# UEFI Development Exploration 99 – Screen capture tool under UEFI Shell

原创 luobing4365 Posted on 2021-08-26 15:47:32 Read 3k Collection 10 Likes 1

Category columns: UEFI Development Article Tags: uefi UEFI Programming Practice bios Low-level application development

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

**Screen capture tool in UEFI Shell**

---

Recently, some programs can only be tested in the UEFI Shell of the actual machine. For example, the diskdump program in the previous article cannot be run in the simulator.

When running on the simulator, you can directly use various screenshot software to take screenshots, and the images are still relatively clear. When running on the actual machine, you can only use your mobile phone to record or take photos. I am still very clear about how bad my photography skills are. The photos taken can only barely show the running status of the program. Therefore, I have to use PS to process it on the computer later.

This kind of situation happens a lot, and we have to find a way to solve it.

On the way to work this morning, an idea came to my mind: why not write a software to take screenshots under UEFI Shell?

There are not many problems that need to be solved, mainly including the following:
1) The screenshot program must be able to reside in the memory to allow other programs to call it out while running;
2) Set keyboard hotkeys to call out the screenshot program running in the background at any time;
3) Store the entire screen in the format of a bmp image on the hard disk or USB flash drive.

It is somewhat similar to the TSR program in the previous DOS system. There were a large number of such programs in the early DOS.

After thinking about the implementation method, I think this screenshot software should be implemented under UEFI. The resident memory should be implemented through the UEFI driver. It is relatively simple to obtain and access BMP images on the screen. The image processing code written before can be slightly modified.

However, the more I thought about it, the more I felt like I had seen this idea before.

I checked the usual development logs and found that Microsoft's Github library provided software with the same function. I had seen it a long time ago, but I had never compiled and tested it.

This UEFI program exists in Microsoft's mu_plus library on Github. The address of the library is: https://github.com/microsoft/mu_plus.git.

The screenshot software is located in mu_plus's MsGraphicsPkg and is named PrintScreenLogger.

Since it is already there, there is no need to write more. This article tries to understand its implementation principle and test it in a real environment.

## 1 PrintScreenLogger code structure

From the overall design point of view, it is pretty much what you would expect. PrintScreenLogger uses the UEFI driver to allow the program to reside in memory.

The code structure is shown in Figure 1.

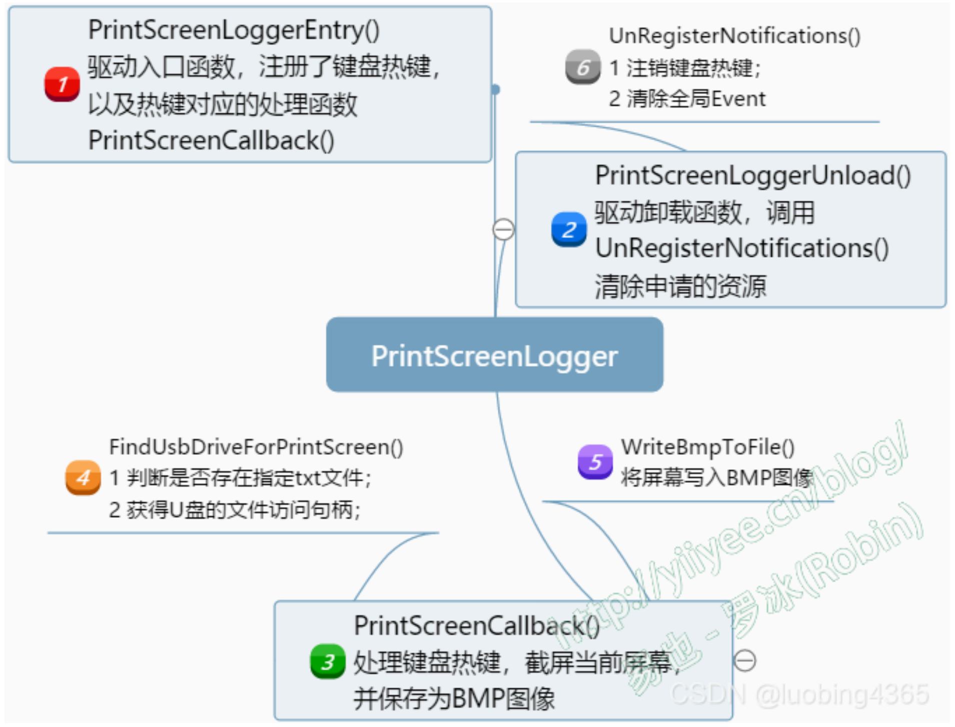


*Figure 1 PrintScreenLogger program structure*

Structurally, it consists of three functions: PrintScreenLoggerEntry(), PrintScreenLoggerUnload(), and PrintScreenCallback(). The global event gTimerEvent is used for synchronization to reserve enough time to store the BMP image to disk.

