分类号_____ 学号 M201276098

学校代码 10487 密级_____

華中科技大學

硕士学位论文

可定制 UEFI 系统的开发与测试

学位申请人 高 强

学 科 专 业：软件工程

指 导 教 师：黄立群 副教授

答 辩 日 期：2014.11.10

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree for the Master of Engineering

Development and Test of Customizable UEFI System

Candidate : Gao Qiang

Major : Software Engineering

Supervisor : Assoc. Prof. Huang Liqun

Huazhong University of Science and Technology

Wuhan 430074, P. R. China

November, 2014

摘 要

完整的计算机系统是由计算机硬件和软件两部分构成，然而在计算机系统中还有一个特殊的重要组成部分，这就是 BIOS，是 Basic Input Output System 的缩写，中文叫做基本输入输出系统，它也是计算机不可或缺的一部分，位于计算机操作系统层和硬件层之间，它的作用是保存基本输入输出的程序，同时还储存系统配置并完成开机自检工作。作为连接操作系统和系统硬件体系之间的桥梁，BIOS 采用汇编语言编写，综合应用多种技术，使用寄存器参数调用的方法，并在很小的内存上进行固定编址。BIOS 在外观上的落后、安全上的薄弱、功能上的羸弱、性能上的不足，都严重制约着它的进一步发展。

在这样的环境下 UEFI（全称为统一的可扩展固件接口，是 Unified Extensible Firmware Interface 的缩写）应运而生，UEFI 和传统 BIOS 相比具有明显的优势，比如使用 C 语言开发，摒弃静态链接而采用动态链接，并以模块化的风格来构建系统，使得 UEFI 扩展性极强，与传统的 BIOS 相比，UEFI 更倾向于是一种规范，UEFI 程序不是固定不变的，可扩展性强，很灵活，有丰富的文档供开发者参考。可定制 UEFI 系统，使得用户可以根据实际需求，添加、删除相应的驱动，并实现了丰富的自定制功能扩展，用户可以依据硬件设备来设置 BIOS，比如配置 DDR2 类型的内存，拥有 10 个 USB 接口，1 个网卡，3 个 SATA 接口等，当用户完成设置后，系统将生成目标 image 文件。

UEFI 在开机后参与了大量的初始化工作，包括硬件设备检测等，此外为了将操作系统解放出来，免去重装操作系统后需要重新安装驱动的复杂工作，UEFI 还承担了加载硬件的驱动程序工作。在使用专业的测试方法和测试框架对 UEFI 进行全方面的测试之后，发现 UEFI 是符合用户需求的，是适应 PC 发展的。总之，UEFI 取代传统 BIOS 已成定局，它已成为公认的、可靠、稳定的新一代 BIOS 标准。

关键词：基本输入输出系统　　统一的可扩展固件接口　　可定制　　测试

Abstract

A complete computer system consists of computer hardware and software. However, there is a special important part in a computer system, it's BIOS(Basic Input Output System), which located between the computer operating system layer and hardware layer, hold the most important basic input and output of computer programs, self-test program, system reboot program and system configure information.

Then, UEFI (Unified Extensible Firmware Interface) emerge as the times require, compared with the traditional BIOS, UEFI has obvious advantages to BIOS, it adopts modularization, dynamic link and parameter stack of C language style to form the system, which achieve more flexible, UEFI is an extensible, standardized firmware interface specification, which is different from the traditional BIOS startup process of fixed, lack of documentation, based entirely on the fact of experience andobscure the agreed. Customizable UEFI system makes user add, delete the corresponding driver according to the actual demand, and realizes custom function expansion rich, user can set the BIOS according to the hardware equipment, such as configuration of DDR2 type memory, with 10 USB ports, 1 Network card, 3 SATA interface, when the user finish configuration, the system will generate the target image file.

The biggest of UEFI change lies in the UEFI initialization after boot, not only finish the task of hardware detection, but also load the hardware driver, and does not require the operating system to load, which eliminates the need to reinstall the driver complex workload after the reinstall the operating system. In addition, after testing, we draw the conclusion that UEFI is consistent with the needs of the user, and adapt to the development of PC. UEFI replace traditional BIOS is a foregone conclusion, UEFI has become a new generation of BIOS standards recognized, reliable and stable.

Key words：Basic Input OutPut System　　Unified Extensible Firmware Interface　　Customizable　　Test

目 录

1 绪论

1.1 研究背景与意义

1.1.1 传统 BIOS 及其不足

一个完整的计算机系统主要由计算机硬件和软件两部分组成，计算机软件分为应用软件和操作系统两部分[1]。计算机硬件部分包括系统中的所有硬件设备和资源，比如 CPU、Memory、I/O 等，通过提供机器指令给操作系统，实现对系统中的各种硬件设备和资源的访问和控制。操作系统位于硬件和用户之间，一方面通过提供接口来供用户使用计算机；另一方面通过管理计算机硬件设备和资源，使得计算机能够充分合理地利用系统资源[2]。运行于操作系统之上的应用软件，帮助用户与计算机系统进行交互，并解决实际的应用问题。

在计算机系统中还有一个特殊的重要组成部分，即 BIOS（Basic I/O System, 基本输入输出系统），它位于计算机硬件层和操作系统层之间，以固件（Firmware）的形式存在，向下负责计算机硬件的管理，向上对操作系统提供统一的硬件使用和管理接口。BIOS 是一组固化到计算机内主板上一个 ROM 芯片上的程序，它保存着计算机最重要的基本输入输出的程序、开机后自检程序、系统自启动程序和系统设置信息[3]。

BIOS 一直是 PC 中不可缺少的一部分，自诞生以来伴随着主板经历了二十多年的风风雨雨，为 PC 的发展做出了重要的贡献。但 BIOS 的发展远远落后于 PC 机的发展，现有的 BIOS 的局限性也制约着 PC 的进一步发展。BIOS 的局限性主要体现在：

（1）对 MBR 的依赖

在引导系统的过程中，传统的 BIOS 要先找到 MBR，然后把 MBR 中的引导指令加载到内存中[4]。但是对于 MBR 的过分依赖，使得传统 BIOS 下的磁盘主分区不

能超 4 个。然而电子技术的快速发展，使得大容量磁盘的应用变得越来越普及，在计算机系统中，传统 BIOS 对 MBR 的依赖严重制约了 PC 的发展。

（2）平台依赖

BIOS 出现之初，主要针对的是 8086 架构，当时处理器和内部数据总线都是 16 位，而地址总线是 20 位，所以使得 BIOS 可以访问 1MB 的内存[5]。但是随着处理器芯片的飞速发展， BIOS 却停滞不前，一直都受限于 8086 架构。

随着处理器更新换代，新的处理器依旧兼容原有处理器的工作方式，甚至在处理器发展到 64 位的时代，处理器在加电启动的时候，还会使用 16 位的实模式[6]。这样来看的话，BIOS 的发展是远远落后于处理器的发展的，这也严重影响了计算机的启动时间和运行速度。

（3）ROM 容量限制

BIOS 的发展同样受到 ROM 芯片容量的严重制约，出于成本的考虑，厂家谋求最小的芯片容量和尽可能多的功能。目前，BIOS 程序主要运行在实模式下，同时 BIOS 代码被限制在 1M 内， BIOS 芯片中没有足够支持一些新的功能。而 BIOS 功能的丰富和发展必然要求更大的 ROM 容量，所以说这也是限制 BIOS 发展的主要原因。

（4）汇编语言实现

BIOS 程序采用汇编语言编写，使得 BIOS 代码的可读性、可移植性和可维护性等都比较差[7]，从事 BIOS 代码的编写和维护工作需要专业的工程师，这也是制约了 BIOS 发展的因素。

（5）安全性问题

安全性问题在 BIOS 的发展过程中越来越引起重视，首先 BIOS 缺乏自身的安全防护功能，一些病毒或黑客利用 BIOS 芯片刷新机制，去破坏 BIOS 代码或添加危害系统安全的恶意代码，由于恶意代码处于系统底层，非常不易检测，给系统造成严重的安全威胁[8]。此外，由于 BIOS 未考虑配置的安全问题，使得远程攻击者可利用本地计算机 BIOS 中的配置漏洞，通过网络来获取系统的控制权限，从而实现对本地

计算机的远程控制和数据存取，而 BIOS 的配置漏洞深入系统底层，远程攻击者甚至可以在本地计算机关机的情况下开启本地计算机并进行存储访问[9]。

1.1.2 UEFI 及其优点

自第一套 BIOS 程序问世以来，BIOS 就采用汇编语言编写，综合应用多种技术，使用寄存器参数调用的方法，并固定编址在很小的内存上[10]。BIOS 在外观上的落后、功能上的羸弱、安全上的薄弱、性能上的不足，都严重制约着它的进一步发展[11]。虽然目前的 BIOS 仍能够实现基本的功能，但 PC 要进步就必须寻求更高更好的技术，相应地原有 BIOS 的发展也就近乎走到了尽头。

EFI（Extensible Firmware Interface）最初由 Intel 针对安腾处理器平台推出，并在 HP、Microsoft 和 Phoenix 的合作下产生。尽管 EFI 是为安腾处理器开发的，但 Intel 在设计 EFI 时没有定位于只适用于安腾处理器，从而使得后来对 IA32、XScale，以及 UEFI 的 x64 和 ARM 的绑定都成为可能[12]。初期 EFI 一直用于 Intel 的安腾处理器，直到 EFI 在 Intel 处理器 IA32 上应用，并在 IDF（Intel Developer Forum）上展示了 EFI 的风采，EFI 才逐渐被大众所了解[13]。EFI 从提出之初就具有足够的前瞻性，随着 Intel 的积极推广，EFI 得到了越来越多的 PC 生产厂商和 BIOS 提供商的支持。2005 年，包括 Intel、AMD 在内的一些 PC 生产厂家决定建立 UEFI 论坛，以共同推动和开发适用于各平台的 PC 固件标准。

本质上 UEFI 是一套崭新的 BIOS 接口，并不对应具体的 BIOS 实现。通常而言，遵照 UEFI 接口规范实现的 BIOS 称为 UEFI BIOS，相应地将不遵循 UEFI 接口规范的早期 BIOS 系统称为传统 BIOS（Legacy BIOS）[14]。除接口定义外，Intel 和其他 UEFI 论坛成员为 UEFI 制定了很多的标准，这里面最有名的是一个开发架构，通过这个开发架构，开发人员可以像开发应用程序一样来开发 UEFI[15]，这个开发架构名字叫“Tiano”。由此发现，UEFI 所拥有的功能远远超过了传统 BIOS，例如：

（1）用户界面友好，UEFI 界面更简单，提供各种友好的交互信息，用户很容易上手，此外 UEFI 上支持 GUI 界面的运行，用户的视觉体验更逼真，并且还支持鼠标操作[16]，非常个性化。可以看出，UEFI 本着用户第一的原则，使得用户使用起来更得心应手。

（2）传统 BIOS 只支持 16 位的处理器，无法充分利用 CPU 性能和内存资源。UEFI.BIOS 可以运行于 64 位的处理器，能够对 64 位的 Windows 系统提供更好的支持[17]。

UEFI 和传统 BIOS 相比有很多优势，比如使用 C 语言开发，摒弃静态链接而采用动态链接，并以模块化的风格来构建系统，使得 UEFI 扩展性极强[18]。与传统的 BIOS 相比，UEFI 更倾向于是一种规范，UEFI 程序不是固定不变的，可扩展性强，很灵活,有丰富的文档供开发者参考[19]。UEFI 驱动程序的实现是由字节代码编写的，字节代码是能够被各种 CPU 所识别的，所以 UEFI 可以兼容各种类型的 CPU[20]。

UEFI 的最大特点是采用模块化设计，它解决了传统 BIOS 的弊端，功能也超出了传统 BIOS 的范围。从技术角度看，UEFI 在开机后参与了大量的初始化工作，包括硬件设备检测等，此外为了将操作系统解放出来，免去重装操作系统后需要重新安装驱动的复杂工作，UEFI 还承担了加载硬件的驱动程序工作[21]。

UEFI 拥有采用结构化 C 语言编写、可视化操作、可扩展性强以及兼容性强等优势[22]。此外，UEFI 界面更简单，提供各种友好的交互信息，用户很容易上手，此外 UEFI 上支持 GUI 界面的运行，用户的视觉体验更逼真，并且还支持鼠标操作，非常个性化。

1.2 国内外研究现状

1998 年，当时英特尔、微软、HP 和其它一些公司正计划使用英特尔安腾系统。这一计划最初被叫做 IBI（the Intel Boot Initiative），即英特尔启动创新项目。从 IBM PC 开始，主流的 PC 机都使用 BIOS（本书称之为 Legacy BIOS，或者传统 BIOS）。但是，在安腾处理器面前，BIOS 的缺点就暴露出来了[23]。比如，BIOS 依靠 8254 定时器和 8259 中断控制器，而它们对于大规模的服务器并不适用。更糟糕的是，BIOS 的可执行内存限制在 1MB，因此只有极其有限的内存空间来执行插槽板卡上的 Option ROMs。再者，BIOS 是基于 16 位的，这使得基于 64 位的安腾系统的优势难以发挥出来。

IBI 后来被称为可扩展固件接口（EFI，Extensible Firmware Interface）。EFI 把现

代计算机的软件架构概念引入固件程序设计，它允许用诸如 C 的高级语言来开发固件，提供对硬件的适当抽象，并具有良好的可扩展性[24]。EFI 以其令人信服的优点，使得微软和业界把它作为基于安腾处理器的计算机系统的唯一启动引导机制。

为了统一 EFI 标准和规范, 由英特尔、微软、惠普等全球著名的计算机软、硬件厂商于 2005 年发起创立了 UEFI 国际论坛。UEFI 是一个标准，它定义了一系列的接口规范 [25]。英特尔对 EFI 的实现定义了 Framework，UEFI 联盟基于 Framework 定义了 PI（平台初始化，Platform Initialization）规范；PI 规范建立了固件内部接口架构以及固件和平台硬件间接口，而后者使得平台硬件驱动程序具有模块化和互操作性[26]。随着 UEFI 和 PI 规范的 Framework 实现，英特尔完成了取代传统 BIOS 的使命，并使得业界超越了传统 BIOS。

