

国内首部UEFI专著，资深UEFI专家兼布道者撰写，英特尔中国研究院院长吴甘沙和大数据和数据经济实验室研究总监周鑫联袂推荐

以实战为导向，全面介绍UEFI的使用、深入剖析UEFI的原理，为开发UEFI应用和驱动程序提供了翔实的指导



戴正华 著

UEFI: Principles and Programming

UEFI 原理与编程



机械工业出版社
China Machine Press

UEFI 原理与编程

戴正华 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

UEFI 原理与编程 / 戴正华著. —北京: 机械工业出版社, 2015.1
(实战)

ISBN 978-7-111-48729-6

I. U… II. 戴… III. 程序设计 IV. TP311.1

中国版本图书馆 CIP 数据核字 (2014) 第 282707 号



UEFI 原理与编程

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 高婧雅

责任校对: 殷虹

印刷:

版次: 2015 年 1 月第 1 版第 1 次印刷

开本: 186mm×240mm 1/16

印张: 26

书号: ISBN 978-7-111-48729-6

定价: 89.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: 010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Preface 序

收到书稿，看到熟悉的作者名字，立刻有种开始仔细阅读的冲动。

翻完最后一页，掩卷体会，其中充满了正华苦心钻研技术的精神与回馈社会的热情。

在 IT 行业各个细分领域里，系统程序员需要埋头于汇编语言 /C 语言的编辑器、调试器和体系结构的大部头手册，堪称最枯燥、最辛苦的工种之一。在系统领域取得成就的程序员需要有丰富的知识、扎实的技巧和踏实的态度，缺一不可。正华在我们团队开始系统程序员的职业生涯，他从一个普通的 C 程序员，以孜孜不倦的学习态度，快速成长为一个出色的编程语言和编译器系统的开发骨干。然后，他去了更广阔的天地，直到今天成为 UEFI 专家。正华的钻研精神和成长经历，是本书在专业内容以外，更值得读者学习和体会的精神食粮。

UEFI 是一个很出色的方向。尤其是对有志于学习计算机体系结构和操作系统的初学者来说，UEFI 是一个复杂度适中、内容涵盖度超高的学习平台。我们这一代的系统程序员，都是从奔腾 /586 CPU、DoS、Windows95、Linux Kernel 2.0 时代成长起来的。仔细阅读、修改、调试这些操作系统平台，是我们成长最有效的途径。但是，现代主流桌面操作系统，比如 Windows、Linux Ubuntu、Fedora，经过 20 多年的发展，功能超级丰富，系统设计超级复杂、代码量超级多，已经不适合作为体系结构和操作系统初学者的学习对象。UEFI 在恰当的时候填补了这个空缺。相比它的前辈 BIOS，UEFI 已经涵盖现代操作系统的核心内核功能和外设模块，可以视为一个最最简化版的操作系统。学通、学透 UEFI，可以帮助操作系统初学者快速入门。同时，对于安全领域和 X86 嵌入式领域来说，UEFI 是不可或缺的知识构成部分。

本书是正华热情回馈社会的成果。书里详细、具体地再现了一个学习者对 UEFI 的学习历程。相对于枯燥的手册，这种历程回顾方式的学习材料，更具有可阅读性、可学习性、可

操作性。事无巨细地列举了所有环境细节、命令参数，能极大地帮助初学者绕过障碍，专注于核心内容的学习，缩短学习过程。这种貌似简单，实则烦琐的学习总结，需要极大的精力和时间来准备与撰写，正华的回馈精神体现无疑。

希望读者们从内容中获取知识的同时，也体会学习和回馈的精神。

周鑫

英特尔中国研究院 大数据和数据经济实验室 研究总监

2014 年 11 月 23 日



BIOS 已经逐渐成为历史，UEFI 目前开始全面取代 BIOS。

UEFI 全称统一可扩展固件接口，是 UEFI 论坛发布的一种操作系统和平台固件之间的标准。它之所以能迅速取代 BIOS，源于硬件平台的发展以及 UEFI 相对于 BIOS 的巨大优势。UEFI 可编程性好，可扩展性好，性能高，安全性高。

随着 64 位 CPU 取代 32 位 CPU，UEFI 也完成了对 BIOS 的取代。市场已经完成了这种转变，对固件开发者来说，这是一种挑战，固件开发模式已经发生了巨大的改变；这也是一个机遇，UEFI 为计算机系统提供了更丰富的功能，为开发者提供了更强大的开发手段。

为什么写这本书

21 世纪什么最重要？大家都知道答案。人才是需要培养的。培养 UEFI 开发者最重要的是相关开发资料。但是目前 UEFI 相关的开发资料十分匮乏。UEFI 是一种全新的技术，是对 BIOS 的一种革命，BIOS 时代的技术积累很难转移给 UEFI。在 BIOS 时代，BIOS 开发采用汇编语言并且与硬件特性息息相关，技术被几大公司垄断，进入 BIOS 开发领域的门槛高，离开的代价大，相关的技术资料很难在广大程序员中流传。进入 UEFI 时代后，这种开发特点因为惯性的作用依然会延续一段时间。

UEFI 开发对广大 C/C++ 开发者敞开了大门，进入和离开 UEFI 的门槛降低了很多。相信会有更多的开发者进入和离开固件开发领域，这种流动性也会促进这个领域的繁荣，相关的开发资料也会越来越多，越来越精彩。

本书特色

本书是国内第一本介绍 UEFI 开发的书籍，希望这本书能改善 UEFI 开发者无处学习的困境。本书以实战为主，辅以相关理论知识，重要的章节还附有完整的应用程序。力图让读

者不仅知道如何开发 UEFI 程序，还让读者了解为什么这样编程。本书提供了丰富的源代码，让读者可以在实践中快速掌握编程要点。

读者对象

本书内容循序渐进，非常适合刚刚接触 UEFI 开发的初学者、大专院校在校的学生，也适合对 UEFI 有一定了解的专业开发者。

如何阅读本书

本书假定读者有基本的 C 和 C++ 知识，但并不要求读者有固件开发的相关知识。本书将一步一步引导读者熟悉 UEFI，随着本书的不断深入，使读者成为 UEFI 开发专家。

本书从实战的角度介绍如何开发 UEFI 应用和驱动，用生动的实例介绍如何使用 UEFI 提供的服务，同时我们将讲述所需的理论知识以及 UEFI 的内部实现。本书既可以作为 UEFI 爱好者的入门教材，也可以帮 UEFI 开发者更加深入了解 UEFI 内部实现。

本书不是 UEFI 开发的参考手册，所以读者在学习过程中最好准备 UEFI Spec 2.4 以备参考。本书提供了大量的实例程序，读者还需下载 TianoCore 的源码，边学习边实践。

本书内容编排如下：

第 1 章 UEFI 概述介绍 UEFI 发展的历史及 UEFI 理论知识，UEFI 系统启动到结束可分为 7 个过程，本章从程序开发的角度阐述了这个过程的执行流程。

第 2 章 介绍 UEFI 开发环境的搭建，EDK2 提供了 Linux、Windows、Cygwin 等多种开发环境，本章逐步讲述了 Linux 和 Windows 下 UEFI 开发环境的搭建，如何制作 OVMF 固件，以及如何制作启动盘。

第 3 章 介绍 EDK2 工程模块及包的概念，主要包括 .inf 文件、.dsc 文件和 .dec 文件的格式与用法。其中，主要的工程模块包括 UEFI 应用程序模块、驱动程序模块、库模块、Shell 应用程序模块。本章最后讲述了如何在模拟器下调试应用程序。

第 4 章 介绍 Protocol 的概念和用法。UEFI 提供的绝大多数服务都是以 Protocol 的形式提供的，由此可见其重要性。

第 5 章 介绍 UEFI 中的基础服务，包括系统表，启动服务和运行时服务。系统表是应用程序进入内核的接口，启动服务用于在启动过程中管理计算机软硬件资源，运行时服务是为操作系统访问固件而提供的服务。

第 6 章 介绍了事件及异步操作方式。除了介绍事件的使用方法，本章还介绍了事件及异步操作在 UEFI 内核的实现机制。最后本章以具体实例介绍了如何使用键盘、鼠标和定时

器事件。

第 7 章 介绍了 GPT 硬盘以及文件系统和文件的操作方法。相比 BIOS, UEFI 的一个重大进步就是开始支持文件系统, FAT32 文件系统是 UEFI 内建文件系统。本章重点讲述 FAT32 文件系统之上的文件操作。

第 8 章 以视频解码服务为例介绍如何利用 Protocol 提供服务。服务型 Protocol 在 UEFI 中占有重要地位, 服务型 Protocol 的开发是 UEFI 开发的一项基本功, 一定要熟练掌握。

第 9 章 介绍驱动开发模型, 并以 AC97 驱动为例介绍开发设备驱动的具体步骤。

第 10 章 介绍如何支持 C++ 语言特性。UEFI 内核是 EDK2 采用 C 语言实现的, UEFI 提供的服务也是以 C 语言形式提供的, 因而 C 天然适用于 UEFI 开发, 但 C++ 会使得开发更有效率。本章首先讲述了如何使用 C++ 基础语法开发 UEFI 应用, 然后讲述如何支持全局构造函数、析构函数, 支持 new、delete 等操作符, 支持标准模板库等高级特性。

第 11 章 介绍开发 GUI 的基础知识, 包括在 UEFI 中如何使用字符串资源、字体和图像。通过使用字符串资源, 可以实现字符串的本地化。本地化后, 字符串的显示需要字体的支持, EDK2 默认支持英文和法文字符的显示, 要显示其他语言的字符串还需要开发者自行开发相应字符的字体。

第 12 章 以视频播放器为例介绍 GUI 开发。目前还没有成熟的 GUI 库可以用于 UEFI 开发。本章从零开始利用第 11 章介绍的基础知识开发 GUI 系统, 主要分两大部分: GUI 事件的派遣和响应; GUI 控件的绘制。

第 13 章 介绍多任务的开发。本章分为两大块: 多核和多线程。首先主要介绍 MPP(MP Services Protocol), MPP 用于管理多核, 本节重点讲述了 MPP 如何在其他 CPU 核心上执行任务。UEFI 没有提供对多线程的支持, 对此我们讲述了如何使用 LongJump 技术实现简单的多线程。

第 14 章 介绍如何开发网络应用。本章首先简单介绍了 UEFI 提供的网络协议栈, 然后讲述了使用 TCP 协议开发网络应用的基本框架。

第 15 章 介绍如何使用 C 标准库中的函数。

第 16 章 介绍应用程序中的 Shell 服务、Shell 脚本的语法以及常用 Shell 命令。

资源和勘误

由于笔者水平与时间有限, 书中可能出现一些错误或不准确的地方, 恳请读者批评、斧正。如果您对本书有任何的意见和指正, 请发邮件至 djx.zhenghua@gmail.com。本书附带的

源代码，可以从如下网址获取：<https://uefi-programming-guider.googlecode.com/svn/trunk/>，或者从华章网站（www.hzbook.com）下载。

致谢

感谢华章公司的高婧雅与杨福川付出的巨大努力，高编辑耐心、细致地反复审阅书稿，使得本书从粗疏渐渐变得精准。

感谢宋风龙在本书写作过程中给予的技术支持。

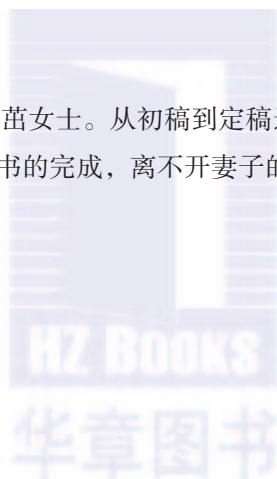
本书最早起始于我在博客园发表的几篇博客，感谢博客园的众多网友的支持和意见，你们的建议让我受益颇深。