## 1) PrintScreenLoggerEntry()

This is the entry function of the driver. In this function, two hotkeys are registered: left Ctrl+PrtScn and right Ctrl+PrtScn, as well as the corresponding hotkey processing function PrintScreenCallback().

In addition, a global EVT_TIMER type event gTimerEvent is created in the function. The implementation code of the function is as follows:

```c
/**
  Main entry point for this driver.

  @param    ImageHandle     Image handle of this driver.
  @param    SystemTable     Pointer to the system table.

  @retval   EFI_STATUS      Always returns EFI_SUCCESS.

**/
EFI_STATUS
EFIAPI
PrintScreenLoggerEntry (
  IN EFI_HANDLE           ImageHandle,
  IN EFI_SYSTEM_TABLE     *SystemTable
  )
{
    EFI_STATUS      Status = EFI_NOT_FOUND;
    INTN            i;

    DEBUG((DEBUG_LOAD, "%a: enter...\n", __FUNCTION__));

    //
    // 1. Get access to ConSplitter's TextInputEx protocol
    //
    if (gST->ConsoleInHandle != NULL) {
        Status = gBS->OpenProtocol (
                        gST->ConsoleInHandle,
                        &gEfiSimpleTextInputExProtocolGuid,
                        (VOID **) &gTxtInEx,
                        ImageHandle,
                        NULL,
                        EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL);
    }
    if (EFI_ERROR(Status)) {
        DEBUG((DEBUG_ERROR, "%a: Unable to access TextInputEx protocol. Code = %r\n", __FUNCTION__, Status));
    }  else {

        //
        // 2.  Register for PrtScn callbacks
        //
        for (i = 0; i < NUMBER_KEY_NOTIFIES; i++) {
             Status = gTxtInEx->RegisterKeyNotify (
                          gTxtInEx,
                          &gPrtScnKeys[i].KeyData,
                          PrintScreenCallback,
                          &gPrtScnKeys[i].NotifyHandle);
            if (EFI_ERROR (Status)) {
                 DEBUG ((DEBUG_ERROR, "%a: Error registering key %d. Code = %r\n", __FUNCTION__, i, Status));
                 break;
            }
        }

        if (!EFI_ERROR(Status)) {
            //
            // 3. Create the PrtScn hold off timer
            //
            Status = gBS->CreateEvent(
                                EVT_TIMER,
                                0,
                                NULL,
                                NULL,
                                &gTimerEvent);
            if (!EFI_ERROR(Status)) {
                //
                // 4. Place event into the signaled state indicating PrtScn is active.
                //
                Status = gBS->SignalEvent (gTimerEvent);
            }
        }

        if (!EFI_ERROR(Status)) {
            DEBUG((DEBUG_INFO, "%a: exit. Ready for Ctl-PrtScn operation\n", __FUNCTION__));
        } else {
            UnRegisterNotifications ();
            DEBUG((DEBUG_ERROR, "%a: exit with errors. Ctl-PrtScn not operational. Code=%r\n", __FUNCTION__, Status));
        }
    }

    return EFI_SUCCESS;
}
```

收起 ^

## 2) PrintScreenLoggerUnload()

PrintScreenLoggerUnload() is the opposite of PrintScreenLoggerEntry(). It is the driver uninstallation function. It unregisters the previously applied keyboard hotkey and deletes the created global event gTimerEvent. The implementation code is as follows:

