UEFI Development Exploration 18 – Using HII to display Chinese characters 3

原创 luobing4365 Posted on 2019-09-16 17:04:10 Read 1.8k Collection 3 Likes 1

Category columns: UEFI Development Article Tags: UEFI Programming UEFI HII UEFI color characters Low-level programming UEFI Chinese character programming

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

Keywords for this blog: Font.

Today I plan to study all aspects of another UEFI font, Font, still focusing on code transplantation, and occasionally solving several problems I raised.

1 Text mode and Graphics mode

Before the experiment, I want to talk about Text mode and Graphics mode. In the UEFI spec, there is little discussion about Text mode and Graphics mode. I have always been confused whether the Text mode and Graphics mode in the spec are what I understand.

On the Legacy BIOS , the earliest Text mode is 25 rows and 80 columns, and the screen can display 2000 characters. Each character on the display screen is represented by two consecutive bytes in the memory, one byte stores the ASCII code, and the other byte stores the character's attributes.

For all display adapters, the principle of displaying characters in Text mode is the same, the difference is that the starting address of the video display memory (that is, video memory) of various adapters is different: for MDA, the starting address is B000:0000; for CGA, EGA, VGA, the starting address is B8000:0000. The monitors we use now are all inherited from VGA, so the starting address of Text mode is B800:0000.

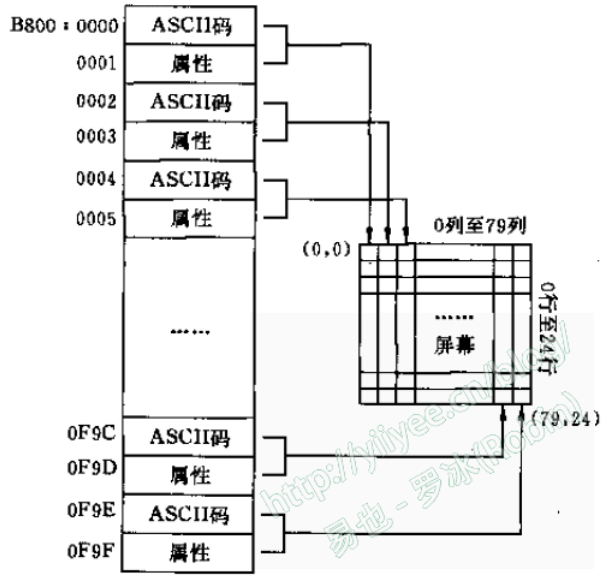

Figure 1 shows the relationship between the storage area and the display position

Whether you call BIOS int 10h directly, DOS int 21h, or use the C library function printf , the data will eventually be written to B800:0000. This is my understanding of Text mode.

Graphics mode is more complicated, please refer to Vesa standard. The starting address of the video memory of graphics mode is A000:0000, and its display mode must be set before use. Different display modes correspond to different resolutions and different numbers of colors.

In the previous blog "Foxdisk11-Displaying Chinese Characters without Font Library 2", I briefly described the programming of graphics mode. I often use the 1024×768, 256-color graphics mode.

Obviously, this mode has a different video memory location and a completely different method of operation. To display Chinese in Text mode, additional support is required.

A long time ago, some people made Chinese cards (such as Lenovo's Ni Guangnan, Founder's Wang Xuan, Giant's Shi Yuzhu, and Kingsoft's Lei Jun), which can add a character generator at the hardware level to display Chinese in Text mode. Some soft Chinese cards use Graphics mode to realize virtual text state for Chinese input.

In the experiment in the previous blog, the UEFI shell should be in Text mode. **The program runs in this state, but it can print Chinese characters.** I suspect that the Text mode under UEFI is not the Text mode defined earlier, at least it is not operating in the B800 segment.

The truth can only be known after reading through the UEFI source code. This question does not affect my subsequent programming, so I will leave it aside for now.

