BIOS Practice: Reading Files

原创 Anthony Posted on 2022-01-11 14:07:38 Read 2.6k Collection 8 Likes 3

Category Column: BIOS learning practice Article Tags: UEFI C

In the UEFI shell, BIOS updates, EC updates, etc. all require reading files. You can learn how to read and write files from UEFI principles and programming. Then, we will move on to the practical part.

First define the variables needed later (for easy understanding):

AI generated projects 登录复制

```
  EFI_STATUS                     Status;
  EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *ptSFS;
  EFI_FILE_PROTOCOL              *ptRootFile;
  EFI_FILE_PROTOCOL              *ptFile;
  EFI_FILE_INFO                  *FileInfo;
  UINTN                          FileInfoSize;
  EFI_HANDLE                     *HandleBuffer;
  UINTN                          NumberOfHandles;
  UINTN                          Index;
  UINT8                          *Buffer;

  Status     = EFI_SUCCESS;
  ptRootFile = NULL;
  ptFile     = NULL;
  FileInfo   = NULL;
  HandleBuffer = NULL;
```

收起

1. First, find all devices that support FileSystemIo by using gBS->LocateHandleBuffer

AI generated projects 登录复制

```
Status = gBS->LocateHandleBuffer (
                 ByProtocol,
                 &gEfiSimpleFileSystemProtocolGuid,
                 NULL,
                 &NumberOfHandles,
                 &HandleBuffer
                 );
  if (EFI_ERROR (Status)) {
    return Status;
  }
```

收起

2. For each device that supports FileSystemIo, open the SimpleFileSystemProtocol on the device

AI generated projects 登录复制

```
for (Index = 0; Index < NumberOfHandles; Index ++) {
    Status = gBS->HandleProtocol(
               HandleBuffer[Index],
               &gEfiSimpleFileSystemProtocolGuid,
               (VOID **)&ptSFS
               );
```

3. Through OpenVolume of SimpleFileSystemProtocol, we can get the root directory handle on the FAT file system, and then we can operate the file according to this handle and open the file for reading and writing:

AI generated projects 登录复制

```
struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
  ///
  /// The version of the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL. The version
  /// specified by this specification is 0x00010000. All future revisions
  /// must be backwards compatible.
  ///
```

```
  UINT64                          Revision;
  EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
};
```



```
Status = ptRootFile->Open(
                  ptRootFile,
                  &ptFile,
                  FileName,
                  EFI_FILE_MODE_READ,
                  0
                  );
```

4. After opening, you can read the file, but there is a problem. How big is the file you are reading and how big is the cache you need? This is an unknown number. Therefore, before reading the file, we need to get some information about the file, such as FileSize.


```
typedef struct {
  ///
  /// The size of the EFI_FILE_INFO structure, including the Null-terminated FileName string.
  ///
  UINT64    Size;
  ///
  /// The size of the file in bytes.
  ///
  UINT64    FileSize;
  ///
  /// PhysicalSize The amount of physical space the file consumes on the file system volume.
  ///
  UINT64    PhysicalSize;
  ///
  /// The time the file was created.
  ///
  EFI_TIME  CreateTime;
  ///
  /// The time when the file was last accessed.
  ///
  EFI_TIME  LastAccessTime;
  ///
  /// The time when the file's contents were last modified.
  ///
  EFI_TIME  ModificationTime;
  ///
  /// The attribute bits for the file.
  ///
  UINT64    Attribute;
  ///
  /// The Null-terminated name of the file.
  ///
  CHAR16    FileName[1];
} EFI_FILE_INFO;
```


2. 读写文件信息

有时我们希望能得到文件的属性等信息，而不是文件的内容。例如，在读文件之前，我们希望能根据文件大小准备好读缓冲区。EFI_FILE_PROTOCOL 提供了 GetInfo 用于读取文件信息；SetInfo 用于设置文件信息。代码清单 7-37 是这两个函数的原型。

代码清单 7-37　FileIo 的 GetInfo、SetInfo 函数原型

```
//获取文件或文件系统的相关信息
typedef EFI_STATUS(EFIAPI *EFI_FILE_GET_INFO)(
  IN EFI_FILE_PROTOCOL *This,     //文件（或目录）句柄
  IN EFI_GUID *InformationType,   //要获取信息的类型标识符
  IN OUT UINTN *BufferSize,       //输入：缓冲区大小；输出：返回数据的长度
  OUT VOID *Buffer                //读缓冲区，数据类型由 InformationType 决定
  );
```



```
struct _EFI_FILE_PROTOCOL {
  ///
  /// 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.
  ///
  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;
  EFI_FILE_READ_EX     ReadEx;
  EFI_FILE_WRITE_EX    WriteEx;
  EFI_FILE_FLUSH_EX    FlushEx;
};
```