c AI generated projects 登录复制 run

```c
/**

  Callback to cleanup the driver on unload.

  @param    Event           Not Used.
  @param    Context         Not Used.

  @retval   None

**/
EFI_STATUS
EFIAPI
PrintScreenLoggerUnload (
  IN  EFI_HANDLE   ImageHandle
  )
{
    UnRegisterNotifications ();
    return EFI_SUCCESS;
}
/**
  Unregister TxtIn callbacks and end the timer

**/
VOID
UnRegisterNotifications (
    VOID
    ) {
    INTN       i;
    EFI_STATUS Status;

    for (i = 0; i < NUMBER_KEY_NOTIFIES; i++) {
        if (gPrtScnKeys[i].NotifyHandle != NULL) {
            Status = gTxtInEx->UnregisterKeyNotify (gTxtInEx,  gPrtScnKeys[i].NotifyHandle);
            if (EFI_ERROR(Status)) {
                DEBUG((DEBUG_ERROR, "%a: Unable to uninstall TxtIn Notify. Code = %r\n", __FUNCTION__, Status));
            }
        }
    }

    if (gTimerEvent != NULL) {
        gBS->SetTimer (gTimerEvent, TimerCancel, 0);
        gBS->CloseEvent (gTimerEvent);

    }
}
```

收起 ^

### 3) PrintScreenCallback()

The main functions of screenshot are concentrated in this function. First post the implementation of the function:

c AI generated projects 登录复制 run

```c
/**
  Handler for hot key notification

  @param KeyData         A pointer to a buffer that is filled in with the keystroke
                         information for the key that was pressed.

  @retval  EFI_SUCCESS   Always - Return code is not used by SimpleText providers.

**/
EFI_STATUS
EFIAPI
PrintScreenCallback (
  IN EFI_KEY_DATA     *KeyData
)
{
    EFI_FILE_PROTOCOL *FileHandle;
    UINTN              Index;
    CHAR16             PrtScrnFileName[] = L"PrtScreen####.bmp";
    EFI_STATUS         Status;
    EFI_STATUS         Status2;
    EFI_FILE_PROTOCOL *VolumeHandle;

    // We only register two keys - LeftCtrl-PrtScn and RightCtrl-PrtScn.
    // Assume print screen function if this function is called.
    DEBUG((DEBUG_INFO,"%a: Starting PrintScreen capture. Sc=%x, Uc=%x, Sh=%x, Ts=%x\n",
        __FUNCTION__,
        KeyData->Key.ScanCode,
        KeyData->Key.UnicodeChar,
        KeyData->KeyState.KeyShiftState,
        KeyData->KeyState.KeyToggleState));

    Status = gBS->CheckEvent (gTimerEvent);

    if (Status == EFI_NOT_READY) {
        DEBUG((DEBUG_INFO,"Print Screen request ignored\n"));

        return EFI_SUCCESS;
    }

    //
    // 1. Find a suitable USB drive - one that has PrintScreenEnable.txt on it.
    //
    Status = FindUsbDriveForPrintScreen(&VolumeHandle);

    if (!EFI_ERROR(Status)) {
        //
        // 2. Find the first value of PrtScreen#### that is available
        //
        Index = 0;

        do {
            Index++;
            if (Index > MAX_PRINT_SCREEN_FILES) {
                goto Exit;
            }

            UnicodeSPrint (PrtScrnFileName, sizeof (PrtScrnFileName), L"PrtScreen%04d.bmp", Index);
            Status = VolumeHandle->Open (VolumeHandle, &FileHandle, PrtScrnFileName, EFI_FILE_MODE_READ, 0);
            if (!EFI_ERROR(Status)) {
                if (Index % PRINT_SCREEN_DEBUG_WARNING == 0) {
                    DEBUG((DEBUG_INFO,"%a: File %s exists.  Trying again\n", __FUNCTION__, PrtScrnFileName));
                }
                Status2 = FileHandle->Close (FileHandle);
                if (EFI_ERROR(Status2)) {
                    DEBUG((DEBUG_ERROR,"%a: Error closing File Handle. Code = %r\n", __FUNCTION__, Status2));
                }
                continue;
            }
            if (Status == EFI_NOT_FOUND) {
                break;
            }
        } while (TRUE);

        //
        // 3. Create the new file that will contain the bitmap
        //
        Status = VolumeHandle->Open (VolumeHandle, &FileHandle, PrtScrnFileName, EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE | EFI_FILE_MODE_CREATE, EFI_FILE_ARCHIVE);
        if (EFI_ERROR(Status)) {
            DEBUG((DEBUG_ERROR,"%a: Unable to create file %s. Code = %r\n", __FUNCTION__, PrtScrnFileName, Status));
            goto Exit;
        }

        //
        // 4. Write the contents of the display to the new file
        //
        Status = WriteBmpToFile (FileHandle);
        if (!EFI_ERROR(Status)) {
            DEBUG((DEBUG_INFO,"%a: Screen captured to file %s.\n", __FUNCTION__, PrtScrnFileName));
        }
        //
        // 4. Close the bitmap file
        //
        Status2 = FileHandle->Close (FileHandle);
        if (EFI_ERROR(Status2)) {
            DEBUG((DEBUG_ERROR,"%a: Error closing bit map file %s. Code = %r\n", __FUNCTION__, PrtScrnFileName, Status2));
        }
Exit:
        //
        // 5. Close the USB volume
        //
        Status2 = VolumeHandle->Close (VolumeHandle);
        if (EFI_ERROR(Status2)) {
            DEBUG((DEBUG_ERROR,"%a: Error closing Vol Handle. Code = %r\n", __FUNCTION__, Status2));
        }
    }

    // Ignore future PrtScn requests for some period.  This is due to the make
    // and break of PrtScn being identical, and it takes a few seconds to complete
    // a single screen capture.
    Status = gBS->SetTimer (gTimerEvent, TimerRelative, PRINT_SCREEN_DELAY);

    return EFI_SUCCESS;
}
```