2 Print directly prints Chinese characters

In the code of the previous blog, a line of test code was added:

Print((const CHAR16*)L"Hello UEFI SimpleFont,My name is Luo Bing~Robin.\n"); The running effect is as follows:

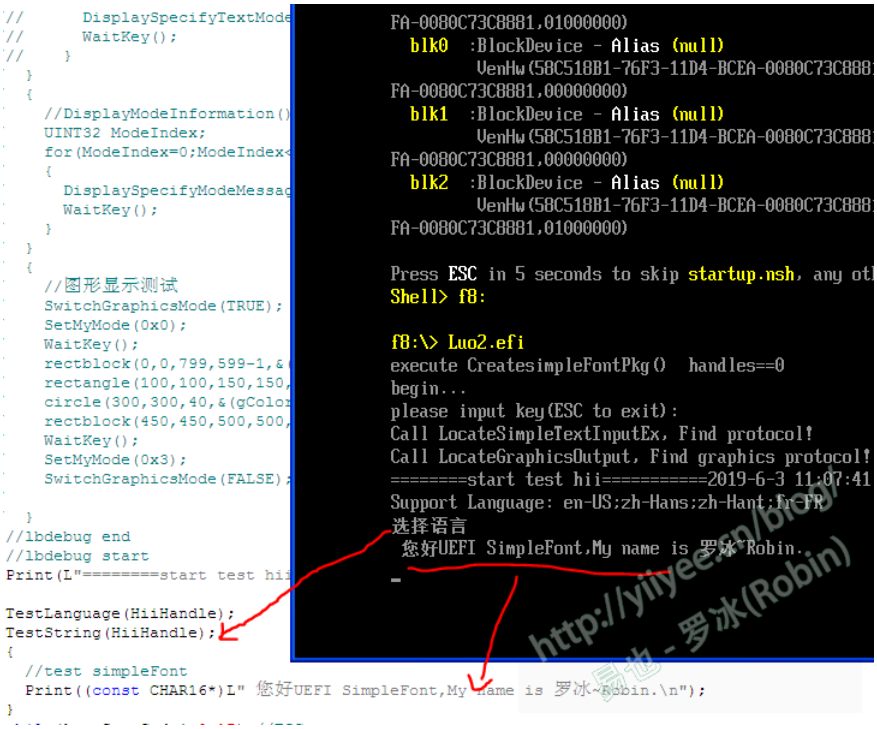

Figure 2 Test Print to print Chinese characters

It seems that Print uses SimpleFont directly for display. From the current reference materials, you can learn a lot of relevant details by carefully reading the display part in MdkModulePkg.

I really want a debugging method that can track execution, similar to windbg , and if there is source code , I can locate the execution function. In this way, it is easy to track how each function runs and which functions are called.

3 Font format

Let's go back to the study of Font format. Font format is more complex than SimpleFont. It no longer limits the dot matrix size and adds margins and step lengths.

```
typedef struct _EFI_HII_GLYPH_INFO {
 UINT16 Width;
 UINT16 Height;
 INT16 OffsetX;
 INT16 OffsetY;
 INT16 AdvanceX;
} EFI_HII_GLYPH_INFO;
```

Width	Width of the character or character cell, in pixels. For fixed-pitch fonts, this is the same as the advance.
Height	Height of the character or character cell, in pixels.
OffsetX	Offset to the horizontal edge of the character cell.
OffsetY	Offset to the vertical edge of the character cell.
AdvanceX	Number of pixels to advance to the right when moving from the *origin* of the current glyph to the origin of the next glyph.