随着 ARM 加入 UEFI 论坛，并支持 OEM 厂商使用 UEFI 开发技术基于 ARM 处理器的解决方案。Intel 公司发表声明：“通过业界多年对 UEFI 的支持和发展，已经在 UEFI 周围形成了一个生态系统。Intel 支持产业界从支持 UEFI 标准，延伸到开发、生产以 Intel 产品和 UEFI 固件为基础的解决方案[27]。”同时，所有业界领先的 BIOS 供应商都向客户和目标市场提供基于 UEFI 固件的解决方案。从这可以看出，新一代的 UEFI BIOS 必将成为一个核心技术的热点[28]。

UEFI 联盟目前有了八十多家企业成员，这些企业成员一共分为三类：推广者、参与者和使用者[29]。推广者主要负责 UEFI 项目的宣传和推广，参与者负责 UEFI 的相关研究工作，使用者是 UEFI 成果的享有者，在 UEFI 成熟技术成果的基础上，将自己的特制的技术与之结合，应用到具体的项目上。每个成员通过对行业推广，促进软、硬件行业更快的了解、认识并采用 UEFI 标准[30]。

UEFI 的发展，带来了 BIOS 技术上翻天覆地的变化。对个人电脑来说，优势体现不是很明显。但是对于企业用户来说，优势巨大。比如，传统的 BIOS 只支持 4 个主分区，当工作需要 100 个主分区时，传统的 BIOS 就遇到瓶颈了。但是 UEFI 支持 128 个主分区，UEFI 的研究和应用潜力巨大[31]。

目前国内外对于UEFI的研究，一方面停留在通用性的层面，即设计的UEFI BIOS能够满足加载不同的操作系统的要求[32]。UEFI 的通用性设计，能够最大限度的兼容

不同的操作系统[33]。但是它同样带来了一个问题，无法结合特定的操作系统发挥出操作系统在加载前和加载后的优势。在启动过程中，如果相关硬件驱动发生问题，无法加载操作系统的时候，这系统的排错和纠错的能力也大大减弱[34]。

另一方面，国内外的企业对于 UEFI 的研究从 PC 领域向着其他电子领域发展，UEFI 在复杂的嵌入式领域的应用也越来越多[35]。UEFI 的应用变得越来越广泛，更多的企业和组织参与到 UEFI 的研究设计和应用中来。

1.3 论文研究内容与结构

本文针对 UEFI 展开深入研究，首先从传统 BIOS 的不足入手，寻求新的解决方案来弥补传统 BIOS 不足，由多家公司联合推出的 UEFI 经过多年发展和演变，已经成为一个成熟的、可靠的新一代 BIOS 标准，且经过多方面的测试，UEFI 完全符合用户的需求，可扩展性也使得 UEFI 适应 PC 未来的发展趋势。而且本论文围绕可定制的 UEFI 系统，研究内容主要有：

（1） UEFI 的层次结构。介绍层次结构中各个模块之间的相互关系和各自的作用，以及这样设计的优点和目标。

（2）UEFI 协议。详细描述了 UEFI 定义的一些 I/O 协议和文件协议，因为 UEFI 的可扩展性在很大程度上依赖于协议。一个 UEFI 驱动程序是一个安装了各种协议的多种句柄来完成工作的可执行映像。UEFI 协议是一组由函数指针和数据结构块，或者由规范定义的 API 构成的集合。

（3）UEFI 的启动流程。详细描述了 UEFI 从开机启动安全检测，到初始化准备，再到驱动加载、设备的启动，直到最后的操作系统加载运行整个过程。

（4）UEFI 系统的设计实现。主要介绍了 SdbTools 的实现过程，讲述系统如何将一个空白的 BIM 文件，经过中间一系列复杂的操作，变成一个最终的解决方案，并生成目标 BIOS 文件的过程，让用户更直观地感受到该系统是如何运作的，对于系统的各个部分都有所认识和了解。

（5）UEFI 的测试。在 UEFI 测试过程中，从单元测试开始，对 API 进行功能性

测试、容错性测试等，保证 API 都能够符合预期设计，在完成单元测试后，对一个功能性模块进行集成测试，验证不同单元的组合最终能否正常运行。当系统最终成型后，依托于一个可靠的测试框架对 UEFI 系统进行了系统测试，包括压力测试等，保证系统功能完善、运行稳定，满足用户的需求和使用习惯。

全文一共分为六章

第一章为绪论，介绍了本文的研究背景以及意义，其中包括传统 BIOS 及其不足、新型 UEFI BIOS 及其优点，还简单介绍了国内外目前的研究现状。

第二章介绍 UEFI 系统中的使用到的一些关键性技术和分析，以及一些核心的 API 功能和分析。

第三章是根据 UEFI 的特点和应用，明确系统的需求。

第四章是可定制 UEFI 系统的设计与实现，介绍了 Config 如何实现对 BIM 文件的操作和 Assembly 如何完成 Build 工作。

第五章是 UEFI 的测试，使用的测试框架，设计测试用例，运行测试，分析、得出测试结果，并通过多种测试方法验证系统的可用性和稳定性。

第六章是总结和展望，对于论文的内容进行全面的、系统的总结，同时对于 UEFI 未来的发展趋势进行展望。

2 相关技术分析

本章主要介绍可定制 UEFI 系统在实现过程中相关的技术。包括 UEFI 的层次结构、UEFI 启动流程和 UEFI 协议的介绍。

2.1 UEFI 的层次结构

由于传统 BIOS 的种种弊端，使得英特尔抛弃了对传统 BIOS 的依赖，设计了 UEFI 的实现方案，即 Platform Innovation Framework for EFI。其中 PI（Platform Initialization, 平台初始化）规范建立了固件内部接口架构以及固件和平台硬件间接口，而后者使得平台硬件驱动程序具有模块化和互操作性。Framework 实现了 UEFI 和 PI 规范。

本节将按照系统架构由低到高的层次，介绍组成系统的各个模块之间的相互关系，如图 2.1 所示。从中可以很清楚地看到平台硬件、PI、UEFI 和操作系统的相互联系和各自的作用：

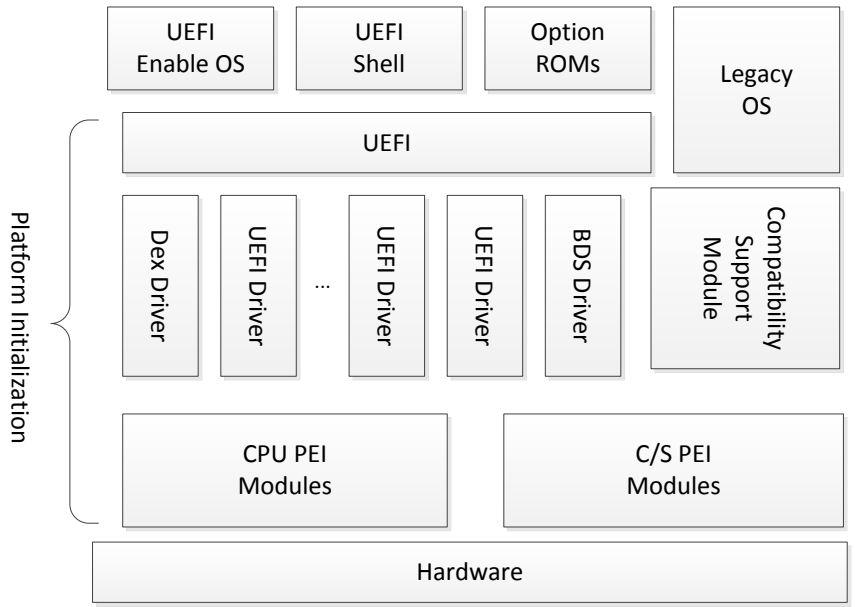

图 2.1 UEFI BIOS 的层次结构

（1）UEFI 初始化准备（Pre-EFI Initialization, PEI）层，是启动最少量的必备硬件资源，这些必备硬件资源可以满足启动固件驱动执行层即可，基本硬件初始化层会把启动的硬件资源信息传递给固件驱动执行层[36]。

（2）固件驱动执行（Driver Execution Environment, DXE）层，彻底完成所有硬件初始化，并为上层接口实现所有 UEFI 规范中定义的各种服务，它在 UEFI BIOS 的层次结构中占有极其重要的位置[37]。

（3）UEFI 接口层，UEFI 规范的接口层，是固件对外的接口，提供标准 UEFI 所规定的各种协议和接口。

（4）UEFI 应用层，基于 UEFI 接口调用，实现操作系统启动前的应用，以及加载操作系统；或者基于兼容性支持模块（CSM），对外提供与传统 BIOS 完全一致的接口，用以支持传统操作系统和基于传统 BIOS 中断调用的应用程序[38]。

从上面的层次结构，可以看到以下几点：

（1）PI 和 UEFI 架构组成了 UEFI BIOS 层次结构的主体，它们往下连接了平台硬件，往上连接了操作系统和其它应用；

（2）PI 主要由 UEFI 初始化准备（PEI）层和固件驱动执行（DXE）层这两层组成。PEI 层初始化系统，并提供最少量的内存；而 DXE 层提供可支持 C 语言代码的 DXE 驱动程序底层架构[39]。PI 使得 DXE 驱动程序标准化和模块化，从而具有很强的互操作性；

（3）PI 架构是 UEFI 得以实现的基础，推动了固件组件提供者之间的互操作能力。从该图可以看出，PI 和 UEFI 各自特色鲜明而又密切相关；

（4）UEFI 接口层就是 UEFI 本身，它仅仅提供接口。UEFI 接口层是一个接口规范，用于启动服务，具有良好的灵活性、扩展性和兼容性。

2.2 UEFI 启动流程

对于UEFI的开发，首选需要研究UEFI的驱动模型和基本框架，有针对性的对每个阶段进行设计。

UEFI 启动流程按照时间先后顺序，通常经过以下几个阶段：

（1）安全检测（SEC）阶段

（2）EFI 初始化准备 (PEI) 阶段

（3）驱动程序执行环境（DXE）阶段

（4）启动设备选择（BDS）阶段

（5）瞬时系统加载（TSL）阶段

（6）系统运行（RT）阶段

（7）运行结束（AL）阶段

下面将依次对各个阶段进行介绍。

2.2.1 安全检测（SEC）阶段

开机后即进入安全检测阶段。在该阶段，一个安全模块将得到控制权，它的作用是对后续部分提供安全校验，并且该安全校验将贯穿整个固件的生命期。该安全模块在自身的初始化完成后，首先对整个固件进行总体校验，然后启动进入EFI初始化准备（PEI）阶段。

2.2.2 EFI 初始化准备 (PEI) 阶段

EFI 初始化准备 (PEI) 在平台初始化过程中主要起两方面的作用：

（1）发现系统的启动资源，并且为后续DXE阶段的执行提供最少量的内存空间。

（2）PEI阶段提供了一套标准方法来加载和调用处理器、芯片组和系统主板的初始配置程序。

PEI阶段发生在SEC阶段之后。该阶段代码运行的主要目的是充分初始化系统，以保证驱动程序执行环境（DXE）阶段能够运行。

PEI阶段工作过程包括调用PEI基础，按顺序调度所有PEIM，以及发现和调用下一个工作状态。在PEI基础初始化期间，PEI基础初始化用于给PEIM提供通用PEI服务的内部数据存储空间和函数[40]。在PEIM调度期间，PEI调度程序根据闪存文件系统的定义仔细检查固件卷，并找出PEIM。然后，PEI调度程序对PEIM进行调度。

2.2.3 驱动程序执行环境（DXE）阶段

DXE阶段包括符合UEFI规范的一个实现。因此，DXE基础和DXE驱动程序都共

享UEFI映像的许多属性。DXE阶段完成大多数系统初始化的工作。PEI阶段负责初始化平台的主存，使得DXE阶段可以被加载和执行。DXE阶段包含下列组件：

（1）DXE基础

（2）DXE调度程序

（3）DXE驱动程序

DXE基础产生一套启动服务、运行时服务和DXE服务。DXE调度程序负责按正确的顺序发现和执行DXE驱动程序。DXE驱动程序负责初始化处理器、芯片组和平台组件，以及为控制台和启动设备提供软件抽象[41]。这些组件一起工作来初始化平台，并提供启动一个OS所需的服务。DXE和启动设备选择（BDS）阶段一起工作来建立可供操作系统启动的平台。当一个OS成功开始启动，即，BDS阶段开启时，DXE就终止了。OS运行时环境只允许由DXE基础提供的运行时服务和由运行时DXE驱动程序提供的服务继续存在。DXE的运行结果是生成一套完整的UEFI接口。

PEI和DXE一个不同之处在于，DXE拥有适量的系统永久RAM可供使用；而PEI仅仅拥有一些有限的临时RAM，并且这些临时RAM在PEI阶段初始化永久内存后可能会被重新配置以作其它的用途，比如缓存（Cache）。因此，PEI没有DXE的资源丰富。

2.2.4 启动设备选择（BDS）阶段

BDS阶段是一个独特的启动管理阶段。UEFI启动管理器是在UEFI固件中的一个组件，它可以决定哪个UEFI驱动和UEFI应用程序应该被明确地加载，何时被加载。一旦UEFI固件初始化完成，控制权被交给启动管理器。接下来，启动管理器决定需要加载什么，以及完成这些动作需要进行的与用户的交互。大多数启动管理器的行为由固件开发者来决定，启动管理器实现的细节不在该规范的考虑范围内。尤其是，与实现方式相关的细节可能包括：任何与引导相关的输入输出接口、集成的引导选择平台管理器、其它内部的应用程序可能的信息或修复的驱动，其它可能的通过启动管理器集成在系统中的内部应用程序或恢复驱动。

BDS阶段实际上是启动管理器选择执行应用模块，可以选择执行的应用模块有：EFI接口操作系统加载器、EFI接口应用模块、传统接口操作系统加载器、传统接口

应用模块[42]。

DXE阶段和BDS阶段一起工作来建立可供操作系统启动的平台。当一个OS成功开始启动，即，BDS阶段开启时，DXE就终止了。如果BDS阶段不能够连接一个控制台设备、加载一个驱动程序或者启动一个启动选择，那么就需要重新调用DXE调度程序。