收起 ︿

The function first checks whether the current storage device is a USB drive and whether there is a file PrintScreenEnable.txt in its root directory. This is achieved through the function FindUsbDriveForPrintScreen().

After processing, the FindUsbDriveForPrintScreen() function will return a pointer variable of type EFI_FILE_PROTOCOL, which will be used as the file Protocol instance for subsequent access to this USB drive.

Then analyze the files in the root directory of the USB drive to see if PrtScreen####.bmp exists (#### value range is 0000 to 0512). then PrtScreen0016.bmp is created).

After successful creation, call WriteBmpToFile() to save the current screenshot into the created bmp file.

It should be noted that at the end of the function, a 3-second trigger time is set for gTimerEvent. This time is used to allow the device to complete the storage of the BMP file to prevent the screenshot process from starting before the file is saved.

For the FindUsbDriveForPrintScreen() and WriteBmpToFile() called in the function, please view the source code in the project given at the end of the article. The implementation will not be posted for analysis.

## 2 Test Run

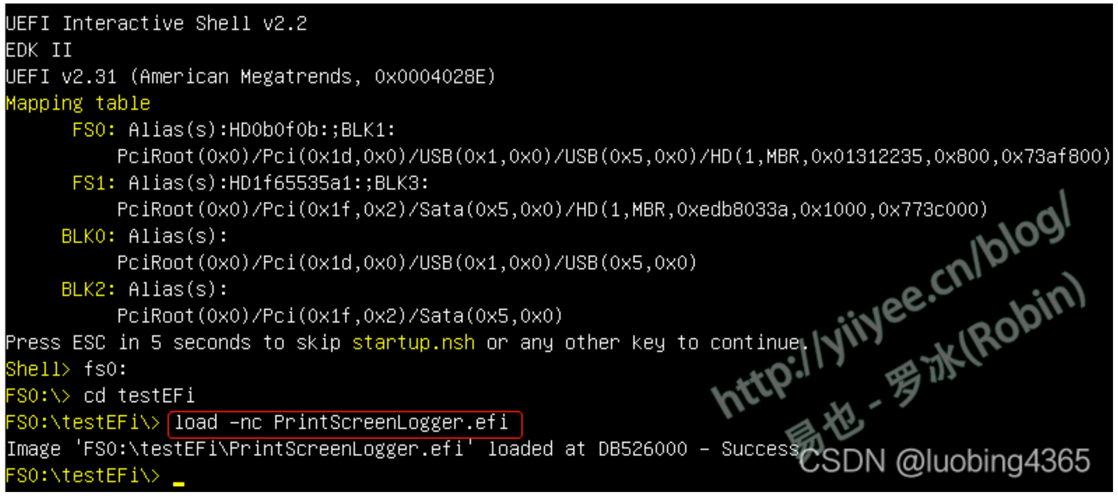
A few days ago, I wrote about hard disk access Diskdump, and the pictures I took were very bad, which I was not satisfied with (Figure 2 of the previous article UEFI Development and Exploration 98). It just so happens that today I want to experiment with a new way of taking screenshots.

