UEFI Development Exploration 23 – File IO (File Reading and Writing)

原创 luobing4365 Modified on 2022-04-24 18:40:34 Read 3.6k Collection 7 Likes 3

Category columns: UEFI Development Article Tags: UEFI Programming UEFI file reading and writing UEFI FileIO Low-level programming EFI_FILE_PROTOCOL

(Please keep-> Author: Luo Bingluobing4365's blog_CSDN blog-UEFI development, assembly language exploration, embedded development field blogger)

In Option ROM development, file reading and writing are rarely used. Therefore, in the early development process, I did not carefully study the protocols related to file processing.

I did get involved in some of this when I was writing a routine to display pictures. I used the example in UDK directly, modified the function to display BMP , and encapsulated the file reading inside the function.

Next I am going to use a few blogs to demonstrate how to display images. The UEFI drawing program has been discussed and demonstrated in previous blogs. What needs to be done next is to parse the image format, decompress the data, and display it on the screen through the drawing function.

The image formats to be processed include BMP and PCX. If you have time, study Jpeg.

Of course, first you have to figure out how to read and write files under UEFI.

1UEFI support for file systems

Compared with Legacy BIOS, UEFI not only supports hard disk access, but also supports file systems, mainly FAT format.

This is a good thing. When I was developing Foxdisk, I dealt with hard disk access the most. Most of the time I spent debugging, besides graphics, was hard disk access.

Moreover, Foxdisk uses the extended function of int 13h to access the disk, and can only be written in C embedded assembly. I borrowed the design of the hard disk drive under Linux , but I also have to be careful about the parameters of various hard disks, especially when debugging under a virtual machine, the heads and cylinders are completely unconventional.

Foxdisk also implements hard disk partitioning and formatting functions, and provides limited support for Fat32 format. When writing code, I was entangled in various details, and the feeling was... hard to describe. UEFI solves these problems very well, so I can spend more time thinking about what to do with UEFI.

(For file access under UEFI, I mainly refer to UEFI Spec and "UEFI Principles and Programming")

Figure 1 shows the structure of the stack of file systems. Each layer continuously shields the details of hard disk reading and writing, and finally builds a protocol for accessing the file system.

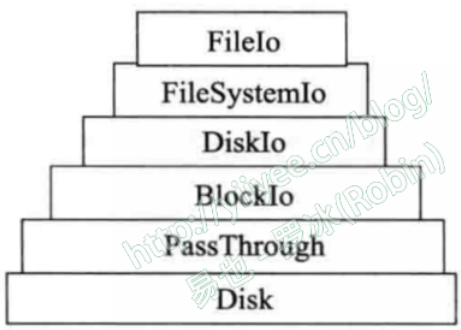

Figure 1 The stack structure of the file system protocol

Don't worry about the protocols of other layers for now, you can study them when needed. Currently, we mainly focus on the FileSystemIo layer, which provides complete file access capabilities.

Accessing files in UEFI is related to two protocols, EFI_SIMPLE_FILE_SYSTEM_PROTOCOL and EFI_FILE_PROTOCOL. As shown in the figure:

GUID

```
#define EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID \
 {0x0964e5b22,0x6459,0x11d2,\
  {0x8e,0x39,0x00,0xa0,0xc9,0x69,0x72,0x3b}}
```

Revision Number

```
#define EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_REVISION  0x00010000
```

Protocol Interface Structure

```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
 UINT64                                Revision;
 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

Figure 2 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL (UEFI Spec 2.8 Page 504)