Figure 3 Font dot matrix information structure (UEFI Spec 2.8 page1813)

The Font package built on Font dot matrix is also more complex than the SimpleFont package. It provides an indexing mechanism, which consists of a Font package header and a dot matrix block list. In my opinion, it actually provides a way to trade space for time, and the continuous storage feature speeds up the search speed.

```
typedef struct _EFI_HII_FONT_PACKAGE_HDR {
 EFI_HII_PACKAGE_HEADER Header;
 UINT32          HdrSize;
 UINT32          GlyphBlockOffset;
 EFI_HII_GLYPH_INFO    Cell;
 EFI_HII_FONT_STYLE    FontStyle;
 CHAR16          FontFamily[];
} EFI_HII_FONT_PACKAGE_HDR;
```

Header	The standard package header, where *Header.Type* = **EFI_HII_PACKAGE_FONTS**.
HdrSize	Size of this header.
GlyphBlockOffset	The offset, relative to the start of this header, of a series of variable-length glyph blocks, each describing information about the bitmap associated with a glyph.
Cell	This contains the measurement of the widest and tallest characters in the font (*Cell.Width* and *Cell.Height*). It also contains the default offset to the horizontal and vertical origin point of the character cell (*Cell.OffsetX* and *Cell.OffsetY*). Finally, it contains the default *AdvanceX*.
FontStyle	The design style of the font, 1 bit per style. See **EFI_HII_FONT_STYLE**.
FontFamily	The null-terminated string with the name of the font family to which the font belongs.

Figure 4 Font header (UEFI Spec 2.8 page1809)

The font glyph block list is composed of multiple glyph blocks, each of which has a different meaning. There are many types of glyph blocks, which are not listed here one by one. You can refer to UEFI spec page 1810~page 1819.

Another code that can be read is \MdeModulePkg\Universal\HiiDatabaseDxe\Font.c, where the function FindGlyphBlock() shows how to locate the corresponding Unicode character in the Font package.

The porting examples still come from "UEFI Principles and Programming". The two pictures on p280 and p281 are very helpful for understanding the program.

4 Code transplantation and compilation and operation

This porting was a little troublesome, mostly because my compilation environment is different from the author's. My compiler requires that all variable definitions must be placed before the executable body, which should be due to the C89 standard, so the author's code cannot be compiled.

In addition, there are problems with the initialization of the structure, which is probably related to C89/C99.

I transplanted FillGLYPH(), FillNarrowGLYPH(), FillWideGLYPH() and CreateDMFontPkg() to Luo2.c, and copied the test code to the main function. I found an external variable extern CHAR16* FontName; I don't know which file it is hidden in, and I can't find it. I changed it to a local variable, gave it a random value, and it compiled successfully.

```
EFI_IMAGE_OUTPUT                    gSystemFrameBuffer;
EFI_IMAGE_OUTPUT*                    pSystemFrameBuffer  = &gSystemFrameBuf
const CHAR16* Str = (const CHAR16*)L"您好UEFI Font";
CHAR16 R1[32];
EFI_FONT_DISPLAY_INFC SystemFont[4]= {{
    {0x00,0x00,0xFF,0x00}, // BGRA ForeColor
    {0xFF,0xFF
    EFI_FONT_I
    {EFI_HII_F
}};
CHAR16 R2[32];
(void)R1;
(void)R2;
(void)Str;
(void)pSystemF
(void)Status;
(void)HiiFont;
(void)gGraphic
(void)gSystemF
(void)SystemFc

Status = gBS->
//Print((const

gBS->LocatePro
//Print((const
gSystemFrameBu
gSystemFrameBu
gSystemFrameBu
CopyMem(System
Status = HiiFo
        HiiFon
        EFI_HI
        EFI_HI
        EFI_HI
        (CHAR1
        (const
```

```
UGA Window 1

            VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)
FA-  您好UEFI Font)0000000)
  blk1  :BlockDevice - Alias (null)
            VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)
FA-0080C73C8881,00000000)
  blk2  :BlockDevice - Alias (null)
            VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)
FA-0080C73C8881,01000000)

Press ESC in 1 seconds to skip startup.nsh, any othe
Shell> fs0:

fs0:\> Luo2.efi
execute CreatesimpleFontPkg()  handles==0
execute CreateDMFontPkg()  handles1==0
begin...
please input key(ESC to exit):
Call LocateSimpleTextInputEx, Find protocol!
Call LocateGraphicsOutput, Find graphics protocol!
========start test hii==========2019-6-7 12:56:21
Support Language: en-US;zh-Hans;zh-Hant;fr-FR
选择语言
 您好UEFI SimpleFont,My name is 罗冰~Robin.
========end test hii==========2019-6-7 12:56:19
```

Figure 5 Program running results

The text appears strangely in the upper left corner, covering the characters behind it. This is to display the string in a graphical way. The principle of StringToImage is to use Blt to output the string bitmap to the screen.

