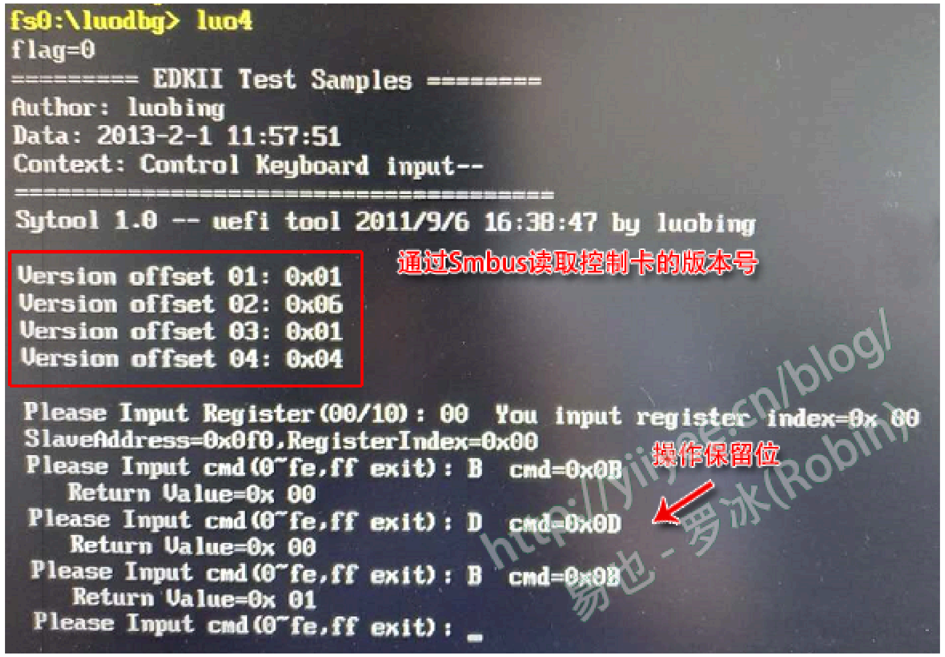
From developing dual-network isolation computers (945 chipset) with Founder to developing H110 chipset products with Lenovo, the Smbus debugging tools have been continuously updated. The earliest was Legacy BIOS, and the access timing of each motherboard was slightly different. I could only ask the BIOS engineer to help me find the smbus access code in the source code and modify and transplant it myself.

Until the emergence of UEFI, this problem was solved once and for all. No longer need to prepare debugging tools every time you develop a project.

The earliest codes were all written in assembly language, and later gradually changed to C mixed assembly language, which contained many tips for accessing hardware and mixed programming.

I'm unlikely to use this code again, but it's still interesting to read. So I also uploaded the code of the last non-UEFI SMBus debugging tool I developed, in case I can't find it later.

(Add photos of the program running on the actual machine)



```
fs0:\luodbg> luo4
flag=0
========= EDKII Test Samples ========
Author: luobing
Data: 2013-2-1 11:57:51
Context: Control Keyboard input--
======================================
Sytool 1.0 -- uefi tool 2011/9/6 16:38:47 by luobing
Version offset 01: 0x01
Version offset 02: 0x06
Version offset 03: 0x01
Version offset 04: 0x04
 Please Input Register(00/10): 00  You input register index=0x 00
 SlaveAddress=0x0f0,RegisterIndex=0x00
 Please Input cmd(0~fe,ff exit): B  cmd=0x0B
    Return Value=0x 00
 Please Input cmd(0~fe,ff exit): D  cmd=0x0D
    Return Value=0x 00
 Please Input cmd(0~fe,ff exit): B  cmd=0x0B
    Return Value=0x 01
 Please Input cmd(0~fe,ff exit): _
```

Figure 7 Running on Lenovo H81 dual-network isolated computer (with control card)

First, read and process the internal version of the control card. The protocol was developed by me a long time ago. When the host program reads Slaveaddress=0xf0, Register=0x01~0x04, the control card firmware returns the corresponding content.

In addition, the control card maintains a byte of internal data, and different bits (D0~D7) represent different meanings. D7 is a reserved bit that allows the host program to set, clear, and read. This internal byte is accessed through SmbusWrite:

1) Write byte 0xB to Slaveaddress=0xf0,register=0x00, then the control card firmware returns the content of D7;
2) Write byte 0xD to Slaveaddress=0xf0,register=0x00, then the control card firmware sets D7 to 1, and the returned content is invalid;
3) Write byte 0xC to Slaveaddress=0xf0,register=0x00, then the control card firmware sets D7 to 0, and the returned content is invalid;

The figure demonstrates the process of setting D7.

Gitee address: https://gitee.com/luobing4365/uefi-explorer
Project code is located in: /14 SMBus HC-tool.
/Luo4: SMBus example under UEFI;

/SY_Intel_SMB (general sy low-level debugging tool): Smbus tool for Legacy BIOS, supports Intel platform. Borland C++3.1 compilation