UEFI Console Splitter

原创 Pedro Posted on 2015-11-08 12:25:37 Read 3.9k Collection 13 Likes 5

Category Column: UEFI-BIOS Article Tags: uefi

Previously, when writing UEFI programs, if you wanted to print information on the terminal or receive user input, you just had to call the ConIn/ConOut protocol under SystemTable.

```
//向终端设备输出字符
Status = gST->ConOut->OutputString (
                        gST->ConOut,
                        L"hehe"
                        );
//接收输入设备的信息  EFI_INPUT_KEY Key;
Status = gST->ConIn->ReadKeyStorke (
                       gST->ConIn,
                       &Key
                       );
```

In this way, the information can be sent to all the connected output terminals of the host/or read all the input devices. But at that time, there was a question: how can the user read the input content of all input devices through a reading interface? So I started to read the spec and found relevant introductions. Then let's start to learn about this UEFI feature —— *Console Splitter*

Console Splitter

You can refer to the detailed introduction of Console Splitter in Beyond BIOS . I will not post the original text here, but simply explain the basic principle of Console Splitter. UEFI provides a splitter/merging mechanism for the system to have multiple input/output devices. Simply put, Console Splitter installs itself on the gST system table and acts as a main Console device. Then the Splitter driver will virtualize three devices virtual ConIn, virtual ConOut, virtual ErrOut, and install the corresponding protocol for them. It will constantly detect these input and output devices, which are specified by the corresponding ConOut, ConIn, ErrOut variables (except hot-swappable devices). In fact, ConOut, ConIn, ErrOut store the PATH of the specified device. The device PATH will be introduced in the blog later.

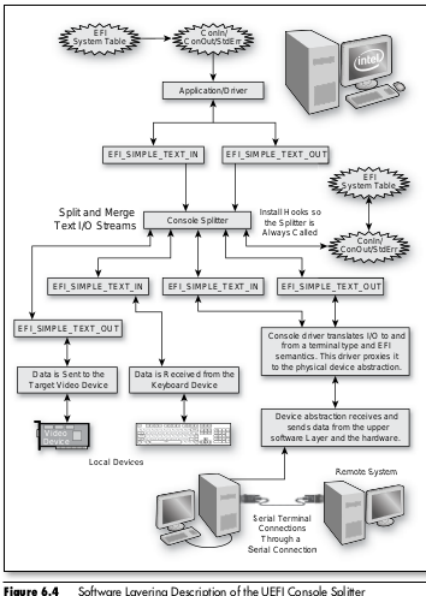

Figure 6.4 Software Layering Description of the UEFI Console Splitter

As shown in the picture, Console Splitter monitors these input and output devices.

Let's start by explaining the principles and practical process of this Splitter driver.

(edk2/MdeModulePkg/Univeral/Console/ConSplitterDxe)

```
这里只贴有ConIn 的code, 而ConOut,ErrOut大同小异就不贴code
EFI_STATUS
EFIAPI
ConSplitterDriverEntry(
  IN EFI_HANDLE           ImageHandle,
  IN EFI_SYSTEM_TABLE     *SystemTable
```

```
  )
{
  EFI_STATUS              Status;

  //
  // Install driver model protocol(s).
  //
  Status = EfiLibInstallDriverBindingComponentName2 (
             ImageHandle,
             SystemTable,
             &gConSplitterConInDriverBinding,
             ImageHandle,
             &gConSplitterConInComponentName,
             &gConSplitterConInComponentName2
             );
  ASSERT_EFI_ERROR (Status);
  Status = ConSplitterTextInConstructor (&mConIn);
  if (!EFI_ERROR (Status)) {
    Status = gBS->InstallMultipleProtocolInterfaces (
                    &mConIn.VirtualHandle,
                    &gEfiSimpleTextInProtocolGuid,
                    &mConIn.TextIn,
                    &gEfiSimpleTextInputExProtocolGuid,
                    &mConIn.TextInEx,
                    &gEfiSimplePointerProtocolGuid,
                    &mConIn.SimplePointer,
                    &gEfiAbsolutePointerProtocolGuid,
                    &mConIn.AbsolutePointer,
                    NULL
                    );
    if (!EFI_ERROR (Status)) {
      //
      // Update the EFI System Table with new virtual console
      // and update the pointer to Simple Text Input protocol.
      //
      gST->ConsoleInHandle  = mConIn.VirtualHandle;
      gST->ConIn            = &mConIn.TextIn;
    }
  }
```