2.2.5 瞬时系统加载（TSL）阶段

TSL阶段使用EFI接口，加载操作系统。在众多应用中，这是最重要的一个应用。该阶段包括无系统应用程序、瞬时OS环境、瞬时OS启动加载器和OS启动加载器等

当操作系统被成功加载后，就进入操作系统运行阶段。

2.2.6 系统运行（RT）阶段

在操作系统运行阶段，只有运行时服务可以被访问，而启动服务不能被访问。这里的运行时服务是指，在启动目标（如操作系统）运行之前，以及在启动目标运行之后都可以使用的函数；而启动服务是指，在启动目标运行之前，或者在ExitBootServices()被调用之前可以访问的函数。该阶段包括操作系统应用程序和最终操作系统环境等。

2.2.7 运行结束（AL）阶段

当操作系统结束运行后，控制权就交还给固件。操作系统结束运行可能是合法的关机，也可能是被强迫的非法关闭。

2.3 UEFI 协议

2.3.1 PCI I/O 协议

PCI I/O协议提供的接口可用于完成对内存、I/O和PCI配置空间的基本操作。系统提供对基本系统资源的抽象访问，以允许一个驱动程序以编程方式来访问它们。这个协议的主要目的是提供一种简化PCI设备驱动程序写操作的抽象。该协议具有如下特征：

（1）提供一个驱动模型。该模型不需要驱动程序去搜索用于设备管理的PCI总

线，而是直接告诉驱动程序需管理的设备的地址，或者当发现一个PCI控制器时，通知驱动程序提供一个设备驱动模型。该模型从PCI设备驱动程序中抽象出I/O地址、内存地址和PCI配置地址，用基地址寄存器 (BAR, Base Address Register)相关寻址方法访问I/O和内存，用设备相关寻址方法访问PCI配置[43]。

（2）如果需要，可提供PCI设备的PCI存储段、PCI总线编号、PCI设备编号和PCI功能编号。可以从PCI设备驱动程序中抽象出这些详细资料。

（3）提供PCI设备的基地址寄存器中未提及的其它任何非标准化地址解码详细资料。

（4）提供对PCI设备作为其成员的用于PCI主机总线的PCI根桥I/O协议的访问。

（5）如果PCI Option ROM在系统内存中存在，就提供它的一个备份。

（6）提供执行总线主控DMA的功能，包括基于包的DMA和公用的缓冲器DMA。

2.3.2 磁盘 I/O 协议

磁盘I/O协议用于将块I/O协议的块访问抽象为一种更加通用的偏移量长度协议。这一固件负责把该协议添加到任何在系统中还没有一个磁盘I/O协议的块I/O接口上去。文件系统和其他磁盘访问代码使用该磁盘I/O协议。

磁盘I/O协议允许进行满足底层设备的块边界或对齐要求的I/O操作。这一功能的实现是当需要提供对块I/O设备的相应请求时，通过复制数据到/从内部缓冲器来完成的。设备句柄上的块I/O协议Flush()函数可用来刷新待写入的缓冲器数据。

该固件自动地把一个磁盘I/O接口添加到所产生的块I/O接口上。它也把文件系统或者逻辑块I/O的接口添加到磁盘I/O接口上，该磁盘I/O接口包含任何可识别的文件系统或者逻辑块I/O设备。适合UEFI的固件必须自动支持下列格式：

（1）UEFI FAT12、FAT16 和FAT32 文件系统类型。

（2）传统的主引导分区记录块。

（3）扩展分区记录块。

（4）El Torito逻辑块设备。

该磁盘I/O接口提供一个非常简单的接口，该接口允许对基本块I/O协议提供一个更加通用的长度偏移量抽象。

2.3.3 块 I/O 协议

块I/O协议用于抽象大容量存储设备，以允许运行在UEFI启动服务环境中的代码在不了解设备具体类型或者不了解管理设备的控制器情况下可以访问它们。已定义了对大容量存储设备以块层次进行数据读写，以及在UEFI启动服务环境中进行设备管理的功能。

2.3.4 简单文件系统协议

简单文件系统协议允许运行在UEFI启动服务环境里的代码对一个设备进行基于文件的访问[44]。简单文件系统协议用于打开一个设备卷，并且返回一个UEFI文件句柄，该句柄提供了访问设备卷文件的接口。这个协议与大多数协议有一点不同，除了简单文件系统对设备进行分层以外，还有一个辅助协议作用于该设备

2.3.5 UEFI 文件协议

当请求一个设备上的文件系统协议时，调用程序得到该卷的简单文件系统协议的实例。在需要时可通过该接口打开文件系统的根目录。调用程序必须在退出前通过Close()关闭该根目录的文件句柄和其它所有已打开的文件句柄。应避免使用文件系统正在抽象的基本设备协议。例如，当一个文件系统处于DISK_IO / BLOCK_IO协议层时，应该避免对组成该文件系统的块设备的直接块访问，即使该设备的打开文件句柄是存在的。

一个文件系统驱动程序可以高速缓存与一个打开的文件相关联的数据。系统提供了一个Flush()函数，该函数将所有和被请求文件相关的文件系统中的脏数据刷新到物理介质中[45]。如果基本设备可以高速缓存数据，那么该文件系统也必须通知该设备进行高速缓存。

2.4 本章小结

本章介绍了与 UEFI 相关的技术，详细介绍了 UEFI 的层次结构中，每一层的作用和关，了解到 UEFI 如何从开机的安全检测、初始化准备、驱动加载一直到最后的操作系统加载运行的整个过程，以及 UEFI 常用的 I/O 协议、文件协议等接口协议。

3 可定制 UEFI 系统的需求分析

3.1 系统需求背景

作为连接操作系统和系统硬件体系之间的桥梁，BIOS 采用汇编语言编写，综合应用多种技术，使用寄存器参数调用的方法，并在很小的内存上进行固定编址。BIOS 在外观上的落后、功能上的羸弱、安全上的薄弱、性能上的不足，都严重制约着它的进一步发展。虽然目前的 BIOS 仍能够实现基本的功能，但 PC 要进步就必须寻求更高更好的技术，相应地原有 BIOS 的发展也就近乎走到了尽头。

在 IT 产业界，系统开发商不断地对 BIOS 功能与设计提出要求，特别是在简洁化与易用性方面要求十分强烈。开发更容易使用的 BIOS，不仅可以加速 IT 产业的创新，对于系统开发商而言更可以通过固件整合特色功能，达到产品差异化的市场需求。

另一方面，芯片厂商也同样渴望有更快速的方法可以将新的系统硬件选项加入 BIOS 中。传统的方法是芯片厂商先必须要将硬件规格或是参考固件代码交给 BIOS 厂商，只有当 BIOS 厂商将达到出货品质的驱动程序加入 BIOS 中之后，芯片厂商才能将芯片交付给系统开发商。有了 UEFI 这种固件的标准接口，无需经过任何特定的附加软件技术就可以将驱动程序植入 BIOS 中，从而消除了过去冗长的沟通联系过程，加快了产品的开发上市过程。

UEFI 得到了 IT 业界各大厂商的大力支持，其前身是 Intel 为其安腾系列处理器开发的“可扩展固件接口”(EFI: Extensible Firmware Interface)。为推动 EFI 技术的发展，Intel 在 tianocore.org 上开源了著名的“Green-H”核心框架，该框架包含了 Intel 对 EFI 的主要参考实现。EFI 标准也从 Intel 的专有标准演进为“统一接口固件标准”UEFI，以现在传统 BIOS 的观点来说，未来将是一个“没有特定 BIOS”的电脑时代，UEFI 也被看做是有近 20 多年历史的 PC BIOS 的继任者和终结者。

随着计算机技术的发展，UEFI 技术广泛的应用到各种电子领域。Phoenix、AMI、Insyde、Byosoft 等 BIOS 公司应用最新的 UEFI 技术，定制出满足不同应用

需求的 UEFI 产品。华硕、技嘉等主板厂商将这些定制的 UEFI 产品固化到最新的主板上，推向市场，UEFI 技术得到了空前的发展。

3.2 可定制 UEFI 系统功能需求

可定制UEFI系统，是一个允许用户自己定制BIOS的系统，目标用户可以根据所使用主板的硬件设备和接口、外设来进行设置，完成设置后，系统运行会得到目标BIOS文件，用户将该image文件借助工具烧录到主板中，就可以正常使用了。

因此，系统需要一个能够可供用户选择的硬件设备底层库函数和接口，用户可以根据自己的设备类型选择可以支持的设备驱动，当然还可以修改原有的设备驱动以支持新的设备，此外还需要良好的可扩展性，允许开发者开发新的驱动以支持最新的设备类型，这些需要一个强大的库函数来完成，底层接口文件中需要定义最基本的变量和设备定义、描述规则等，需要有一定硬件基础的开发人员从事开发工作，用户在这些基础之上，只需对系统有一个大概的认识和基本的硬件设备知识，就可以自由选择驱动文件。以上所说的用户的这些操作是针对一个具体的实例，这是建立在一个项目的基础上，先创建一个空的项目，然后用户完成设置之后，就把用户的需求添加到项目中，对于用户的其他操作，也放在项目里，由项目来统一管理、记录用户操作，并根据操作生成结果，最终生成解决方案，比如，打开一个新的项目，然后添加需要的驱动文件，调用不同的接口完成设备的连接和驱动的底层支持文件的最优解决方案，至此完成了对于一个具体实例的操作，下一步就是根据这个项目的解决方案进行Build工作，生成最终目标BIOS文件。所以在生成项目的解决方案之后，需要一个工具来完成最终的Build任务，在这一个过程中，要根据项目中的驱动底层支持文件来生成整个Build操作所需要的基本变量和规则，最终依据这些变量和规则的要求生成二进制BIOS文件，也就是最终的BIOS，至此整个系统完成。

根据上述要求，可定制UEFI系统主要由两大模块构成：模块一的功能是提供各种类型的底层设备接口和一些通用的设备接口，供用户选择，用户根据自己的需求来完成目标BIOS配置，这一部分工作主要由EDK II来实现；模块二的功能是根据用户的设置，动态生成解决方案，并得到目标BIOS文件(.fd文件)，这项工作主要由

SdbTools来完成。

EDK II（EFI Development Kit，EFI 开发工具）是一个包含一系列基本的底层库函数和接口的 UEFI 规范标准的框架接口，是 EFI BIOS 的开源的发布框架。开源核心代码包含了三种平台架构，分别是 DUET、NT32、Unix，除了开源核心部分，它同时也是一个开发、调试和测试 EFI 驱动程序的综合平台。最初的版本是 EDK，随着近几年的发展，UEFI 联盟根据 EFI 用户的反馈信息对 EDK 进行了进一步的功能增强和优化，形成了新的功能更为强大的 EDK II。EDK II 主要着眼于如何让用户设计出更为规范和相对容易的自定义模块，在开发库、类机制、编译控制优化、元数据控制以及 PCD 机制方面进行了扩展。EDK II 中各个模块的是一个独立的 Package，他们之间存在非常紧密的依赖关系，如图 3.1 所示。

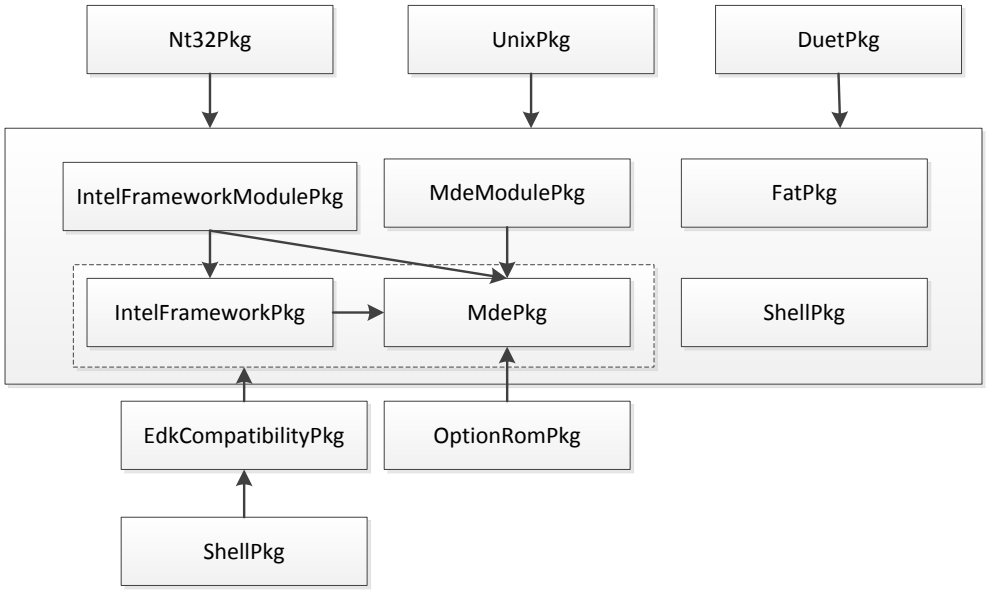

图 3.1　EDK II 主要 Package 的依赖关系

EDK II主要模块和功能分析：

BaseTools：包含代码编译所需要的二进制编译工具和编译环境的配置文件。

Conf：保存编译环境信息、编译目标路径以及编译器参数。

MdeModulePkg：提供跨平台的底层库示例的模块，例如MdePkg。

MdePkg：EDK II环境下IA32、X64、IPF和EBC平台的底层库函数、工业标准和

协议。

FatPkg：包含针对不同CPU架构的原始FAT驱动。

ShellPkg：提供一个平台通用的UEFI Shell应用程序开发环境。

DuetPkg：提供基于传统BIOS的Runtime环境的支持库。

NetworkPkg：网络驱动模型。

Nt32Pkg：在windows操作系统下通过加载 32 位模拟器来提供Runtime的运行环境。

UnixPkg：在linux操作系统下通过加载 32 位模拟器来提供Runtime的运行环境。

UefiCpuPkg：CPU驱动模型。

EdkCompatibilityPkg：提供兼容传统BIOS和EDK的协议和库。

OptionRomPkg：针对不同CPU和PCI的实例程序。

edksetup.bat文件：该文件提供配置EDK II的运行环境，包括工作区的配置和编译工具使用环境的配置。

利用EDK II开发工具，用户可以很容易的设计出符合自己要求并且规范的驱动和应用程序模块。用户要根据EDK II给出了相应的INF文件规范，设计符合要求的Module Inf文件，同时还要遵守DEC文件、DSC文件的相应规范，通过该工具，可以实现UEFI系统对不同底层设备的支持，比如，当前系统中UefiCpuPkg下支持E620 和E640 两种型号的CPU，但是现在市场上研发出了一种新型号的CPU，是E680 类型，那么用户就可以按照规范在UefiCpuPkg新加入E680 型号CPU的Inf文件和相应的DEC、DSC等其他关联文件，那么更新后的系统中，用户就可以选择支持新的E680型号的CPU。同样，用户还可以在MdeModulePkg中选择支持新的内存、硬盘等。

