

第 3 章 Chapter 3

UEFI 工程模块文件

第 2 章讲述了如何编译 EDK2，下面开始介绍如何在 EDK2 环境下编程。编程之前首先要了解 EDK2 的两个概念：模块（Module）和包（Package）。

在 EDK2 根目录下，有很多以 *Pkg 命名的文件夹，每一个这样的文件夹称为一个 Package。当然，这种说法不准确，准确地说，“包”是一组模块及平台描述文件（.dsc 文件）、包声明文件（.dec 文件）组成的集合。模块是 UEFI 系统的一个特色。模块（可执行文件，即 .efi 文件）像插件一样可以动态地加载到 UEFI 内核中。对应到源文件，EDK2 中的每个工程模块由元数据文件（.inf 文件）和源文件（有些情况下也可以包含 .efi 文件）组成。

在 Linux 下编程，除了编写源代码之外，还要编写 Makefile 文件。在 Windows 下使用 VS（Visual Studio）时通常要建立工程文件和源文件。与之相似，在 EDK2 环境下，我们除了要编写源文件外，还要为工程编写元数据文件（.inf）。.inf 文件与 VS 的工程文件及 Linux 下的 Makefile 文件功能相似，用于自动编译源代码。包相当于 VS 中的项目，.dsc 文件则相当于 VS 项目的 .sln 文件；模块相当于 VS 项目中的工程，.inf 文件则相当于 VS 工程的 .proj 文件。

UEFI 模块类型众多，表 3-1 列出了 EDK2 中主要的模块类型。

表 3-1 UEFI 模块

模块类型	说 明
标准应用程序工程模块	在 DXE 阶段运行的应用程序（Shell 环境下也可以运行）
ShellAppMain 应用程序工程模块	Shell 环境下运行的应用程序
main 应用程序工程模块	Shell 环境下运行的应用程序，并且应用程序链接了 StdLib 库

(续)

模块类型	说 明
UEFI 驱动模块	符合 UEFI 驱动模型的驱动, 仅在 BS 期间有效
库模块	作为静态库被其他模块调用
DXE 驱动模块	DXE 环境下运行的驱动, 此类驱动不遵循 UEFI 驱动模型
DXE 运行时驱动模块	进入运行期仍然有效的驱动
DXE SAL 驱动模块	仅对安腾 CPU 有效的一种驱动
DXE SMM 驱动模块	系统管理模式驱动, 模块被加载到系统管理内存区。系统进入运行期该驱动仍然有效
PEIM 模块	PEI 阶段的模块
SEC 模块	固件的 SEC 阶段
PEI_CORE 模块	固件的 PEI 阶段
DXE_CORE 模块	固件的 DXE 阶段

本章主要讲述 UEFI 编程常用的几种模块, 包括 3 种应用程序工程模块、UEFI 驱动模块和库模块。

3.1 标准应用程序工程模块

标准应用程序工程模块是其他应用程序工程模块的基础, 也是 UEFI 中常见的一种应用程序工程模块。每个工程模块由两部分组成: 工程文件和源文件, 标准应用程序工程模块也不例外。源文件包括 C/C++ 文件、.asm 汇编文件, 也可以包括 .uni (字符串资源文件) 和 .vfr (窗体资源文件) 等资源文件。下面以一个简单的标准应用程序工程模块为例来介绍它的格式。

一个简单的标准应用程序工程模块包含一个 C 程序源文件 (本例中为 Main.c) 以及一个工程文件 (Main.inf)。该示例程序在 inf\UefiMain 目录下。

3.1.1 入口函数

示例 3-1 就是这个简单模块的源程序, 它仅有一个函数 UefiMain。UefiMain 就是这个模块的入口函数, 其功能是向标准输出设备输出字符串 “HelloWorld”。

【示例 3-1】简单的标准应用程序。

```
#include <Uefi.h>
EFI_STATUS UefiMain(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
{
    SystemTable->ConOut-> OutputString(SystemTable->ConOut, L"HelloWorld\n");
    return EFI_SUCCESS;
}
```

一般来说,标准应用程序至少要包含以下两个部分。

1) **头文件**:所有的 UEFI 程序都要包含头文件 Uefi.h。Uefi.h 定义了 UEFI 基本数据类型及核心数据结构。

2) **入口函数**:UEFI 标准应用程序的入口函数通常是 UefiMain。之所以说通常是 UefiMain 而不是说必须是 UefiMain,是因为入口函数可有开发者指定。UefiMain 只是一个约定俗成的函数名。入口函数由工程文件 UefiMain.inf 指定。虽然入口函数的函数名可以变化,但其函数签名(即返回值类型和参数列表类型)不能变化。

①入口函数的返回值类型是 EFI_STATUS。

❑ 在 UEFI 程序中基本所有的返回值类型都是 EFI_STATUS。它本质上是无符号长整数。

❑ 最高位为 1 时其值为错误代码,为 0 时表示非错误值。通过宏 EFI_ERROR(Status) 可以判断返回值 Status 是否为错误码。若 Status 为错误码,EFI_ERROR(Status) 返回真,否则返回假。

❑ EFI_SUCCESS 为预定义常量,其值为 0,表示没有错误的状态值或返回值。

②入口函数的参数 ImageHandle 和 SystemTable。

❑ .efi 文件(UEFI 应用程序或 UEFI 驱动程序)加载到内存后生成的对象称为 Image(映像)。ImageHandle 是 Image 对象的句柄,作为模块入口函数参数,它表示模块自身加载到内存后生成的 Image 对象。

❑ SystemTable 是程序同 UEFI 内核交互的桥梁,通过它可以获得 UEFI 提供的各种服务,如启动(BT)服务和运行时(RT)服务。SystemTable 是 UEFI 内核中的一个全局结构体。

向标准输出设备打印字符串是通过 SystemTable 的 ConOut 提供的服务完成的。ConOut 是 EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL 的一个实例。而 EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL 的主要功能是控制字符输出设备。向输出设备打印字符串是通过 ConOut 提供的 OutputString 服务完成的。该服务(函数)的第一个参数是 This 指针,指向 EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL 实例(此处为 ConOut)本身;第二个参数是 Unicode 字符串。关于 Protocol 和 This 指针将在第 4 章详细介绍。简而言之,这条打印语句的意义就是通过 SystemTable → ConOut → OutputString 服务将字符串 L “HelloWorld” 打印到 SystemTable → ConOut 所控制的字符输出设备。

3.1.2 工程文件

要想编译 Main.c,还需要编写 .inf (Module Information File) 文件。.inf 文件是模块的工程文件,其作用相当于 Makefile 文件或 Visual Studio 的 .proj 文件,用于指导 EDK2 编译工具自动编译模块。



注意 在工程文件中，字符 # 后面的内容为注释。

工程文件分为很多个块，每个块以 “[块名]” 开头，“[块名]” 必须单独占一行。有些块是所有工程文件都必需的块，这些块包括 [Defines]、[Sources]、[Packages] 和 [LibraryClasses]，见表 3-2。其他的块并不是每个模块都一定要编写的块，仅在用到的时候需要编写这些块，表 3-3 列出了一些非必需块。

表 3-2 工程文件必需块

必需块	块描述
[Defines]	定义本模块的属性变量及其他变量，这些变量可在工程文件其他块中引用
[Sources]	列出本模块的所有源文件及资源文件
[Packages]	列出本模块引用到的所有包的包声明文件。可能引用到的资源包括头文件、GUID、Protocol 等，这些资源都声明在包的包声明文件 .dec 中
[LibraryClasses]	列出本模块要链接的库模块

表 3-3 工程文件非必需块

非必需块	块描述
[Protocols]	列出本模块用到的 Protocol
[Guids]	列出本模块用到的 GUID
[BuildOptions]	指定编译和链接选项
[Pcd]	Pcd 全称为平台配置数据库（Platform Configuration Database）。[Pcd] 用于列出本模块用到的 Pcd 变量，这些 Pcd 变量可被整个 UEFI 系统访问
[PcdEx]	用于列出本模块用到的 Pcd 变量，这些 Pcd 变量可被整个 UEFI 系统访问
[FixedPcd]	用于列出本模块用到的 Pcd 编译期常量
[FeaturePcd]	用于列出本模块用到的 Pcd 常量
[PatchPcd]	列出的 Pcd 变量仅本模块可用

下面以我们编写的简单标准应用程序工程模块为例分别讲述几个必需块及其他常用块的使用法。

1. [Defines] 块

[Defines] 块用于定义模块的属性和其他变量，块内定义的变量可被其他块引用。

(1) 属性定义语法

属性名 = 属性值

(2) 属性

块内必须定义的属性包括：

- ❑ **INF_VERSION** : INF 标准的版本号。EDK2 的 build 会检查 INF_VERSION 的值并根据这个值解释 .inf 文件。最新的 INF 标准版本号为 0x00010016, 前半部分为主版本号, 后半部分为次版本号。通常将 INF_VERSION 设置为 0x00010005 即可。
- ❑ **BASE_NAME** : 模块名字字符串, 不能包含空格。它通常也是输出文件的名称。例如, Base_Name 为 UefiMain, 则最终生成的文件为 UefiMain.efi。
- ❑ **FILE_GUID** : 每个工程文件必须有一个 8-4-4-4-12 格式的 GUID, 用于生成固件。例如, FILE_GUID = 6987936E-ED34-ffdb-AE97-1FA5E4ED2117。
- ❑ **VERSION_STRING**: 模块的版本号字符串。例如, 可以设置为 VERSION_STRING=1.0。
- ❑ **MODULE_TYPE** : 定义模块的模块类型, 可以是 SEC、PEI_CORE、PEIM、DXE_CORE、DXE_SAL_DRIVER、DXE_SMM_DRIVER、UEF_DRIVER、DXE_DRIVER、DXE_RUNTIME_DRIVER、UEFI_APPLICATION、BASE 中的一个。对标准应用程序工程模块来说, MODULE_TYPE 的值为 UEFI_APPLICATION。
- ❑ **ENTRY_POINT** : 定义模块的入口函数。在上文中我们提到 UefiMain 是模块的入口函数, 就是在此处通过设置 ENTRY_POINT 的值为 UefiMain 得到的。