这就是Splitter driver的入口函数，首先在这个UEFI driver的Image上安装UefiDriverBindingProtocol。构造一个vistual ConIn device，然后在这个device 的handl上安装上TextIn ,TextInEx, SimplePointer, AbsolutePointer 相应的protocol.
看一下ConSpliterConInDriverBinding 的 Support 函数

```
EFI_STATUS
EFIAPI
ConSplitterConInDriverBindingSupported (
  IN  EFI_DRIVER_BINDING_PROTOCOL     *This,
  IN  EFI_HANDLE                      ControllerHandle,
  IN  EFI_DEVICE_PATH_PROTOCOL        *RemainingDevicePath
  )
{
  return ConSplitterSupported (
          This,
          ControllerHandle,
          &gEfiConsoleInDeviceGuid
          );
}
EFI_STATUS
ConSplitterSupported (
  IN  EFI_DRIVER_BINDING_PROTOCOL     *This,
  IN  EFI_HANDLE                      ControllerHandle,
  IN  EFI_GUID                        *Guid
  )
{
  EFI_STATUS  Status;
  VOID        *Instance;

  //
  // Make sure the Console Splitter does not attempt to attach to itself
  //
  if (ControllerHandle == mConIn.VirtualHandle  ||
      ControllerHandle == mConOut.VirtualHandle ||
      ControllerHandle == mStdErr.VirtualHandle
      ) {
    return EFI_UNSUPPORTED;
  }

```

```
35
36    //
37    // Check to see whether the specific protocol could be opened BY_DRIVER
38    //
39    Status = gBS->OpenProtocol (
40                    ControllerHandle,
41                    Guid,
42                    &Instance,
43                    This->DriverBindingHandle,
44                    ControllerHandle,
45                    EFI_OPEN_PROTOCOL_BY_DRIVER
46                    );
47
48    if (EFI_ERROR (Status)) {
49      return Status;
50    }
51
52    gBS->CloseProtocol (
53          ControllerHandle,
54          Guid,
55          This->DriverBindingHandle,
56          ControllerHandle
57          );
58
59    return EFI_SUCCESS;
}
```