SdbTools是可定制UEFI系统的关键，首先，SdbTools要根据用户的选择和要求，将用户的设置放入BIM文件中，其次，根据设备的连接规则生成设备连接方案，同时要在EDK II的Package中寻找出支持用户选择的底层库接口，并评估选取最小、最优的解决方案来满足用户的选择，最后要利用编译工具将BIM文件编译为最终的.fd文件，也即目标BIOS文件。SdbTools是Intel研发的针对可定制UEFI系统的特定工具，核心主要由三大模块组成：Config、Assembly和UI，各部分的功能如下：

Config：完成对BIM文件的操作，设置Question、Module、PCD，生成设备连接方案，添加第三方文件，生成最终解决方案等。

Assembly：在Config完成之后调用，利用Build规则将BIM文件直接转化成最终的BIOS文件。

UI：封装Config和Assembly，为用户提供可视操作界面。

除此之外，Logger模块，能够记录系统运行过程中每一步的信息，帮助用户理解整个运行过程的同时，还能记录出错信息、定位出错的位置。总之，SdbTools提供了用户多种必要且易用功能，使得系统更完善、更简洁。

3.3 可定制 UEFI 系统非功能需求

一个系统的好坏不仅仅表现在功能的足够强大，还需要有优越的性能、良好的易用性和其他展现系统的特点的非功能性需求，这样的系统才称得上为优秀的系统，在提供强大功能的同时，使得用户使用体验非常良好，用户与系统交互方便，系统使得用户很容易上手，并提供各种提示信息，对于用户的请求系统反应迅速，所以，对于可定制 UEFI 系统，也提出了以下的要求：

3.3.1 系统的性能需求

作为一款应用软件，用户需要安装系统，系统的安装过程是这样的，首先安装各种准备环境，比如 Java 虚拟机，添加系统变量等，至少需要 30s，然后安装 EDK II 的各种 Package 和 Build 工具，该过程需要较长时间，因为目前有 30 个 Package，平均每个 Package 需要 8~10s，所以需要 4~5min，总计 5min 左右的时间，经过多次安装测试，发现正常情况下 5min 之内可以完成安装，有时由于 Package 的错误，会使得系统安装失败，用户需要等待更长时间，所以要求系统安装尽量控制在五分钟之内，如果有新添加的 Package，可以稍微延长一点等待的时间，如果系统在 5min 内没有完成安装，则需要优化系统。

同时，安装 Package 的过程中，每一个 Package 下面都有很多的驱动文件的驱动支持文件，使得 Package 的容量会比较大，安装的时候默认安装在系统盘（通常是电

脑的 C 盘），每一个 Package 平均需要 10M，30 个需要 300M，此外还有 Build 工具和其他工具，也需要 100M 的空间，同时考虑到后续的开发工作和需求，所以要求系统盘至少要留有 500M 的空间供系统安装使用。

用户在使用过程中的每一步操作，都希望系统能够尽快的处理并回应，有些操作很快只是一些界面切换，可能系统响应时间很短，不到 1s，有些操作系统需要在后台运行处理然后将最后结果呈现给用户，这个可能需要一些时间，经过对一些 API 的分析和测试，发现系统只要能够对于用户的基本请求在 3s 内给出回应，就使得用户有一个比较好的操作体验，但是有一个比较特殊的操作，就是 Build 操作，系统需要调用多个 API，生成中间文件，然后使用工具完成最后的工作，这个过程平均需要 2min 的时间。

因此，综上所述，对系统的性能提出以下要求：

（1）系统安装时间尽量不超过 5min。

（2）系统盘（C 盘）剩余容量不少于 500M。

（3）系统对于用户的操作响应时间不超过 3s，Build 操作生成最终 BIOS 的时间不能超过 2min。

3.3.2 易用性需求

为了使得用户能够对系统有好感，以及很快熟悉并迅速上手，那么首先系统的界面要尽量做到简洁、大方、美观，试想如果一个系统将界面做的很华丽、很复杂，用户要进行一个操作需要在界面上找半天，那么这个系统就已经注定要失败，优秀的系统是能够用尽量少的界面元素实现尽可能多的丰富的功能，最简单的例子就是苹果，乔布斯的信条：至繁归于至简，用户接触到系统之后，只有操作一个按钮就实现了各种各样的功能，那么用户有什么理由拒绝使用呢？所以，可定制UEFI系统要做到界面简单，当然，UEFI系统还是需要提供一些简单的指导操作和功能说明，使得用户能够理解系统是如何运行的，通过这些指导和提示，用户能够更快地熟悉系统，当然对于不熟练的用户，系统必须能够有较强的容错能力，因为用户可能不小心做了一个错误的操作，抑或是不合理的操作，那么系统能够提醒用户操作失误，

并继续运行，用户看到这些就会明白自己的操作有问题，就回去尝试其它的操作，所以，系统还需要能够满足一下几点要求：

（1）操作界面美观、简洁。

（2）提供用户手册和操作提示信息。

（3）如果用户出现操作失误，系统能够返回错误并提醒用户，同时维持操作前的状况，并继续正常运行。

3.3.3 运行环境需求

系统除了自身需要有丰富、完善的功能，可靠、易用的性能，对于支持系统运行的环境，也有一定的要求，俗话说得好：好马配好鞍，只有一个好的系统是不够的，同时系统还要运行在一个足够支持系统、展示系统功能和性能的环境之上，之前系统可以支持WinXP，但是由于微软已经不再支持WinXP的更新，所以目前系统主要运行在Win7 操作系统之上，同时为了系统能够快速处理并响应用户请求，对于CPU和内存也有一个最基本的要求，所以系统的运行环境要求如下：

（1）操作系统： Win7 及以上

（2）CPU：主频 2.4GHZ及以上

（3）内存： 512M及以上

3.4 本章小结

本章主要介绍了可定制UEFI系统的功能需求，系统的主要功能是提供底层库函数和接口给用户，并生成最终的目标BIOS文件，同时结合系统的特点和应用，对系统提出了性能、易用性、运行环境等方面的需求。

4 可定制 UEFI 系统的设计与实现

根据上一章中描述的系统的功能，可知 UEFI 系统位于硬件设备和操作系统之间，具体的功能结构如图 4.1 所示。

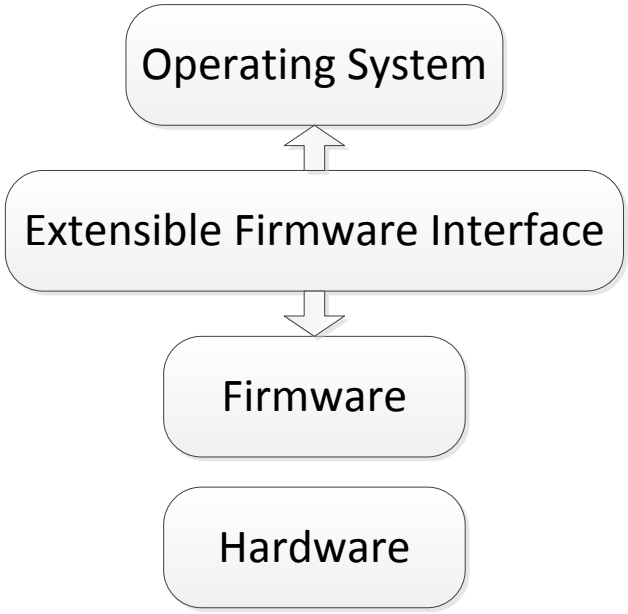

图 4.1 UEFI 功能示意图

从图中可以十分明确的看出来，UEFI 向下负责计算机硬件的管理，向上对操作系统提供统一的硬件使用和管理接口，使得用户能够方便的操作底层硬件设备，并为操作系统的资源管理和调配提供丰富的 API。

综合 UEFI 的功能以及上一章节中 UEFI 系统的需求，以及通过对需求的分析，完成系统进行的设计和实现工作。

在需求中提到了 UEFI 系统的两个关键模块，一个模块提供底层库函数和接口，供用户根据硬件设备来选择驱动文件，另一个模块通过两部分组成，一部分完成对二进制镜像管理（BIM）文件的操作，一部分根据前面的结果完成目标 BIOS 文件的生成，因此这样来设计系统，如图 4.2 所示。

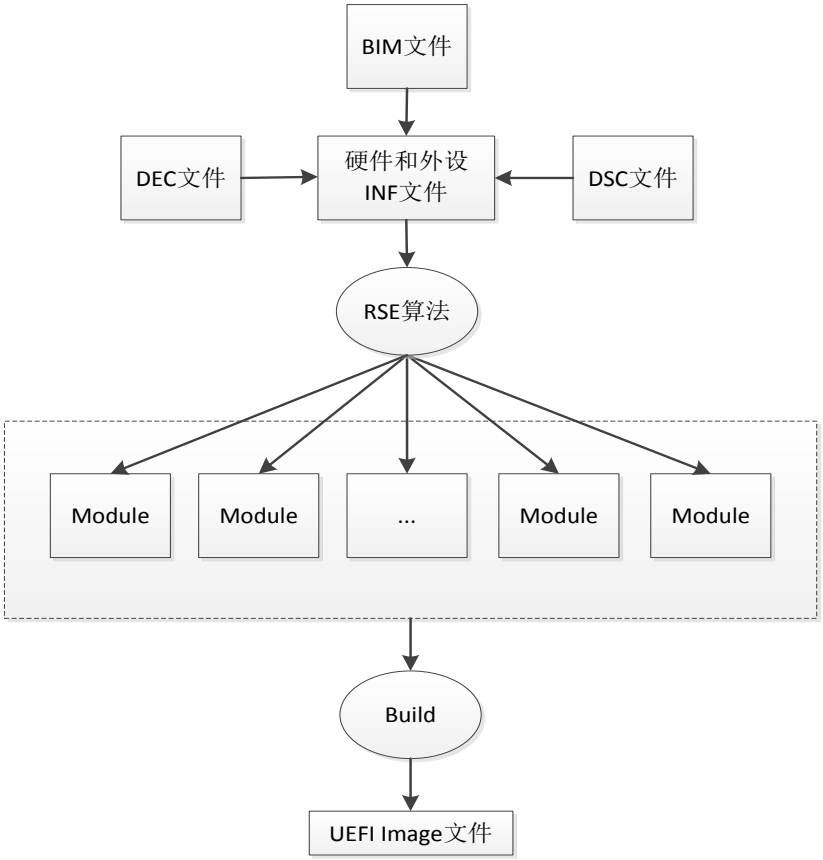

图 4.2　系统设计框架

SdbTools 主要实现框架中的 BIM 文件操作、RES 算法和 Build 工作。SdbTools 有两个关键模块，Config 和 Assembly。Config 顾名思义，完成系统配置工作，主要是实现对 BIM 文件的各种操作，为 Assembly 做准备工作，Assembly 在 Config 之后调用，生成 Build 的中间文件（dsc 和 fdf），这是 EDK II Build 的基础，然后调用 Build 得到最终的 BIOS 文件，具体的实现如下：

4.1　Config 的设计与实现

二进制镜像管理文件 BIM（Binary Image Management）文件，该文件用来记录每个硬件设备和外设以及所需要的驱动。对于用户来说， BIM 文件就是初级 BIOS，不同的 BIOS 就需要不同的 BIM 文件，一个 BIM 文件唯一对应的一个 BIOS，也就是系统中的一个新的项目，一个 BIM 文件中包含很多 section，有些 section 下面的信息正是由 Config 的函数来实现的，主要的 section 有：

（1）[Globals]

Name = Project

ReadOnly = FALSE

Name 是一个项目的唯一标识，最终生成的 BIOS 也是这样的名字（Project.fd），ReadOnly 表明该项目是只读的还是可读的，只读的不允许用户修改，可读的允许修改。

（2）[Tool]和[Repository]

ToolPath = C:\Program Files (x86)\Intel\Unified Binary Management Suite\0.1.0.72994\Bin\Win32

[Repository]

RepositoryPath = C:\Users\Public\Repository\Intel_UBMS

其中 ToolPath 是最后调用的 Build 工具的路径，该路径必须正确，否则生成最终的 BIOS 会失败，RepositoryPath 就是提供各种底层库函数和接口的 Package 的路径。

（3）[ModuleSettings]

PCD_1 = BomModuleInf_1|False|gEfiBinaryDistributionSupportPkgTokenSpaceGuid.PcdSymbolicDebugEnabled|TRUE

PCD_2 = 9727502C-034E-472b-8E1B-67BB28C6CFDB|False|gUart16550PciTokenSpaceGuid.PcdSerialPciDeviceInfo|{0x17, 0x00, 0xa4, 0x00, 0x00, 0x00, 0x44, 0x00, 0x0a, 0x01, 0x54, 0x00, 0xff}

该 section 下是 PCD 的值，是由 Config 的 SetPCD 和 FindModuleSolution 实现添加的，PCD 分为两部分，一部分是在用户在添加完 ModuleInf 后，通过 SetPCD 将对应的 PCD 的值也加进来，在 PCD 的值中也可以看到的对应的 ModuleInf，另一部分是在调用 FindModuleSolution 找到的新的 ModuleInf 后，同时添加这些新的 ModuleInf 里面使用到的 PCD 的值。