This proves my guess to some extent that the Text mode of UEFI Shell is actually running in a certain Graphics mode, running the shell in a simulated way, otherwise Blt cannot output the bitmap. At least UEFI Text mode should not be the traditional Text mode.

My expectation to use Font in UEFI Shell was dashed, at least not in the way I imagined. Actually, it is true that the font size is relatively free, so how can it be compatible with other texts in Shell? The text display in UEFI Shell should call the SimpleFont font library.

5 One more thing

I saw the drive letter displayed in UEFI Shell is yellow on a black background. I wondered how to achieve this effect? Can I use Print directly to achieve this?

The processing of Legacy BIOS is relatively simple, and its core principle is to modify the attribute bytes of the video memory. Whether it is a BIOS interrupt or a DOS interrupt, it is ultimately processed in this way.

printf() also provides methods for font color and background processing, although the syntax is rather strange.

Print under UEFI is somewhat different from printf. I tried the same method but it didn't work. I don't have the energy to track down the implementation of Print.

Finally, I found that ConOut provides SetAttribute, which can be used to modify the foreground and background colors of text (UEFI Spec 2.8 page449). I wrote a few test codes and the results are as follows:

```
//test simpleFont
Print((const CHAR16*)L" 您好UEFI SimpleFont,My name is 罗冰~Robin.\n");
gST->ConOut->SetAttribute(gST->ConOut,EFI_BACKGROUND_RED|EFI_WHITE);
gST->ConOut->OutputString(gST->ConOut,L"begin...\n\r");
Print((const CHAR16*)L" 测试彩色文字\n");
```

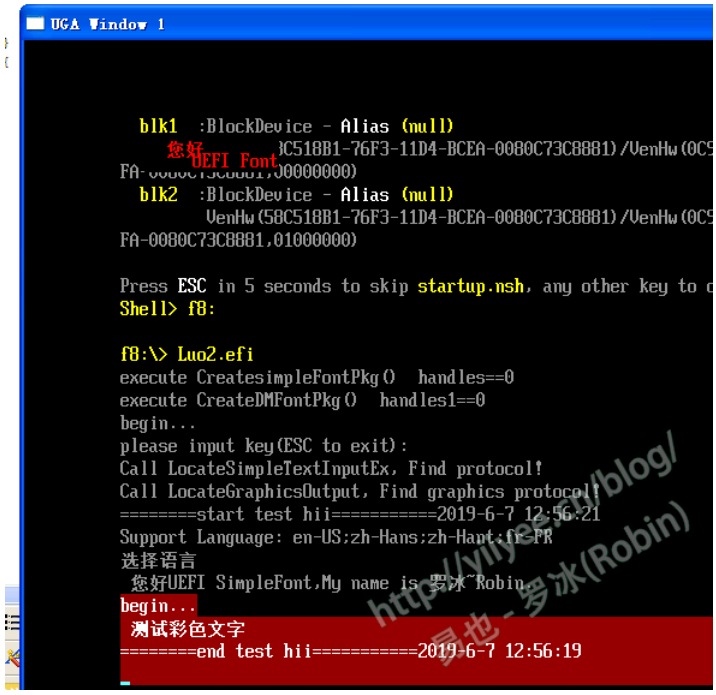

Figure 6 Printing color text

Gitee address: https://gitee.com/luobing4365/uefi-explorer
The project code is located under: /11 HiiShellPrint-Font.