UEFI - Accessing PCI/PCIE devices (Part 2)

原创 Practitioner xxl Posted on 2024-09-12 16:37:12 Read 1.9k Collection 30 Likes: 33

Category columns: BIOS Development Article Tags: UEFI firmware

1. Protocol that supports access to PCI/ PCIE devices

UEFI provides two main modules to support PCI bus, one is PCI Host Bridge controller driver, the other is PCI bus driver. These two modules are bound to specific platform hardware. Under this mechanism, the differences between different CPU architectures are shielded, providing software developers with a relatively consistent Protocol interface.

The UEFI standard provides two types of protocols for accessing PCI/PCIE devices: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL and EFI_PCI_IO_PROTOCOL . The former provides an abstract IO function for the PCI root bridge, which is generated by the PCI Host Bus Controller (PCI main bus driver) and is generally used by the PCI/PCIe bus driver to enumerate devices, obtain Option ROM, allocate PCI device resources, etc.; the latter is generated by the PCI/PCIE bus driver for PCI/PCIE devices, and is generally used by PCI/PCIE device drivers to access the IO space, memory space, and configuration space of PCI/PCIE devices.

1.1 EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL provides basic memory, input/output (I/O), PCI configuration, and direct memory access interfaces that are used to abstract access to the PCI root bridge controller behind the PCI controller.

```
typedef struct _EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL;
```

```
1 /// Provides the basic Memory, I/O, PCI configuration, and DMA interfaces that are
2 /// used to abstract accesses to PCI controllers behind a PCI Root Bridge Controller.
3 ///提供了基本内存，I/O，PCI配置和直接内存访问（DMA）接口，这些接口用于抽象PCI控制器后PCI根桥的访问
4 struct _EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL {
5   ///
```

```
  /// The EFI_HANDLE of the PCI Host Bridge of which this PCI Root Bridge is a member.
/// 这是PCI根桥所属的PCI主机桥的EFI_HANDLE
  EFI_HANDLE                                ParentHandle; //包含这个PCI根桥的PCI主机桥的Handle
```

展开

Here we only introduce the interface Mem for accessing memory, the interface Io for accessing I/O space, and the interface Pci for accessing PCI configuration space. The parameter types of these three interfaces are the same, all of which are EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS


```
typedef struct {

  /// Read PCI controller registers in the PCI root bridge memory space.
  /// 读数据
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM    Read;

  /// Write PCI controller registers in the PCI root bridge memory space.
  /// 写数据
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM    Write;
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS;

typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM)(
  IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL              *This, //指向protocol实例
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH    Width, //标识内存操作的宽度，一般有8位、16位、32位、64位几种
  IN     UINT64                                   Address, //内存操作的基地址
  IN     UINTN                                    Count, //读写的数据个数，以width为单位
  IN OUT VOID                                     *Buffer
  );
```

收起

The parameter Adress is different when accessing I/O space, memory space, and configuration space.

For the configuration space, Address is determined by the BDF address and Register offset, and the macro EFI_PCI_ADDR
Register offset. In EDK2, the macro definition is as follows:

```
#define EFI_PCI_ADDRESS(bus, dev, func, reg) \
  (UINT64) ( \
  (((UINTN) bus) << 24) | \
  (((UINTN) dev) << 16) | \
  (((UINTN) func) << 8) | \
  (((UINTN) (reg)) < 256 ? ((UINTN) (reg)) : (UINT64) (LShiftU64 ((UINT64) (reg), 32))))
```

For IO space, the parameter Address refers to the IO address of the PCI device IO space;

For the Memory space, the parameter Address refers to the Memory address of the PCI device Memory space.

They are determined by BAR and offset.

1.2 EFI_PCI_IO_PROTOCOL

In PCI/PCIE device drivers, EFI_PCI_IO_PROTOCOL is generally used to access the internal resources of the device. The Protocol is mounted on the PCI/PCIE controller and runs in the EFI boot environment to access the memory space and I/O space of the PCI/PCIE device.


```
/// Global ID for the PCI I/O Protocol
///
#define EFI_PCI_IO_PROTOCOL_GUID \
  { \
    0x4cf5b200, 0x68b8, 0x4ca5, {0x9e, 0xec, 0xb2, 0x3e, 0x3f, 0x50, 0x2, 0x9a } \
  }

typedef struct _EFI_PCI_IO_PROTOCOL EFI_PCI_IO_PROTOCOL;
```