（4）[GlobalSettings]

gEfiUefiSpecificationFeaturesTokenSpaceGuid.PcdPlatformAuthenticatedVariableSupportEnabled = True|FALSE

gEfiCpuTokenSpaceGuid.PcdIsPowerOnReset = False|FALSE

GlobalSettings 下描述了 Question 和 GlobalPCD 的值，SetQuestion 完成了用户对 Question 的添加，GlobalPCD 则由 FindModuleSolution 在找到并完成添加。Question 描述了 BIOS 的一些属性，比如大小、架构、支持的操作系统、能否快速重启等。PCD 则是底层的库函数和接口的所使用到的局部的或全局的变量。

（5）[BillOfMaterials]

BomModuleInf_1 = IntelHardwarePkg_64260E85-B9F0-4fb7-BE5E-0A64607E903A_0.10\Cpu\E6xx\E640AtomProcessor.inf

BomModuleInf_2 = HardwarePkg_F9D480FC-75A5-4ac3-B0A4-9D16B61C5263_0.10\Hardware\Ddr2Memory\Ddr2Memory.inf

BomModuleInf_3 = HardwarePkg_F9D480FC-75A5-4ac3-B0A4-9D16B61C5263_0.10\Peripherals\Usb\UsbMouse\UsbMouse.inf

BillOfMaterials 下记录了用户的设置信息，如上，用户选择了 E640AtomProcessor 类型的 CPU，Ddr2Memory 类型的内存，并且支持 UsbMouse，Config 通过调用 SetModuleInf 将用户的选择加入到[BillOfMaterials]下面。AddConnection 就是根据该 section 下面的用户的选择来获取设备连接方案，RSE 算法的核心也是根据这些内容来生成最终的解决方案的。

（6）[Modules]

ModuleInf_1 = MdeModulePkg_BA0D78D6-2CAF-414b-BD4D-B6762A894288_0.92\Bus\Usb\UsbMouseDxe\UsbMouseDxe\DEBUG_LOG\UsbMouseDxe.inf

[Modules]下的信息就是 RSE 算法的结果，Repository Search Engine（RSE）算法是为了满足一组指定的需求，根据硬件和外部设备的要求，生成一组驱动模块（Module Solution），使这组模块能够驱动这些硬件和外设，算法的核心代码就是上面提到的 FindModuleSolution，首先，根据用户在[BillOfMaterials]下的设置，将与用户相关的 ModuleInf 的 PCD 分别添加到[ModuleSettings]和[GlobalSettings]下，然

后在[Repository]下提供的路径下搜索能够支持[BillOfMaterials]下 ModuleInf 的驱动文件，包括所有的 HardwareInf、PeripheralInf 和 ConnectorInf，最后对所找到的结果进行合并得到最终的驱动解决方案。比如通过 FindModuleSolution 找到的 UsbMouseDxe.inf 就是 UsbMouse.inf 的驱动文件。

（7）[NetList]

List_1 = BomModuleInf_1.MEMORY,BomModuleInf_2.MEMORY

NetList 下记录了 ModuleInf 是如何连接的，用户在选择完 ModuleInf 后，通过调用 Config 的 AddConnection 接口，实现 ModuleInf 的连接，他们的连接需要遵循一定的规范，每一个 ModuleInf 文件中有规定这模块可以和哪些连接以及如何连接，是一对一的还是可以支持多连接。

（8）[ExternalModules]、[PayloadFiles]和[UiFiles]

[ExternalModules]

SourceInfs_1= C:\TestCase\shell.inf | C:\TestCase| C:\TestCase\shell.dsc

BinaryInfs_1= C:\TestCase\shell.inf | C:\TestCase

BinaryDrivers_1= C:\UEFI\AddFileElement\TestCase\shell.efi

[PayloadFiles]

PayloadFiles_1 = C:\TestCase\shell.inf |e95a12f0-0783-47a2-a5fa-e4ac57e74996

PayloadFiles_2 = C:\TestCase\shell.c |e95a12f0-0783-47a2-a5fa-e4ac57e74997

[UiFiles]

UiFiles_1 = C:\TestCase\shell.inf

UiFiles_2 = C:\TestCase\shell.c

这几个 section 完成了一个功能的记录，就是添加第三方文件，因为添加文件的类型不同所以存放在不同的 section 下面，都是通过 AddFileElement 来实现的，如果添加的是 SourceInf，则还需调用 BuildUefiDrivers，将 source 文件 Build 成 binary 文件，然后再调用 PomUefiDrivers 对文件进行分析。

除了上面介绍的 API 外，Config 还提供了其它的功能函数：

（1）Get/Set

Get/Set 方法，针对的不是某一个 section，而是对所有的 section 都适用，Set 方

法实现了对于各种属性的设置，Get 方法可以让用户了解到各种属性都是是如何设置的，比如用户添加了 ModuleInf 之后，又想要去掉该 ModuleInf，那么就可以调用 Set 方法，将该 ModuleInf 的属性值设为 0 或者 False（表示去掉），则就可以实现删除功能，那到底是否已经去掉了，然后调用 Get 方法来获取所有的 Module 来查看，如果 Set 成功的话，那么结果就没有该 ModuleInf，如果结果中还存在该 ModuleInf，说明 Set 失败。Get/Set 是实现了某种意义上的查询、删除功能。

（2）SaveFile 和 SaveFileAs

SaveFile 和 SaveFileAs 都实现了保存 BIM 文件的功能，但是两者的应用场景是不一样的。SaveFile 的实现允许用户在任何步骤调用，保存用户的操作结果，用户只知道自己最后一次调用 SaveFile 后保存 BIM 文件的区别，中间做了那些操作，每一步操作 BIM 是如何变化的，用户是不清楚的，因为自始至终用户只在一个文件上面进行操作，最后的结果会覆盖前面所有的过程，而 SaveFileAs 不仅实现了保存操作结果，使得用户可以选择保存中间某一步操作的结果，甚至是每一步的结果，最后能够比较不同的文件，得出 BIM 的变化，清楚地理解每一步操作是如何引起 BIM 文件的变化的，两者的不同是一个只在一个文件上操作，一个是可以保存结果到中间的文件，用户可以是具体的情况而定如何调用。比如，用户新建了一个项目，在对应的新的 BIM 文件上操作，最后就可以调用 SaveFile，只需要保存最后结果即可。如果用户打开了一个已经存在的项目，在该 BIM 文件上操作完之后，调用 SaveFileAs，将结果保存到一个新的 BIM 文件中，而不影响原有的 BIM 文件。

（3）GetModuleSolution

GetModuleSolution 是在 FindModuleSolution 的基础上进行了，目的是获取 BIM 文件中[Modules]的所有 Solution 结果，如果没有进行 FindModuleSolution，就强行调用 GetModuleSolution，那么结果将为空，什么也得不到。GetModuleSolution 的应用场景是如果[Modules]中有添加或删除新的 Module，通过 FindModuleSolution 得到新的解决方案，可以调用 GetModuleSolution 来比较变化前后的结果，看是否有新的变化，同时也为测试提供了方便，每一个测试用例都有一套规范的答案，如果随着版本的变化和 API 的改动，通过 GetModuleSolution 获取到解决方案与标准的解决方案

不一样，那么说明改动出了问题，可以帮助寻找系统的 bug。

4.2 Assembly 的设计与实现

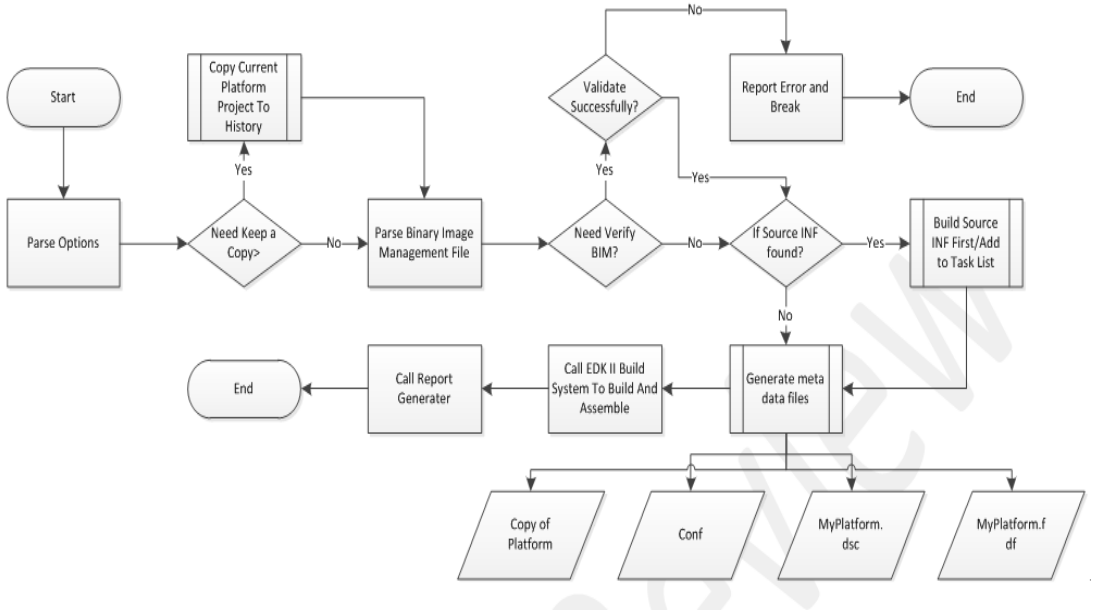

图 4.3　Assembly 的状态图

Assembly 所要操作的文件是最终的 BIM 文件，所以 Assembly 是系统在 Config 之后再调用的，从图 4.3 中（Assembly 的状态图），可以得出以下信息：

Assembly 的输入：BIM 文件

Assembly 的输出：

（1）History/MyPlatformProject_YYYY_MM_DD_HH_MM_SS.bim – 这是一个 BIM 文件的备份，带有时间戳，存放在 History 目录下。

（2）Conf –这是 EDK II Build 所需要的配置文件。

（3）MyPlatformProject.dsc –这是 EDK II Build 所需要的.dsc 文件。

（4）MyPlatfromProject.fdf –这是 EDK II Build 所需要的.fdf 文件。

（5）<ProjectName>_Report_YYYY_MM_DD_HH_MM_SS.scv – 这是 Assembly 调用 RGT（Report Generation Tool）后生成的报告。

（6）<ProjectName>.fd – 这是最终的 BIOS 文件

Assembly 在开始时先备份 BIM 文件，然后调用 Parser 函数对 BIM 文件进行解

析，判断 BIM 文件里面的内容是否合法，如果不合法则生成错误报告并立即终止，如果合法则继续往下进行，下一步会对 BIM 文件中是否包含 SourceInf 进行判断，如果存在，则会先调用 BuildUefiDrivers 将其编译为 binary 的文件，否则继续，这是 Assembly 的核心部分，在这一步骤中，Assembly 会调用 ConfGenerator、BimDscGenerator 和 FDFGenerator 来分别生成相应的元数据文件，这些文件是 EDK II Build 的基础，在 EDK II Build 的时候，至少需要一个含有 PCDs 值的 DSC 文件和一个对 PCDs 进行排列规则进行规定的 FDF 文件。如果有一个 PCD 在 FDF 文件中用到，但是没有在 DSC 文件中定义，Build 过程将会被终止，所以没有这些文件，EDK II Build 一定会失败。

总之，SdbTools 提供了一系列的 API，包括 RSE 算法、SetQuestion、SetPCD、SetModuleInf、AddConnection 等，帮助实现对 BIM 文件的操作和最后的 Build 工作，还提供了多种有用的 API，实现系统的同时还有助于帮助更好的测试主要 API 的功能。

4.3 本章小结

本章主要介绍了 UEFI 系统的设计和实现，以关键模块 SdbTools 为核心，介绍了 SdbTools 的实现过程，包括两个部分 Config 和 Assembly，Config 主要实现了对二进制镜像管理文件（BIM）的操作，Assembly 主要实现了 Build 工作，生成最终的目标 BIOS 文件。

5 可定制 UEFI 系统的测试

5.1 需求测试

UEFI 系统的测试，从项目开始时就已经进行了，首先，就是需求测试，对于需求中提到的要求，进行可行性测试，在现有的技术和工具的帮助下，是否能够实现需求中的问题，当前系统版本是否需要这样的功能，当然对于需求测试，一大部分工作可能是基于集体讨论的结果做出判断和选择的，但是还有一部分是需要有技术验证的，说到这个问题，我就具体阐述一下自己亲身经历过的一次需求测试的例子，也是对于能否实现和是否需要的一个合理的解释。

有一次系统的需求文档有了更新，需求报告里面要求系统在添加指定的文件夹或文件的时候，支持中文路径、中文名和中文内容，在最开始的实现过程中是没有这点需求的，这也是随着系统开发实现，需要添加进去的，在拿到这个需求变动的时候，当然首先要按照需求来完成任务，就是添加中文路径和中文内容这些测试用例，因为之前没有实现这样的功能，所以测试时，新添加的这些测试用例的测试结果是失败的，然后就和开发人员沟通，和他们一起核对了一下需求，确定之后，然后就对代码进行了修改，他们自己尝试的结果是可以支持中文，但是提交代码后，经过测试，设计的测试用例显示的结果却还是不能支持中文，然后一起又讨论了测试用例的设计，以及这样设计是否有什么问题，分析之后认为测试用例是正确的，完全符合要求，并且也比较完整，所以就排除了测试用例的问题，然后就只好单步调试，结果发现在代码里中文的编解码有点问题，把中文重新使用 utf-8 编码，然后再去使用，但是在系统使用过程中，用户经常会使用 Unicode 编码的中文，所以才出现了这样的问题，后来在测试代码中对中文路径和中文内容进行 utf-8 编码之后再使用，就没有了这个问题。

当然这个问题可以用这样的方法尝试解决，但是用户是不会知道应该使用什么编码格式的中文的，用户不关心这些，所以最终还是要系统内部来完成处理，系统内部如何处理，这又是一个问题，因为在测试的几个 API 中，可以支持中文，但是另