也要感谢工作在 CryptoMill 的同事。

特别鸣谢

最后要特别感谢我的妻子张一茁女士。从初稿到定稿近一年的时间里，我将大部分的节假日和周末用在了这本书上，这本书的完成，离不开妻子的付出和支持。

戴正华



Contents 目 录

序	
前 言	
第 1 章 UEFI 概述	1
1.1 BIOS 的前世今生	1
1.1.1 BIOS 在计算机系统中的作用	1
1.1.2 BIOS 缺点	2
1.2 初识 UEFI	2
1.2.1 UEFI 系统组成	3
1.2.2 UEFI 的优点	4
1.2.3 UEFI 系统的启动过程	5
1.3 本章小结	12
第 2 章 UEFI 开发环境搭建	14
2.1 配置 Windows 开发环境	14
2.1.1 安装所需开发工具	15
2.1.2 配置 EDK2 开发环境	15
2.1.3 编译 UEFI 模拟器和 UEFI 工程	17
2.1.4 运行模拟器	19
2.2 配置 Linux 开发环境	21
2.2.1 安装所需开发工具	22
2.2.2 配置 EDK2 开发环境	22
2.2.3 编译 UEFI 模拟器和 UEFI 工程	23
2.2.4 运行模拟器	24
2.3 OVMF 的制作和使用	25
2.4 UEFI 的启动	27
2.5 本章小结	28
第 3 章 UEFI 工程模块文件	29
3.1 标准应用程序工程模块	30
3.1.1 入口函数	30
3.1.2 工程文件	31
3.1.3 编译和运行	37
3.1.4 标准应用程序的加载过程	37
3.2 其他类型工程模块	43
3.2.1 Shell 应用程序工程模块	43
3.2.2 使用 main 函数的应用程序工程模块	46
3.2.3 库模块	47
3.2.4 UEFI 驱动模块	49
3.2.5 模块工程文件小结	50
3.3 包及 .dsc、.dec、.fdf 文件	51
3.3.1 .dsc 文件	51

3.3.2	.dec 文件	56	6.1.2	生成事件的服务	
3.4	调试 UEFI	59		CreateEvent	106
3.5	本章小结	61	6.1.3	CreateEventEx 服务	110
第 4 章	UEFI 中的 Protocol	62	6.1.4	事件相关的其他函数	112
4.1	Protocol 在 UEFI 内核中的表示	64	6.2	定时器事件	113
4.2	如何使用 Protocol 服务	65	6.3	任务优先级	114
4.2.1	OpenProtocol 服务	66	6.3.1	提升和恢复任务优先级	115
4.2.2	HandleProtocol 服务	67	6.3.2	UEFI 中的时钟中断	116
4.2.3	LocateProtocol 服务	69	6.3.3	UEFI 事件 Notification 函数的派发	126
4.2.4	LocateHandleBuffer 服务	69	6.4	鼠标和键盘事件示例	127
4.2.5	其他一些使用 Protocol 的服务	71	6.5	本章小结	128
4.2.6	CloseProtocol 服务	72	第 7 章	硬盘和文件系统	129
4.3	Protocol 服务示例	73	7.1	GPT 硬盘	129
4.4	本章小结	75	7.1.1	基于 MBR 分区的传统硬盘	129
第 5 章	UEFI 的基础服务	76	7.1.2	GPT 硬盘详解	130
5.1	系统表	76	7.2	设备路径	134
5.1.1	系统表的构成	77	7.3	硬盘相关的 Protocol	139
5.1.2	使用系统表	79	7.3.1	BlockIo 解析	140
5.2	启动服务	82	7.3.2	BlockIo2 解析	142
5.2.1	启动服务的构成	82	7.3.3	DiskIo 解析	146
5.2.2	启动服务的生存期	91	7.3.4	DiskIo2 解析	147
5.3	运行时服务	93	7.3.5	PassThrough 解析	150
5.4	本章小结	102	7.4	文件系统	152
第 6 章	事件	103	7.5	文件操作	153
6.1	事件函数	104	7.5.1	打开文件	154
6.1.1	等待事件的服务		7.5.2	读文件	156
	WaitForEvent	105	7.5.3	写文件	159
			7.5.4	关闭文件 (句柄)	160
			7.5.5	其他文件操作	160

7.5.6 异步文件操作	162	第 10 章 用 C++ 开发 UEFI 应用	222
7.5.7 EFI_SHELL_PROTOCOL 中的 文件操作	166	10.1 从编译器角度看 C 与 C++ 的 差异	222
7.6 本章小结	170	10.2 在 EDK2 中支持 C++	224
第 8 章 开发 UEFI 服务	171	10.2.1 使 EDK2 支持 C++ 基本 特性	224
8.1 Protocol 服务接口设计	172	10.2.2 在 Windows 系统下的程序 启动过程	226
8.2 Protocol 服务的实现	174	10.2.3 在 Windows 系统下支持 全局构造和析构	229
8.3 服务型驱动的框架	178	10.2.4 在 Linux 系统下的程序启动 过程	231
8.4 ffmpeg 的移植与编译	179	10.2.5 在 Linux 系统下支持全局 构造和析构	240
8.4.1 libavcodec 的建立和移植	181	10.2.6 支持 new 和 delete	242
8.4.2 其他库的建立与移植	182	10.2.7 支持 STL	243
8.4.3 在驱动型服务中使用 StdLib	186	10.3 GcppPkg 概览	243
8.5 使用 Protocol 服务	188	10.4 测试 GcppPkg	246
8.6 本章小结	190	10.5 本章小结	248
第 9 章 开发 UEFI 驱动	191	第 11 章 GUI 基础	249
9.1 UEFI 驱动模型	192	11.1 字符串	249
9.1.1 EFI Driver Binding Protocol 的 构成	192	11.1.1 字符串函数	249
9.1.2 EFI Component Name Protocol 的作用和构成	196	11.1.2 字符串资源	251
9.2 编写设备驱动的步骤	197	11.1.3 管理字符串资源	255
9.3 PCI 设备驱动基础	199	11.2 管理语言	260
9.4 AC97 控制器芯片的控制接口	202	11.3 包列表	262
9.5 AC97 驱动	206	11.4 图形界面显示	263
9.5.1 AC97 驱动的驱动服务 EFI_AUDIO_PROTOCOL	206	11.4.1 显示模式	264
9.5.2 AC97 驱动的框架部分	213	11.4.2 Block Transfer (Blt) 传输 图像	267
9.5.3 AC97 驱动实验	220		
9.6 本章小结	221		

11.4.3 在图形界面下显示字符串	269	13.1.2 启动 AP 的过程	324
11.5 用 SimpleFont 显示中文	272	13.2 内联汇编基础和寄存器上下文的保存与恢复	333
11.5.1 SimpleFont 格式	273	13.2.1 内联汇编基础	333
11.5.2 如何生成字体文件	275	13.2.2 寄存器上下文的保存与恢复	335
11.5.3 如何注册字体文件	276	13.3 多线程	336
11.6 开发 SimpleFont 字库程序	277	13.3.1 生成线程	337
11.7 字体 Font	278	13.3.2 调度线程	340
11.7.1 Font 的格式	279	13.3.3 等待线程结束	341
11.7.2 字体包的格式	279	13.3.4 SimpleThread 服务	341
11.7.3 为什么 Font 性能高于 SimpleFont	281	13.4 本章小结	345
11.8 本章小结	284		
第 12 章 GUI 应用程序	285	第 14 章 网络应用开发	346
12.1 UEFI 事件处理	285	14.1 在 UEFI 中使用网络	348
12.1.1 键盘事件	285	14.2 使用 EFI_TCP4_PROTOCOL	350
12.1.2 鼠标事件	292	14.2.1 生成 Socket 对象	352
12.1.3 定时器事件	293	14.2.2 连接	356
12.1.4 UI 事件服务类	294	14.2.3 传输数据	358
12.2 事件处理框架	297	14.2.4 关闭 Socket	361
12.3 鼠标与控件的绘制	302	14.2.5 测试 Socket	362
12.3.1 鼠标的绘制	303	14.3 本章小结	363
12.3.2 控件的绘制	305		
12.4 控件系统包 GUIPkg	306	第 15 章 使用 C 标准库	364
12.5 简单视频播放器的实现	309	15.1 为什么使用 C 标准库函数	364
12.6 本章小结	315	15.2 实现简单的 Std 函数	365
		15.2.1 简单标准库函数包 sstdPkg	366
第 13 章 深入了解多任务	317	15.2.2 使用 sstdPkg	368
13.1 多处理器服务	317	15.3 使用 EDK2 的 StdLib	369
13.1.1 EFI_MP_SERVICES_PROTOCOL 功能及用法	317	15.3.1 main 函数工程	369

15.3.2 非 main 函数工程	374	16.4.1 调试设备的相关命令	388
15.4 本章小结	376	16.4.2 驱动相关命令	390
第 16 章 Shell 及常用 Shell 命令	377	16.4.3 网络相关命令	392
16.1 Shell 的编译与执行	377	16.5 本章小结	394
16.2 Shell 服务	379	附录 A UEFI 常用术语及简略语	395
16.3 Shell 脚本	385	附录 B RFC 4646 常用语言列表	397
16.3.1 Shell 脚本语法简介	385	附录 C 状态值	398
16.3.2 自动运行指定应用程序	388	附录 D 参考资料	400
16.4 Shell 内置命令	388		





UEFI 概述

从我们按下开机键到进入操作系统，对用户来说是一个等待的过程，而对计算机来说是一个复杂的过程。在 BIOS 时代，这个过程重复了一年又一年，操作系统已经从枯燥的文本界面演化到丰富多彩的图形界面，BIOS 却一直延续着枯燥的过程，BIOS 设置也一直是单调的蓝底白字格式。BIOS 的坚持出于两个原因：外因是 BIOS 基本能满足市场需求，内因是 BIOS 的设计使得 BIOS 的升级和扩增变得非常困难。随着 64 位 CPU 逐渐取代 32 位 CPU，BIOS 越来越不能满足市场的需求，这使得 UEFI 作为 BIOS 的替代者，逐渐开始取代 BIOS 的地位。

1.1 BIOS 的前世今生

BIOS 诞生于 1975 年的 CP/M 计算机，诞生之初，也曾是一种先进的技术，并且是系统中相当重要的一个部分。随着 IBM PC 兼容机的流行，BIOS 也逐渐发展起来。它“统治”了计算机系统 20 多年的时间，在这段时间里，CPU 每 18 个月性能提升一倍。计算机软硬件都已经繁衍了无数代，BIOS 诞生之初与之配套的 8 位 CPU 和 DOS 系统都已经退出历史舞台，而 BIOS 依然顽强地存在于计算机中。

1.1.1 BIOS 在计算机系统中的作用

BIOS 全称为“基本输入/输出系统”，它是存储在主板 ROM 里的一组程序代码，这些代码包括：

- ❑ 加电自检程序，用于开机时对硬件的检测。
- ❑ 系统初始化代码，包括硬件设备的初始化、创建 BIOS 中断向量等。
- ❑ 基本的外围 I/O 处理的子程序代码。
- ❑ CMOS 设置程序。

BIOS 程序运行在 16 位实模式下，实模式下最大的寻址范围是 1MB, 0x0C0000 ~ 0x0FFFFF 保留给 BIOS 使用。开机后，CPU 跳到 0x0FFFF0 处执行，一般这里是一条跳转指令，跳到真正的 BIOS 入口处执行。BIOS 代码首先做的是“加电自检”（Power On Self Test, POST），主要是检测关机设备是否正常工作，设备设置是否与 CMOS 中的设置一致。如果发现硬件错误，则通过喇叭报警。POST 检测通过后初始化显示设备并显示显卡信息，接着初始化其他设备。设备初始化完毕后开始检查 CPU 和内存并显示检测结果。内存检测通过以后开始检测标准设备，例如硬盘、光驱、串口设备、并口设备等。然后检测即插即用设备，并为这些设备分配中断号、I/O 端口和 DMA 通道等资源。如果硬件配置发生变化，那么这些变化的配置将更新到 CMOS 中。随后，根据配置的启动顺序从设备启动，将启动设备主引导记录的启动代码通过 BIOS 中断读入内存，然后控制权交到引导程序手中，最终引导进入操作系统。