The interface of EFI_PCI_IO_PROTOCOL is as follows:

```
///
/// The EFI_PCI_IO_PROTOCOL provides the basic Memory, I/O, PCI configuration,
/// and DMA interfaces used to abstract accesses to PCI controllers.
/// There is one EFI_PCI_IO_PROTOCOL instance for each PCI controller on a PCI bus.
/// A device driver that wishes to manage a PCI controller in a system will have to
```

```
/// retrieve the EFI_PCI_IO_PROTOCOL instance that is associated with the PCI controller.
///
struct _EFI_PCI_IO_PROTOCOL {
  EFI_PCI_IO_PROTOCOL_POLL_IO_MEM           PollMem;
  EFI_PCI_IO_PROTOCOL_POLL_IO_MEM           PollIo;
  EFI_PCI_IO_PROTOCOL_ACCESS                Mem;
  EFI_PCI_IO_PROTOCOL_ACCESS                Io;
  EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS         Pci;
  EFI_PCI_IO_PROTOCOL_COPY_MEM              CopyMem;
  EFI_PCI_IO_PROTOCOL_MAP                   Map;
  EFI_PCI_IO_PROTOCOL_UNMAP                 Unmap;
  EFI_PCI_IO_PROTOCOL_ALLOCATE_BUFFER       AllocateBuffer;
  EFI_PCI_IO_PROTOCOL_FREE_BUFFER           FreeBuffer;
  EFI_PCI_IO_PROTOCOL_FLUSH                 Flush;
  EFI_PCI_IO_PROTOCOL_GET_LOCATION          GetLocation;
  EFI_PCI_IO_PROTOCOL_ATTRIBUTES            Attributes;
  EFI_PCI_IO_PROTOCOL_GET_BAR_ATTRIBUTES    GetBarAttributes;
  EFI_PCI_IO_PROTOCOL_SET_BAR_ATTRIBUTES    SetBarAttributes;

  ///
  /// The size, in bytes, of the ROM image.
  ///
  UINT64                                    RomSize;

  ///
  /// A pointer to the in memory copy of the ROM image. The PCI Bus Driver is responsible
  /// for allocating memory for the ROM image, and copying the contents of the ROM to memory.
  /// The contents of this buffer are either from the PCI option ROM that can be accessed
  /// through the ROM BAR of the PCI controller, or it is from a platform-specific location.
  /// The Attributes() function can be used to determine from which of these two sources
  /// the RomImage buffer was initialized.
  ///
  VOID    *RomImage;
};
```


【Note!!!】It should be noted here that there is only one instance of EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL in most office
global pointer variable gPCIRootBridgeIO; while there are multiple instances of EFI_PCI_IO_PROTOCOL, generally there are

so the global pointer array gPCIIOArray[256] is used to store these instances.

2. Simple implementation of accessing PCI/PCIE devices

2.1 EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL traverses PCI devices

When using EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL to obtain PCI/PCIE devices, you first need to obtain all handles of EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL. You can use the gBS->LocateHandleBuffer() function to obtain the handle. After obtaining the handle, use the gBS->HandleProtocol() function to obtain the protocol instance, and then you can call the function interface in it. Use BDF to find a device and read the device's configuration space.

The configuration space of the PCI device is represented by the structure PCI_TYPE00, and its code prototype is:


```
/// PCI Device Configuration Space

typedef struct {
  PCI_DEVICE_INDEPENDENT_REGION    Hdr;
  PCI_DEVICE_HEADER_TYPE_REGION    Device;
} PCI_TYPE00;
```

Hdr represents the general header area of the PCI configuration space, which includes the following fields:


```
// Common header region in PCI Configuration Space

typedef struct {
  UINT16    VendorId;
  UINT16    DeviceId;
  UINT16    Command;
  UINT16    Status;
  UINT8     RevisionID;
  UINT8     ClassCode[3];
  UINT8     CacheLineSize;
  UINT8     LatencyTimer;
  UINT8     HeaderType;
  UINT8     BIST;
} PCI_DEVICE_INDEPENDENT_REGION;
```

Device represents the header area in the PCI device configuration space


```
/// PCI Device header region in PCI Configuration Space

typedef struct {
  UINT32    Bar[6];
  UINT32    CISPtr;
  UINT16    SubsystemVendorID;
  UINT16    SubsystemID;
  UINT32    ExpansionRomBar;
  UINT8     CapabilityPtr;
  UINT8     Reserved1[3];
  UINT32    Reserved2;
  UINT8     InterruptLine;
  UINT8     InterruptPin;
  UINT8     MinGnt;
  UINT8     MaxLat;
} PCI_DEVICE_HEADER_TYPE_REGION;
```


Simple implementation of application using EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL to access PCIe devices:

Write the MyPCIEProtocol.c code:

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/ShellCEntryLib.h>
#include <Library/DebugLib.h>

#include <Protocol/PciIo.h>
#include <Protocol/PciRootBridgeIo.h>
#include <IndustryStandard/Pci.h>
```