There were some minor issues in the original PrintScreenLogger that prevented it from compiling. These were mainly header file inclusions and a few cast issues. I have now modified the project file in RobinPkg and can compile it using the following command:

```c
C:\vUDK2018\edk2>build -p RobinPkg\RobinPkg.dsc -m RobinPkg\Drivers\PrintScreenLogger\PrintScreenLogger.inf -a X64
```

Copy it to a USB flash drive with UEFI Shell, and create PrintScreenEnable.txt in the root directory of the USB flash drive. The file content should be empty.

Start UEFI Shell and load PrintScreenLogger.efi, as shown in Figure 2.

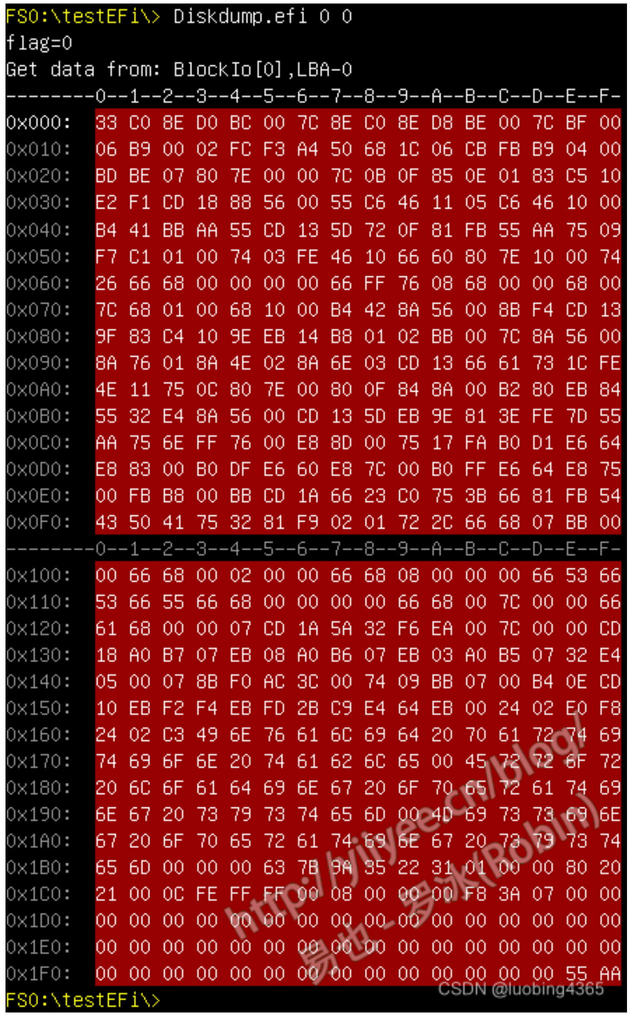


```
UEFI Interactive Shell v2.2
EDK II
UEFI v2.31 (American Megatrends, 0x0004028E)
Mapping table
      FS0: Alias(s):HD0b0f0b:;BLK1:
          PciRoot(0x0)/Pci(0x1d,0x0)/USB(0x1,0x0)/USB(0x5,0x0)/HD(1,MBR,0x01312235,0x800,0x73af800)
      FS1: Alias(s):HD1f65535a1:;BLK3:
          PciRoot(0x0)/Pci(0x1f,0x2)/Sata(0x5,0x0)/HD(1,MBR,0xedb8033a,0x1000,0x773c000)
     BLK0: Alias(s):
          PciRoot(0x0)/Pci(0x1d,0x0)/USB(0x1,0x0)/USB(0x5,0x0)
     BLK2: Alias(s):
          PciRoot(0x0)/Pci(0x1f,0x2)/Sata(0x5,0x0)
Press ESC in 5 seconds to skip startup.nsh or any other key to continue.
Shell> fs0:
FS0:\> cd testEFi
FS0:\testEFi\> load -nc PrintScreenLogger.efi
Image 'FS0:\testEFi\PrintScreenLogger.efi' loaded at DB526000 - Success
FS0:\testEFi\> _
```

*Figure 2 Loading the screenshot tool*

After loading successfully, you can use Ctrl+PstScn to take a screenshot. The image will be stored in the root directory of the USB drive with the name PrtScreen####.bmp (#### ranges from 0000 to 0512). In fact, Figure 2 was captured using this method.