外的几个 API 却不能支持，原因是这些 API 是不同的开发人员完成的，各自的设计里面对于中文的处理是不一样的，所以目前的系统还不能对中文进行统一的处理，经过一番讨论和协商，最终讨论的结果是，修改需求文档，暂时先不支持中文，将中文支持放在下一个版本中去实现，因为中文对于当前的版本影响不大，客户群的语言主要是英语，而且当前版本的系统马上就要上线了，去修改中文的问题是可以解决，但是时间却又有限，考虑到这些因素，所以决定目前版本暂时不支持中文，下一版本再把中文的问题给解决掉。

其实以前的项目过程中，有接触过测试工作，但是都只是在系统实现完成以后，对系统的功能做一些针对性的测试，因为功能都是需求文档里面要求的，只要保证这些功能都是可用的、正确的就可以了，往往就会忽略了需求测试这样的一个环节，这其实也是非常重要的一个测试工作，说的更通俗一些，项目第一个工作就是要做好需求分析，分析过程中也包括测试，对需求的测试，这部分工作不仅仅只是停留在讨论之后，比如讨论决定需要这样的功能，应该也可以实现这样的功能，但更多的要考虑的其实很简单，就是如何去让系统变得更简单易用，甚至可以摒弃一些很少使用的功能，至繁归于至简，这是乔布斯的信条，开发者也应该学会如何做到归于至简，不仅使得系统简单，开发者也会减少不必要的工作，节省更多的时间用在必要的地方。

5.2 实现测试

UEFI 实现的测试，其实就是在系统实现完成的同时（与实现一起进行或者实现之后进行，通常在实现完成后），对实现的功能进行测试，看是否满足系统预期的要求，对于实现的测试，主要是依据系统的详细设计文档制定的测试文档来完成，在测试文档中，有对 API 着重测试的点有比较详细的说明，测试中的 function case 的设计就是针对这些重要的功能点来进行设计的，当然对于 API 测试的还有一些理论上的额外要求，可以将它们设计为 conformance case 来测试，但是如果功能不受影响的话，conformance case 的测试结果并不是判断系统有无 bug 的依据，但是 conformance case 从一定程度上反映了系统的稳定性、容错能力和恢复能力，因此也

尽量做到满足 conformance case 的设计期望。

对于 UEFI 的实现的测试，主要从一下几个方面进行：

5.2.1 单元测试

单元测试（又叫模块测试）是测试人员编写的一小段代码，用于检验被测代码的一个很小的、很明确的功能是否正确。通常而言，一个单元测试是用于判断某个特定条件（或者场景）下某个特定函数的行为，执行单元测试，就是为了证明这段代码的行为和期望的一致，所以根据测试文档里面的需求，对指定的 API 进行单元测试，我所测试的 API 是针对某一特殊用途的，更具体一点是在 build 之前完成添加第三方的文件，并完成分析、编译第三方文件的工作。首先是添加第三方文件，利用的函数 AddFileElement 来完成添加文件的功能，该函数的具体描述如下：

AddFileElement(self, TypeString, Path1, WorkspacePath=None, DscPath=None)

If TypeString is "UefiDriver", adds a driver to the current external module list.

If WorkspacePath is None, then the file at Path1must be a binary driver file.

If WorkspacePath is a valid Workspace path, then Path1 must be a .INF file.

If WorkspacePath path is other, then the function will return False.

If DscPath is None, then WorkspacePath must be valid,

and Path1 must be a AsBuiltInf file.

If DscPath is a valid DSC file and WorkspacePath is a a workspace

that DSC file is in, then Path1 muast be a SourceInf file.

If DscPath is not a DSC file, then the function must return False.

If TypeString is "UiFiles", then add a UI configuration file.

This must be moved into the directory that the Platform Project PDO file is in.

This file's information is recorded, but it is not parsed at all by config.

If the file cannot be found and moved or if WorkspacePath or DscPath is not None,

then return False, otherwise return True.

If TypeString is "PayloadFiles", then add a new payload file.

Path1 must point to a file (not a directory) and both

DscPath and WorkspacePath must both be None.

If the element is successfully created, then reuturn True, otherwise return False.

If Type is other than those specified, return False.

If the file is a duplicate, return False.

Keywork arguments:

Path1 - the path to the .EFI, AsBuiltInf, SourceInf, or Configuration file.

WorkspacePath - the path to the Workspace is Path1 is AsBuiltInf or SourceInf file, otherwise None.

DscPath - the path to the DSC file to use in a source build. WorkspacePath must be valid and Path1 must be a SourceInf. The DSC file must be in the workspace specified by workspacePath

从上面的函数描述中，可以看到该函数的主要功能是实现将指定路径下的第三方文件添加到 BIOS 目标 BIM 文件中。应用场景举例，比如用户想要在 BIOS 里面加入一个第三方的 Shell 菜单，那么就可以使用这个函数，将指定的 Shell 文件加入到 BIM 文件中，当系统 Build 完成之后，把生成的目标 BIOS 烧录到测试板子中，启动之后，就会在 BIOS 界面看到 Shell 菜单。在函数描述中，函数参数“TypeString”表示所要添加的文件类型，有三种类型，分别是“UefiDriver”、“PayloadFiles”和“UiFiles”。如果添加“UefiDrivers”的类型文件，那么若文件是 binary driver file（.efi 文件）的话，则后面的参数 WorkspacePath 必须为空，DSCPath 也必须为空，如果后面的两个参数有一个不为空，那么就会添加失败；若添加的文件是.INF file（.inf 文件）的话，那么参数 WorkspacePath 必须有值，并且必须是一个合法的值，详细的理解就是，首先路径必须存在，不能是一个不存在的路径，其次路径必须合法，也就是这个路径下存在所要添加的这个文件，然后才有添加成功可能，否则如果有任何一个条件并不满足，就一定会添加失败。对于不同类型的 INF file 又会有不同的操作，如果添加的文件是 binary inf file 类型，那么 WorkspacePath 是合法的，同时 DSCPath 必须为空；如果添加的文件时 source inf file 类型，那么 WorkspacePath 和 DSCPath 都必须是合法的，而且 DSCPath 必须和 WorkspacePath 是一致对应的。

在设计测试用例的时候，先设计测试正常的用例，也就是前面提到的 API 的正常功能，所以按照要求，若添加一个 binary driver file，那么就可以这样来写 AddFileElement(“UefiDrivers”,“C:\UEFI\AddFileElement\TestCase\shell.efi”)，这个用例

指定添加一个“UefiDrivers”类型的文件，是一个 binary driver file，在一个指定的路径“C:\UEFI\AddFileElement\TestCase”下面有一个 shell.efi 文件，如果路径正确，该路径下也有这样的一个文件，且其他条件也正确，那么调用这个函数成功之后，保存 BIM 文件，然后就会在保存的 BIM 文件中找到这样的一个改动：

[ExternalModules]

BinaryDrivers_1= C:\UEFI\AddFileElement\TestCase\shell.efi

看到这个就说明添加成功了。

同样的做法，如果要添加一个 INF file，若是 binary inf，这样来设计测试用例，AddFileElement(“UefiDrivers”,“C:\TestCase\shell.inf”, “C:\TestCase”)，该测试用例中，因为 shell.inf 是 binary inf 类型，所以 WorkspacePath 有值，是一个合法的路径，文件 shell.inf 在该路径下，同时 DSCPath 必须为空，如果添加成功，则在 BIM 文件中的结果为：

[ExternalModules]

BinaryInfs_1= C:\TestCase\shell.inf | C:\TestCase

如果要添加的是一个 source inf 类型的文件，那么测试用例应该是这样的 AddFileElement(“UefiDrivers”,“C:\TestCase\shell.inf”,“C:\TestCase”,“C:\TestCase\shell.dsc”)，由于添加的 shell.inf 是 source inf 类型的文件，所以 WorkspacePath 和 DSCPath 都不能为空，且都必须是合法的，DSCPath 是与 shell.inf 对应的，并且和 WorkspacePath 是保持一致的，如果添加成功，那么 BIM 中的显示是这样的：

[ExternalModules]

SourceInfs_1= C:\TestCase\shell.inf | C:\TestCase| C:\TestCase\shell.dsc

当然，除了测试这正常的功能外，还应该测试系统对错误输入的处理和恢复，比如，在添加 binary driver fie 的时候，这样来设计测试用例，WorkspacePath 不为空，AddFileElement(“UefiDrivers”,“C:\TestCase\shell.efi”, “C:\TestCase”)，那么按照文档的要求，系统应该返回 False，同理，在添加 binary inf file 的时候，我把 WorkspacePath 设为空，AddFileElement(“UefiDrivers”,“C:\TestCase\shell.inf”)，还有就是如果添加的文件不属于任何类型，比如要添加了一个第三方的 C 文件，测试用例这样写：AddFileElement(“UefiDrivers”,“C:\TestCase\shell.c”)，那么这个时候，再看系统是不是

也是返回为 False，上面的这些测试用例的设计是为了验证系统的稳定性和容错能力，因为要求系统要能够应对各种各样的复杂情况和异常情况，所以对于这些异常的输入，系统必须能够合理、正确、快速的反馈，使得用户能够继续操作下去，而不是遇到问题就停止或者退出。这些测试用例则满足了对于系统这些方面的测试和考察。

上面列举的三个测试用例只是针对 API 中的参数 TypeString 为“UefiDrivers”的类型进行的用例设计，如果参数 TypeString 为“PayLoadFiles”或者“UiFiles”，又要如何设计测试用例？在 API 中的描述里面，这两个类型的文件的添加过程是类似的，都是一个只有个文件路径，而且要求 WorkspacePath 和 DSCPath 都必须为空，否则就返回 False，那么对于 function case，就这样来设计：

AddFileElement(“ PayLoadFiles”,“C:\TestCase\shell.inf”)

AddFileElement(“ PayLoadFiles”,“C:\TestCase\shell.c”)

AddFileElement(“ UiFiles”,“C:\TestCase\shell.inf”)

AddFileElement(“ UiFiles”,“C:\TestCase\shell.doc”)

首先两者对于要添加的文件类型都不做限制，不像 UefiDrivers 那样，要求文件类型必须是 binary efi file，source inf file 或者 binary inf file 等，这两种类型所要添加的文件是任意的，比如测试用例中就加了.C 文件和.doc 文件，其次，文件的路径是正确的，该路径下存在这样的文件，最后，函数其它的参数都为空，这样的测试用例满足的 API 中的描述，那么结果就是这些文件都会被成功加入：

[PayloadFiles]

PayloadFiles_1 = C:\TestCase\shell.inf |e95a12f0-0783-47a2-a5fa-e4ac57e74996

PayloadFiles_2 = C:\TestCase\shell.c |e95a12f0-0783-47a2-a5fa-e4ac57e74997

[UiFiles]

UiFiles_1 = C:\TestCase\shell.inf

UiFiles_2 = C:\TestCase\shell.c

对于 conformance case 的设计，可以这样来设计：

AddFileElement(“ PayLoadFiles”,“ shell.inf”)

AddFileElement(“ PayLoadFiles”,“C:\TestCase\shell.c”, “C:\TestCase”)

AddFileElement(“ UiFiles”,“C:\TestCase”)

AddFileElement(“ UiFiles”,“C:\Test\shell.inf”, “C:\Test”, “C:\Test\shell.dsc”)

比如添加 PayLoadFiles 的时候，文件路径只给了文件的名字，而不是一个完整的绝对路径，或者文件路径是正确的，但是参数里面的 WorkspacePath 不为空；而添加 UiFiles 的时候，文件的路径是存在的，但是没有指定要添加的文件的名字，或者文件路径正确，但是 WorkspacePath 和 DSCPath 都不为空，这几个测试用力的返回结果都为 False，符合预期的要求。

单元测试的优点是将使测试者从用户的角度去观察、思考，能够帮助理解用户的行为，帮助提升用户体验。上面是单元测试的一个例子，当然还有很多其他的 API 也是遵循同样的设计思路和理念，透过单元测试，可以发现，单元测试能够帮助发现很多在开发阶段出现的 bug，很容易定位问题出现的位置和原因，使得开发人员也很容易修改，这大大降低了修改的成本和时间，在完成了单元测试的基础上再进行系统的集成测试，会使得系统的集成过程大大简化，避免了在集成测试过程中对于 API 功能正确性的怀疑，让测试人员更集中精神去定位其他地方的问题。

5.2.2 集成测试

集成测试，是在单元测试的基础上，将所有模块按照设计要求组装成为子系统或系统，进行测试。集成测试的目标是按照设计要求使用那些通过单元测试的构件来构造程序结构。

所以在完成了单元测试之后，就对单元测试的结果进行进一步的验证，以及对于不同单元的组合完成一个更大的功能进行考察和检验，集成测试就是基于上面单元测试的 API 的一个集成，完成的一个大的功能是，添加第三方文件到 BIM 文件中，然后对 BIM 文件进行 BuildUefiDrivers 操作，把 source inf 文件 Build 成为 binary inf 文件，然后在调用 PomUefiDrivers 进行分析，分析 binary inf 是否合法，是否满足系统所要的第三方的文件的条件和格式，然后进行 Save 保存结果到 BIM 文件中，最后进行 Build image 的操作，最终到的 BIOS image 文件，然后烧到板子里面，看添加的文件是否是可以正确的表现出来。

集成测试也是对一个大的功能的一个完整的操作流程的测试，根据系统要求，首先需要打开一个指定的已经存在的项目，或者新建一个新的项目（相当于新建一个项目，然后打开），这涉及到了 OpenProject 函数，然后再调用 AddFileElement 进

行文件添加的操作，添加完成后再做 BuildUefiDrivers 操作，将 source 文件都编译成 binary 文件，然后调用 PomUefiDrivers 来对编译完的 binary 文件进行解析，最后调用 Save 操作保存 BIM 文件，然后利用 Build 去编译 BIM 文件生成最终的 image，然后调用 CloseProject 去关闭项目，是这样的一个完整流程，然而在做第一步 OpenProject 的时候就出问题了，这个结果是在后面的时候表现出来的，出现的问题是在打开项目之后，发现原本项目里面的一个 source inf file，已经不再是 source inf file，因为某些测试用例的原因，会事先在 BIM 文件中放入 source inf file，按照前面的描述，打开项目之后应该是：