示例 3-2 是标准应用程序工程模块示例工程文件中完整的 [Defines] 块。

【示例 3-2】 标准应用程序工程模块工程文件的 [Defines]。

```
[Defines]
INF_VERSION = 0x00010005
BASE_NAME = UefiMain
FILE_GUID = 6987936E-ED34-ffdb-AE97-1FA5E4ED2117
MODULE_TYPE = UEFI_APPLICATION
VERSION_STRING = 1.0
ENTRY_POINT = UefiMain
```

2. [Sources] 块

[Sources] 用于列出模块的所有源文件和资源文件。

(1) 语法

块内每一行表示一个文件, 文件使用相对路径, 根路径是工程文件所在的目录。作为示例的标准应用程序工程模块仅含有一个源文件。[Sources] 块如下所示:

```
[Sources.$(Arch)]
    UefiMain.c
```

(2) 体系结构相关块

\$(Arch) 是可选项, 可以是 IA32、X64、IPF、EBC、ARM 中的一个, 表示本块适用的体系结构。[Sources] 块适用于任何体系结构。例如, 编译 32 位模块时 (即在 build 命令选项中使用 -a IA32 选项), 工程将包含 [Sources] 和 [Sources.IA32] 中的源文件; 编译 64 位

模块时，工程将包含 [Sources] 和 [Sources.X64] 中的源文件。

(3) 示例

示例 3-3 包含了三个块：[Sources]、[Sources.IA32] 和 [Sources.X64]。在编译 32 位模块时，模块包含 [Sources.IA32] 中的文件 Cpu32.c 和 [Sources] 中的 Common.c；在编译 64 位模块时，模块包含文件 [Sources.X64] 中的 Cpu64.c 和 [Sources] 中的 Common.c。

【示例 3-3】 标准应用程序工程模块的 [Sources]。

```
[Sources]
    Common.c
[Sources.IA32]
    Cpu32.c
[Sources.X64]
    Cpu64.c
```

(4) 编译工具链相关的源文件

有时文件名后面会有一个该号符号，该符号后面会跟工具链名字，如示例 3-4 所示。

【示例 3-4】 工具链相关的源文件。

```
[Sources]
    TimerWin.c | MSFT
    TimerLinux.c | GCC
```

这表示 TimerWin.c 仅在使用 Visual Studio 编译器时有效，TimerLinux.C 仅在使用 GCC 编译器时有效。除了 MSFT 和 GCC 外，EDK2 还定义了 INTEL 和 RVCT 两种工具链。INTEL 工具链是 ICC 编译器或 Intel EFI 字节码编译器；RVCT 是 ARM 编译器。

3. [Packages] 块

[Packages] 列出本模块引用到的所有包的包声明文件（.dec 文件）。

(1) 语法

[Packages] 块内每一行列出一个文件，文件使用相对路径，相对路径的根路径为 EDK2 的根目录。若 [Sources] 块内列出了源文件，则在 [Packages] 块必须列出 MdePkg/MdePkg.dec，并将其放在本块的首行。

(2) 示例

在本节介绍的这个简单标准应用程序工程模块示例中，UefiMain.c 仅仅引用了 MdePkg 中的头文件 Uefi.h，因而 [Packages] 仅列出 MdePkg/MdePkg.dec 即可，如示例 3-5 所示。

【示例 3-5】 工程文件的 [Packages] 块。

```
[Packages]
    MdePkg/MdePkg.dec
```


4. [LibraryClasses]

[LibraryClasses] 块列出本模块要链接的库模块。

(1) 语法

块内每一行声明一个要链接的库（库定义在包的 .dsc 文件中，定义方法将在下文讲述）。

语法如下：

```
[LibraryClasses]
    库名称
```

(2) 常用库

应用程序工程模块必须链接 UefiApplicationEntryPoint 库；驱动模块必须链接 UefiDriverEntryPoint 库。

(3) 示例

本示例中，UefiMain.c 文件的 UefiMain 函数没有使用其他库函数，因而在 [LibraryClasses] 块列出 UefiApplicationEntryPoint 即可，如下所示：

```
[LibraryClasses]
    UefiApplicationEntryPoint
```

5. [Protocols] 块

[Protocols] 列出模块中使用的 Protocol，严格来说，列出的是 Protocol 对应的 GUID。如果模块未使用任何 Protocol，则此块为空。

(1) 语法

块内每一行声明一个在本模块中引用的 Protocol。语法如下：

```
[LibraryClasses]
    Protocol 的 GUID
```

(2) 示例

如果在程序中使用了某个 Protocol 的 GUID，例如，源程序中使用了如下代码：

```
Status = gBS->LocateProtocol ( &gEfiHiiDatabaseProtocolGuid,
                                NULL, (VOID **) &HiiDatabase );
```

则在 [Protocols] 块必须列出 gEfiHiiDatabaseProtocolGuid，如示例 3-6 所示。

【示例 3-6】 工程文件的 [LibraryClasses]。

```
[LibraryClasses]
    gEfiHiiDatabaseProtocolGuid
```

6. [BuildOptions] 块

[BuildOptions] 指定本模块的编译和连接选项。

(1) 语法

```
[BuildOptions]
```

```
[ 编译器家族 ]: [$(Target)]_[TOOL_CHAIN_TAG]_[$(Arch)]_[CC|DLINK]_FLAGS[|=|==] 选项
```

说明如下:

- ❑ 编译器家族可以是 MSFT (Visual Studio 编译器家族)、INTEL (Intel 编译器家族)、GCC (GCC 编译器家族) 和 RVCT (ARM 编译器家族) 中的一个。
- ❑ Target 是 DEBUG、RELEASE 和 * 中的一个, * 为通配符, 表示对 DEBUG 和 RELEASE 都有效。
- ❑ TOOL_CHAIN_TAG 是编译器名字。编译器名字定义在 Conf\tools_def.txt 文件中, 预定义的编译器名字有 VS2003、VS2005、VS2008、VS2010、GCC44、GCC45、GCC46、CYGGCC、ICC 等, * 表示对指定编译器家族内的所有编译器都有效。
- ❑ Arch 是体系结构, 可以是 IA32、X64、IPF、EBC 或 ARM, * 表示对所有体系结构都有效。
- ❑ CC 表示编译选项。DLINK 表示连接选项。
- ❑ = 表示选项附加到默认选项后面。== 表示仅使用所定义的选项, 弃用默认选项。
- ❑ = 或 == 号后面是编译选项或连接选项。

(2) 示例

示例 3-7 表示使用 Visual Studio 编译器编译时添加 /wd4804 编译选项, 连接时添加 /BASE:0 选项。

【示例 3-7】 使用 “=” 的 [BuildOptions]。

```
[BuildOptions]
```

```
MSFT:*_*_*_CC_FLAGS = /wd4804
```

```
MSFT:*_*_*_DLINK_FLAGS = /BASE:0
```

示例 3-8 表示使用 VS2010 编译 32 位 DEBUG 版本时仅使用指定的编译选项, 忽略所有默认的编译选项。

【示例 3-8】 使用 “==” 的 [BuildOptions]。

```
[BuildOptions]
```

```
MSFT:DEBUG_VS2010_IA32_CC_FLAGS == /nologo /c /WX /GS- /W4 /Gs32768 /D  
UNICODE /Olib2 /GL /Ehs-c- /GR- /GF /Gy /Zi /Gm /D EFI_SPECIFICATION_VERSION=  
0x0002000A /D TIANO_RELEASE_VERSION=0x00080006 /FAs /Oi-
```

最后将所有的块放在一起, 组成完整的标准应用程序工程文件, 如示例 3-9 所示。

【示例 3-9】 标准应用程序 HelloWorld 的完整工程文件。

```
[Defines]
```

```
INF_VERSION = 0x00010005
```



```
BASE_NAME = UefiMain
FILE_GUID = 6987936E-ED34-ffdb-AE97-1FA5E4ED2117
MODULE_TYPE = UEFI_APPLICATION
VERSION_STRING = 1.0
ENTRY_POINT = UefiMain
[Sources]
    main.c
[Packages]
    MdePkg/MdePkg.dec
[LibraryClasses]
    UefiApplicationEntryPoint
    UefiLib
```

3.1.3 编译和运行

源文件和工程文件都已经编写完成，下面编译和运行这个标准应用程序工程模块。将 UefiMain.inf 添加到 Nt32Pkg.dsc 或 UnixPkg.dsc 的 [Components] 部分，例如添加下面一行代码（uefi 目录在 EDK2 下）：

```
[Components]
uefi/book/inf/UefiMain.inf
```

然后就可以使用 BaseTools 下的 build 工具进行编译了。


Windows 下执行如下命令进行编译：

```
C:\EDK2>edksetup.bat --nt32
C:\EDK2>build -p Nt32PkgNt32Pkg.dsc -a IA32
```

Linux 下执行如下命令进行编译：

```
$>source ./edksetup.sh BaseTools
$>build -p UnixPkg/UnixPkg.dsc -a IA32
```

在 UEFI 模拟器中执行 UefiMain 命令，输出如图 3-1 所示。



```
Shell> fs0:
FS0:\> UefiMain.efi
HelloWorld
FS0:\> _
```