收起 ^

AI generated projects 登录复制

```
Status = ptFile->GetInfo (
                      ptFile,
                      &gEfiFileInfoGuid,
                      &FileInfoSize,
                      FileInfo
                      );
   if(Status == EFI_BUFFER_TOO_SMALL){
     FileInfo = AllocateZeroPool(FileInfoSize);
     if(FileInfo == NULL){
       Status = EFI_OUT_OF_RESOURCES;
     } else {
       Status = ptFile->GetInfo(
                           ptFile,
                           &gEfiFileInfoGuid,
                           &FileInfoSize,
                           FileInfo
                           );
```

收起 ^

Finally, the actual reading of the file

AI generated projects 登录复制

```
Buffer = AllocateZeroPool((UINTN)FileInfo->FileSize);

Status = ptFile->Read(ptFile, &FileInfo->FileSize, Buffer);
```

The overall process is as follows:

```
EFI_STATUS
ReadFileInFS (
  IN      CHAR16  *FileName,
  OUT     UINT8   **FileData,
  IN OUT  UINTN   *BufferSize
  )
{
  EFI_STATUS                     Status;
  EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *ptSFS;
  EFI_FILE_PROTOCOL              *ptRootFile;
  EFI_FILE_PROTOCOL              *ptFile;
  EFI_FILE_INFO                  *FileInfo;
  UINTN                          FileInfoSize;
  EFI_HANDLE                     *HandleBuffer;
  UINTN                          NumberOfHandles;
  UINTN                          Index;
  UINT8                          *Buffer;

  Status     = EFI_SUCCESS;
  ptRootFile = NULL;
  ptFile     = NULL;
  FileInfo   = NULL;
```

```
  HandleBuffer = NULL;
  Status = gBS->LocateHandleBuffer (
                 ByProtocol,
                 &gEfiSimpleFileSystemProtocolGuid,
                 NULL,
                 &NumberOfHandles,
                 &HandleBuffer
                 );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  for (Index = 0; Index < NumberOfHandles; Index ++) {
    Status = gBS->HandleProtocol(
               HandleBuffer[Index],
               &gEfiSimpleFileSystemProtocolGuid,
               (VOID **)&ptSFS
               );
    if (!EFI_ERROR (Status)) {
      Status = ptSFS->OpenVolume(ptSFS, &ptRootFile);
      if (!EFI_ERROR (Status)) {
        Status = ptRootFile->Open(
                    ptRootFile,
                    &ptFile,
                    FileName,
                    EFI_FILE_MODE_READ,
                    0
                    );
        if (!EFI_ERROR (Status)) {
          break;
        } else {
          if (ptRootFile != NULL){
            ptRootFile->Close(ptRootFile);
            ptRootFile = NULL;
          }
        }
      }
    }
  }

  if (HandleBuffer != NULL) {
    FreePool(HandleBuffer);
    HandleBuffer = NULL;
  }

  if (!EFI_ERROR (Status)) {
    FileInfo = NULL;
    FileInfoSize = 0;
    Status = ptFile->GetInfo (
                       ptFile,
                       &gEfiFileInfoGuid,
                       &FileInfoSize,
                       FileInfo
                       );
    if(Status == EFI_BUFFER_TOO_SMALL){
      FileInfo = AllocateZeroPool(FileInfoSize);
      if(FileInfo == NULL){
        Status = EFI_OUT_OF_RESOURCES;
      } else {
        Status = ptFile->GetInfo(
                           ptFile,
                           &gEfiFileInfoGuid,
                           &FileInfoSize,
                           FileInfo
                           );
      }
    }
  }
  if (EFI_ERROR (Status)) {
    goto ProcExit;
  }

  if(FileInfo->Attribute & EFI_FILE_DIRECTORY){
    Status = EFI_INVALID_PARAMETER;
    goto ProcExit;
  }

  Buffer = AllocateZeroPool((UINTN)FileInfo->FileSize);
  if (Buffer == NULL) {
    goto ProcExit;
  }
```

```
  Status = ptFile->Read(ptFile, &FileInfo->FileSize, Buffer);
  if (EFI_ERROR (Status)) {
    FreePool (Buffer);
    goto ProcExit;
  }
  *FileData = Buffer;
  *BufferSize = FileInfo->FileSize;

ProcExit:
  if (ptFile != NULL){
    ptFile->Close(ptFile);
  }
  if (ptRootFile != NULL){
    ptRootFile->Close(ptRootFile);
  }
  if (FileInfo != NULL) {
    FreePool (FileInfo);
  }

  return Status;
}
```

收起 ^

In the APP, as long as this function is added, the file can be read. We can read BIOS files, EC files, image files, etc. However, this function has a disadvantage. For example, if I compile a DisplayBmpTest.efi and run it to display a bmp image, then this image and DisplayBmpTest.efi must be in the root directory, not in the subdirectory. If in the subdirectory, the path must be added. For example, in the root directory, I directly use DisplayBmpTest.efi Tianocore.bmp, but in the subdirectory, I need to run DisplayBmpTest.efi \xxx\xxx\Tianocore.bmp, so how can I run DisplayBmpTest.efi Tianocore.bmp in the subdirectory? Then we need to find this path through code. OK, let's continue to find the path in the next section.