Simply analyzing this function , this function tests whether all controller handles have gEfiConsoleInDeviceGuid installed. If the support function returns success, then the start function will be called. It will detect all controller handles. If these handles have device paths installed and the path of this device is the same as the variable mentioned in ConIn, it is a mutilate instance path. If there is a path that matches one of the instances, then this guid will be installed for this controller. The hot-swappable input device will not determine whether its path matches the paths in ConIn). So assuming there is a device that meets the requirements of this support function, the Start function will be called. Let's take a look at the start function.

```
1  EFI_STATUS
2  EFIAPI
3  ConSplitterStdErrDriverBindingStart (
4    IN  EFI_DRIVER_BINDING_PROTOCOL     *This,
5    IN  EFI_HANDLE                      ControllerHandle,
6    IN  EFI_DEVICE_PATH_PROTOCOL        *RemainingDevicePath
7    )
8  {
9    EFI_STATUS                      Status;
10   EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL  *TextOut;
11
12   //
13   // Start ConSplitter on ControllerHandle, and create the virtual
14   // agrogated console device on first call Start for a StandardError handle.
15   //
16   Status = ConSplitterStart (
17             This,
18             ControllerHandle,
19             mStdErr.VirtualHandle,
20             &gEfiStandardErrorDeviceGuid,
twen           &gEfiSimpleTextOutProtocolGuid,
twen           (VOID **) &TextOut
twen           );
twen   if (EFI_ERROR (Status)) {
25       return Status;
26     }
27   }
28   EFI_STATUS
29   ConSplitterStart (
30     IN  EFI_DRIVER_BINDING_PROTOCOL     *This,
31     IN  EFI_HANDLE                      ControllerHandle,
32     IN  EFI_HANDLE                      ConSplitterVirtualHandle,
33     IN  EFI_GUID                        *DeviceGuid,
34     IN  EFI_GUID                        *InterfaceGuid,
35     OUT VOID                            **Interface
36     )
37   {
38     EFI_STATUS  Status;
39     VOID        *Instance;
40
41     //
42     // Check to see whether the ControllerHandle has the DeviceGuid on it.
43     //
44     Status = gBS->OpenProtocol (
45                     ControllerHandle,
```

```
                  DeviceGuid,
                  &Instance,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_BY_DRIVER
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Open the Parent Handle for the child.
  //
  Status = gBS->OpenProtocol (
                  ControllerHandle,
                  DeviceGuid,
                  &Instance,
                  This->DriverBindingHandle,
                  ConSplitterVirtualHandle,
                  EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER
                  );
  if (EFI_ERROR (Status)) {
    goto Err;
  }

  //
  // Open InterfaceGuid on the virtul handle.
  //
  Status =  gBS->OpenProtocol (
                ControllerHandle,
                InterfaceGuid,
                Interface,
                This->DriverBindingHandle,
                ConSplitterVirtualHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );

  if (!EFI_ERROR (Status)) {
    return EFI_SUCCESS;
  }

  //
  // close the DeviceGuid on ConSplitter VirtualHandle.
  //
  gBS->CloseProtocol (
        ControllerHandle,
        DeviceGuid,
        This->DriverBindingHandle,
        ConSplitterVirtualHandle
        );

Err:
  //
  // close the DeviceGuid on ControllerHandle.
  //
  gBS->CloseProtocol (
        ControllerHandle,
        DeviceGuid,
        This->DriverBindingHandle,
        ControllerHandle
        );

  return Status;
}
```

In the ConSplitterStart function, it mainly returns an interface, which is the interface of SimpleInProtocol on each device. So this function mainly looks at

```
Status = gBS->OpenProtocol (
                ControllerHandle,
                InterfaceGuid,
                Interface,
                This->DriverBindingHandle,
                ConSplitterVirtualHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
```

Actually, OpenProtocol can be understood as follows: DriverBindingHandle is the driver and ConSplitterVirtualHandle. They open the SimpleInProtocol protocol interface on the ControllerHandle, which is the handle of the device. So the returned Interface is the independent protocol interface on the device.

EFI_OPEN_PROTOCOL_GET_PROTOCOL only records that ConSplitterVirtualHandle and ControllerHandle are the child and parent. For details about OpenProtocol, please refer to UEFI spec

Next, the driver will call ConSplitterTextInAddDevic(&mConIn, TextIn); to store this interface in an array . If there are multiple devices, then this array will store the instance of SimpleTextInputProtocol for each device.

In ConSplitterDriverEntry function, we found
gST->ConsoleInHandle = mConIn.VirtualHandle;
gST->ConIn = &mConIn.TextIn;
So the unified interface mentioned at the beginning of the article that we call to read the information of the input device in App is the interface provided by the mConIn.TextIn protocol. Then let's take a look at the implementation of these interfaces:
(ReadKeyStroke corresponds to ConSplitterTextInReadKeyStroke)
In ConSplitterTextInReadKeyStroke, we mainly look at ConSplitterTextInPrivateReadKeyStroke(Private, Key). In this function, we see a for loop that calls the interfaces provided by each device.

```
for (Index = 0; Index < Private->CurrentNumberOfConsoles; Index++){
  Status = Private->TextInList[Index]->ReadKeyStroke (
                                      Private->TextInList[Index],
                                      &CurrentKey
                                      );
}
```