[ExternalModules]

SourceInfs_1= C:\TestCase\shell.inf | C:\TestCase| C:\TestCase\shell.dsc

然而结果却是：

[ExternalModules]

BinaryInfs_1= C:\TestCase\shell.inf | C:\TestCase

分析结果发现，项目中的 source inf 文件再打开后就被系统给编译成了 binary inf 文件，就是还没有进行 BuildUefiDrivers 的操作，就已经出现了 Build 后的结果，后来发现是因为 OpenProject 的时候，里面有调用 BuildUefiDrivers，所以才出现了这样的结果，这也是设计文档里面的设计，后来经过讨论，决定 OpenProject 应该简化，只做打开项目的操作和一些其它的初始化准备工作，而把 Build 的操作完全放由 BuildUefiDrivers 来处理，于是就能够正确的打开项目了，项目打开之后，接着要进行 AddFileElement 操作去添加第三方文件，在添加第三方文件的过程中，出现了一个小的插曲，就是对于一些类型的文件，即使指定了 TypeString 为“UefiDrivers”，但是添加完成之后，会发现原本放入[ExternalModules]这个 section 的文件最后却添加到了[PayLoadFiles]这个 section 下面去了，后来验证发现，原来设计的测试用例有一个特殊的例子，之前的文件类型都是 driver 类型，但是有一个 case 是 application 类型，按照需求即使指定添加到[ExternalModules]（通过把参数值设为“UefiDrivers”来指定），但是如果文件本身是 application 类型的，最后也要添加到[PayLoadFiles]下面去，所以才出现了上面的情况，不过这是一个正确的结果，在后面的测试用例中又

加了一些来专门测试这样的一个需求。

在添加完第三方文件之后，进行 Build 操作，通过测试这些也都没什么问题，接下来的一步 Pom 又出问题了，因为 Pom 主要是完成 binary inf 文件的解析工作，对于添加的文件需要进行分析，同时会把分析的日志记录到一个 log 文件中保存下来，如果有什么问题，可以知道是哪里出了问题，然后再对第三方文件的问题进行定位和修改，但是在测试过程中，却怎么也找不到 log 文件，后来发现，在做单元测试的时候，log 文件时放在指定的绝对路径下面的，直接就可以在指定目录下看到 log 文件，做集成的时候，log 文件是和 project 放在一起的，每一个运行一个 case 生成文件，然后在跑下一个 case 的时候会对环境进行一个清理，为跑下一个 case 做准备，同时就把 log 文件也给删除了，所以一直都找不到，后来在清理之前，做了一个判断，如果有 log 文件，就把文件给备份到指定的路径下，然后就在指定的路径下发现了 log 文件。

上面的这些操作都是为最终的 Build image 做铺垫，接下来就是先调用 Save 来保存一下 BIM 文件，然后再 Build 完成工作，在这两步操作中也遇到了阻碍，问题是这样的，做完一系列从操作后保存 BIM，然后可以看到 BIM 文件中已经添加成功的第三方文件，而且有些是已经编译好的，在指定目录下也可以看到 Pom 后的 log 文件，表明第三方文件是正确的，且已经添加成功，然而对于最后生成的 image 烧录到板子里面做测试，却发现没有第三方文件却没有进去，比如加入了一个 shell.efi，那么最终会在板子启动后，在 BIOS 菜单选项里面找到这样的一个选项 shell.efi，然而结果却没有，首先 Save 之前是没有问题的，Save 之后没问题，然后查看了一下 Save 之后保存的 BIM 文件，发现那些添加的文件都在 BIM 文件里面，说明是添加成功了，而且单独做 Save 操作和单独做 Build 操作都是正确的，但是 Save 和 Build 一起进行的结果却不是正确的，说明问题是出在 Save 和 Build 结合这一步上，通过比对、单步调试等多种方法，最终发现了问题的所在，原来 Save 操作里面有对 BIM 文件获取内容的操作，里面有 FindModuleSolution，而 Build 的里面也有 FindModuleSolution，在 Save 完成 FindModuleSolution 之后，会对 BIM 文件进行清理，同时为下一个 FindModuleSolution 做准备，所以当 Build 操作进行的时候，事实

上它是以 Save 的结果作为基础，由于 FindModuleSolution 的结果被清理了，所以 Build 的 FindModuleSolution 的结果为空，也就是最初的 BIM 文件，因此生成的 image 是没有添加的第三方文件的。所以把 Build 过程中的 FindModuleSolution 处理给去掉，只在 Save 的时候调用，Build 只用专一的完成编译工作就可以了。

通过这一系列的集成测试，保证了第三方文件的添加、编译、分析，BIM 文件的保存，以及最后 BIOS image 的生成这一过程得到了完整的测试，同时也发现了之前没有发现的一些问题，使得问题得到及早的解决，确保了系统功能的可行性和稳定性。

5.2.3 系统测试

系统测试是将经过集成测试的软件，作为系统计算机的一个部分，与系统中其他部分结合起来，在实际运行环境下对计算机系统进行的一系列严格有效地测试，系统测试的目的是验证最终软件系统是否满足用户规定的需求。

在 UEFI 的系统测试过程中，使用了一套完整的测试框架，重点介绍一下这个框架，这个框架跟一些现成的、比较成熟的框架比较，还有很多的不足，尚待补充完善，这样才能使它有更广泛的应用空间，但是对于当前系统的测试是绰绰有余的，该测试框架有三部分构成，第一部分是测试用例的设计和处理，通过第一部分可以得到处理过的测试用例，将它作为第二部分的输入；第二部分是系统的逻辑处理，将第一部分的测试用例输入系统，经过系统的逻辑的处理，得到测试结果；第三部分是结果的汇报整理与输出，第三部分将对于第二部分得到的测试结果进行整理、分析，将以报告的形式生成出来，比如 excel 文档或者 log 文件，通过邮件或其它方式呈献给用户。

首先，测试用例的设计和处理，在这一部分中，将设计的测试用例通过 excel 文档完美地展现出来，excel 的每一行是一个单独的测试用例，excel 的每一列是一个测试项，行列交叉处的值表明该测试用例对于该测试项的要求，比如规定值为 1 表示该用例包含本测试项，为 0 表示不包含本测试项，在设计过程中，可以使用控制变量法来保证任何一个测试项都能得到 case 的包含，使得一个 case 可以包含一个或多个同类别或者不同类别测试项，一个测试项可以出现在一个或多个测试用例中。比

如对于 BIOS 是否支持 debug 模式和是否支持 log 记录，就可以设计这么几个测试用例，如图 5.1 所示。

	A	B	C
1	CaseName	debug	log
2	case1	1	0
3	case2	0	1
4	case3	1	1
5	case4	0	0

图 5.1　测试用例设计表

case1：支持 debug 模式，不支持 log 记录；

case2：不支持 debug 模式，支持 log 记录；

case3：既支持 debug 模式，又支持 log 记录；

case4：既不支持 debug 模式，也不支持 log 记录；

当然，这只是一个举例说明一下测试用例的设计，当然在具体的应用过程中会更复杂。对于这样的一个测试用例表，通过第一部分的处理，主要就是读取 excel 表格，获取每一个 case 对应的测试项，因为做测试使用的是 Python 语言，所以会将从表格中得到的测试用例保存在一个字典中，像这样{case1:{debug:1,log:0}, case2:{debug:0,log:1}，case3:{debug:1,log:1}，case4:{debug:0,log:0}}，然后通过访问字典的关键字可以获取每一个 case，在每一个 case 中，对应的测试项的配置也是一个字典，通过访问 case 的每一个测试项得到该测试用例的每一项的设置，然后在第二部分根据这个来进行逻辑上的操作，每一个系统的配置表都大同小异，不同的系统根据不同的需求来对测试用例表进行自己的设计，比如，图 5.2 所示的 UEFI 系统的测试用例表：

TCSName	Note:	Question									
		Enable Size Optimization	Include UEFI Shell Boot Support	Include Yocto Boot Support	Enable UEFI X64 CPU Support	Enable Recovery File Generation	Enable Firmware Update File Generation	Firmware Recovery and Firmware Update Authentication Support	Firmware Recovery and Firmware Update Custom Signing Tool Path	Enable Image Authentication Support	Enable Authenticated Variable Support
TCS1		1	1	1	0	0	0	NA	NA	0	0
TCS2		1	1	1	0	0	0	NA	NA	0	0
TCS3		1	1	1	0	0	0	NA	NA	0	0
TCS4		1	1	1	0	0	0	NA	NA	0	0
TCS5		1	1	1	0	0	0	NA	NA	0	0
TCS6		1	1	1	0	0	0	NA	NA	0	0
TCS7		1	1	1	0	0	0	NA	NA	0	0
TCS8		1	1	1	0	0	0	NA	NA	0	0
TCS9		1	1	1	0	0	0	NA	NA	0	0
TCS10		1	1	1	0	0	0	NA	NA	0	0
TCS11		1	1	1	0	0	0	NA	NA	0	0
TCS12		1	1	1	0	0	0	NA	NA	0	0
TCS13		1	1	1	0	0	0	NA	NA	0	0
TCS14		1	1	1	0	0	0	NA	NA	0	0
TCS15		1	1	1	0	0	0	NA	NA	0	0
TCS16		1	1	1	0	0	0	NA	NA	0	0

表 5.2　UEFI 系统测试用例设计表

通常对于系统测试来说，每一个用例的测试过程是基本类似的，不同的是测试用例的测试配置项不一样，所以拿到所有的测试用例后，通过一个 for 循环来运行所有的测试用例，如何运行测试用例，就是第二部分要解决的问题了。

当通过第一步拿到测试用例后，第二步就要针对测试用例中的配置项进行判断，然后决定如何运行，比如上面举的例子，case1：（debug：1，log：0），那么在第二步中，判断如果 debug 值为 1，那么就调用 debug 函数来开启 debug 模式，否则不管；如果 log 值为 1，则调用 log 函数来记录 log，否则忽略。因此 case1 就会调用 debug 函数，case2 会调用 log 函数，case3 两个函数都会调用，case4 两个函数都不调用。每个 case 的处理是一样的，只是在完成判断之后，每个 case 的结果是不一样的。

对于 UEFI 系统，就比较复杂了，每个 case 要有 Question 的判断，还要有 ModuleInf 的判断，所以把同一类判断放在一起在进行，再通过一个循环来完成对每一个小的测试项的判断即可，比如 SetQuestion 的测试代码：

SetQuestion：

```
def SetQuestion(self, ConnectDict, InfQuestionDict, PlatformPdo2Obj):
        Result = True
        for InfQuestion in InfQuestionDict:
```

```
for PcdSettingElement in PcdSettingElements:
    PcdGuidName = PcdSettingElement.GetPcdGuidName()
    CurrentValue = PcdSettingElement.GetCurrentValue()
    if PcdGuidName.split(".")[1] == HdInfPcd:
        if type(HdInfPcdValue) is str:
            if HdInfPcdValue.isdigit():
                HdInfPcdValue = bool(int(HdInfPcdValue))
            elif "TRUE" in HdInfPcdValue.upper():
                HdInfPcdValue = True
            elif "FALSE" in HdInfPcdValue.upper():
                HdInfPcdValue = False
            else:
                HdInfPcdValue = int(float(HdInfPcdValue))
```

通过两层 for 循环，然后在对测试项的值进行 if 判断，不同的类型做不同的处理，所以每一个测试用例的结果都是不一样的。当然，测试结果是要保存起来的，然后在第三步对测试结果进行分析，得出测试报告。

在第二部分中，完成了测试用例的运行，将会得到测试结果，但是所得到的测试结果只是一个比较笼统的结果，如果结果失败了，那么应该是哪一步出现的问题导致这样的结果，为什么会出现这样的结果，这就是要对测试结果进行的操作，对于每一个 case 的每一步测试结果进行整理和分析，比如上面的例子中 case3 出了问题，那么到底是在调用 debug 出现问题，还是 log 出现问题，就要对 debug 函数的结果做记录，同时对 log 函数的结果做记录，最终的结果是由这两个结果一起决定的，只有两个都正确，结果才正确，任何一个失败，结果就是失败。如何将结果展示给用户，也使用 excel 表格，在测试用例的表格后面加入几列，分别记录每一步的测试结果，然后对与最终结果进行汇总，又前面的结果得到最终结果，如表 5.3：

	A	B	C	D	E	F
1	caseName	debug	log	debug_res	log_res	result
2	case1	1	0	TRUE	NA	TRUE
3	case2	0	1	NA	FALSE	FALSE
4	case3	1	1	TRUE	FALSE	FALSE
5	case4	0	0	NA	NA	NA

表 5.3　测试结果表

从表中可以发现每一个 case 每一步的测试结果，如果没有调用，则结果为“NA”，表示默认结果；对每一步结果做&操作得到 result，所以只要有一步失败，则 result 结果为 FALSE，而且也清楚地知道哪一步出现了错误，这样在定位错误的时候就更具有针对性。同样的，对于 UEFI 也是如此，如表 5.4：

TCSName	Note:	ConfigResult	FindModuleSolution Result	AssemblyResult
TCS270	all feature	Pass	Pass	Pass
TCS271	all feature	Pass	Pass	Pass
TCS272	part feature	Pass	Pass	Pass
TCS273	part feature	Pass	Pass	Pass
TCS274	Shell1.0 for	Pass	Pass	Pass
TCS275	Shell1.0 for	Pass	Pass	Pass
TCS5		Pass	Pass	Pass
TCS110		Pass	Pass	Fail
TCS26		Pass	Pass	Pass

表 5.4　UEFI 系统测试结果表

UEFI 的系统测试就是采用上面的测试框架完成的，当然，这个测试框架的应用还不止如此，在 Increment Build、clean Build、UPT 等系统的测试中都使用了这样的框架，整个框架对于测试用例的处理、测试用例的运行和测试结果的汇总都是大同小异的，会有一些通用的函数提供最基本的功能，当然在每一个系统中，具体的实现是有一定差异的，但这并不妨碍测试框架的整体，所以通过验证发现，该测试框