1.1.2 BIOS 缺点

随着 CPU 及其他硬件设备的革新，BIOS 逐渐成为计算机系统发展的瓶颈，主要体现在如下几个方面：

- 1) **开发效率低**：大部分 BIOS 代码使用汇编开发，开发效率不言而喻。汇编开发的另一个缺点是使得代码与设备的耦合程度太高，代码受硬件变化的影响大。
- 2) **性能差**：BIOS 基本输入 / 输出服务需要通过中断来完成，开销大，并且 BIOS 没有提供异步工作模式，大量的时间消耗在等待上。
- 3) **功能扩展性差，升级缓慢**：BIOS 代码采用静态链接，增加硬件功能时，必须将 16 位代码放置在 0x0C0000 ~ 0x0DFFFF 区间，初始化时将其设置为约定的中断处理程序。而且 BIOS 没有提供动态加载设备驱动的方案。
- 4) **安全性**：BIOS 运行过程中对可执行代码没有安全方面的考虑。
- 5) **不支持从硬盘 2 TB 以上的地址引导**：受限于 BIOS 硬盘的寻址方式，BIOS 硬盘采用 32 位地址，因而引导扇区的最大逻辑块地址是 2^{32} （换算成字节地址，即 $2^{32} \times 512 = 2\text{TB}$ ）。

1.2 初识 UEFI

UEFI（Unified Extensible Firmware Interface，统一可扩展固件接口）定义了操作系统和平台固件之间的接口，它是 UEFI Forum 发布的一种标准。它只是一种标准，没有提供实现。

其实现由其他公司或开源组织提供，例如英特尔公司提供的开源 UEFI 实现 TianoCore 和 Phoenix 公司的 SecureCore Tiano。UEFI 实现一般可分为两部分：

- ❑ 平台初始化（遵循 Platform Initialization 标准，同样由 UEFI Forum 发布）。
- ❑ 固件 – 操作系统接口。

UEFI 发端于 20 世纪 90 年代中期的安腾系统。相对于当时流行的 32 位 IA32 系统，安腾是一种全新的 64 位系统，BIOS 的限制对这种 64 位系统变得不可接受（BIOS 也正是随着 32 位系统被 64 位系统取代而逐渐退出市场的）。因为 BIOS 在 64 位系统上的限制，1998 年英特尔公司发起了 Intel Boot Initiative 项目，后来更名为 EFI（Extensible Firmware Interface）。2003 年英特尔公司的安腾 CPU 计划遭到 AMD 公司的 x86_64 CPU 顽强阻击，x86_64 CPU 时代到来，市场更愿意接受渐进式的变化，英特尔公司也开始发布兼容 32 位系统的 x86_64 CPU。安腾虽然没有像预期那样独占市场，EFI 却显示出了它的价值。2005 年，英特尔公司联合微软、AMD、联想等 11 家公司成立了 Unified EFI Forum，负责制定统一的 EFI 标准。第一个 UEFI 标准——UEFI 2.0 在 2006 年 1 月发布。目前最新的 UEFI 标准是 2013 年发布的 UEFI 2.4。

1.2.1 UEFI 系统组成

UEFI 提供给操作系统的接口包括启动服务（Boot Services，BS）和运行时服务（Runtime Service，RT）以及隐藏在 BS 之后的丰富的 Protocol。BS 和 RT 以表的形式（C 语言中的结构体）存在。UEFI 驱动和服务以 Protocol 的形式通过 BS 提供给操作系统。

图 1-1 展示了基于 EFI 的计算机系统的组成。

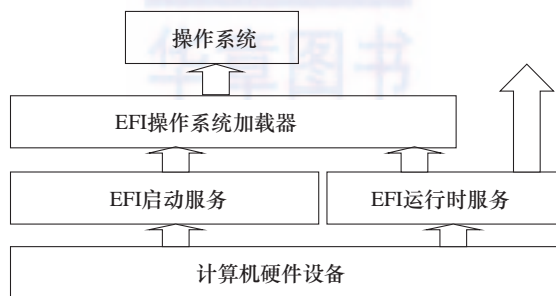


图 1-1 EFI 系统组成

从操作系统加载器（OS Loader）被加载，到 OS Loader 执行 ExitBootServices() 的这段时间，是从 UEFI 环境向操作系统过渡的过程。在这个过程中，OS Loader 可以通过 BS 和 RT 使用 UEFI 提供的服务，将计算机系统资源逐渐转移到自己手中，这个过程称为 TSL（Transient System Load）。

当 OS Loader 完全掌握了计算机系统资源时，BS 也就完成了它的使命。OS Loader 调用 ExitBootServices() 结束 BS 并回收 BS 占用的资源，之后计算机系统进入 UEFI Runtime 阶段。

在 Runtime 阶段只有运行时服务继续为 OS 提供服务，BS 已经从计算机系统中销毁。

在 TSL 阶段，系统资源通过 BS 管理，BS 提供的服务如下。

1) **事件服务**：事件是异步操作的基础。有了事件的支持，才可以在 UEFI 系统内执行并发操作。

2) **内存管理**：主要提供内存的分配与释放服务，管理系统内存映射。

3) **Protocol 管理**：提供了安装 Protocol 与卸载 Protocol 的服务，以及注册 Protocol 通知函数（该函数在 Protocol 安装时调用）的服务。

4) **Protocol 使用类服务**：包括 Protocol 的打开与关闭，查找支持 Protocol 的控制器。例如要读写某个 PCI 设备的寄存器，可以通过 OpenProtocol 服务打开这个设备上的 PciIo Protocol，用 PciIo->Io.Read() 服务可以读取这个设备上的寄存器。

5) **驱动管理**：包括用于将驱动安装到控制器的 connect 服务，以及将驱动从控制器上卸载的 disconnect 服务。例如，启动时，如果我们需要网络支持，则可以通过 loadImage 将驱动加载到内存，然后通过 connect 服务将驱动安装到设备。

6) **Image 管理**：此类服务包括加载、卸载、启动和退出 UEFI 应用程序或驱动。

7) **ExitBootServices**：用于结束启动服务。

RT 提供的服务主要包括如下几个方面。

1) **时间服务**：读取 / 设定系统时间。读取 / 设定系统从睡眠中唤醒的时间。

2) **读写 UEFI 系统变量**：读取 / 设置系统变量，例如 BootOrder 用于指定启动项顺序。通过这些系统变量可以保存系统配置。

3) **虚拟内存服务**：将物理地址转换为虚拟地址。

4) **其他服务**：包括重启系统的 ResetSystem，获取系统提供的下一个单调单增值等。

1.2.2 UEFI 的优点

UEFI 能迅速取代 BIOS，得益于 UEFI 相对 BIOS 的几大优势。

(1) UEFI 的开发效率

BIOS 开发一般采用汇编语言，代码多是硬件相关的代码。而在 UEFI 中，绝大部分代码采用 C 语言编写，UEFI 应用程序和驱动甚至可以使用 C++ 编写。UEFI 通过固件 - 操作系统接口（BS 和 RT 服务）为 OS 和 OS 加载器屏蔽了底层硬件细节，使得 UEFI 上层应用可以方便重用。

(2) UEFI 系统的可扩展性

UEFI 系统的可扩展性体现在两个方面：一是驱动的模块化设计；二是软硬件升级的兼容性。

大部分硬件的初始化通过 UEFI 驱动实现。每个驱动是一个独立的模块，可以包含在固

件中，也可以放在设备上，运行时根据需要动态加载。

UEFI 中每个表、每个 Protocol(包括驱动)都有版本号，这使得系统的平滑升级变得简单。

(3) UEFI 系统的性能

相比 BIOS，UEFI 有了很大的性能提升，从启动到进入操作系统的时间大大缩短。性能的提高源于以下几个方面：

1) UEFI 提供了异步操作。基于事件的异步操作，提高了 CPU 利用率，减少了总的等待时间。

2) UEFI 舍弃了中断这种比较耗时的操作外部设备的方式，仅仅保留了时钟中断。外部设备的操作采用“事件+异步操作”完成。

3) 可伸缩的遍历设备的方式，启动时可以仅仅遍历启动所需的设备，从而加速系统启动。

(4) UEFI 系统的安全性

UEFI 的一个重要突破就是其安全方面的考虑。当系统的安全启动功能被打开后，UEFI 在执行应用程序和驱动前会先检测程序和驱动的证书，仅当证书被信任时才会执行这个应用程序或驱动。UEFI 应用程序和驱动采用 PE/COFF 格式，其签名放在签名块中。

1.2.3 UEFI 系统的启动过程

UEFI 系统的启动遵循 UEFI 平台初始化 (PlatformInitialization) 标准^①。UEFI 系统从加电到关机可分为 7 个阶段：

SEC (安全验证) → PEI (EFI 前期初始化) → DXE (驱动执行环境)
 → BDS (启动设备选择) → TSL (操作系统加载前期)
 → RT (Run Time)
 → AL (系统灾难恢复期)

图 1-2 展示了 UEFI 系统从加电到关机的 7 个阶段^② (以图中竖线为界)。

前三个阶段是 UEFI 初始化阶段，DXE 阶段结束后 UEFI 环境已经准备完毕。

BDS 和 TSL 是操作系统加载器作为 UEFI 应用程序运行的阶段。

操作系统加载器调用 ExitBootServices() 服务后进入 RT 阶段，RT 阶段包括操作系统加载器后期和操作系统运行期。

当系统硬件或操作系统出现严重错误不能继续正常运行时，固件会尝试修复错误，这时系统进入 AL 期。但 PI 规范和 UEFI 规范都没有规定 AL 期的行为。“?” 号表示其行为由系统供应商自行定义。

① www.uefi.org。

② http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=File:PI_Boot_Phases.jpg。

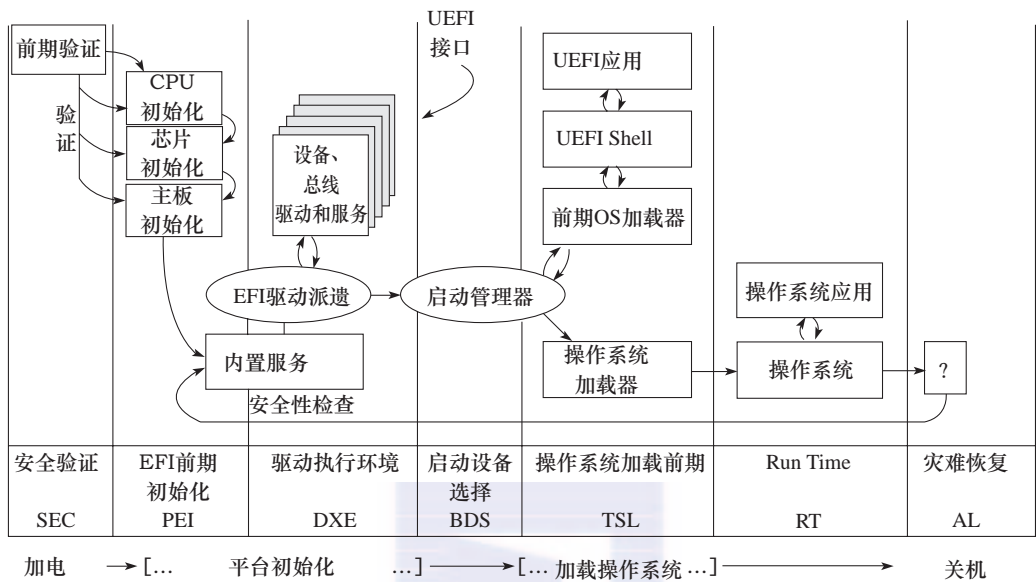


图 1-2 UEFI 系统的 7 个阶段

1. SEC 阶段

SEC (Security Phase) 阶段是平台初始化的第一个阶段，计算机系统加电后进入这个阶段。

(1) SEC 阶段的功能

UEFI 系统开机或重启进入 SEC 阶段，从功能上说，它执行以下 4 种任务。

1) 接收并处理系统启动和重启信号：系统加电信号、系统重启信号、系统运行过程中的严重异常信号。

2) 初始化临时存储区域：系统运行在 SEC 阶段时，仅 CPU 和 CPU 内部资源被初始化，各种外部设备和内存都没有被初始化，因而系统需要一些临时 RAM 区域，用于代码和数据的存取，我们将之称为临时 RAM，以示与内存的区别。这些临时 RAM 只能位于 CPU 内部。最常用的临时 RAM 是 Cache，当 Cache 被配置为 no-eviction 模式时，可以作为内存使用，读命中时返回 Cache 中的数据，读缺失时不会向主存发出缺失事件；写命中时将数据写入 Cache，写缺失时不会向主存发出缺失事件，这种技术称为 CAR (Cache As Ram)。