图 3-1 标准应用程序 UefiMain.efi 的输出

3.1.4 标准应用程序的加载过程

下面深入分析一下应用程序的加载和调用过程。开始介绍之前，还要了解一下应用程序是如何被编译成 .efi 文件的，整个过程分为以下三步。

1) UefiMain.c 首先被编译成目标文件 UefiMain.obj。

2) 连接器将目标文件 UefiMain.obj 和其他库连接成 UefiMain.dll。

3) GenFw 工具将 UefiMain.dll 转换成 UefiMain.efi。

整个过程由 build 命令自动完成。第 2) 和 3) 步的命令如图 3-2 所示。

```
link.exe /OUT:d:\edk2\Build\...\DEBUG\UefiMain.dll /NOLOGO /NODEFAULTLIB
/IGNORE:4001 /OPT:REF /OPT:ICF=10 /MAP /ALIGN:32 /SECTION:.xdata,D
/SECTION:.pdata,D /MACHINE:X86 /LTCG /DLL /ENTRY:_ModuleEntryPoint
/SUBSYSTEM:EFI_BOOT_SERVICE_DRIVER /SAFESEH:NO /BASE:0 /DRIVER/DEBUG
/EXPORT:InitializeDriver=_ModuleEntryPoint /BASE:0x10000 /ALIGN:4096
/FILEALIGN:4096 /SUBSYSTEM:CONSOLE @d:\edk2\Build\...\OUTPUT\static_
library_files.lst
"GenFw" -e UEFI_APPLICATION -o d:\edk2\Build\...\DEBUG\UefiMain.efi d:\edk2\Build\
...\DEBUG\UefiMain.dll
```

图 3-2 标准应用程序连接和 GenFw 过程

说明：连接器在生成 UefiMain.dll 时使用了 /dll/entry:_ModuleEntryPoint。efi 是遵循 PE32 格式的二进制文件，_ModuleEntryPoint 便是这个二进制文件的入口函数。

3.1.1 节讲到模块的入口函数是 UefiMain，_ModuleEntryPoint 与 UefiMain 是什么关系呢？让我们带着这个疑问来看应用程序的加载过程。

1. 将 UefiMain.efi 文件加载到内存

首先来看 UefiMain.efi 文件是如何加载到内存的。当在 Shell 中执行 UefiMain.efi 时，Shell 首先用 gBS->LoadImage() 将 UefiMain.efi 文件加载到内存生成 Image 对象，然后调用 gBS->StartImage(Image) 启动这个 Image 对象。具体加载过程如代码清单 3-1 所示。

代码清单 3-1 应用程序的加载

```
//@file ShellPkg\Application\Shell\ShellProtocol.c
EFI_STATUS EFIAPI InternalShellExecuteDevicePath(
    IN CONST EFI_HANDLE*ParentImageHandle,
    IN CONST EFI_DEVICE_PATH_PROTOCOL *DevicePath,           // UefiMain.efi 的设备路径
    IN CONST CHAR16 *CommandLine OPTIONAL,                  // 应用程序所需的命令行参数
    IN CONST CHAR16 **Environment OPTIONAL,                  // UEFI 环境变量
    OUT EFI_STATUS *StatusCode OPTIONAL                       // 程序 UefiMain.efi 的返回值
)
{
    EFI_STATUS          Status;
    EFI_HANDLE          NewHandle;
    EFI_LOADED_IMAGE_PROTOCOL *LoadedImage;
    LIST_ENTRY          OrigEnvs;
    EFI_SHELL_PARAMETERS_PROTOCOL ShellParamsProtocol;
    ...
}
```

// 第一步：将 UefiMain.efi 文件加载到内存，生成 Image 对象，NewHandle 是这个对象的句柄

```
Status = gBS->LoadImage(  
    FALSE,  
    *ParentImageHandle,  
    (EFI_DEVICE_PATH_PROTOCOL*)DevicePath,  
    NULL,  
    0,  
    &NewHandle);  
  
if (EFI_ERROR(Status)) {  
    if (NewHandle != NULL) {  
        gBS->UnloadImage(NewHandle);  
    }  
    return (Status);  
}  
  
// 第二步：取得命令行参数，并将命令行参数交给 UefiMain.efi 的 Image 对象，即 NewHandle  
Status = gBS->OpenProtocol(  
    NewHandle,  
    &gEfiLoadedImageProtocolGuid,  
    (VOID*)&LoadedImage,  
    gImageHandle,  
    NULL,  
    EFI_OPEN_PROTOCOL_GET_PROTOCOL);  
  
if (!EFI_ERROR(Status)) {  
    ASSERT(LoadedImage->LoadOptionsSize == 0);  
    if (CommandLine != NULL) {  
        LoadedImage->LoadOptionsSize = (UINT32)StrSize(CommandLine);  
        LoadedImage->LoadOptions = (VOID*)CommandLine;  
    }  
}  
...  
  
// 第三步：启动所加载的 Image  
if (!EFI_ERROR(Status)) {  
    if (StatusCode != NULL) {  
        *StatusCode = gBS->StartImage(NewHandle, NULL, NULL);  
    } else {  
        Status = gBS->StartImage(NewHandle, NULL, NULL);  
    }  
}  
...  
  
// 退出应用程序后清理资源  
}
```

加载应用程序中最重要的一步，也是我们最关心的部分，就是 `gBS->StartImage(NewHandle, NULL, NULL)`。`StartImage` 的主要作用是找出可执行程序映像（Image）的入口函数并执行找到的入口函数。`gBS->StartImage` 是个函数指针，它实际指向 `CoreStartImage` 函数。

2. 进入映像的入口函数

CoreStartImage 的主要作用是调用映像的入口函数。下面来看 CoreStartImage 是如何做的, 具体如代码清单 3-2 所示。

代码清单 3-2 启动应用程序

```
//@file MdeModulePkg\Core\Dxe\Image\Image.c
EFI_STATUS EFIAPI CoreStartImage (IN EFI_HANDLE ImageHandle,
    OUT UINTN *ExitDataSize, OUT CHAR16 **ExitData OPTIONAL)
{
    EFI_STATUS Status;
    LOADED_IMAGE_PRIVATE_DATA *Image;
    LOADED_IMAGE_PRIVATE_DATA *LastImage;
    UINT64 HandleDatabaseKey;
    UINTN SetJumpFlag;
    UINT64 Tick;
    EFI_HANDLE Handle;

    // 设置 LongJump, 用于退出此程序
    Image->JumpBuffer = AllocatePool (sizeof (BASE_LIBRARY_JUMP_BUFFER) +
        BASE_LIBRARY_JUMP_BUFFER_ALIGNMENT);
    if (Image->JumpBuffer == NULL) {
        return EFI_OUT_OF_RESOURCES;
    }
    Image->JumpContext = ALIGN_POINTER (Image->JumpBuffer, BASE_LIBRARY_JUMP_
        BUFFER_ALIGNMENT);
    SetJumpFlag = SetJump (Image->JumpContext);
    // 首次调用 SetJump() 返回 0。通过 LongJump (Image->JumpContext) 跳转到此处时返回非零值
    if (SetJumpFlag == 0) {
        // 调用 Image 的入口函数
        Image->Started = TRUE;
        Image->Status = Image->EntryPoint (ImageHandle, Image->Info.SystemTable);
        // 设置 Image 执行后的状态, 然后通过 LongJump 跳到应用程序退出点
        CoreExit (ImageHandle, Image->Status, 0, NULL);
    }
    // 此处是应用程序退出点
    // 程序通过 LongJump 跳转到此处, 然后根据 Image->Status 进行错误处理
    ...
}
```

在 gBS->StartImage 中, SetJump/LongJump 为应用程序的执行提供了一种错误处理机制, 执行流程如图 3-3 所示。

gBS->StartImage 的核心是 Image->EntryPoint(…), 它就是程序映像的入口函数, 对应用程序来说, 就是 _ModuleEntryPoint 函数。进入 _ModuleEntryPoint 后, 控制权才转交给应用程序 (此处就是我们的 UefiMain.efi)。代码清单 3-3 是 _ModuleEntryPoint 的代码。

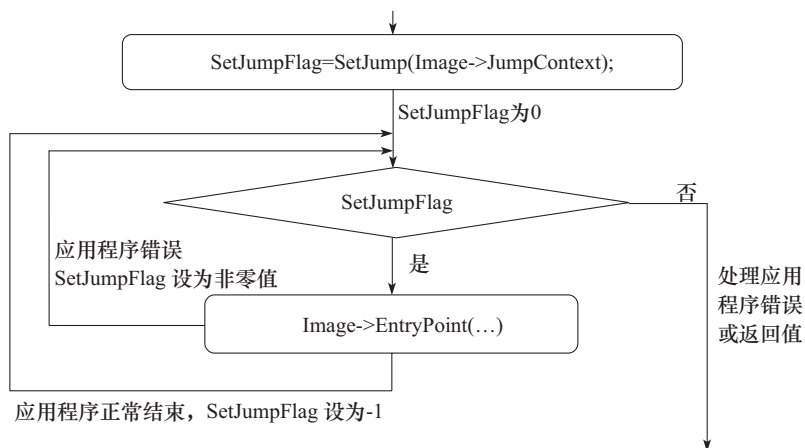


图 3-3 gBS->StartImage 执行流程

代码清单 3-3 程序映像的入口函数 _ModuleEntryPoint

```
//@file MdePkg\UefiApplicationEntryPoint\ApplicationEntryPoint.c
EFI_STATUS EFIAPI _ModuleEntryPoint ( IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable )
{
    EFI_STATUS Status;
    if ( _gUefiDriverRevision != 0 ) {
        // 确保系统平台的 UEFI 版本号大于或等于 ImageHandle 的 UEFI 版本号
        if ( SystemTable->Hdr.Revision < _gUefiDriverRevision ) {
            return EFI_INCOMPATIBLE_VERSION;
        }
    }
    // 所有将被使用的库的构造函数
    ProcessLibraryConstructorList (ImageHandle, SystemTable);
    // 调用 Image 的入口函数
    Status = ProcessModuleEntryPointList (ImageHandle, SystemTable);
    // 所有库的析构函数
    ProcessLibraryDestructorList (ImageHandle, SystemTable);
    return Status;
}
```