Run the Diskdump program in the previous article, and the screenshot is shown in Figure 3:



```
FS0:\testEFi\> Diskdump.efi 0 0
flag=0
Get data from: BlockIo[0],LBA-0
--------0--1--2--3--4--5--6--7--8--9--A--B--C--D--E--F-
0x000:   33 C0 8E D0 BC 00 7C 8E C0 8E D8 BE 00 7C BF 00
0x010:   06 B9 00 02 FC F3 A4 50 68 1C 06 CB FB B9 04 00
0x020:   BD BE 07 80 7E 00 00 7C 0B 0F 85 0E 01 83 C5 10
0x030:   E2 F1 CD 18 88 56 00 55 C6 46 11 05 C6 46 10 00
0x040:   B4 41 BB AA 55 CD 13 5D 72 0F 81 FB 55 AA 75 09
0x050:   F7 C1 01 00 74 03 FE 46 10 66 60 80 7E 10 00 74
0x060:   26 66 68 00 00 00 00 66 FF 76 08 68 00 00 68 00
0x070:   7C 68 01 00 68 10 00 B4 42 8A 56 00 8B F4 CD 13
0x080:   9F 83 C4 10 9E EB 14 B8 01 02 BB 00 7C 8A 56 00
0x090:   8A 76 01 8A 4E 02 8A 6E 03 CD 13 66 61 73 1C FE
0x0A0:   4E 11 75 0C 80 7E 00 80 0F 84 8A 00 B2 80 EB 84
0x0B0:   55 32 E4 8A 56 00 CD 13 5D EB 9E 81 3E FE 7D 55
0x0C0:   AA 75 6E FF 76 00 E8 8D 00 75 17 FA B0 D1 E6 64
0x0D0:   E8 83 00 B0 DF E6 60 E8 7C 00 B0 FF E6 64 E8 75
0x0E0:   00 FB B8 00 BB CD 1A 66 23 C0 75 3B 66 81 FB 54
0x0F0:   43 50 41 75 32 81 F9 02 01 72 2C 66 68 07 BB 00
--------0--1--2--3--4--5--6--7--8--9--A--B--C--D--E--F-
0x100:   00 66 68 00 02 00 00 66 68 08 00 00 00 66 53 66
0x110:   53 66 55 66 68 00 00 00 00 66 68 00 7C 00 00 66
0x120:   61 68 00 00 07 CD 1A 5A 32 F6 EA 00 7C 00 00 CD
0x130:   18 A0 B7 07 EB 08 A0 B6 07 EB 03 A0 B5 07 32 E4
0x140:   05 00 07 8B F0 AC 3C 00 74 09 BB 07 00 B4 0E CD
0x150:   10 EB F2 F4 EB FD 2B C9 E4 64 EB 00 24 02 E0 F8
0x160:   24 02 C3 49 6E 76 61 6C 69 64 20 70 61 72 74 69
0x170:   74 69 6F 6E 20 74 61 62 6C 65 00 45 72 72 6F 72
0x180:   20 6C 6F 61 64 69 6E 67 20 6F 70 65 72 61 74 69
0x190:   6E 67 20 73 79 73 74 65 6D 00 4D 69 73 73 69 6E
0x1A0:   67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74
0x1B0:   65 6D 00 00 00 63 7B 9A 35 22 31 01 00 00 80 20
0x1C0:   21 00 0C FE FF FF 00 08 00 00 00 F8 3A 07 00 00
0x1D0:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x1E0:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x1F0:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA
FS0:\testEFi\>
```

*Figure 3 Screenshot of Diskdump running*

Compared with the test result chart at the end of the previous blog, this chart is obviously clearer. Figure 3 is composed of two pictures spliced together, mainly because one screen cannot fully display 512 bytes of data. When splicing, I did not do any beautification, just deleted the redundant content.

At this point, we have a tool to take screenshots under UEFI.

Considering the usual needs, I think I can make a simple screen recording software to record the operations under UEFI Shell. Of course, it can also be simply processed to collect 24 pictures in 1 second (or less, 8-12 frames per second). The pictures are then integrated and processed frame by frame using software to get the operation process.

When you have time, make some minor changes to meet this requirement.

The project code address of this article is as follows:

*Gitee address: https://gitee.com/luobing4365/uefi-explorer*
*Project code is located in: /FF RobinPkg/RobinPkg/Drivers/PrintScreenLogger*

---