```
9
10
11 EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *gPciRootBridgeIo; //创建一个EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL类型的指针变量
12
13 EFI_STATUS LocatePciRootBridgeIo(void); //函数声明，告诉编译器函数名称、返回值类型、参数类型；如果函数定义在函数调用之后，那么在函数调用之前必须进行函数声明
14
15 //函数声明
16 EFI_STATUS PciDevicePresent(
17   IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL * PciRootBridgeIo,
18   OUT PCI_TYPE00 * Pci,
19   IN UINT8 Bus,
20   IN UINT8 Device,
21   IN UINT8 Func
22   );
23
24 EFI_STATUS ListPciInformation(void); //函数声明
25
26
27 EFI_STATUS
28 EFIAPI
29 UefiMain (
30   IN EFI_HANDLE        ImageHandle,
31   IN EFI_SYSTEM_TABLE  *SystemTable
32   )
33 {
34   EFI_STATUS Status;
35
36
37   Status = LocatePciRootBridgeIo(); //调用LocatePciRootBridgeIo()函数，定位根桥
38   if(EFI_ERROR(Status))
39   {
40     DEBUG((DEBUG_ERROR, "[CSDN]Call LocatePciRootBridgeIo failed,Can't find protocol!\n"));
41   }
42   else
43   {
44     DEBUG((DEBUG_ERROR, "[CSDN]Call LocatePciRootBridgeIo successed,Find protocol!\n"));
45   }
46
47   //列出PCI设备的信息
48   ListPciInformation();
```

```
   50    return EFI_SUCCESS;
}

EFI_STATUS LocatePciRootBridgeIo()
{
  EFI_STATUS Status;
  EFI_HANDLE *PciHandleBuffer = NULL;
  UINTN      HandleIndex = 0;
  UINTN      HandleCount = 0;

  //获取 EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL的所有句柄
  Status = gBS->LocateHandleBuffer(
    ByProtocol, //查找句柄的方式，这里采用返回指定协议的句柄
    &gEfiPciRootBridgeIoProtocolGuid, //要查找的协议的GUID
    NULL,
    &HandleCount, //输出参数，缓冲区中返回的句柄数量
    &PciHandleBuffer  //输出参数，指向用于返回支持协议的请求句柄数组的缓冲区的指针。

    );
  if(EFI_ERROR(Status))  return Status; //如果没成功找到，直接结束程序

  //查找支持 EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL句柄的protocol实例
  for(HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++)
  {
    Status = gBS->HandleProtocol(
      PciHandleBuffer[HandleIndex],
      &gEfiPciRootBridgeIoProtocolGuid,
      (VOID **)&gPciRootBridgeIo
      );
    if(EFI_ERROR(Status))  continue; //如果没找到就继续寻找
    else                   return EFI_SUCCESS; //如果找到了就退出程序
    //这里找到就退出的原因是大部分办公用的个人电脑中就只存在一个EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL
  }

  return Status;

}

EFI_STATUS ListPciInformation()
{
```

```
  EFI_STATUS Status = EFI_SUCCESS;
  PCI_TYPE00 Pci; //PCI_TYPE00是一个结构体，表示PCI的配置空间
  UINT16 Dev,Func,Bus,PciDevicecount = 0;

  DEBUG((DEBUG_ERROR, "[CSDN] PciDeviceNo\tBus\tDev\tFunc | Vendor.Device.ClassCode\n"));
  for(Bus=0; Bus<256; Bus++)
    for(Dev=0; Dev<= PCI_MAX_DEVICE; Dev++)
      for(Func=0; Func<=PCI_MAX_FUNC; Func++)
      {
        //判断设备是否存在
        Status = PciDevicePresent(gPciRootBridgeIo,&Pci,(UINT8)Bus,(UINT8)Dev,(UINT8)Func);
          if(Status == EFI_SUCCESS)
          {
            //如果找到了设备，PCI设备数量加1
            PciDevicecount++;
             //打印设备信息
             DEBUG((DEBUG_ERROR, "[CSDN] %d\t\t%x\t%x\t%x\t",PciDevicecount,(UINT8)Bus,(UINT8)Dev,(UINT8)Func));
             DEBUG((DEBUG_ERROR, "%x\t%x\t%x\n",Pci.Hdr.VendorId,Pci.Hdr.DeviceId,Pci.Hdr.ClassCode[0]));
          }
      }

  return EFI_SUCCESS;
}

EFI_STATUS
PciDevicePresent (
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL     *PciRootBridgeIo,
  OUT PCI_TYPE00                          *Pci,
  IN  UINT8                               Bus,
  IN  UINT8                               Device,
  IN  UINT8                               Func
  )
{
  UINT64      Address;
  EFI_STATUS  Status;

  //
  // Create PCI address map in terms of Bus, Device and Func
  //
  Address = EFI_PCI_ADDRESS (Bus, Device, Func, 0);

```