_ModuleEntryPoint 主要处理以下三个事情。

- 1) 初始化：在初始化函数 ProcessLibraryConstructorList 中会调用一系列的构造函数。
- 2) 调用本模块的入口函数：在 ProcessModuleEntryPointList 中会调用应用程序工程模块真正的入口函数（即我们在 .inf 文件中定义的入口函数 UefiMain）。
- 3) 析构：在析构函数 ProcessLibraryDestructorList 中会调用一系列析构函数。

那么这三个 Process* 函数是在哪里定义的呢？在命令行执行 build 命令的时候，build 命令会解析模块的工程文件（即 .inf 文件），然后生成 AutoGen.h 和 AutoGen.c，这三个函数便

是 AutoGen.c 中的一部分。一般而言,在 .inf 文件的 [LibraryClasses] 段声明了某个库后,如果这个库有构造函数,AutoGen 便会在 ProcessLibraryConstructorList 中加入这个库的构造函数。另外,ProcessLibraryConstructorList 还会加入启动服务和运行时服务的构造函数。代码清单 3-4 是标准应用程序工程模块 HelloWorld 的 ProcessLibraryConstructorList 函数。

代码清单 3-4 标准应用程序工程模块 HelloWorld 的 ProcessLibraryConstructorList 函数

```
VOID EFIAPI ProcessLibraryConstructorList (IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable)
{
    EFI_STATUS Status;
    // 初始化全局变量 gBS、gST 和 gImageHandle
    Status = UefiBootServicesTableLibConstructor (ImageHandle, SystemTable);
    ASSERT_EFI_ERROR (Status);
    // 初始化全局变量 gRT
    Status = UefiRuntimeServicesTableLibConstructor (ImageHandle, SystemTable);
    ASSERT_EFI_ERROR (Status);
    // 初始化 UefiLib, Print 函数就是在 UefiLib 中实现的
    Status = UefiLibConstructor (ImageHandle, SystemTable);
    ASSERT_EFI_ERROR (Status);
}
```

gBS 指向启动服务表, gST 指向系统表 (System Table), gImageHandle 指向正在执行的驱动或应用程序, gRT 指向运行时服务表, 这几个全局变量在开发应用程序和驱动时会经常用到。使用 gBS、gST、gImageHandle 前需加入 #include<include/UefiBootServicesTableLib.h>。使用 gRT 之前需加入 #include<include/UefiRuntimeServicesTableLib.h>

与构造函数相似, AutoGen 会在析构函数中调用相应 Library 的析构函数。代码清单 3-5 是 Hello World 标准应用程序工程模块的析构函数。Hello World 模块的析构函数 ProcessLibraryDestructorList 函数为空, 因为 UefiBootServicesTableLib、UefiRuntimeServicesTableLib、UefiLib 这三个 Library 都没有析构函数。

代码清单 3-5 标准应用程序工程模块 HelloWorld 的析构函数 ProcessLibraryDestructorList

```
VOID EFIAPI ProcessLibraryDestructorList (IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable )
{
}
```

3. 进入模块入口函数

在 ProcessModuleEntryPointList 中, 调用了应用程序工程模块的真正入口函数 UefiMain, 如代码清单 3-6 所示。

代码清单 3-6 标准应用程序工程模块 HelloWorld 的 ProcessModuleEntryPointList 函数

```
EFI_STATUS EFIAPI ProcessModuleEntryPointList ( IN EFI_HANDLE ImageHandle,
IN EFI_SYSTEM_TABLE *SystemTable )
{
    return UefiMain (ImageHandle, SystemTable);
}
```

至此，我们已经了解了标准应用程序工程模块入口函数调用的整个过程，再来简单回顾一下整个过程：StartImage → _ModuleEntryPoint → ProcessModuleEntryPointList → UefiMain。

通过本节的学习，我们还了解了标准应用程序工程模块的组成、入口函数 UefiMain 的结构，以及 .inf 文件的结构。下面继续学习其他类型工程模块的编写方法。

3.2 其他类型工程模块

常用的工程模块除了标准应用程序工程模块外，还有 Shell 应用程序工程模块、使用 main 函数的应用程序工程模块、库模块和驱动模块，下面分别介绍这几种模块。

3.2.1 Shell 应用程序工程模块

从 3.1.4 节的讲述可以看出，标准应用程序处理命令行参数很不方便。但是，能在 Shell 中执行的命令（命令也是应用程序）通常都会带有命令行参数，为了方便开发者开发能在 Shell 环境下执行的应用程序，EDK2 提供了一种特殊的应用程序工程模块，这种模块以 INTN ShellAppMain(IN UINTN Argc, IN CHAR16**Argv) 作为入口函数。我们称这种模块为 Shell 应用程序工程模块。一个简单的示例如示例 3-10 所示。

【示例 3-10】 Shell 应用程序工程模块的入口函数。

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
INTN ShellAppMain (IN UINTN Argc, IN CHAR16 **Argv)
{
    gST -> ConOut-> OutputString(gST -> ConOut, L"HelloWorld\n");
    return 0;
}
```

因为在入口函数的参数中没有了 SystemTable，所以要通过全局变量 gST 使用系统表。入口函数 ShellAppMain 的第一个参数 Argc 是命令行参数个数，第二个参数 Argv 是命令行参数列表，Argv 列表中的每个参数都是 Unicode 字符串（CHAR16* 类型的字符串）。在 UEFI 中使用的字符串通常都是 Unicode 字符串。

在介绍标准应用程序工程模块（参见 3.1 节）时讲过，每一种模块都由两部分组成：工

程文件和源文件, Shell 应用程序工程模块也不例外。上文介绍了源文件的格式, 下面介绍一下工程文件的编写方法。

1) 首先处理的是 [Defines] 块。在 [Defines] 块中, 将 MODULE_TPYE 设置为 UEFI_APPLICATION, 这一点与标准应用程序工程模块相同。然后将 ENTRY_POINT 设置为 ShellCEntryLib, 这里可以看出与标准应用程序工程模块的不同, 标准应用程序工程模块的 ENTRY_POINT 为 UefiMain, 同时在源程序中开发者需实现 UefiMain 函数, 也就是说, 开发者需提供由 ENTRY_POINT 指定的入口函数。在 Shell 应用程序工程模块中, ENTRY_POINT 必须为 ShellCEntryLib, 而在源程序中开发者必须提供 ShellAppMain。

2) 在 [Packages] 块中, 必须列出 MdePkg/MdePkg.dec 和 ShellPkg/ShellPkg.dec。

3) 在 [LibraryClasses] 块中, 必须列出 ShellCEntryLib, 通常还要列出 UefiBootServicesTableLib 和 UefiLib。

完整的工程文件如示例 3-11 所示。

【示例 3-11】 Shell 应用程序工程模块的工程文件。

```
[Defines]
INF_VERSION = 0x00010006
BASE_NAME = Main
FILE_GUID = 4ea97c46-7491-4dfd-b442-747010f3ce5f
MODULE_TYPE = UEFI_APPLICATION
VERSION_STRING = 0.1
ENTRY_POINT = ShellCEntryLib
[Sources]
Main.c
[Packages]
MdePkg/MdePkg.dec
ShellPkg/ShellPkg.dec
[LibraryClasses]
ShellCEntryLib
UefiBootServicesTableLib
UefiLib
```

UEFI 的入口函数 ShellCEntryLib 究竟做了哪些工作呢? 可以分析一下 ShellCEntryLib 的源码。该函数位于 ShellPkg/Library/UefiShellCEntryLib/UefiShellCEntryLib.c 中, 其源码如代码清单 3-7 所示。

代码清单 3-7 ShellCEntryLib 函数

```
// ShellCEntryLib 库提供的应用程序入口函数, 该函数最终会调用用户的入口函数 ShellAppMain
EFI_STATUS EFIAPI ShellCEntryLib (
    IN EFI_HANDLE ImageHandle,                // UEFI 应用程序的 ImageHandle
    IN EFI_SYSTEM_TABLE *SystemTable          // UEFI 系统的 EFI 系统表
)
```

```
{
    INTN ReturnFromMain;
    EFI_SHELL_PARAMETERS_PROTOCOL *EfiShellParametersProtocol;
    EFI_SHELL_INTERFACE *EfiShellInterface;
    EFI_STATUS Status;
    ReturnFromMain = -1;
    EfiShellParametersProtocol = NULL;
    EfiShellInterface = NULL;
    // 首先取得命令行参数
    Status = SystemTable->BootServices->OpenProtocol(ImageHandle,
        &gEfiShellParametersProtocolGuid,
        (VOID **)&EfiShellParametersProtocol,
        ImageHandle,
        NULL,
        EFI_OPEN_PROTOCOL_GET_PROTOCOL);
    if (!EFI_ERROR(Status)) {
        // 通过 Shell 2.0 接口获得命令行参数
        // 调用用户的入口函数
        ReturnFromMain = ShellAppMain ( EfiShellParametersProtocol->Argc,
            EfiShellParametersProtocol->Argv);
    } else {
        // 打开 Shell 2.0 Protocol 失败, 尝试 Shell 1.0
        Status = SystemTable->BootServices->OpenProtocol(ImageHandle,
            &gEfiShellInterfaceGuid,
            (VOID **)&EfiShellInterface,
            ImageHandle,
            NULL,
            EFI_OPEN_PROTOCOL_GET_PROTOCOL
        );
        if (!EFI_ERROR(Status)) {
            // 通过 Shell 1.0 接口获得命令行参数
            // 调用用户的入口函数
            ReturnFromMain = ShellAppMain ( EfiShellInterface->Argc, EfiShellInterface->Argv);
        } else {
            ASSERT(FALSE);
        }
    }
    if (ReturnFromMain == 0) {
        return (EFI_SUCCESS);
    } else {
        return (EFI_UNSUPPORTED);
    }
}
```

可以看出, ShellCEntryLib 函数的输入参数与我们前面用到的 UefiMain 函数的输入参数完全相同, 就是标准的 UEFI application 入口函数。在 ShellCEntryLib 中, 首先打开 EFI_SHELL_PARAMETERS_PROTOCOL, 通过 EFI_SHELL_PARAMETERS_PROTOCOL 可以获

得命令行参数个数 `Argc` 及命令行参数数组 `Argv`，然后调用 `ShellAppMain` 函数。代码清单 3-8 是 `EFI_SHELL_PARAMETERS_PROTOCOL` 数据结构，在可执行文件被 UEFI 通过 `Load Image Protocol` 载入 UEFI 系统时，会在可执行文件的 `Image Handle` 上安装 `Shell Protocol` 和 `Shell Parameter Protocol`。

代码清单 3-8 `EFI_SHELL_PARAMETERS_PROTOCOL` 数据结构