3) 作为可信系统的根：作为取得对系统控制权的第一部分，SEC 阶段是整个可信系统的根。SEC 能被系统信任，以后的各个阶段才有被信任的基础。通常，SEC 在将控制权转移给 PEI 之前，可以验证 PEI。

4) 传递系统参数给下一阶段 (即 PEI)：SEC 阶段的一切工作都是为 PEI 阶段做准备，最终 SEC 要把控制权转交给 PEI，同时要将现阶段的成果汇报给 PEI。汇报的手段就是将如

下信息作为参数传递给 PEI 的入口函数。

- ❑ 系统当前状态，PEI 可以根据这些状态判断系统的健康状况。
- ❑ 可启动固件（Boot Firmware Volume）的地址和大小。
- ❑ 临时 RAM 区域的地址和大小。
- ❑ 栈的地址和大小。

(2) SEC 阶段执行流程

上面介绍了 SEC 的功能，下面再来看看 SEC 的执行流程，如图 1-3 所示。

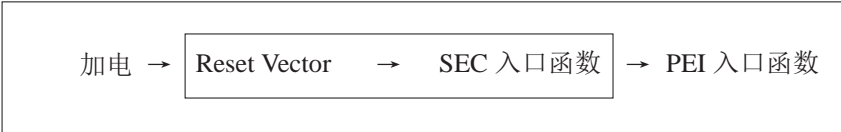


图 1-3 SEC 阶段执行流程

以临时 RAM 初始化为界，SEC 的执行又分为两大部分：临时 RAM 生效之前称为 Reset Vector 阶段，临时 RAM 生效后调用 SEC 入口函数从而进入 SEC 功能区。

其中 Reset Vector 的执行流程如下。

- 1) 进入固件入口。
- 2) 从实模式转换到 32 位平坦模式（包含模式）。
- 3) 定位固件中的 BFV（Boot Firmware Volume）。
- 4) 定位 BFV 中的 SEC 映像。
- 5) 若是 64 位系统，从 32 位模式转换到 64 位模式。
- 6) 调用 SEC 入口函数。

下面的代码描述了从固件入口 Reset Vector 到 SEC 入口函数的执行过程：

```
; file: UefiCpuPkg/ResetVector/Vtf0/Ia16/ResetVectorVtf0.asm
resetVector:
jmp     short EarlyBspInitReal16

; file: UefiCpuPkg/ResetVector/Vtf0/Ia16/Init16.asm EarlyBspInitReal16:
mov     di, 'BP'
jmp     short Main16

; file: UefiCpuPkg/ResetVector/Vtf0/Main.asm
Main16:
OneTimeCall EarlyInit16
OneTimeCall TransitionFromReal16To32BitFlat;    从实模式转换到 32 位平坦模式
OneTimeCall Flat32SearchForBfvBase;            定位固件中的 BFV
```

```

OneTimeCall Flat32SearchForSecEntryPoint;           定位 BFV 中的 SEC 映像
;esi 寄存器存放了 SEC 的入口地址, ebp 寄存器存放了 BFV 起始地址
#ifdef ARCH_IA32
mov eax, esp
jmp esi; 跳到 SEC 入口
#else
OneTimeCall Transition32FlatTo64Flat;             从 32 位模式转换到 64 位模式
...
jmp rsi;                                           跳到 SEC 入口
#endif

```

在 Reset Vector 部分, 因为系统还没有 RAM, 因而不能使用基于栈的程序设计, 所有的函数调用都使用 jmp 指令模拟。OneTimeCall 是宏, 用于模拟 call 指令。例如, 宏调用 OneTimeCall EarlyInit16, 如图 1-4 所示。

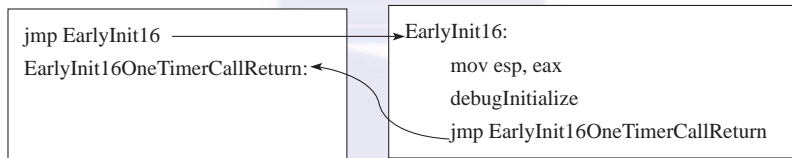


图 1-4 OneTimeCall 示例

进入 SEC 功能区后, 首先利用 CAR 技术初始化栈, 初始化 IDT, 初始化 EFI_SEC_PEI_HAND_OFF, 将控制权转交给 PEI, 并将 EFI_SEC_PEI_HAND_OFF 传递给 PEI。

不同的硬件平台, SEC 代码会有不同的实现方式, 但大致执行过程相似。下面以 OVMF (具体介绍参见第 2 章) 为例, 介绍 SEC 功能区的执行过程。

```

# file: OvmfPkg/Sec/X64/SecEntry.S
ASM_PFX(_ModuleEntryPoint):
# 临时 RAM 已经初始化, 设置栈地址。PcdOvmfSecPeiTempRamBase 和 PcdOvmfSecPeiTempRamSize
  在 OvmfPkgIa32X64.fdf 中定义
# 分别为 0x010000 和 0x008000
.set SEC_TOP_OF_STACK, FixedPcdGet32 (PcdOvmfSecPeiTempRamBase) +
FixedPcdGet32 (PcdOvmfSecPeiTempRamSize)
movq $SEC_TOP_OF_STACK, %rsp
movq %rbp, %rcx#rcx: BFV 首地址, rbp 为传入参数
movq %rsp, %rdx#rdx: 栈起始地址
subq $0x20, %rsp
call ASM_PFX(SecCoreStartupWithStack)# 此时栈已可用, 故可使用 call 指令

```



```

# file: OvmfPkg/Sec/X64/SecEntry.S
VOID EFI_API SecCoreStartupWithStack(
IN EFI_FIRMWARE_VOLUME_HEADER *BootFv,

```

```

IN VOID *TopOfCurrentStack
){
    EFI_SEC_PEI_HAND_OFF SecCoreData;
    // 初始化浮点寄存器
    // 初始化 IDT
    // 初始化 SecCoreData, 将临时 RAM 地址、栈地址、BFV 地址赋值给 SecCoreData
    SecStartupPhase2(&SecCoreData);
}

```



```

# file: OvmfPkg/Sec/X64/SecEntry.S
VOID EFI_API SecStartupPhase2(IN VOID *Context)
{
    EFI_SEC_PEI_HAND_OFF      SecCoreData;
    EFI_PEI_CORE_ENTRY_POINT  PeiCoreEntryPoint;
    SecCoreData = (EFI_SEC_PEI_HAND_OFF *) Context;
    // 从 BFV 中找出 PEI 的入口函数
    FindAndReportEntryPoints (&SecCoreData->BootFirmwareVolumeBase,
    &PeiCoreEntryPoint);
    // 调用 PEI 入口函数, SecCoreData 包含了临时 RAM、栈、BFV 地址和大小,
    // mPrivateDispatchTable 包含了 EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI
    (*PeiCoreEntryPoint) (SecCoreData,
    (EFI_PEI_PPI_DESCRIPTOR *)&mPrivateDispatchTable);
}

```

2. PEI 阶段

PEI (Pre-EFI Initialization) 阶段资源仍然十分有限, 内存到了 PEI 后期才被初始化, 其主要功能是为 DXE 准备执行环境, 将需要传递到 DXE 的信息组成 HOB (Handoff Block) 列表, 最终将控制权转交到 DXE 手中。PEI 执行流程如图 1-5 所示。

从功能上讲, PEI 可分为以下两部分。

- ❑ PEI 内核 (PEI Foundation): 负责 PEI 基础服务和流程。
- ❑ PEIM (PEI Module) 派遣器: 主要功能是找出系统中的所有 PEIM, 并根据 PEIM 之间的依赖关系按顺序执行 PEIM。PEI 阶段对系统的初始化主要是由 PEIM 完成的。

每个 PEIM 是一个独立的模块, 模块的入口函数类型定义如下所示:

```

typedef EFI_STATUS(EFI_API *EFI_PEIM_ENTRY_POINT2)(
    IN EFI_PEI_FILE_HANDLE FileHandle, IN CONST EFI_PEI_SERVICES **PeiServices
);

```

通过 PeiServices, PEIM 可以使用 PEI 阶段提供的系统服务, 通过这些系统服务, PEIM 可以访问 PEI 内核。PEIM 之间的通信通过 PPI (PEIM-to-PEIM Interfaces) 完成。

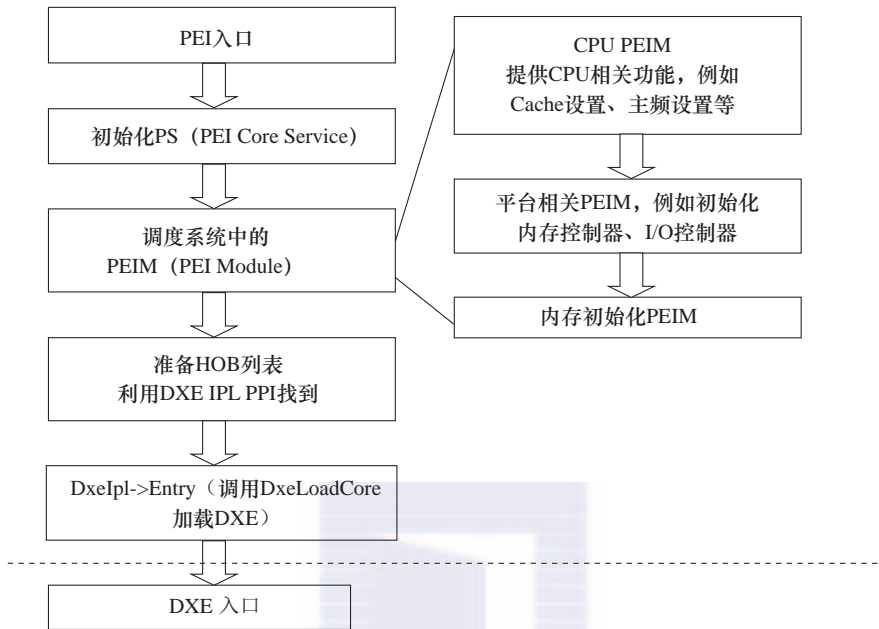


图 1-5 PEI 执行流程

PPI 与 DXE 阶段的 Protocol 类似，每个 PPI 是一个结构体，包含了函数指针和变量，例如：

```

struct _EFI_PEI_DECOMPRESS_PPI {
    EFI_PEI_DECOMPRESS_DECOMPRESS Decompress;
}
extern EFI_GUID    gEfiPeiDecompressPpiGuid;
  
```

每个 PPI 都有一个 GUID。根据 GUID，通过 PeiServices 的 LocatePpi 服务可以得到 GUID 对应的 PPI 实例。

UEFI 的一个重要特点是其模块化的设计。模块载入内存后生成 Image。Image 的入口函数为 `_ModuleEntryPoint`。PEI 也是一个模块，PEI Image 的入口函数 `_ModuleEntryPoint`，位于 `MdePkg/Library/PeimEntryPoint/PeimEntryPoint.c`。`_ModuleEntryPoint` 最终调用 PEI 模块的入口函数 `PeiCore`，位于 `MdeModulePkg/Core/Pei/PeiMain/PeiMain.c`。进入 `PeiCore` 后，首先根据从 SEC 阶段传入的信息设置 Pei Core Services，然后调用 `PeiDispatcher` 执行系统中的 PEIM，当内存初始化后，系统会发生栈切换并重新进入 `PeiCore`。重新进入 `PeiCore` 后使用的内存为我们所熟悉的内存。所有 PEIM 都执行完毕后，调用 PeiServices 的 `LocatePpi` 服务得到 DXE IPL PPI，并调用 DXE IPL PPI 的 `Entry` 服务，这个 `Entry` 服务实际上是 `DxeLoadCore`，它找出 DXE Image 的入口函数，执行 DXE Image 的入口函数并将 HOB 列表传递给 DXE。

3. DXE 阶段

DXE (Driver Execution Environment) 阶段执行大部分系统初始化工作, 进入此阶段时, 内存已经可以被完全使用, 因而此阶段可以进行大量的复杂工作。从程序设计的角度讲, DXE 阶段与 PEI 阶段相似, 执行流程如图 1-6 所示。