```
  //
  // Read the Vendor ID register
  //
  Status = PciRootBridgeIo->Pci.Read (
                                  PciRootBridgeIo,
                                  EfiPciWidthUint32,
                                  Address,
                                  1,
                                  Pci
                                  );

  if (!EFI_ERROR (Status) && (Pci->Hdr).VendorId != 0xffff) { //0xffff通常标识一个无效的，未分配的VentorId,如果能读取设备信息就进行打印。
    //
    // Read the entire config header for the device
    //
    Status = PciRootBridgeIo->Pci.Read (
                                    PciRootBridgeIo,
                                    EfiPciWidthUint32,
                                    Address,
                                    sizeof (PCI_TYPE00) / sizeof (UINT32),
                                    Pci
                                    );

    return EFI_SUCCESS;
  }

  return EFI_NOT_FOUND;
}
```


Write an INF file and run it. The result is:

```
InstallProtocolInterface: 752F3136-4E16-4FDC-A22A-E5F46812F4CA FE6D828
[CSDN]Call LocatePciRootBridgeIo successed,Find protocol!
[CSDN] PciDeviceNo      Bus     Dev     Func | Vendor.Device.ClassCode
[CSDN] 1                0       0       0       8086    1237    0
[CSDN] 2                0       1       0       8086    7000    0
[CSDN] 3                0       1       1       8086    7010    80
[CSDN] 4                0       1       2       8086    7020    0
[CSDN] 5                0       1       3       8086    7113    0
[CSDN] 6                0       2       0       1234    1111    0
FSOpen: Open '\' Success
FS0:\>
```


2.2 EFI_PCI_IO_PROTOCOL traverses PCI devices

It is relatively simple to traverse the device using EFI_PCI_IO_PROTOCOL. Therefore, the instance of the Protocol obtained previously is generated for the PCI/PCIE device. In fact, it is equivalent to finding the host device. You only need to print out the device information.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/ShellCEntryLib.h>
#include <Library/DebugLib.h>

#include <Protocol/PciIo.h>
#include <Protocol/PciRootBridgeIo.h>
#include <IndustryStandard/Pci.h>


EFI_PCI_IO_PROTOCOL *gPciIoArray[256];
UINTN gPciIoCount = 0;
EFI_STATUS LocatePciIo(void);

EFI_STATUS ListPciInformation(void);


EFI_STATUS
EFIAPI
UefiMain (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
```

```
  EFI_STATUS Status;
                    26
  Status = LocatePciIo();
  if(EFI_ERROR(Status))
  {
    DEBUG((DEBUG_ERROR, "[CSDN]Call LocatePciIo failed,Can't find protocol!\n"));
  }
  else
  {
    DEBUG((DEBUG_ERROR, "[CSDN]Call LocatePciIo successed,Find protocol!\n"));
  }

  ListPciInformation();

  return EFI_SUCCESS;
}

EFI_STATUS LocatePciIo()
{
  EFI_STATUS Status;
  EFI_HANDLE *PciHandleBuffer = NULL;
  UINTN      HandleIndex = 0;
  UINTN      HandleCount = 0;

  Status = gBS->LocateHandleBuffer(
    ByProtocol,
    &gEfiPciIoProtocolGuid,
    NULL,
    &HandleCount,
    &PciHandleBuffer
    );
  if(EFI_ERROR(Status))  return Status;

  gPciIoCount = HandleCount;
  DEBUG((DEBUG_ERROR, "[CSDN] the number of PCIn devices is %d", HandleCount));

  for(HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++)
  {
    Status = gBS->HandleProtocol(
        PciHandleBuffer[HandleIndex],
```

```
        &gEfiPciIoProtocolGuid,
                                  (VOID **)&(gPciIoArray[HandleIndex])
        );
  }

  return Status;

}

EFI_STATUS ListPciInformation()
{
  UINTN i, count = 0;
  PCI_TYPE00 Pci;
  DEBUG((DEBUG_ERROR, "[CSDN] PciDeviceNo VendorID DeviceID ClassCode\n"));
  for (i = 0; i < gPciIoCount;i++)
  {
    gPciIoArray[i]->Pci.Read(gPciIoArray[i],EfiPciIoWidthUint32,0,sizeof(Pci)/sizeof(PCI_TYPE00),&Pci);
    ++count;
    DEBUG((DEBUG_ERROR, "[CSDN] %d\t\t%x\t%x\t%x\n", count, Pci.Hdr.VendorId, Pci.Hdr.DeviceId, Pci.Hdr.ClassCode[0]));
  }

  return EFI_SUCCESS;
}
```

收起 ^

The running results are as follows. It is found that the data of Class Code[0] read is different. I don't know why?