```
// @file ShellPkg\Include\Protocol\EfiShellParameters.h
typedef struct _EFI_SHELL_PARAMETERS_PROTOCOL {
    CHAR16 **Argv;           // 命令行参数数组，第一个参数是可执行文件的全路径
    UINTN Argc;              // Argv 数组的元素个数
    SHELL_FILE_HANDLE StdIn; // 标准输入句柄
    SHELL_FILE_HANDLE StdOut; // 标准输出句柄
    SHELL_FILE_HANDLE StdErr; // 标准错误句柄
} EFI_SHELL_PARAMETERS_PROTOCOL;
```

3.2.2 使用 `main` 函数的应用程序工程模块

对 C 语言程序员来说，最熟悉的程序莫过于 `main` 函数。EDK2 也提供了使用 `main` 函数的应用程序工程模块，通常在此类应用程序中都会使用 C 标准库（`StdLib`）中的函数。示例 3-12 是一个使用 `main` 函数的应用程序工程模块的简单示例。

【示例 3-12】使用 `main` 函数的应用程序工程模块的源文件。

```
#include <stdio.h>
int main (int argc, char **argv)
{
    printf("HelloWorld\n");
    return 0;
}
```

在工程文件中要进行如下设置。

- ❑ 在 [Defines] 块中，设置 `MODULE_TPYE` 为 `UEFI_APPLICATION`。
- ❑ 在 [Defines] 块中，设置 `ENTRY_POINT` 为 `ShellCEntryLib`。
- ❑ 在 [Packages] 块中，列出 `MdePkg/MdePkg.dec`、`ShellPkg/ShellPkg.dec` 和 `StdLib/StdLib.dec`。
- ❑ 在 [LibraryClasses] 块中，列出 `ShellCEntryLib`（提供 `ShellCEntryLib` 函数）、`LibC`（提供 `ShellAppMain` 函数）及 `LibStdio`（提供 `printf` 函数）库。
- ❑ 在 [Sources] 中，列出源文件 `main.c`。

回忆一下 3.2.1 节，`Shell` 应用程序工程模块使用了 `ShellCEntryLib`，然后我们自己实现了 `ShellAppMain` 函数作为程序的入口函数。

在使用 `main` 函数的应用程序工程模块中使用了 `StdLib`，而 `StdLib` 提供了 `ShellAppMain` 函数。开发者要实现 `int main(int Argc, char** Argv)` 作为程序的入口函数以供 `ShellAppMain`

调用。而真正的模块入口函数是 ShellCEntryLib，调用过程为 ShellCEntryLib->ShellAppMain->main。完整的工程文件如示例 3-13 所示。

【示例 3-13】 使用 main 函数的应用程序工程模块的工程文件。

```
[Defines]
  INF_VERSION = 0x00010006
  BASE_NAME = Main
  FILE_GUID = 4ea97c46-7491-4dfd-b442-747010f3ce5f
  MODULE_TYPE = UEFI_APPLICATION
  VERSION_STRING = 0.1
  ENTRY_POINT = ShellCEntryLib
[Sources]
  main.c
[Packages]
  MdePkg/MdePkg.dec
  ShellPkg/ShellPkg.dec
  StdLib/StdLib.dec
[LibraryClasses]
  LibC
  LibStdio
  ShellCEntryLib
```

还要再说明一点，如果用户的程序中用到了 printf(...) 等标准 C 的库函数，那么一定要使用此种类型的应用程序工程模块。ShellCEntryLib 函数中会调用 StdLib 的 ShellAppMain(...)，这个 ShellAppMain 函数会对 StdLib 进行初始化。StdLib 的初始化完成后才可以调用 StdLib 的函数。关于 StdLib 的使用将在后面章节详细介绍。

通常，使用 main 函数的应用程序工程模块在 AppPkg 环境下才能成功编译。首先将 main.inf 添加到 AppPkg\AppPkg.dsc 文件的 [Components]。

```
## @file AppPkg.dsc
[Components]
  uefi\book\infs\main\main.inf
```

然后可以用如下命令编译这个工程：

```
build -p AppPkg\AppPkg.dsc -m uefi\book\infs\main\main.inf
```

3.2.3 库模块

开发大型工程的时候经常会用到库，例如我们要开发视频解码程序，会用到 zlib 库。在库模块的工程文件中，需要设置 MODULE_TYPE 为 BASE；设置 LIBRARY_CLASS 为 library 的名字，例如 zlib。同时，不要设置 ENTRY_POINT。[Packages] 块列出库引用到的包，[LibraryClasses] 列出包所依赖的其他库。示例 3-14 是 zlib 库的工程文件。

【示例 3-14】 zlib 库的工程文件。

```
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = zlib
  FILE_GUID = 348aaa62-BFBD-4882-9ECE-C80BBbbb736
  VERSION_STRING = 1.0
  MODULE_TYPE = BASE
  LIBRARY_CLASS = zlib

[Sources]
  adler32.c
  # 此处不一一列举 zlib 中的源文件，感兴趣的读者可以参考本书附带的源码 (book\ffmpeg\zlib\zlib.inf)

[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  StdLib/StdLib.dec

[LibraryClasses]
  MemoryAllocationLib
  BaseLib
  UefiBootServicesTableLib
  BaseMemoryLib
  UefiLib
```

有些库仅能被某些特定的模块调用，编写这种库时需工程文件中声明库的适用范围，声明方法是在 [Defines] 块的 LIBRARY_CLASS 变量中定义，格式如下：

```
LIBRARY_CLASS = 库名字 | 适用模块类型 1 适用模块类型 2
```

例如，如果想使 zlib 库仅能被应用程序工程模块调用，那么 LIBRARY_CLASS 需设置为：

```
LIBRARY_CLASS = zlib | UEFI_APPLICATION
```

编写了库之后，要使库能被其他模块调用，还要在包的 .dsc 文件中声明该库，例如要使得 AppPkg 中的模块能调用 zlib 库，需将 zlib\zlib-1.2.6\zlib.inf（假设 zlib-1.2.6 在 EKD2 的根目录下）放到 AppPkg\AppPkg.dsc 文件 [LibraryClasses] 中，如下所示：

```
[LibraryClasses]
  zlib\zlib-1.2.6\zlib.inf
```

调用 zlib 时，在需要链接 zlib 的工程模块的工程文件 [LibraryClasses] 中添加 zlib 即可。

```
[LibraryClasses]
  zlib
```

如果库使用之前需要进行初始化，在库的工程文件需指定 CONSTRUCTOR 和 DESTRUCTOR，CONSTRUCTOR 函数会加入到 ProcessLibraryConstructorList 中，这个 CONSTRUCTOR 函数会在 ENTRY_POINT 之前执行；DESTRUCTOR 函数会加入到 ProcessLibraryDestructorList 中，这个 DESTRUCTOR 就会在 ENTRY_POINT 之后执行。

例如, 如果 zlib 库在被调用之前需在 InitializeLib() 中初始化, 程序结束之前需调用 LibDestructor() 清理 zlib 库占用的资源, 那么要在工程文件中做如下设置:

```
[Defines]
...
CONSTRUCTOR =InitializeLib
DESTRUCTOR  =LibDestructor
```

然后还需在库的源文件中提供这两个函数, 如示例 3-15 所示。

【示例 3-15】 zlib 库的构造函数和析构函数。

```
RETURN_STATUS EFI_API InitializeLib()
{
    EFI_STATUS Status;
    ...// 初始化库
    return Status;
}

RETURN_STATUS EFI_API LibDestructor ()
{
    EFI_STATUS Status;
    ...// 清理库所占资源
    return Status;
}
```

3.2.4 UEFI 驱动模块

在 UEFI 中, 驱动分为两类: 一类是符合 UEFI 驱动模型的驱动, 模块类型为 UEFI_DRIVER, 称为 UEFI 驱动; 另一类是不遵循 UEFI 驱动模型的驱动, 模块类型包括 DXE_DRIVER、DXE_SAL_DRIVER、DXE_SMM_DRIVER、DXE_RUNTIME_DRIVER, 称为 DXE 驱动。

驱动与应用程序的模块入口函数 (ENTRY_POINT) 类型一样, 函数原型如代码清单 3-9 所示。

代码清单 3-9 应用程序工程模块和驱动程序模块入口函数的函数原型

```
typedef EFI_STATUS API (*UEFI_ENTRYPOINT) (
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable);
```

驱动与应用程序的最大区别是驱动会常驻内存, 而应用程序执行完毕后就会从内存中清除。本书重点讲述第一类驱动。UEFI 驱动模型将在第 9 章详细讲述, 本节主要讲述 UEFI 驱动模块工程文件的格式。

❑ 在 [Defines] 块, 将 MODULE_TYPE 设置为 UEFI_DRIVER。其他宏变量如 INF_VERSION、BASE_NAME、FILE_GUID、VERSION_STRING 和 ENTRY_POINT, 相信读者已经明白其含义和设置方法, 在此不再赘述。

□ 在 [Sources] 块, 通常含有 ComponentName.c, 在此文件中定义了驱动的名字, 驱动安装后, 这个名字将显示给用户。

□ 在 [LibraryClasses] 块, 必须包含 UefiDriverEntryPoint。

代码清单 3-10 是 DiskIo 驱动的工程文件。

代码清单 3-10 DiskIo 驱动的工程文件