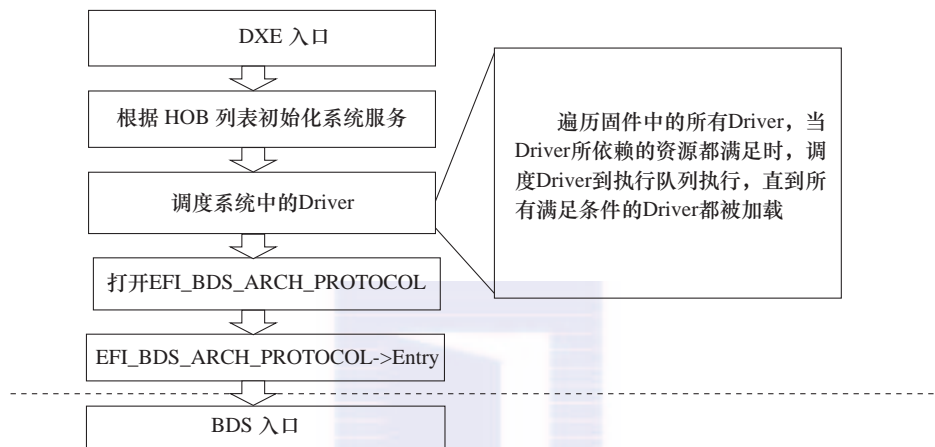


图 1-6 DXE 执行流程

与 PEI 类似, 从功能上讲, DXE 可分为以下两部分。

❑ DXE 内核: 负责 DXE 基础服务和执行流程。

❑ DXE 派遣器: 负责调度执行 DXE 驱动, 初始化系统设备。

DXE 提供的基础服务包括系统表、启动服务、Run Time Services。

每个 DXE 驱动是一个独立的模块, 模块入口函数类型定义为:

```

typedef EFI_STATUS(EFI_API *EFI_IMAGE_ENTRY_POINT)(
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
);
  
```

DXE 驱动之间通过 Protocol 通信。Protocol 是一种特殊的结构体, 每个 Protocol 对应一个 GUID, 利用系统 BootServices 的 OpenProtocol, 并根据 GUID 来打开对应的 Protocol, 进而使用这个 Protocol 提供的服务。

当所有的 Driver 都执行完毕后, 系统完成初始化, DXE 通过 EFI_BDS_ARCH_PROTOCOL 找到 BDS 并调用 BDS 的入口函数, 从而进入 BDS 阶段。从本质上讲, BDS 是一种特殊的 DXE 阶段的应用程序。

4. BDS 阶段

BDS (Boot Device Selection) 的主要功能是执行启动策略, 其主要功能包括:

- ❑ 初始化控制台设备。
- ❑ 加载必要的设备驱动。
- ❑ 根据系统设置加载和执行启动项。

如果加载启动项失败，系统将重新执行 DXE dispatcher 以加载更多的驱动，然后重新尝试加载启动项。

BDS 策略通过全局 NVRAM 变量配置。这些变量可以通过运行时服务的 `GetVariable()` 读取，通过 `SetVariable()` 设置。例如，变量 `BootOrder` 定义了启动顺序，变量 `Boot####` 定义了各个启动项（#### 为 4 个十六进制大写符号）。

用户选中某个启动项（或系统进入默认的启动项）后，OS Loader 启动，系统进入 TSL 阶段。

5. TSL 阶段

TSL（Transient System Load）是操作系统加载器（OS Loader）执行的第一阶段，在这一阶段 OS Loader 作为一个 UEFI 应用程序运行，系统资源仍然由 UEFI 内核控制。当启动服务的 `ExitBootServices()` 服务被调用后，系统进入 Run Time 阶段。

TSL 阶段之所以称为临时系统，在于它存在的目的就是为操作系统加载器准备执行环境。虽然是临时系统，但其功能已经很强大，已经具备了操作系统的雏形，UEFI Shell 是这个临时系统的人机交互界面。正常情况下，系统不会进入 UEFI Shell，而是直接执行操作系统加载器，只有在用户干预下或操作系统加载器遇到严重错误时才会进入 UEFI Shell。

6. RT 阶段

系统进入 RT（Run Time）阶段后，系统的控制权从 UEFI 内核转交到 OS Loader 手中，UEFI 占用的各种资源被回收给 OS Loader，仅有 UEFI 运行时服务保留给 OS Loader 和 OS 使用。随着 OS Loader 的执行，OS 最终取得对系统的控制权。

7. AL 阶段

在 RT 阶段，如果系统（硬件或软件）遇到灾难性错误，系统固件需要提供错误处理和灾难恢复机制，这种机制运行在 AL（After Life）阶段。UEFI 和 UEFI PI 标准都没有定义此阶段的行为和规范^①。

1.3 本章小结

相对 BIOS，UEFI 有更好的可编程性，强大的可扩展性，出色的安全性，并且其设计更

① http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=PI_Boot_Flow。

能适应 64 位平台。这些优势使得 UEFI 可以迅速取代 BIOS。

UEFI 定义了操作系统和平台固件之间的接口。UEFI 接口可分为以下两个部分。

1) 启动服务：启动服务的主要服务对象是操作系统加载器以及其他 UEFI 应用程序和 UEFI 驱动。操作系统加载器通过启动服务逐步取得对整个计算机系统资源的控制。当加载器完全控制计算机软硬件资源后，系统结束启动服务，进入运行时。启动服务主要包括：事件服务、内存管理、Protocol 管理、Protocol 使用类服务、驱动管理、Image 管理以及 ExitBootServices 服务。

2) 运行时服务：运行时服务的主要服务对象是操作系统、操作系统加载器以及 UEFI 应用和 UEFI 驱动。运行时服务主要包括：时间服务、读写 UEFI 系统变量的服务、虚拟内存服务、重启系统的服务。

基于 UEFI 的计算机系统，从启动到关机可以分为 7 个阶段，本书分别对这 7 个阶段的功能和执行流程进行了简单介绍。启动服务和运行时服务只有在系统进入 DXE 阶段后才生成。本书重点讲述的 UEFI 应用程序和 UEFI 驱动就是运行在这一阶段。

通过本章的学习，读者应该对 UEFI 有了初步的认识。下一章主要介绍 UEFI 开发环境的搭建。

UEFI 开发环境搭建

通过前面的学习，我们已经知道 UEFI 是一种标准，它没有给出具体的实现。软件厂商可以根据 UEFI 标准开发自己的 UEFI 实现，其中常用的开源实现是 EDK2。EDK2 是遵循 UEFI 标准和 PI 标准的跨平台固件开发环境。UEFI 的目标是完全取代 BIOS，因而它要能完全支持所有类型的 CPU^①，并让所有的硬件厂商接受这种变化。来自不同厂商的开发者使用各种不同的开发环境开发自己的产品。为了让这些不同的开发者愉快地接受 EDK2 来开发自己平台上的 UEFI 固件或应用，EDK2 对每种平台都提供了对应的开发工具。EDK2 支持在多种操作系统下的开发，例如 Windows、Linux、Darwin、UNIX 等，并支持跨平台编译，如在 Windows 开发环境下可以编译出 Arm 平台上的 UEFI 应用程序。下面我们分别介绍在 Windows 和 Linux 下如何使用 EDK2 进行开发。

2.1 配置 Windows 开发环境

EDK2 目前支持 Windows 7、Windows 8、Windows 8.1。开发 UEFI 应用和驱动之前要建立开发环境，分为如下几步。

1) 首先需安装 EDK2 依赖的开发工具：Windows SDK、C 编译器、IASL 编译器，然后下载 EDK2 源码，EDK2 源码包里包含了开发所需的源码和工具。

2) 配置 EDK2：主要是设置开发工具的路径。然后就可以通过 EDK2 提供的源码和工

① 目前 EDK2 支持 x86（32 位和 64 位）CPU、安腾 CPU 和 Arm CPU。

具开发 UEFI 应用和驱动。

3) 编译 UEFI 模拟器和 UEFI 工程。

4) 运行 UEFI 模拟器。

下面详细介绍动手开发之前必须进行的这几个步骤。

2.1.1 安装所需开发工具

首先需要安装 EDK2 所依赖的开发工具以及 EDK2 本身。以下是主要的安装步骤。

1) 安装 Windows SDK。Windows SDK 可从如下地址下载：

<http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=24826>

2) 安装 C 编译器。推荐安装微软公司的 Visual Studio 编译器或英特尔公司的 ICC 编译器，也可以安装 Cygwin 及其 gcc 编译器。

3) 安装 IASL 编译器 (https://www.acpica.org/downloads/binary_tools)。

4) IASL 用于编译 .asl 文件。asl 是高级配置与电源接口 (Advanced Configuration and Power Interface) 源文件。

5) 下载 EDK2 开发包：可以从 TianoCore 官方网站下载 EDK2 发行版 (https://sourceforge.net/projects/edk2/files/UDK2014_Releases/UDK2014/UDK2014.Complete.MyWorkSpace.zip/download)，也可以用 subversion 工具或命令行获得最新源码。下面是两个简单的示例。

1) 用 subversion 命令行获得 EDK2 源码：

```
svn co https://svn.code.sf.net/p/edk2/code/trunk/edk2
```

2) 或者使用 TortoiseSvn 下载源码，如图 2-1 所示。

2.1.2 配置 EDK2 开发环境

下面讲解如何配置 EDK2 开发环境。

1) 首先进入 EDK2 目录并运行 edksetup.bat，此步骤用于建立 Conf 目录下的 target.txt、tools_def.txt 等文件。

```
C:\>EDK2\edksetup.bat
```

2) 编辑 Conf\target.txt。

根据用户的编译器环境修改编译工具 TOOL_CHAIN_TAG。此处以 x86_64 平台的 32 位的 Visual Studio 2008 为例，需设置 TOOL_CHAIN_TAG=VS2008x86。

图 2-2 是设置了 TOOL_CHAIN_TAG 为 VS2008x86 的 target.txt 文件。

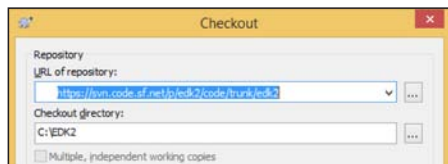


图 2-1 用 TortoiseSvn 下载 EDK2 源码

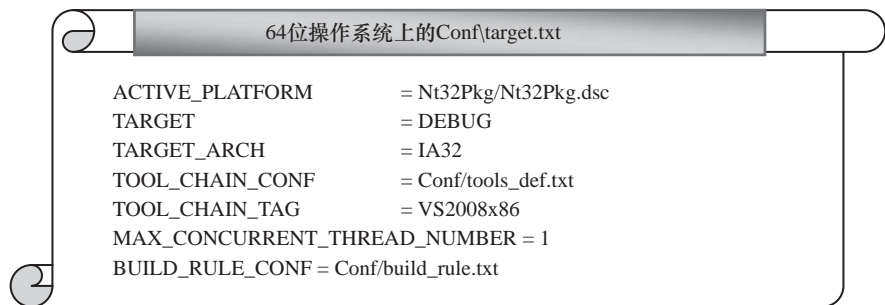



图 2-2 Conf\target.txt 配置文件

 **注意** TOOL_CHAIN_TAG 的设置要根据 Visual Studio 的安装路径来确定。

IA32 平台下 TOOL_CHAIN_TAG 的设置见表 2-1。

表 2-1 IA32 平台下的 TOOL_CHAIN_TAG

Visual Studio 版本	TOOL_CHAIN_TAG 的值
Visual Studio 2003 (VC7)	VS2003
Visual Studio 2005 (VC8)	VS2005
Visual Studio 2008 (VC9.0)	VS2008
Visual Studio 2010 (VC10.0)	VS2010

AMD64 平台下的 TOOL_CHAIN_TAG 的设置见表 2-2。

表 2-2 AMD64 平台下的 TOOL_CHAIN_TAG

Visual Studio 版本	TOOL_CHAIN_TAG 的值 (64 位 VS)	TOOL_CHAIN_TAG 的值 (32 位 VS)
Visual Studio 2003 (VC7)	VS2003	VS2003x86
Visual Studio 2005 (VC8)	VS2005	VS2005x86
Visual Studio 2008 (VC9.0)	VS2008	VS2008x86
Visual Studio 2010 (VC10.0)	VS2010	VS2010x86

如果用户安装的是 ICC 编译器，则需要将 TOOL_CHAIN_TAG 设置为 ICC 或 ICC11，具体配置需根据 ICC 版本号确定。例如，安装了 ICC11 编译器，则做如下设置：

```
TOOL_CHAIN_TAG = ICC11
```