So the above can be understood as this splitter In driver queries all input devices. If found, it will put the interface implemented by SimpleTextInputProtocol on the device into an array. When the application calls gST->ConIn->ReadKeyStroke, the driver will call the ReadKeyStroke function on all devices in a loop. This is a simple process, but our analysis is not over yet, because when reading user input, we must block the current program to wait for user input. So the general UEFI program is written to read input information like this:

```
EFI_STATUS
WaitForKeyStroke(
  IN OUT EFI_INPUT_KEY  *Key
  )
{
    EFI_STATUS  Status;
    UINTN       Index;

    while (TRUE) {

      Status = gST->ConIn->ReadKeyStroke(gST->ConIn, Key);
      if (!EFI_ERROR(Status)) {
        break;
      }

      if (Status != EFI_NOT_READY) {
        continue;
      }

      gBS->WaitForEvent(1, &gST->ConIn->WaitForKey, &Index);
    }

    return Status;
}
```

We can see that this function will call gBS->WaitForEvent, which will block the current program from executing until the event gST->ConIn->WaitForKey is signaled. So next, we can see when this event is signaled. Trace code We found that this event is registered in the ConSplitterTextInConstructor function.

```
  Status = gBS->CreateEvent (
                  EVT_NOTIFY_WAIT,
                  TPL_NOTIFY,
                  ConSplitterTextInWaitForKey,   ---callback
                  ConInPrivate,
                  &ConInPrivate->TextIn.WaitForKey ---Event
                  );
```

An event of type EVT_NOTIFY_WAIT is created here. This type of event means that when this event is WaitForEvent or CheckEvent, if the event is not signaled at that time, its callback will be called. I think there is a little difference between the two. When using WaitForEvent, if this event has no signal, the callback will be called until the event is signaled, and when using CheckEvent, the callback will be called once. If you are interested, you can take a look. WaitForEvent has a while loop inside and keeps calling CheckEvent.
Because we called gBS->WaitForEvent(1, &gST->ConIn->WaitForKey, &Index); in the application, the ConSplitterTextInWaitForKey function will run continuously.

```
VOID
EFIAPI
ConSplitterTextInWaitForKey (
  IN  EFI_EVENT                     Event,
```

```
  IN  VOID                          *Context
  )
{
  EFI_STATUS                    Status;
  TEXT_IN_SPLITTER_PRIVATE_DATA *Private;
  UINTN                         Index;

  Private = (TEXT_IN_SPLITTER_PRIVATE_DATA *) Context;

  if (Private->KeyEventSignalState) {
    //
    // If KeyEventSignalState is flagged before, and not cleared by Reset() or ReadKeyStroke()
    //
    gBS->SignalEvent (Event);
    return ;
  }

  //
  // If any physical console input device has key input, signal the event.
  //
for(Index = 0; Index < Private->CurrentNumberOfConsoles; Index++){
 Status=gBS->CheckEvent (Private->TextInList[Index]->WaitForKey);
    if (!EFI_ERROR (Status)) {
      gBS->SignalEvent (Event);
      Private->KeyEventSignalState = TRUE;
     }
   }
 }
```

We can see that there is a for loop inside it, which checks the waitforkey events of the input devices recorded in the array. If any of them have an event signaled, this function will call gBS->SignalEvent (Event) to signal its own event, so the callback will not be called any more. The events of each device are actually written by their corresponding drivers. When there is input information, their respective events will be signaled. When we call CheckEvent here, it will return True, and our callback will stop calling any more.

So here we have finished analyzing the ConIn function in Console Splitter. The basic principles of the other drivers are the same.

In the next article, we will practice it, otherwise it will be a bit of a whirlwind tour. In the next article, we will write a virtual input device and then bind the Console Splitter driver so that we can read the input information in the application.

> Reference to the content about Splitter in Beyond BIOS Second Editon