```
// @file MdeModulePkg/Universal/Disk/DiskIoDxe/DiskIoDxe.inf
[Defines]
  INF_VERSION = 0x00010005
  BASE_NAME = DiskIoDxe
  FILE_GUID = 6B38F7B4-AD98-40e9-9093-ACA2B5A253C4
  MODULE_TYPE = UEFI_DRIVER
  VERSION_STRING = 1.0
  ENTRY_POINT = InitializeDiskIo
[Sources]
  ComponentName.c
  DiskIo.h
  DiskIo.c
[Packages]
  MdePkg/MdePkg.dec
[LibraryClasses]
  UefiDriverEntryPoint
  UefiBootServicesTableLib
  MemoryAllocationLib
  BaseMemoryLib
  BaseLib
  UefiLib
  DebugLib
[Protocols]
  gEfiDiskIoProtocolGuid ## BY_START
  gEfiBlockIoProtocolGuid ## TO_START
```

3.2.5 模块工程文件小结

前面我们讲述了模块的工程文件 .inf, 现在我们知道, .inf 就像 Visual Studio 里的工程文件或者 Linux 里面的 Makefile 文件, 用于帮助我们组织和编译工程。 .inf 文件有 [Defines]、[Sources]、[Packages]、[LibraryClasses]、[Protocols]、[BuildOptions] 几个部分。

□ [Defines] 部分定义了模块类型 (MODULE_TYPE)、模块的名字 (BASE_NAME)、版本号 (VERSION_STRING)、入口函数 (ENTRY_POINT) 等。

□ [Sources] 部分定义了本模块包含的源文件或目标文件。

□ [Packages] 指明了要引用的包, 包中的头文件可以在库的源文件中引用。

□ [LibraryClasses] 列出了需要链接的库。

- ❑ [Protocols] 里列出了本模块用到的 Protocol。
- ❑ [BuildOptions] 列出了编译本模块中的源文件时用到的编译选项。

3.3 包及 .dsc、.dec、.fdf 文件

前面我们介绍了 .inf 文件，如果说 .inf 文件相当于 Visual studio 中的工程文件，.dsc (Platform Description File) 则相当于 Visual studio 中的 solution 文件。每个包包含一个 .dec (Package Declaration File) 文件、一个 .dsc 文件。如过一个包用于生成固件 Image 或 Option Rom Image，这个包还要包含 .fdf (Flash Description Files)，.fdf 用于生成固件 Image、Option Rom Image 或可启动 Image。

- ❑ build 命令用于编译包，它需要一个 .dsc 文件、一个 .dec 文件以及一个或多个 .inf 文件。
- ❑ GenFW 命令用于制作固件或 Option Rom Image，它需要一个 .dec 文件和一个 .fdf 文件。

图 3-4 展示了 EDK2 文件与 EDK2 工具链命令之间的关系。

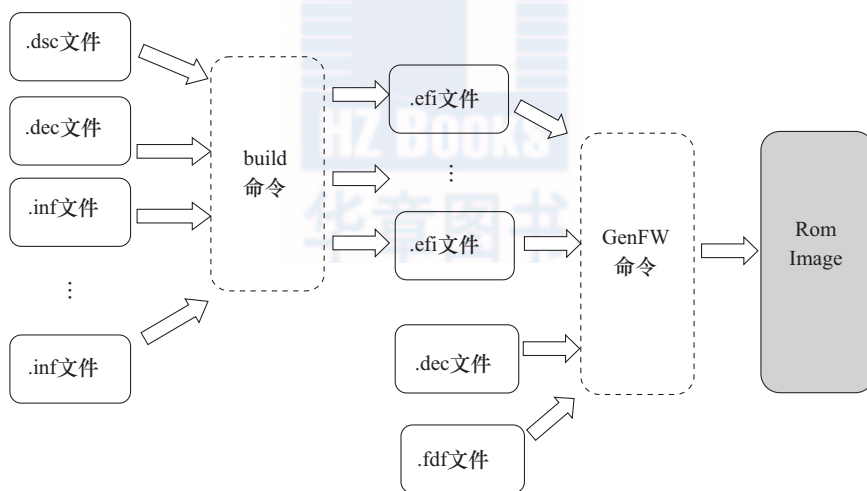


图 3-4 EDK2 文件与 EDK2 工具链关系图

下面讲述常用的两种文件：.dsc 与 .dec 文件。

3.3.1 .dsc 文件

.inf 用于编译一个模块，而 .dsc 文件用于编译一个 Package，它包含了 [Defines]、[LibraryClasses]、[Components] 几个必需部分以及 [PCD]、[BuildOptions] 等几个可选部分。

1. [Defines] 块

[Defines] 用于设置 build 相关的全局宏变量，这些变量可以被 .dsc 文件的其他模块引用。
[Defines] 必须是 .dsc 文件的第一个部分，格式如下。

```
[Defines]
宏变量名 = 值
DEFINE 宏变量名 = 值
EDK_GLOBAL 宏变量名 = 值
```

[Defines] 中通过 DEFINE 和 EDK_GLOBAL 定义的宏可以在 .dsc 文件和 .fdf 文件中通过 \$(宏变量名) 使用。表 3-4 列出了 .dsc 文件中 [Defines] 必须定义的宏变量。

表 3-4 .dsc 文件中 [Defines] 必须定义的宏变量

宏变量名	值类型	说 明
DSC_SPECIFICATION	数值	DSC Spec 1.22 对应的值为 0x00010006。目前（UEFI Spec 2.3/2.4）常用值为 0x00010005。DSC 必须保证向后兼容
PLATFORM_GUID	GUID	平台 GUID，每个 .dsc 文件必须有一个独一无二的 GUID
PLATFORM_VERSION	数值	.dsc 文件变化时，增加此数值
PLATFORM_NAME	标识符	标识符字符串中只能包含英文字符、数字、横线和下划线
SKUID_IDENTIFIER	标识符	该宏可以通过命令行在 build 时传入。可以是 Default。如果不是 Default，值必须是 [SkuIds] 中的一个
SUPPORTED_ARCHITECTURES	列表	通过“ ”分隔的列表，该 .dsc 所支持的平台的体系结构。例如 IA32 或者 IA32X64
BUILD_TARGETS	列表	通过“ ”分隔的列表，该 .dsc 所支持的编译目标，例如 BUILD 或者 BUILD RELEASE

表 3-5 是 .dsc 文件 [Defines] 可选宏变量。

表 3-5 .dsc 文件 [Defines] 可选宏变量

宏变量名	值类型	说 明
OUTPUT_DIRECTORY	路径	目标文件路径。可以是相对路径，也可以是绝对路径。默认为 \$(WORKSPACE)/Build/\$(PLATFORM_NAME)
FLASH_DEFINITION	文件名	FDF 文件。可以是文件名，也可以是带路径的文件名。如果仅仅是文件名，则该文件必须在 .dsc 所在的目录
BUILD_NUMBER	最多 4 个字符	用于 Makefile 文件
FIX_LOAD_TOP_MEMORY_ADDRESS	地址	驱动、应用程序在内存中的起始地址
TIME_STAMP_FILE	文件名	时间戳文件。可以是文件名，也可以是带路径的文件名。该文件包含了一个时间戳，所有编译过程中生成的文件使用该文件戳

(续)

宏变量名	值类型	说 明
DEFINE	MACRO= PATH Value	宏定义。定义的宏可以在 .dsc 文件中调用。例如， DEFINE DEBUG_ENABLE_OUTPUT = FALSE 在后续的 .dsc 文件中可以这样使用： !if \$(DEBUG_ENABLE_OUTPUT) !endif
EDK_GLOBAL	MACRO= PATH Value	仅用于 EDK 模块。EDK2 模块忽略该值
RFC_LANGUAGES	RFC 4646 语言代 码列表	RFC 4646 语言代码字符串，各个语言代码之间用分号 “;” 分隔。用于在 AutoGen 阶段处理 Unicode 字符串。字 符串必须用 " 包裹 "
ISO_LANGUAGES	ISO-639-2 语言代 码列表	ISO-639-2 语言代码字符串，语言代码之间无任何分 隔，每个语言代码为 3 字符。用于在 AutoGen 阶段处理 Unicode 字符串。字符串必须用 " 包裹 "
VPD_TOOL_GUID	GUID	在 AutoGen 阶段调用此 GUID 对应的 VPD 程序。VPD 程序定义在 ConfTools_def.txt 文件中，默认为 BPDG
PCD_INFO_GENERATION	布尔值	TRUE 表示在 PCD 数据库中生存 PCD 信息

代码清单 3-11 是 Nt32Pkg 中 .dsc 文件的 [Defines] 部分。

代码清单 3-11 Nt32Pkg 中 .dsc 文件的 [Defines] 块

[Defines]	
PLATFORM_NAME	= NT32
PLATFORM_GUID	= EB216561-961F-47EE-9EF9-CA426EF547C2
PLATFORM_VERSION	= 0.4
DSC_SPECIFICATION	= 0x00010005
OUTPUT_DIRECTORY	= Build/NT32
SUPPORTED_ARCHITECTURES	= IA32
BUILD_TARGETS	= DEBUG RELEASE
SKUID_IDENTIFIER	= DEFAULT
FLASH_DEFINITION	= Nt32Pkg/Nt32Pkg.fdf

2. [LibraryClasses] 块

在 [LibraryClasses] 块中定义了库的名字以及库 .inf 文件的路径。这些库可以被 [Components] 块内的模块引用。

(1) 语法

[LibraryClasses] 块语法如下：

```
[LibraryClasses.$(Arch).$(MODULE_TYPE)]  
LibraryName | Path/LibraryName.inf
```

或者