架是可行的，可以应该用到 UEFI 的系统测试中。

在 UEFI 的系统测试中，主要测试了系统从开始到关闭的一个完整的流程，模拟了用户的一个常规操作：启动系统，打开一个项目，完成 Question 设置，选择需要的 Module INF，完成 AddConnection 操作，然后设置某些 Module 的 PCD 值，再调用 Save 保存 BIM 文件，最后 Build 生成 image 文件，关闭系统。在这一轮的测试中，系统功能正常，有较强的容错能力和恢复能力。

在系统测试中，还做了压力测试，因为系统测试的测试用例设计并不是完全覆盖所有的测试点，首先，对所有应该测试的测试项都设计了测试用例，然后在剩下的测试项中，随机组合生成一个新的测试用例，所以初步的测试用例有 100 多个；其次，每一个测试用例的运行时间大概有 2 分钟左右，有些 case 甚至更长，所以跑议论测试需要三四个小时，但是要求每天都要跑测试报告。所以最终决定系统测试用例个数为 100+，覆盖了大部分的测试点，但是要做到覆盖所有测试项，总共需要 1300 多个 case，通常需要跑一天多，等同于压力测试，所以为了验证系统的稳定性，设计了压力测试，将所有的测试点的各种合理的组合测试一遍，同样使用了测试框架，通过测试，最终得出结论：系统功能正常、运行稳定，可以交付。

5.2.4 UI 测试

UI 测试，也即用户界面测试，测试用户界面的功能模块的布局是否合理，整体风格是否一致和各个控件的放置位置是否符合客户使用习惯，更重要的是要符合操作便捷，导航简单易懂，界面中文字是否规范，功能是否可以正常使用，是否符合公司或行业的标准。

UEFI 的 UI 测试主要是验证系统的功能，按钮是否都可以使用，输入框可以输入内容，复选框是否可以选中，下拉菜单是否可以正常使用等等，此外，从用户体验方面来讲，界面的风格是否简洁大方，操作起来是否简单方便，能够和容易找到一些功能键，对于一些输入的提示是否足够清楚明白，系统的解释和帮助信息是否通俗易懂。

通过 UI 测试，发现用户界面比较简洁，快速可以找到菜单，打开项目，改变视图方式，进行 Build 操作，经过测试，各个功能也都是可用的，输入框、复选框、下

拉菜单等都是正常的，鼠标放到指定位置会有相应的解释，因此，该系统的界面设计是比较成功的。

5.3 本章小结

本章对 UEFI BIOS 系统进行了全面的测试，使用基本的测试方法，在深入理解系统的基础上，从不同的角度寻找系统中潜在的缺陷和 bug，发现并修正系统出现的问题，使得系统能够满足用户的要求，经过测试，得出结论，系统功能正常，运行稳定，具备较强的容错能力和恢复能力，可以交付使用。

6 总结与展望

6.1 全文总结

作为连接操作系统和系统硬件体系之间的桥梁，BIOS 采用汇编语言编写，综合应用多种技术，使用寄存器参数调用的方法，并在很小的内存上进行固定编址。BIOS 在各个方面都有着严重的不足，这也制约着它的进一步发展。在这样的情况下需要一个更为先进、更为合适的替代品出现，于是 UEFI 就应运而生了，UEFI 和传统 BIOS 相比具有明显的优势，比如使用 C 语言开发，摒弃静态链接而采用动态链接，并以模块化的风格来构建系统，使得 UEFI 扩展性极强，与传统的 BIOS 相比，UEFI 更倾向于是一种规范，UEFI 程序不是固定不变的，可扩展性强，很灵活，有丰富的文档供开发者参考。另外，UEFI 驱动程序的实现是由字节代码编写的，字节代码是能够被各种 CPU 所识别的，所以 UEFI 可以兼容各种类型的 CPU。

UEFI 系统，使得用户可以根据实际需求，添加、删除相应的驱动，并实现了丰富的自定制功能扩展，用户可以依据自己的硬件设备和需求来设置自己的 BIOS 的功能。UEFI 在开机后参与了大量的初始化工作，包括硬件设备检测等，此外为了将操作系统解放出来，免去重装操作系统后需要重新安装驱动的复杂工作，UEFI 还承担了加载硬件的驱动程序工作，这也大大缩短了开机时间。

经过多方面的测试，已经证明了 UEFI 是符合用户需求、功能完善、运行稳定、性能良好的新一代 BIOS，UEFI BIOS 取代传统 BIOS 已成定局。

6.2 展望

UEFI 定义了很多接口规范，使用 C 语言开发，摒弃静态链接而采用动态链接，并以模块化的风格来构建系统，使得 UEFI 扩展性极强。UEFI 规范可移植性好的优点，大多数 UEFI 代码在做 X86 平台和 ARM 平台之间互相移植的时候都几乎不需要修改源代码，只需要重新编译一次既可。

在 UEFI 框架流程的第一个阶段：SEC 阶段，会做一些检查工作。SEC 也就成

为固件平台的可信链的可信根，从而顺序建立平台的可信链。

UEFI 利用了公钥密码、单向函数、数字签名、公钥证书和可信根等密码体制实现了完整性和鉴别授权验证，包括 UEFI 安全启动（Secure Boot）

UEFI 安全启动是为了提供通用认证信息设计的协议。它包括 UEFI 用户身份鉴别(User Identification)和 UEFI 认证变量(Authenticated Variable)。UEFI 用户身份鉴别是针对不同用户认证设备（如智能卡、智能令牌和指纹识别设备等）的标准认证框架，使用 UEFI HII 向用户显示信息，引入了可选的策略控制机制，允许对设备连接、映像加载和配置接口访问等进行细粒度的权限控制。UEFI 变量是平台有价值的数据，定义了基于非对称密钥技术的写保护变量服务，UEFI 预定义变量允许平台模式的切换以及固件和操作系统间进行密钥交换。使用 UEFI 安全启动这些安全特性有助于实现更好的基于平台与身份的鉴别，比如操作系统加载前的认证、单点登录、多因子认证，以及实现更加灵活的访问控制策略等。在 UEFI 安全启动过程中，系统通过 UEFI 变量对驱动和映像进行签名验证，保证了系统平台的安全性。因此，可以预见 UEFI 在平台安全性方面还有很大的发展空间。

致 谢

光阴似箭，日月如梭，二年半的研究生生涯转瞬即逝，随着论文的尘埃落定，自己的学生生涯也将宣告结束，内心还是会有些留恋和不舍，留恋大学生活的美好，不舍自己的同学、导师和母校。在研究生期间，自己得到了很多人的帮助和指导，无论是在学习中、生活上，还是在外实习期间，借此机会，对他们表示真诚的感谢和由衷的祝福。

首先，我要衷心感谢我的导师黄立群老师，本科的时候黄老师就给我留下深刻的印象，研究生有幸成为黄老师的学生，得到了黄老师的关心、指导和信任，记得一次去企业参加年会，黄老师关心我们的出行安全，直接和公司联系，要求公司对我们安全作出保障，这让我十分感动。黄老师和蔼可亲的性格、治学严谨的态度，低调内敛的品质也一直影响着我，为我树立了学习的榜样，在黄老师的教导下，自己学到了不少的知识，并且能力也得到了极大的锻炼和提高。此次论文的撰写也得到黄老师极大地帮助，再次感谢黄老师。

其次，要感谢我的同学，和我一起保送本院的我的同学傅强、刘平、白兆宁，此外还有我宿舍的好伙伴牛龙，感谢你们对我的帮助，不仅是学习上的，还有生活上的和精神上的，人生有你们才精彩。感谢我们实验室的其他同学，杜想、陈书仪和李静贵，融洽的实验室氛围，使得我们这个集体更加团结和谐，谢谢你们对我的帮助。

然后，我要感谢自己在 Intel 实习期间，帮助过我的各位同事。感谢他们在我刚进去的时候对我的指导和关心，让我能够很快的适应了实习的生活，感谢他们对我工作上的引导和帮助，使我能够高质量、高效率的完成自己的任务，得到 manager 的信任和表扬，谢谢你们让我的实习生活变得更有意义。

最后要感谢我的家人，感谢我的女朋友，谢谢你们给我的关心和爱护，你们的支持和信任是我勇往直前的巨大动力。

参考文献

[1] 张道华. 传统 BIOS 终结者——UEFI. 电脑爱好者, 2013(13): 72-73

[2] 韩德强, 马骏, 张强. UEFI 驱动程序的研究与开发. 电子技术应用, 2014(5): 10-13

[3] 裘文锋. UEFI 让系统安全从启动开始. 电脑迷, 2013(8): 44-45

[4] 潘登, 刘光明. EFI 结构分析及 Driver 开发. 计算机工程与科学, 2006(2): 115-117

[5] 余超志, 朱泽民. 新一代BIOS——EFI、CSSBIOS技术研究. 科技信息, 2006(5): 14-15

[6] 吴松青, 王典洪. 基于 UEFI 的 Application 和 Driver 的分析与开发. 计算机应用与软件, 2007(2): 98-100

[7] 周洁, 谢智勇, 余涵等. 基于UEFI的国产计算机平台BIOS研究. 计算机工程, 2011(S1): 355-358

[8] 王晓箴, 刘宝旭, 潘林. BIOS 恶意代码实现及其检测系统设计. 计算机工程, 2010(21): 17-18

[9] 付思源, 刘功申, 李建华. 基于UEFI固件的恶意代码防范技术研究. 计算机工程, 2012(9): 117-120

[10] 刘佳, 辛晓晨, 沈钢钢等. 基于 UEFI 的 Flash 更新的开发研究. 计算机工程与设计, 2011(1): 114-117

[11] 王晓箴, 于磊, 刘宝旭. 基于 EDK2 平台的数据备份与恢复技术. 中国计算机报, 2011(15): 262-264

[12] 刘冬, 文伟平. EFI 及其安全性分析. 信息网络安全, 2009(5): 32-34

[13] 黄海彬, 金晶. 基于 EFI 系统的多文件系统解决方案, 2010(6): 122-126

[14] 张朝华. 基于 EFI/Tiano 的协处理器模型的设计与实现: [硕士学位论文]. 上海: 上海交通大学图书馆, 2007

[15] 崔莹, 辛晓晨, 沈钢钢. 基于 UEFI 的嵌入式驱动程序的开发研究. 计算机工

程与设计, 2010, 31(10): 2384-2387

[16] 李振华. 基于 USBKEY 的 EFI 可信引导的设计与实现: [硕士学位论文]. 北京: 北京交通大学图书馆, 2008

[17] 万象. 基于 UEFI 系统的 LINUX 通用应用平台的设计与实现: [硕士学位论文]. 上海: 上海交通大学图书馆, 2012

[18] 杨荣伟. 基于 Intel 多核平台的 EFI/Tiano 图形界面系统研究: [硕士学位论文]. 上海: 上海交通大学图书馆, 2007

[19] 邱忠乐. 基于PC主板下一代BIOS的研究与开发: [硕士学位论文]. 南京: 南京航空航天大学图书馆, 2005

[20] 吴广, 何宗键. 基于UEFI Shell的Pre-OS Application的开发与研究. 科技信息, 2010(1): 15-17

[21] 邢卓媛. 基于 UEFI 的网络协议栈的研究与改进: [硕士学位论文]. 上海: 上海交通大学图书馆, 2011

[22] 曾颖明, 谢小权. 基于 UEFI 的可信 Tiano 设计与研究. 计算机工程与设计, 2009(11): 2645-2648

[23] 任超. EFI 技术的实现与应用研究: [硕士学位论文]. 上海: 上海交通大学图书馆, 2005

[24] 王兴欣. 基于 EFI 和多核体系的软件运用架构: [硕士学位论文]. 成都: 电子科技大学图书馆, 2011

[25] 唐文彬, 陈熹, 陈嘉勇. UEFI Bootkit 模型与分析. 计算机科学, 2012(4): 8-10

[26] 陈文钦. BIOS 研发技术剖析. 第 1 版. 北京: 清华大学出版社, 2001: 30-44

[27] 周伟东. 基于 EFI BIOS 的计算机网络接入认证系统的研究与实现: [硕士学位论文]. 西安: 西安电子科技大学图书馆, 2008

[28] 高云岭, 庄克良, 丁守芳. UEFI+BIOS 全局配置数据库的设计与实现. 物联网技术, 2014(10): 52-54

[29] Intel Corporation. Legacy BIOS and UEFI Boot Process. Intel Corporation SSG, March, 2008

[30] Intel Corporation. EDKII user manual Revision 1. 0. The United States: Intel

Corporation, 2011: 10-12

[31] Intel Corporation. Intel 64 and IA-32 Architectures Software Develop's Manual. The United States: Intel Corporation, 2009: 10-778

[32] Intel Corporation. EDK II Extended INF File Specification Revision. Version 1. 24, 2013

[33] Intel Corporation. EDK II Package Declaration (DEC) File Specification Revision. Version 1. 24, 2013

[34] Intel Corporation. EDK II Platform Description (DSC) File Specification Revision. Version 1. 24, 2013

[35] Intel Corporation. EDK II Flash Description (FDF) File Specification Revision. Version 1. 24, 2013

[36] Intel Corporation. Driver Writer’s Guide for UEFI 2. 0. Revision 0. 95, 2009: 9-51

[37] Intel Corporation. EDKII Platform Configuration Datebase Infrastructure Description. Revision 0. 55, 2009: 11-13

[38] Intel Corporation. Pre-EFI Initialization (PEI) Overview. Intel Corporation SSG, March, 2008

[39] Intel Corporation. Driver Execution Environment (DXE) Overview. Intel Corporation SSG, March, 2008

[40] Intel Corporation. Intel UEFI Packaging Tool. Revision 040, 2013

[41] Intel Corporation. EDK II Build Specification Revision. Version 1. 24, 2013

[42] Mark Lutz. Programming Python, 3rd Edition. O'Reilly, August 2006

[43] Intel, HP, Microsoft, Advanced Configuration and Power Interface Specification. Revision4. 0, 2009: 5-496

[44] Intel Corporation. UEFI Driver Development Driver Development Training, Adding Code Lab. Intel Corporation SSG, March, 2008

[45] Intel Corporation. Supplemental OS-BIOS interface-ACPI Interface. Intel Corporation SSG, March, 2008