如果用户安装的是 Cygwin，则需要将 TOOL_CHAIN_TAG 设置为 CYGWIN：

```
TOOL_CHAIN_TAG = CYGWIN
```

3) 检查 Conf\tools_def.txt，确保编译器路径正确。

tools_def.txt 工具为 EDK2 自动生成的文件，里面预定义了几种常用的编译器。

例如，我们在步骤2中设置了 `TOOL_CHAIN_TAG` 为 `VS2008x86`，在 `tools_def.txt` 查找 `VS2008x86_BIN`，看其路径是否正确，如果路径不正确，需设置为 `cl.exe` 的正确路径。

图2-3为 `tools_def.txt` 文件中关于 `VS2008x86` 的部分，其中 `DEFINE VS2008x86_BIN` 定义了编译器 `cl.exe` 的路径。当编译32位EDK程序时，`build` 工具会使用 `VS2008x86_BIN` 下的 `cl` 编译器；当编译 `x86_64` 程序时，会使用 `VS2008x86_BINX64` 下的 `cl` 编译器。

`DEFINE WIN_ASX_BIN_DIR` 定义了 `IASL` 编译器的路径。

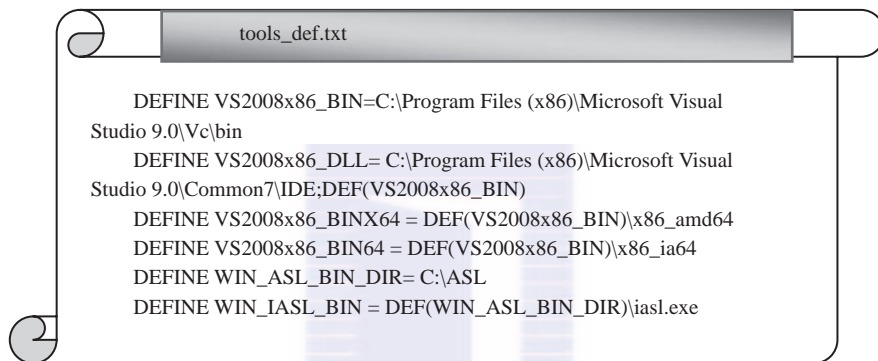


图2-3 `conf\tools_def.txt` 文件 `VS2008x86` 部分，该文件定义了编译器的路径

2.1.3 编译UEFI模拟器和UEFI工程

下面讲解如何利用EDK2提供的工具链进行编译，并对其中的重要参数进行讲解。

编译UEFI代码是在CMD命令行中通过运行EDK2工具链命令完成的，分两种情况：一是编译 `Nt32Pkg` 工程，二是编译非 `Nt32Pkg` 的UEFI工程。

1. 编译UEFI模拟器，即 `Nt32Pkg`

首先需要运行 `edksetup.bat--nt32` 以设置EDK2环境变量，如下所示：

```
Setting environment for using Microsoft Visual Studio 2008 x86 tools.
C:\Program Files(x86)\Microsoft Visual Studio 9.0\VC>cd c:\EDK2
C:\EDK2>edksetup.bat --nt32
```

设置好环境变量后，就可以使用EDK2的工具链编译UEFI模拟器了。编译UEFI模拟器的命令非常简单，在命令行执行 `build` 命令即可，如下所示：

```
C:\EDK2> build
```

`build` 命令相当于Visual Studio的 `nmake` 命令，它分析UEFI的工程文件，根据分析结果

自动执行相应编译和连接命令。

不带参数的 build 命令等同于 build -a IA32 -p Nt32Pkg\Nt32Pkg.dsc，这是因为 build 命令使用了 conf\target.txt 中定义的默认参数 TARGET_ARCH（对应 -a 参数）和 ACTIVE_PLATFORM（对应 -p 参数）。

2. 编译非 Nt32Pkg 工程

首先设置环境变量，打开 Visual Studio 2008 command prompt，进入 EDK2 目录并运行 edksetup.bat，如下所示：

```
Setting environment for using Microsoft Visual Studio 2008 x86 tools.
C:\Program Files(x86)\Microsoft Visual Studio 9.0\VC>cd c:\EDK2
C:\>EDK2>edksetup.bat
```

设置好环境变量后，就可以使用 EDK2 提供的 build 工具编译 UEFI 代码了。例如，要编译 MdePkg：

```
C:\EDK2> build -a X64 -p MdePkg\MdePkg.dsc
```

3. build 命令

build 命令是编译 UEFI 工程常用的命令。它有三个重要参数：-a、-p 和 -m。

- ❑ -a 用来选择目标平台。可供选择的选项有 IA32（32 位 x86 CPU）、X64（64 位 x86_64 CPU）、IPF（Itanium Processor Family）、ARM 和 EBC（EFI byte code）；默认的参数在 Conf/target.txt 中设置。
- ❑ -p 用来指定要编译的 package 或 Platform。-p 的参数是这个 package 或 Platform 的 .dsc 文件。默认的参数在 Conf/target.txt 中设置。
- ❑ -m 用来指定要编译的模块。如果不指定 -m 选项，build 将编译 .dsc 文件指定的所有模块。

例如，编译 32 位的 Shell 后，Shell.efi 存放于 edk2\Build\ShellPkg\DEBUG_VS2008x86\IA32\。

```
C:\EDK2> build -a IA32 -p ShellPkg\ShellPkg.dsc -m ShellPkg/Application/Shell/Shell.inf
```

编译 64 位的 Shell 后，Shell.efi 存放于 edk2\Build\ShellPkg\DEBUG_VS2008x86\X64\。

```
C:\EDK2> build -a X64 -p ShellPkg\ShellPkg.dsc -m ShellPkg/Application/Shell/Shell.inf
```

表 2-3 列出了 build 命令的常用参数及用法。

表 2-3 build 命令的常用参数

build 命令参数	参数用法
-a ARCH	选择目标平台, ARCH 可以是 IA32、X64、IPF、ARM 或 EBC, 该选项将会取代 Conf\target.txt 文件中的 TARGET_ARCH
-DMACROS	定义宏, 例如 -D NETWORK_ENABLE
-h	显示帮助信息
-j LOGFILE	将编译信息输出到文件
-b TARGET	选择编译成 DEBUG 还是 RELEASE 例如: -b DEBUG -b RELEASE
-t TOOLCHAIN	选择 tools_def.txt 中定义的编译工具, 例如要使用 Visual Studio 2010: -t vs2010
-n ThreadNumber	编译器使用的线程数量
-p PlatformFile	通过指定 .dsc 文件指定要编译的 Package, 该选项将会取代 Conf\target.txt 文件中的 ACTIVE_PLATFORM。例如, 要编译 AppPkg, 可以使用如下选项: -p AppPkg\AppPkg.dsc
-m ModuleFile	指定要编译的模块, build 工具将只编译此模块
-q	编译过程中只显示严重错误信息
-s	使用沉默模式执行 make 或 nmake
-u	跳过 AutoGen 这一步
-c	文件名不区分大小写

2.1.4 运行模拟器

使用以下命令运行 UEFI 模拟器:

```
C:\EDK2> build run
```

因为在 target.txt 中已经设置了 TARGET_ARCH 与 ACTIVE_PLATFORM, 所以 build run 与 build -a IA32 -p Nt32Pkg\Nt32Pkg.dsc run 命令等同, 可以用来运行 UEFI 模拟器。或者直接运行 Build\NT32\DEBUG_VS2008x86\IA32 目录下的 SecMain.exe。

通常模拟器最终会进入 UEFI Shell, 有关 UEFI Shell 内容我们会在第 16 章详细讲述。EDK2 默认将目录 C:\edk2\Build\NT32\DEBUG_VS2008x86\IA32\ 映射为文件系统 FSNT0, 在 Shell 中执行命令 FSNT0:, Shell 将会打开 FSNT0 分区并将当前目录切换到 FSNT0 分区的根目录, 如下列代码所示:

```
Shell>FSNT0:
FSNT0:\>
```

执行 “?” 命令将会列出所有 Shell 支持的命令，如图 2-4 所示。

```
FSNT0:\> ?
alias          - Displays, Creates, or deletes UEFI Shell aliases.
attrib         - Displays or changes the attributes of files or directories.
...
```

图 2-4 Shell 中的 “?” 命令

图 2-5 显示了 UEFI 模拟器启动时的界面。图 2-6 显示了 UEFI 模拟器进入 Shell 时的界面。



图 2-5 NT32 UEFI 模拟器启动界面图

图 2-5 中第二行显示了 Current running mode 1.1.2，这说明 UEFI Shell 是 EDK2 开发包中预先编译的 Shell。我们可以用如下方式进入刚才编译的 Shell。

1) 将 Shell.efi 文件从 edk2\Build\ShellPkg\DEBUG_VS2008x86\IA32\ 复制到 Build\NT32\DEBUG_VS2008x86\IA32 目录。

2) 在模拟器窗口的 Shell 中执行以下命令：

```
Shell>FSNT0:
FSNT0:\>shell
```

执行完毕后将会进入新的 UEFI Shell，新 Shell 如图 2-7 所示。

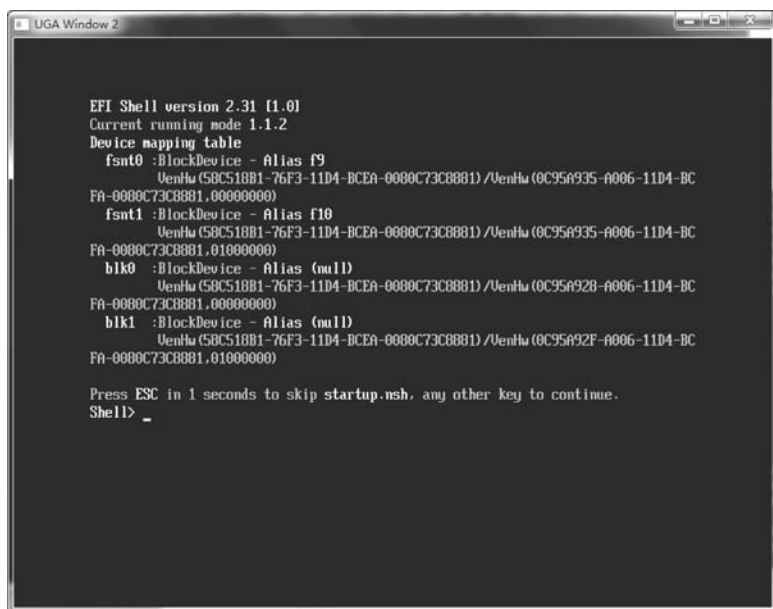


图 2-6 NT32 UEFI 模拟器 Shell 界面

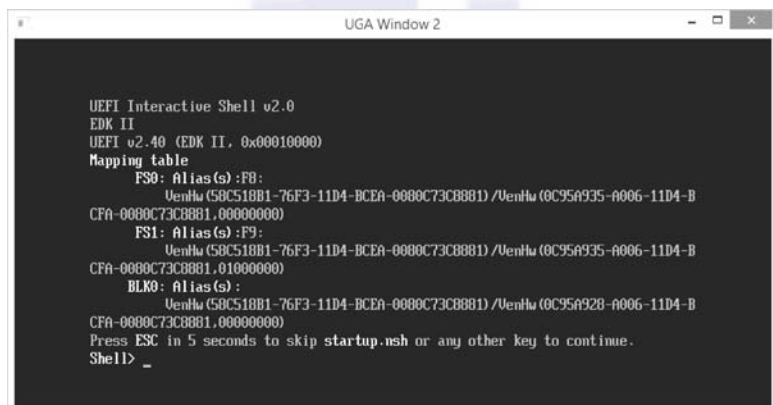


图 2-7 UEFI 2.4 标准下的 UEFI Shell

2.2 配置 Linux 开发环境

与配置 Windows 开发环境相似，配置 Linux 开发环境也包括如下步骤。

1) 安装 gcc 编译器，并下载 EDK2 源码。

2) 配置 EDK2，包括编译 EDK2 工具链和设置编译器路径。在配置 Windows 开发环境时，我们没有编译 EDK2 工具链，因为 EDK2 源码包中包含了 Windows 下的 EDK2 工具链可执行文件。但是，由于 Linux 发行版众多，因此，EDK2 源码包中没有包含 EDK2 工具链