```
[LibraryClasses.$(Arch).$(MODULE_TYPE), LibraryClasses.$(Arch1).$(MODULE_TYPE1)]  
LibraryName | Path/LibraryName.inf
```

\$(Arch) 和 \$(MODULE_TYPE) 是可选项。逗号表示并列关系, 块内的库对 Library Classes. \$(Arch).\$(MODULE_TYPE) 和 LibraryClasses.\$(Arch1).\$(MODULE_TYPE1) 都有效。

通常, .dsc 文件中都有 [LibraryClasses] 区块, 表示块内定义的库对所有体系结构和所有类型的模块都有效。

\$(Arch) 表示体系结构, 可以是下列值之一: IA32、X64、IPF、EBC、ARM、common。common 表示对所有体系结构有效。

\$(MODULE_TYPE) 表示模块的类别, 块内列出的库只能供 \$(MODULE_TYPE) 类别的模块链接。\$(MODULE_TYPE) 可以是下列值: SEC、PEI_CORE、PEIM、DXE_CORE、DXE_SAL_DRIVER、BASE、DXE_SMM_DRIVER、DXE_DRIVER、DXE_RUNTIME_DRIVER、UEFI_DRIVER、UEFI_APPLICATION、USER_DEFINED。

(2) 示例

例如, 在 Nt32Pkg.dsc 中有:

```
[LibraryClasses.common.PEIM, LibraryClasses.common.PEI_CORE]  
HobLib|MdePkg/Library/PeiHobLib/PeiHobLib.inf
```

在 ShellPkg.dsc 中有:

```
[LibraryClasses.ARM]  
NUL|ArmPkg/Library/CompilerIntrinsicsLib/CompilerIntrinsicsLib.in
```

引用库是在 .inf 文件的 [LibraryClasses] 块中完成的。例如, 本章第一个示例程序的 .inf 文件中引用了 UefiApplicationEntryPoint 和 UefiLib, 如下所示:

```
[LibraryClasses]  
UefiApplicationEntryPoint  
UefiLib
```

3. [Components] 块

在该区块内定义的模块都会被 build 工具编译并生成 .efi 文件, 格式如下:

```
[Components.$(Arch)]  
Path\Exectuables.inf
```

或者

```
[Components.$(Arch)]  
Path\Exectuables.inf{  
<LibraryClasses> # 嵌套块  
LibraryName|Path/LibraryName.inf
```



```
<BuildOptions>    # 嵌套块
    # 子块中还可以包含 <Pcds*>
}
```

如果 Path 是相对路径, 则相对路径起始于 \$(WORKSPACE), \$(WORKSPACE) 通常是 EDK2 的根目录。在 Path 中可以使用通过 DEFINE 命令定义的宏。例如:

```
[Components]
    DEFINE MYSOURCE_PATH = D:/Source
    $(MYSOURCE_PATH)/Hello.inf # 相当于 D:/Source/Hello.inf
```

上述格式中的大括号内的内容仅对本模块有效, 例如, 示例 3-16 中 [Components] 声明的 DevicePathDxe.inf 会调用在 MdePkg/Library/UefiDevicePathLib/UefiDevicePathLib.inf 定义的 DevicePathLib 库, 其他模块会使用在全局 [LibraryClasses] 块中声明的 UefiDevicePathLib DevicePathProtocol.inf 对应的 DevicePathLib 库。

【示例 3-16】 .dsc 文件中的嵌套块。

```
[LibraryClasses]
DevicePathLib|MdePkg/Library/UefiDevicePathLibDevicePathProtocol/UefiDevicePath
    LibDevicePathProtocol.inf
[Components]
...
MdeModulePkg/Universal/DevicePathDxe/DevicePathDxe.inf {
    <LibraryClasses>
        DevicePathLib|MdePkg/Library/UefiDevicePathLib/UefiDevicePathLib.inf
    }
```

下面是在 MdeModulePkg.dsc 中的一个实例, 这个块内的模块对 IA32、X64 和 IPF 体系结构有效。

```
[Components.IA32, Components.X64, Components.IPF]
    MdeModulePkg/Universal/Network/UefiPxeBcDxe/UefiPxeBcDxe.inf
    MdeModulePkg/Universal/DebugSupportDxe/DebugSupportDxe.inf
    MdeModulePkg/Universal/EbcDxe/EbcDxe.inf
```

4. [BuildOptions] 块

[BuildOptions] 格式在前面的 .inf 文件已经介绍过, .dsc 文件的 [BuildOptions] 与 .inf 文件的 [BuildOptions] 格式大致相同, 区别在于 .dsc 文件的 [BuildOptions] 对 .dsc 文件内的所有模块有效。[BuildOptions] 格式如下:

```
[BuildOptions.$(Arch).$(CodeBase)]
    [ 编译器 ] : [$(Target)]_[Tool]_[$(Arch)]_[CC|DLINK]_FLAGS=
```

\$(Arch) 与 \$(CodeBase) 都是可选项。\$(CodeBase) 是 EDK 和 EDK2 之一。\$(Arch) 是体

系结构, 如 IA32、X64 等。

\$(Target) 是 DEBUG、RELEASE、* 中的一个。Tool 是编译工具的名字, 如 VS2012、GCC44 等。CC 表示编译选项, DLINK 表示连接选项。

代码清单 3-12 是 AppPkg.dsc 中的 [BuildOptions] 块内容。

代码清单 3-12 AppPkg.dsc 中的 [BuildOptions] 块

```
[BuildOptions]
!ifndef $(EMULATE)
    INTEL:*_*_*_CC_FLAGS      = /Qfreestanding /D UEFI_C_SOURCE
    MSFT:*_*_*_CC_FLAGS      = /X /Zc:wchar_t /D UEFI_C_SOURCE
    ...
!else
    INTEL:*_*_IA32_CC_FLAGS   = /Od /D UEFI_C_SOURCE
    MSFT:*_*_IA32_CC_FLAGS   = /Od /D UEFI_C_SOURCE
    ...
!endif
```

在 .dsc 文件中可以使用 ! 命令。!include 用于加载其他文件。!if、!ifdef、!ifndef、!else、!end 是条件处理语句。

最后简单介绍一下 [PCD] 块。[PCD] 块用于定义平台配置数据。其目的是在不改动 .inf 文件和源文件的情况下完成对平台的配置。例如在 UEFI 模拟器 Nt32Pkg 的 .dsc 文件 Nt32Pkg.dsc 中, 可以通过 PCD 的 PcdWinNtFileSystem 配置模拟器文件系统路径, 如下所示:

```
gEfiNt32PkgTokenSpaceGuid.PcdWinNtFileSystem|L"..!\..\..\..\EdkShellBinPkg\
Bin\Ia32\Apps"|VOID*|106
```

该项配置被竖线 “|” 分为 4 个部分。第一部分中 gEfiNt32PkgTokenSpaceGuid 是名字空间, PcdWinNtFileSystem 是变量名。第二部分是值。第三部分是变量类型。第四部分是变量数据的最大长度。

在源文件中可以通过 LibPcdGetPtr(_PCD_TOKEN_PcdWinNtFileSystem) 获得 gEfiNt32PkgTokenSpaceGuid.PcdWinNtFileSystem 定义的值。

3.3.2 .dec 文件

.dec 文件定义了公开的数据和接口, 供其他模块使用。它包含了必需区块 [Defines] 以及可选区块 [Includes]、[LibraryClasses]、[Guids]、[Protocols]、[Ppis] 和 [PCD] 几个部分。

1. [Defines] 块

[Defines] 区块用于提供 package 的名称、GUID、版本号等信息, 格式如下:

```
[Defines]
Name = Value
```

Name 可以是下面 4 个：DEC_SPECIFICATION、PACKAGE_NAME、PACKAGE_GUID、PACKAGE_VERSION。

例如，MdePkg.dsc 中的 [Defines] 部分如下所示：

```
[Defines]
DEC_SPECIFICATION = 0x00010005
PACKAGE_NAME = MdePkg
PACKAGE_GUID = 1E73767F-8F52-4603-AEB4-F29B510B6766
PACKAGE_VERSION = 1.03
```

2. [Includes] 块

[Includes] 中列出了本 Package 提供的头文件所在的目录，格式如下：

```
[Includes.$(Arch)]
Path
```

Path 只能是相对路径，该相对路径起始于本 Package 的 .dsc 所在的目录。

例如，MdePkg.dec 文件中的 [Includes] 部分如下所示：

```
[Includes]
Include
[Includes.IA32] # 编译 32 位程序时的头文件路径
Include/Ia32
[Includes.X64] # 编译 x86_64 位程序时的头文件路径
Include/X64
```

3. [LibraryClasses] 块

Package 可以通过 .dec 文件对外提供库，每个库都必须有一个头文件，放在 Include\Library 目录下。本区块用于明确库和头文件的对应关系。其格式如下：

```
[LibraryClasses.$(Arch)]
LibraryName | Path/LibraryHeader.h
```

例如，下面的代码是 MdePkg.dec 中的 [LibraryClasses] 的一部分。

```
[LibraryClasses]
UefiUsbLib|Include/Library/UefiUsbLib.h
...
[LibraryClasses.IA32, LibraryClasses.X64]
SmmLib|Include/Library/SmmLib.h
```

4. [Guids] 块

在 Package\Include\Guid 目录中有很多文件，每个文件内定义了一个或几个 GUID，例如

在 MdePkg\Include\Gpt.h 文件中定义了 Gpt 分区相关的 GUID，如下所示：

```
extern EFI_GUID gEfiPartTypeUnusedGuid;  
extern EFI_GUID gEfiPartTypeSystemPartGuid;  
extern EFI_GUID gEfiPartTypeLegacyMbrGuid
```

可以看出这些定义仅仅是声明。那么真正的常量定义在什么地方呢？真正的常量定义在 AutoGen.c 中，其值定义在 .dec 文件的 [Guids] 区块。其格式为：

```
[Guids.$(Arch)]  
    GUIDName = GUID
```

回到前面讲的 Gpt 相关的 GUID，在 MdePkg.dec 中，这些 GUID 定义如下：