```
typedef struct _EFI_FILE_PROTOCOL {
 UINT64          Revision;
 EFI_FILE_OPEN      Open;
 EFI_FILE_CLOSE     Close;
 EFI_FILE_DELETE    Delete;
 EFI_FILE_READ      Read;
 EFI_FILE_WRITE     Write;
 EFI_FILE_GET_POSITION GetPosition;
 EFI_FILE_SET_POSITION SetPosition;
 EFI_FILE_GET_INFO   GetInfo;
 EFI_FILE_SET_INFO   SetInfo;
 EFI_FILE_FLUSH     Flush;
 EFI_FILE_OPEN_EX    OpenEx; // Added for revision 2
 EFI_FILE_READ_EX    ReadEx; // Added for revision 2
 EFI_FILE_WRITE_EX   WriteEx; // Added for revision 2
 EFI_FILE_FLUSH_EX   FlushEx; // Added for revision 2
} EFI_FILE_PROTOCOL;
```

Parameters

Revision	The version of the **EFI_FILE_PROTOCOL** interface. The version specified by this specification is **EFI_FILE_PROTOCOL_LATEST_REVISION**. Future versions are required to be backward compatible to version 1.0.
Open	Opens or creates a new file. See the Open() function description.
Close	Closes the current file handle. See the Close() function description.
Delete	Deletes a file. See the Delete() function description.
Read	Reads bytes from a file. See the Read() function description.
Write	Writes bytes to a file. See the Write() function description.
GetPosition	Returns the current file position. See the GetPosition() function description.
SetPosition	Sets the current file position. See the SetPosition() function description.
GetInfo	Gets the requested file or volume information. See the GetInfo() function description.
SetInfo	Sets the requested file information. See the SetInfo() function description.
Flush	Flushes all modified data associated with the file to the device. See the Flush() function description.
OpenEx	Opens a new file relative to the source directory's location.
ReadEx	Reads data from a file.
WriteEx	Writes data to a file.
FlushEx	Flushes all modified data associated with a file to a device.

Figure 3 EFI_FILE_PROTOCOL (UEFI Spec 2.8 Page 507)

In the UEFI Spec, a GUID is provided for EFI_SIMPLE_FILE_SYSTEM_PROTOCOL, and its instance can be found through functions such as LocateProtocol. However, the instance of EFI_FILE_PROTOCOL is not located by GUID, but obtained through the function OpenVolume of EFI_SIMPLE_FILE_SYSTEM_PROTOCOL.

2. Programming

The file access provided by UEFI is relatively simple, divided into synchronous access and asynchronous access. When I implemented it, I only implemented the synchronous access code. The asynchronous access code (functions with the word "Ex", such as OpenEx) can be modified according to the examples in the book.

It should be noted that when a program accesses a file, there is a concept of "file read/write position". Each time a file is read or written, the read/write position is automatically updated according to the read/write situation. I made this mistake at the beginning. After writing data to a file, I read it directly, and the data read out was always 0. When the read/write position was positioned to the beginning of the file, the required data was read out.

```
//2019-06-11 09:23:48 luobing

#ifndef _FILERW_H
#define _FILERW_H
#include "Common.h"

EFI_STATUS OpenFile(EFI_FILE_PROTOCOL **fileHandle, CHAR16 *fileName, UINT64 OpenMode);
EFI_STATUS ReadFile(EFI_FILE_PROTOCOL *fileHandle, UINTN *bufSize, VOID *buffer);
EFI_STATUS WriteFile(EFI_FILE_PROTOCOL *fileHandle, UINTN *bufSize, VOID *buffer);
EFI_STATUS SetFilePosition(EFI_FILE_PROTOCOL *fileHandle, UINT64 position);
EFI_STATUS GetFilePosition(EFI_FILE_PROTOCOL *fileHandle, UINT64 *position);

#endif
```

Figure 4 Function writing

As needed, I wrote several access functions, including opening files, reading files, writing files, setting read and write positions, and getting read and write positions.

3. Compile and run

The program reads readme.txt , writes data into it, and reads it back out.

The operation effect is as follows:

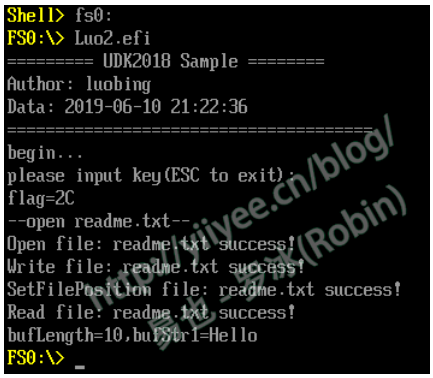

Figure 5: Operation effect

During the writing process, you need to pay attention to the pointers in various function parameters. It is best to be consistent with the parameters given in the Spec. Please refer to the code in Baidu Cloud for details.

Gitee address: https://gitee.com/luobing4365/uefi-explorer
The project code is located under: / 15 FileIo.