可执行文件。然后就可以通过 EDK2 提供的源码和工具开发 UEFI 应用和驱动了。

3) 编译 UEFI 模拟器和 UEFI 工程。

4) 运行 UEFI 模拟器。

下面详细介绍配置 Linux 开发环境的这几个步骤。

2.2.1 安装所需开发工具

1) 安装 gcc: 在 Ubuntu 下可以使用如下命令安装 gcc。

```
EDK2:$apt-get install gcc
```

2) 下载 EDK2 开发包。

① 可以从 TianoCore 官方网站下载 EDK2 发行版:

[https://sourceforge.net/projects/edk2/files/UDK2014_Releases/UDK2014/UDK2014.Complete.](https://sourceforge.net/projects/edk2/files/UDK2014_Releases/UDK2014/UDK2014.Complete.MyWorkspace.zip/download)

MyWorkspace.zip/download

② 也可以用代码管理工具获得最新源码。

3) 用 subversion 命令行获得 EDK2 源码:

```
EDK2:$svn co https://svn.code.sf.net/p/edk2/code/trunk/edk2edk2
```

或用 git 命令行下载源码:

```
EDK2:$git clone https://github.com/tianocore/edk2
```

2.2.2 配置 EDK2 开发环境

1) 编辑 Conf/target.txt。根据用户的编译器环境修改编译工具 TOOL_CHAIN_TAG。此处以 x86_32 Ubuntu 的 gcc 4.4 为例, 需设置 TOOL_CHAIN_TAG=GCC44, 如图 2-8 所示。

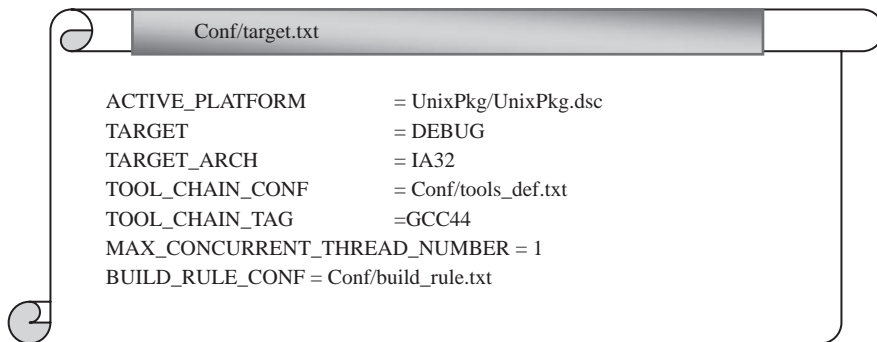


图 2-8 Conf/target.txt 文件, 操作系统为 x86_32 Ubuntu, 编译器为 gcc 4.4

2) 检查 Conf\tools_def.txt (见图 2-9), 确保编译器路径正确。

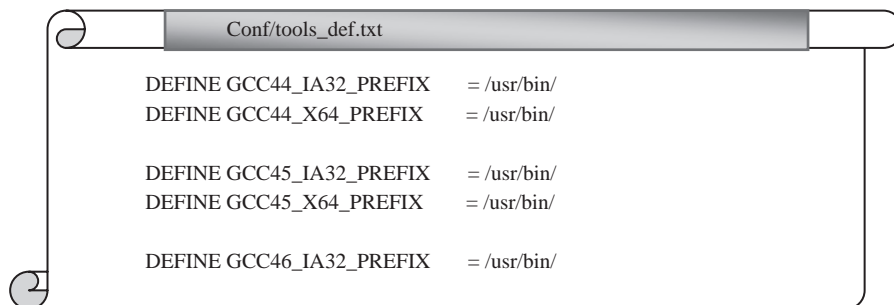


图 2-9 Conf\tools_def.txt 文件 GCC44 相关部分, 该文件定义了编译器的路径

3) 编译 EDK2 工具链。

Linux 环境里 EDK2 工具链位于 BaseTools/BinWrappers/PosixLike/ 目录下。EDK2 工具链包括 build、GenFv、GenFw 等工具。如果该目录下没有这些工具, 就要编译 BaseTools 来获得这些工具。

要编译 BaseTools, 首先需保证系统中已经安装了 make、gcc 等编译工具。下面的命令用于安装编译 BaseTools 所需的编译工具:

```
EDK2$ sudo apt-get install build-essential uuid-dev
```

然后进入 BaseTools 目录并使用 make 命令编译, 如下所示:

```
EDK2$ cd BaseTools
EDK2/BaseTools$ make
```

编译后, BaseTools/BinWrappers/PosixLike/ 目录如图 2-10 所示。



图 2-10 Linux 环境下 EDK2 工具链

2.2.3 编译 UEFI 模拟器和 UEFI 工程

UEFI 开发环境已经配置完毕, 下面就可以编译 UEFI 模拟器和 UEFI 应用程序了。

1) 首先设置环境变量。打开命令行工具, 并进入 EDK2 目录, 然后执行 source edksetup.sh, 如下所示:

```
EDK2$ source edksetup.sh
```

2) 环境变量配置好后就可以编译 UEFI 模拟器和其他 UEFI 工程了。

① 编译 UEFI 模拟器。

Linux 下的模拟器是 UnixPkg，编译模拟器就是编译 UnixPkg。可以使用 build 命令或带 -p 参数的 build 命令编译 UnixPkg，如下所示：

```
EDK2$build
```

或

```
EDK2$build -p UnixPkg/UnixPkg.dsc
```

编译后，模拟器 SecMain 将输出到 EDK2/Build/Unix/DEBUG_GCC44/IA32 目录。

build 命令的用法在 2.1 节配置 Windows 开发环境时已经做了详细说明，Linux 环境下 build 命令与之完全相同，在此不再重复说明。

② 编译其他 UEFI 工程。

编译其他 UEFI 工程也是通过 build 命令完成的。下面是使用 build 命令编译 Shell.efi 的示例。

【示例 2-1】 用如下命令可以编译 32 位的 Shell。编译后，Shell.efi 存放于 Build/Unix/DEBUG_GCC44/IA32/ 目录下。

```
EDK2$build -a IA32 -p ShellPkg/ShellPkg.dsc -m ShellPkg/Application/Shell/Shell.inf
```

【示例 2-2】 用如下命令可以编译 64 位的 Shell。编译后，Shell.efi 存放于 Build/Unix/DEBUG_GCC44/X64/ 目录下。

```
EDK2$build -a X64 -p ShellPkg/ShellPkg.dsc -m ShellPkg/Application/Shell/Shell.inf
```

2.2.4 运行模拟器

有两种方式可以运行 UEFI 模拟器。

第一种方式是在 Linux 图形界面的命令行中运行 build run 命令。

```
EDK2$build run
```

第二种方式是执行 EDK2/Build/Unix/DEBUG_GCC44/IA32 目录下的 SecMain 命令，如下所示。

```
EDK2$Build/Unix/DEBUG_GCC44/IA32/SecMain
```

图 2-11 是 Linux 下的 UEFI 模拟器。

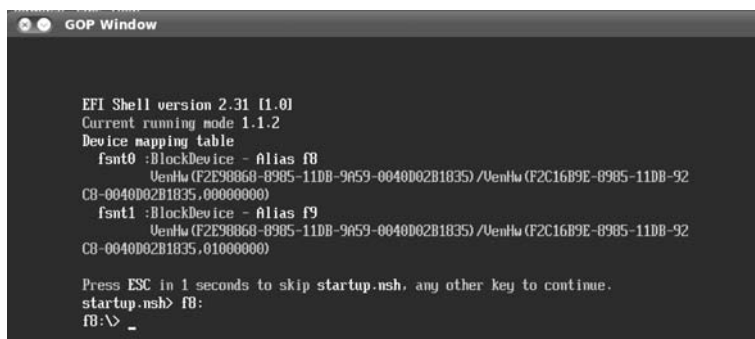


图 2-11 Linux 下的 UEFI 模拟器

2.3 OVMF 的制作和使用

OVMF(Open Virtual Machine Firmware, 开放虚拟机固件)是用于虚拟机上的 UEFI 固件。在开发过程中,我们需要不断地测试所开发的产品。在模拟器中测试非常方便,但模拟器功能有限,并且模拟器只能测试 32 位程序。另外,在真实的 UEFI 环境中,测试又往往比较烦琐。在虚拟机中测试无疑是一种方便、快捷的方式,它既能较好地模拟真实环境,又可以做到快速、方便。EDK2 提供了制作虚拟机固件的方法,称为 OVMF。下面介绍如何编译和使用虚拟机固件。

1. 制作 OVMF

编译 OVMF 包,分如下两种情况。

1) 用如下命令编译 64 位 OVMF 固件。

```
C:\EDK2>build -a X64 -p OvmfPkg\OvmfPkgX64.dsc
```

编译后的固件为 edk2\Build\OvmfX64\DEBUG_VS2008x86\FV\ 目录下的 OVMF.fd。

2) 用如下命令编译 32 位 OVMF 固件。

```
C:\EDK2>build -a IA32 -p OvmfPkg\OvmfPkgIa32.dsc
```

编译后的固件为 edk2\Build\OvmfIa32\DEBUG_VS2008x86\FV\ 目录下的 OVMF.fd。

2. 在 QEMU 虚拟机中使用 OVMF

QEMU 是目前广泛使用的计算机仿真器和虚拟机。在 QEMU 虚拟机中,用户可以使用自定义的固件,利用这个特性我们可以测试 OVMF。首先,将存放于 edk2\Build\OvmfPkgIa32\DEBUG_VS2008x86\FV 下的文件 OVMF.fd 复制到 QEMU 的安装目录。然

后，在 QEMU 虚拟机中加载 OVMF，有两种方式：一种是使用 QEMU Manager（管理器）运行 QEMU 虚拟机，在图形界面中指定 OVMF 固件；另一种是使用 QEMU 命令行指定 OVMF 固件。

（1）使用 QEMU Manager

在 QEMU Manager 控制面板选择 Advanced → BIOS Filename，然后选择 OVMF.fd，如图 2-12 所示。

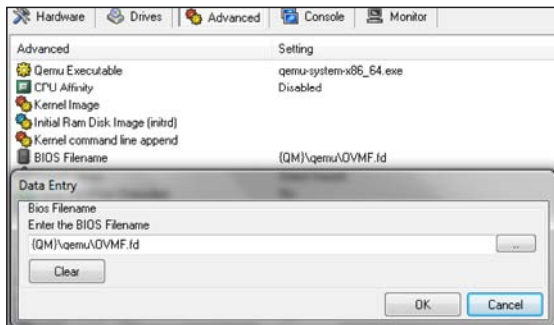


图 2-12 在 QEMU Manager 中选择固件 OVMF.fd

运行虚拟机，运行后将进入 UEFI Shell 界面，如图 2-13 所示。

（2）在 CMD 命令行运行 QEMU 命令

在 CMD 命令行运行如下 QEMU 命令。

```
C:\Program files\Qemu>qemu-system-x86_64.exe-bios "OVMF.fd" -M "pc" -m 256 -cpu
"qemu64" -vga cirrus -serial vc -parallel vc -name "UEFI" -boot order=dc
```

该命令运行后同样会进入图 2-13 所示的 UEFI Shell 界面。

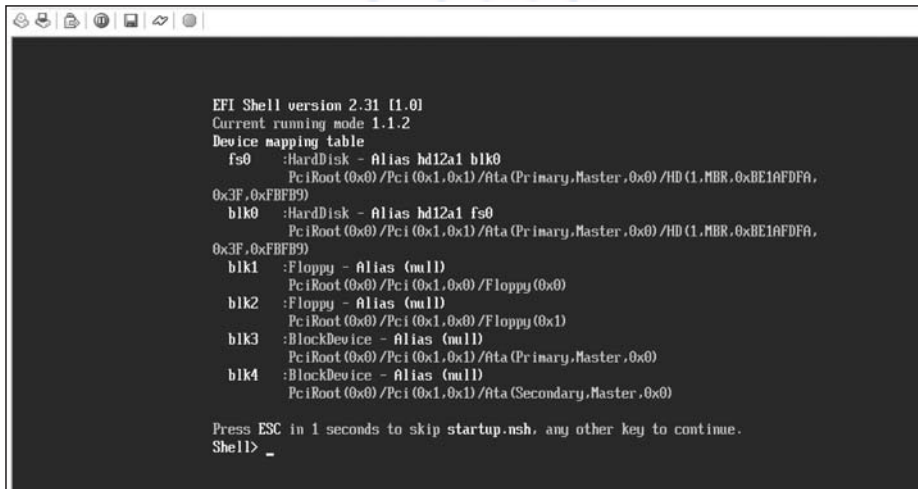


图 2-13 基于固件 OVMF.fd 的虚拟机

2.4 UEFI 的启动

虽然现在已经进入了 UEFI 时代，但是因为 UEFI 刚刚开始取代 BIOS，目前依然有很多运行着的 BIOS 系统。为了方便开发者从 BIOS 转移到 UEFI，EDK2 提供了 DUET，用于在 BIOS 系统上模拟 UEFI 执行环境。下面介绍一下 DUET。