```
[Guids]  
...  
## Include/Guid/Gpt.h  
    gEfiPartTypeLegacyMbrGuid = { 0x024DEE41, 0x33E7, 0x11D3, { 0x9D, 0x69,  
        0x00, 0x08, 0xC7, 0x81, 0xF3, 0x9F }}  
    gEfiPartTypeSystemPartGuid = { 0xC12A7328, 0xF81F, 0x11D2, { 0xBA, 0x4B, 0x00,  
        0xA0, 0xC9, 0x3E, 0xC9, 0x3B }}  
    gEfiPartTypeUnusedGuid = { 0x00000000, 0x0000, 0x0000, { 0x00, 0x00, 0x00,  
        0x00, 0x00, 0x00, 0x00, 0x00 }}
```

当在模块工程文件的 [Guids] 中引用这些 Guid 时，这些值就会复制到 AutoGen.c 中。

5. [Protocols] 块

与 Guids 类似，在 Package\Include\Protocols 目录下有很多头文件，每个头文件定义了一个或多个 Protocol，这些 Protocol 的 GUID 值就定义在 .dec 文件的 [Protocols] 区块，格式如下：

```
[Protocols.$(Arch)]  
    ProtocolName = GUID
```

例如，在 MdePkg\Include\Protocol 目录下的 BlockIo.h 定义了 BlockIo Protocol。

```
extern EFI_GUID gEfiBlockIoProtocolGuid;
```

gEfiBlockIoProtocolGuid 的值就定义在 MdePkg.dec 的 [Protocols] 块内。

```
[Protocols]  
    gEfiBlockIoProtocolGuid = { 0x964E5B21, 0x6459, 0x11D2, { 0x8E, 0x39, 0x00,  
        0xA0, 0xC9, 0x69, 0x72, 0x3B }}
```

下面简单说明 [Ppis]、[PCD] 和 [UserExtensions]。[Ppis] 用于定义源文件中用到的 PPI（回忆一下，PPI 是 PEI 阶段 PEI 模块之间通信的接口），语法类似于 [Protocols]。[PCD] 块是 .dsc 文件中 [PCD] 块的补充。

3.4 调试 UEFI

UEFI 有两种调试方式,一种是在模拟环境 Nt32Pkg 下调试,另一种是通过串口调试真实环境中的 UEFI 程序。下面介绍在 Nt32Pkg 下的调试。

在需要调试的代码前面加入 “_asm int 3” 后编译,然后在模拟环境 (Nt32Pkg) 中运行该程序,当模拟器执行到 “int 3” 指令时,会弹出对话框,然后就可以调试了。

下面以一个标准应用程序工程模块为例演示如何调试。示例 3-17 是这个模块的源文件,详细工程文件和代码在本书附带的代码的 book\infs\Debug 目录中。该示例中,打印语句前加入了 _asm int 3 指令。

【示例 3-17】 在源文件中加入调试代码。

```
#include <Uefi.h>
EFI_STATUS
UefiMain (IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
{
    _asm int 3
    SystemTable->ConOut->OutputString(SystemTable->ConOut, L"HelloWorld\n");
    return EFI_SUCCESS;
}
```

需要注意的是,使用 _asm int 3 只能调试 32 位的应用程序或驱动。

另一个要注意的地方是, Visual C 编译时默认打开了优化选项,优化后 “_asm int 3” 的位置可能发生变动,导致无法进入正确的调试位置。如果出现这种情况,需要设置 .inf 文件中的 [BuildOptions], 关闭优化选项。设置方法如下:

```
[BuildOptions]
MSFT:DEBUG_*_IA32_CC_FLAGS = /Od
```

示例 3-18 是这个模块的工程文件,在该工程文件中优化选项被关闭。

【示例 3-18】 关闭优化选项的工程文件。

```
[Defines]
INF_VERSION = 0x00010006
BASE_NAME = UefiMain
FILE_GUID = 4ea97c46-7491-4dfd-b442-747010f3ce5f
MODULE_TYPE = UEFI_APPLICATION
VERSION_STRING = 0.1
ENTRY_POINT = UefiMain

[Sources]
Main.c

[Packages]
MdePkg/MdePkg.dec

[LibraryClasses]
```

```
UefiApplicationEntryPoint
UefiLib
[BuildOptions]
  MSFT:DEBUG_*_IA32_CC_FLAGS = /Od
```

编译后生成 DebugMain.efi 文件。按如下步骤进入调试 DebugMain.efi 的调试界面。

1) 首先启动 UEFI 模拟器, 进入 UEFI Shell, 然后在 UEFI Shell 中执行 DebugMain.efi 程序, 如图 3-5 所示。

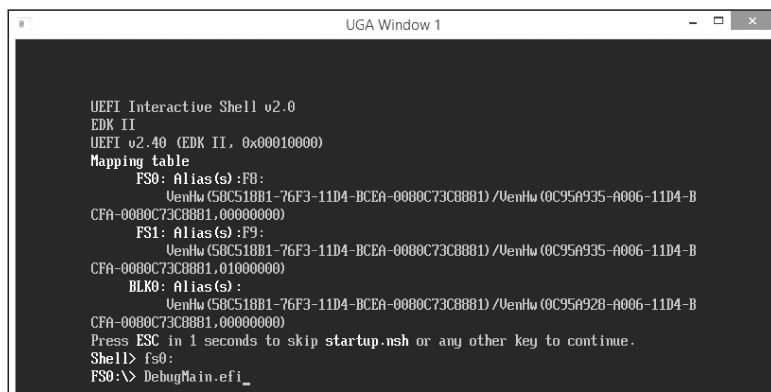


图 3-5 在 UEFI Shell 中执行 DebugMain.efi 程序

2) 执行 DebugMain.efi 程序后, 遇到 “int 3” 指令后会弹出系统调试对话框, 如图 3-6 所示。

3) 单击调试对话框的 “Debug” 按钮, 进入 Visual Studio 调试器, 如图 3-7 所示。

4) 单击 Visual Studio 调试器的 “Yes” 按钮, 进入 Visual Studio 调试准备界面, 如图 3-8 所示。

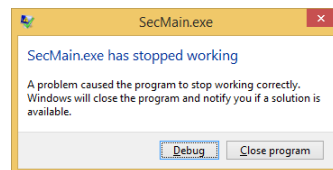


图 3-6 程序遇到 “int 3” 指令后弹出的系统调试对话框

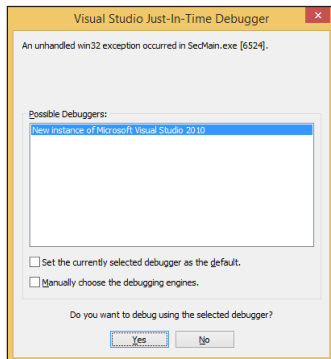


图 3-7 Visual Studio 调试器

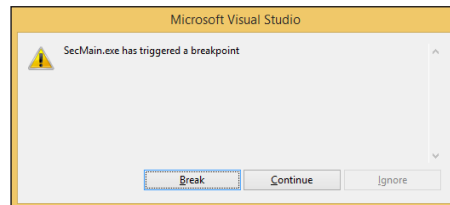


图 3-8 Visual Studio 调试准备界面

5) 单击 Visual Studio 调试准备界面中的“Break”按钮, 进入 Visual Studio 调试界面, 如图 3-9 所示。

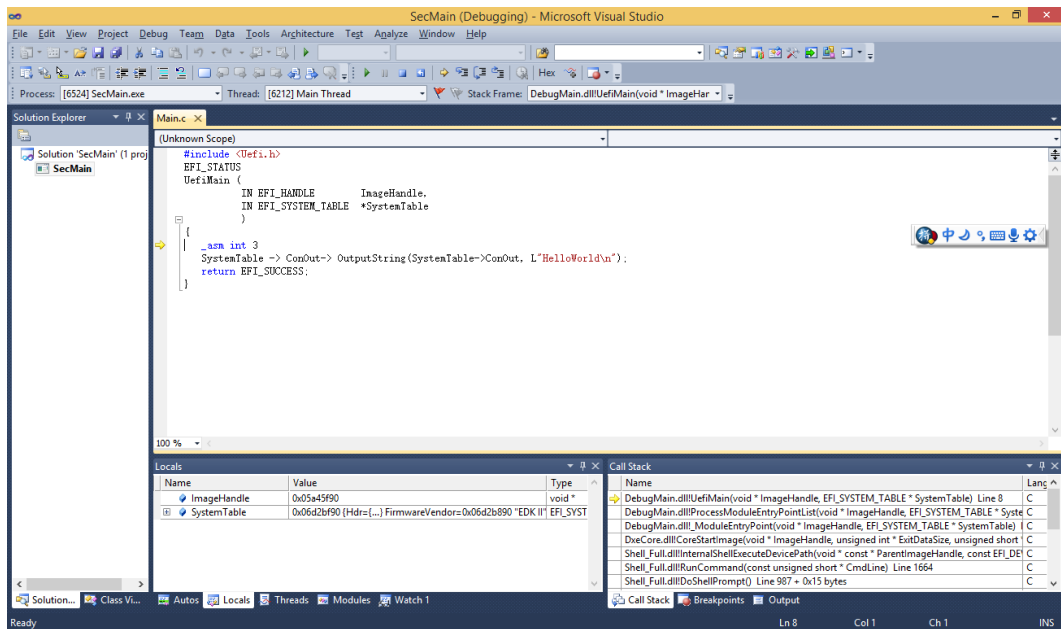


图 3-9 Visual Studio 调试界面

然后就可以利用 Visual Studio 调试器调试 DebugMain.efi 程序了。

3.5 本章小结

本章讲述了 .dsc、.dec 和 .inf 文件的格式。 .dsc 文件相当于 Visual Studio 的项目文件, 用于指导编译工具编译整个包; .inf 文件相当于 Visual Studio 的工程文件, 用于指导编译工具编译这个模块; .dec 文件则用于向其他工程模块提供本 Package 的资源。

现在我们已经“扫除”了编译 UEFI 应用程序的所有障碍。

在下一章, 将讲述 UEFI 开发中一定会遇到的系统服务。