1. DUET 简介

DUET (Developer's UEFI Emulation) 是基于 Legacy BIOS 系统的 UEFI 模拟器，主要为 UEFI 开发者提供一个在传统 BIOS 系统上的 UEFI 运行环境。DUET 支持基于 MBR 的启动方式，从 MBR 启动后进入 UEFI 执行环境。

下面开始制作传统 BIOS 平台下的模拟 UEFI 启动盘。

如果目标平台是 Legacy BIOS，则需要在 U 盘中制作 MBR 和引导文件，可按如下步骤来进行。

1) 编译 DuetPkg，在 CMD 命令行输入如下命令。

```
C:\EDK2>build -a IA32 -p DuetPkg\DuetPkgIa32.dsc
```

或者

```
C:\EDK2>build -a X64 -p DuetPkg\DuetPkgX64.dsc
```

2) 通过如下命令生成引导文件。

```
C:\EDK2> cd DuetPkg
C:\EDK2\DuetPkg>postbuild.bat Ia32
```

或者

```
C:\EDK2\DuetPkg>postbuild.bat X64
```

3) 插入 U 盘，假设 J: 是 U 盘盘符，通过如下命令向 U 盘写入 MBR。

```
C:\EDK2\DuetPkg> createbootdisk usb J: FAT32 IA32
```

或者

```
C:\EDK2\DuetPkg> createbootdisk usb J: FAT32 X64
```

4) 拔出并重新插入 U 盘，通过如下命令向 U 盘复制 UEFI 文件。

```
C:\EDK2\DuetPkg> createbootdisk usb J: FAT32 IA32 step2
```

或者

```
C:\EDK2\DuetPkg> createbootdisk usb J: FAT32 X64 step2
```

此命令向 U 盘根目录复制了 `efldr20` (该文件用于引导系统进入 UEFI 环境), 并向 `efi\boot` 目录复制了引导文件 `bootia32.efi` (源文件位于 `ShellBinPkg\UefiShell\Ia32\Shell.efi`) 或 `bootx64.efi` (源文件位于 `ShellBinPkg\UefiShell\X64\Shell.efi`)。

总结一下, 要制作传统 BIOS 平台下的模拟 UEFI 启动盘 (64 位), 需在 CMD 命令行执行如下命令。

```
C:\EDK2>build -a X64 -p DuetPkg\DuetPkgX64.dsc
C:\EDK2> cd DuetPkg
C:\EDK2\DuetPkg>postbuild.bat X64
C:\EDK2\DuetPkg> createbootdisk usb J: FAT32 X64
```

拔出并重新插入 U 盘, 继续执行如下命令。

```
C:\EDK2\DuetPkg> createbootdisk usb J: FAT32 X64 step2
```

32 位模拟 UEFI 启动盘与 64 位相似, 此处不再赘述。

有时需要使用最新的 Shell, 此时要编译 `ShellPkg`, 然后将编译好的 `Shell.efi` 复制到 U 盘 `efi\boot` 目录并重命名为引导文件 (`bootia32.efi` 或 `bootx64.efi`)。

接下来就可以用 U 盘来引导进入 UEFI 世界了。

2. 制作 UEFI 平台下的 USB 启动盘

如果目标平台是 UEFI 平台, 那么启动盘的制作就变得非常简单, 可按如下步骤来进行。

- 1) 格式化 U 盘为 FAT (FAT、FAT16 或 FAT32) 格式。
- 2) 在 U 盘上建立目录 `efi\boot`。
- 3) 将 `efi` 的应用程序复制到 `efi\boot` 目录, 并改名为 `bootx64.efi` 或者 `bootia32.efi`。

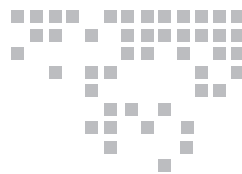
与 Legacy BIOS 需要 MBR 来引导操作系统不同, UEFI 的启动文件是 FAT 盘内 `efi\boot` 目录中的 `bootx64.efi` 或 `bootia32.efi`。

2.5 本章小结

本章主要是为开发 UEFI 应用和驱动准备开发环境, 主要内容包括:

- ❑ Windows 和 Linux 下的 UEFI 开发环境的搭建 (主要是 EDK2 的配置)。
- ❑ `build` 命令的用法。
- ❑ 制作和使用 OVMF 固件。
- ❑ 利用 DUET 制作传统 BIOS 系统上的 UEFI 启动盘。

第 3 章我们将开始介绍 UEFI 工程模块, 主要包括 UEFI 应用程序工程、UEFI 驱动工程和 UEFI 类库工程。



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 infs\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。

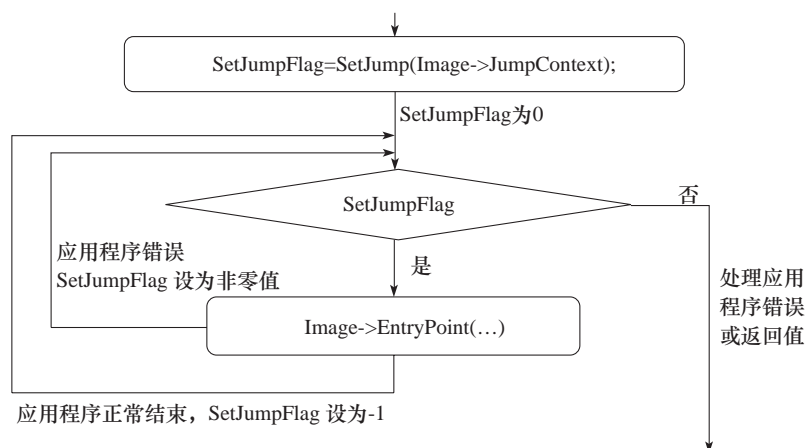


图 3-3 gBS->StartImage 执行流程

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。

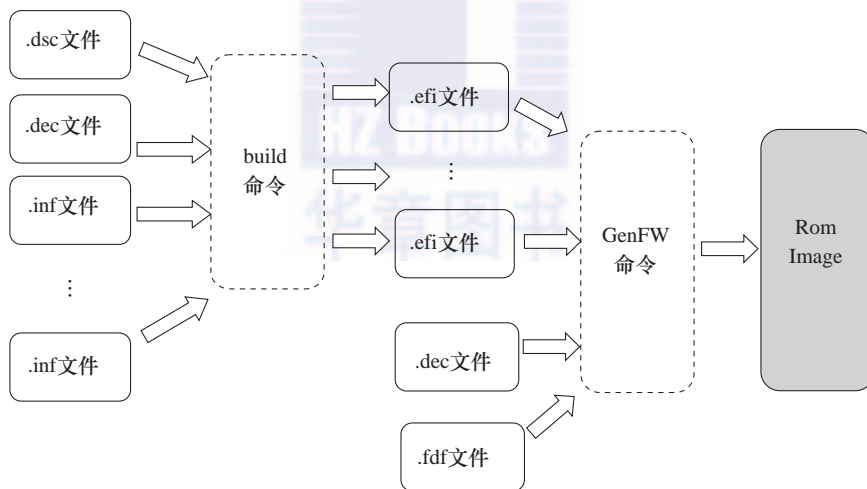


图 3-4 EDK2 文件与 EDK2 工具链关系图

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_ADD RESS	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 Conf\tools_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]
  NULL | 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。

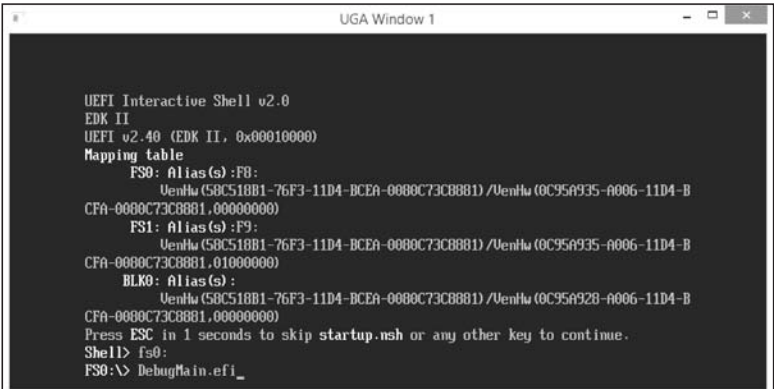


图 3-5 在 UEFI Shell 中执行 DebugMain.efi 程序

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。

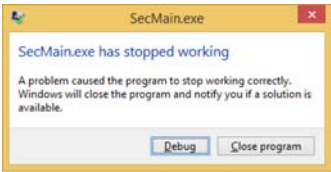


图 3-6 程序遇到“int 3”指令后弹出的系统调试对话框

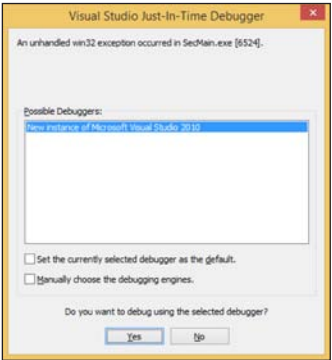


图 3-7 Visual Studio 调试器



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。

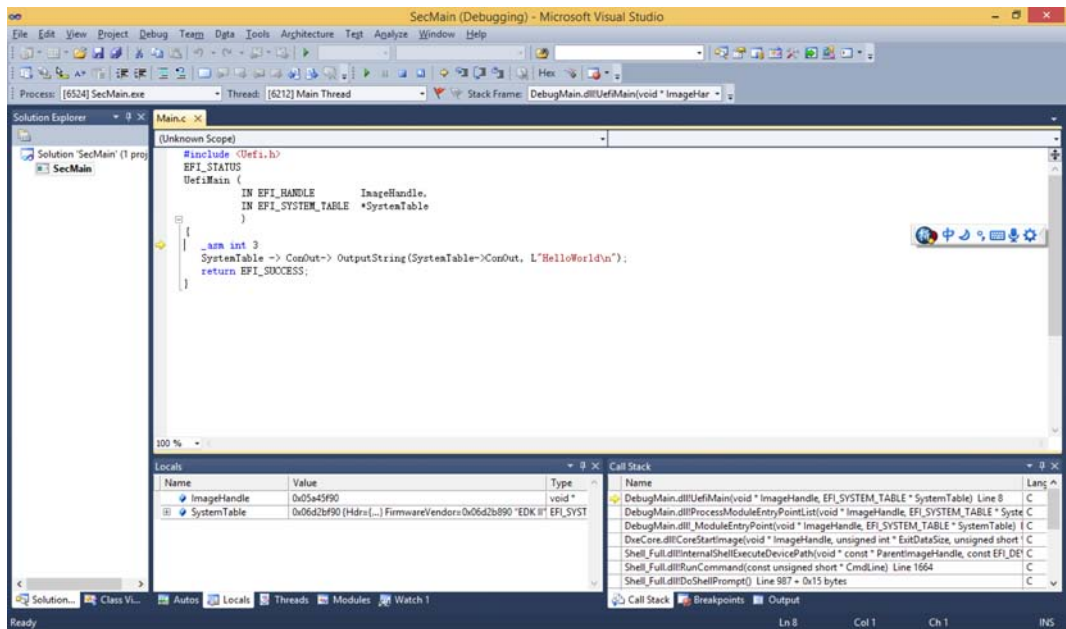


图 3-9 Visual Studio 调试界面

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。