

分类号_____

学校代码 10487

学号 M201175989

密级_____

华中科技大学

硕士学位论文

基于 EDKII 框架的 固件生成工具设计与实现

学位申请人 胡侠情

学 科 专 业：软件工程

指 导 教 师：肖来元 教授

答 辩 日 期：2014.1.6

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

Design and Implementation of the Firmware Build Tool Based on EDK II

Candidate : Hu Xiaqing

Major : Software Engineering

Supervisor : Prof. Xiao Laiyuan

Huazhong University of Science and Technology

Wuhan 430074, P. R. China

January, 2014

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名:

日期: 年 月 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密 ☐， 在 _____ 年解密后适用本授权书。
 本论文属于
 不保密 ☐。

(请在以上方框内打“√”)

学位论文作者签名:

日期: 年 月 日

指导教师签名:

日期： 年 月 日

摘要

随着计算机硬件和体系结构的不断发展，人们对计算机性能的追求越来越高。为了替代传统 BIOS，业界制定出了下一代 BIOS 固件技术的国际标准 UEFI(Unified Extensible Firmware Interface)：统一可扩展固件接口。

EDKII 是一个开源的 UEFI BIOS 的发布框架，它里面包含了各种具有依赖关系的功能模块、描述文件和底层库函数。固件生成工具可以根据开发源码自动构建适用于不同架构平台的固件，从而大大提高开发效率。本课题即是设计和实现一个基于此目的，可以在 Windows 和 Linux 平台上运行，为 IA32 和 X64 两种目标架构生成固件文件的工具。本文研究的主要内容有以下几个方面：

介绍了 UEFI 的背景和发展现状，分析了 UEFI BIOS 的工作流程和体系结构，然后详细分析了 EDKII 开源框架的内部结构和功能模块。

确定了固件生成工具的三个主要功能模块：自动生成模块、编译模块和固件生成模块。自动生成模块负责提取配置信息、构建目标和构建规则并生成编译链文件、中间文件和 makefile 文件；编译模块负责调用编译工具、链接工具和第三方工具将上一阶段输出的文件生成符合 UEFI 规范的 EFI 文件；最后固件生成模块根据构建目标和规则生成最终的固件镜像文件。

最终完成了预期目标，实现了一个跨平台运行的、可为 IA32 和 X64 两种目标架构生成固件文件的工具。

关键词：固件生成工具 跨平台 统一可扩展固件接口

Abstract

With the continuous development of computer hardware and architecture, the pursuit of computer performance is higher and higher. In order to replace the traditional BIOS, International standards developed next-generation BIOS firmware UEFI : Unified Extensible Firmware Interface.

EDKII is an open source release framework of UEFI BIOS, it contains a variety of dependency relationship function modules, description documents and library function. Build tool can be suitable for different structure platform firmware according to the automatic source construction development, and then the develop efficiency improved greatly. This subject is to design and implement a build tool based on this purpose, it can operate in both Windows and Linux platform, which can generate the firmware file construction tools for IA32 and X64 two target structure. The main studies of this paper has the following several aspects:

The article introduced the background and the developing situation of UEFI , and also analyzed the working process and system structure of UEFI BIOS, then analysed the internal structure and function module of EDKII open source framework in detail;

We design the three main functional modules: AutoGen module, build module and imageGen module. AutoGen module is in charge of extracting configuration information, constructing targets and rules and then generate compilation chain files, temporary files and makefile files; Make modules is in charge of calling compilation tools, linking tools and the third-party tools, and then use the output files from previous stage to generate EFI files in accord with the standards. Finally, imageGen module generate the final firmware image files according to the constructing target and rules.

This article achieved its expected aim finally, accomplished a build tool which can run cross-platform for two targeted architecture IA32 and X64 to generate firmware files.

Key words: Firmware build tool Cross-platform

Unified extensible firmware interface (UEFI)

目 录

摘 要.....	I
Abstract	II
1 绪论	
1.1 研究背景	(1)
1.2 研究目的和意义	(2)
1.3 国内外研究现状和发展趋势.....	(2)
1.4 本文的主要研究内容.....	(5)
2 相关技术分析	
2.1 UEFIBIOS 框架分析.....	(7)
2.2 EDKII 开发框架介绍.....	(14)
2.3 本章小结	(18)
3 固件生成工具总体设计	
3.1 系统需求分析	(19)
3.2 系统结构设计	(19)
3.3 系统功能模块设计	(20)
3.4 平台配置数据库设计.....	(22)
3.5 文件类型设计	(23)
3.6 本章小结	(27)
4 固件生成工具详细设计	
4.1 自动生成模块的设计.....	(28)
4.2 编译模块的设计	(33)
4.3 固件生成模块的设计.....	(35)
4.4 本章小结	(39)

5 固件生成工具的实现

5.1 系统开发环境 (40)

5.2 系统功能实现 (40)

5.3 本章小结 (55)

6 系统测试

6.1 WINDOWS7 64 位平台上的系统测试 (56)

6.2 LINUX 平台上的系统测试..... (62)

6.3 本章小结 (64)

7 总结与展望

7.1 全文总结 (65)

7.2 展望..... (66)

致 谢..... (67)

参考文献..... (68)

1 绪论

1.1 研究背景

传统的 BIOS 是由汇编语言编写，在编写、调试和维护阶段都很不方便。同时，随着计算机硬件和体系结构的不断发展，传统 BIOS 对一些新的硬件产品的支持也越来越差。所以急需出现一种新的 BIOS 技术。Intel 作为芯片生产商的领头者，在传统 BIOS 与新架构的处理器越来越不兼容的情况下，提出了可扩展固件接口 EFI(Extensible Firmware Interface)的固件结构，同时开启了 Tiano 项目作为对 EFI 的技术支持^[1]。在看到了 EFI 对比于传统 BIOS 的巨大优势后，众多的计算机、软件的著名公司如微软、IBM、Intel、惠普等组建了一个基于 EFI 的论坛-UEFI(Unified Extensible Firmware Interface)，Intel 将 EFI 技术的核心部分 EDK(EFI Develop Kit)开源，业界也基于此制定出了下一代 BIOS 固件技术的国际标准 UEFI(Unified Extensible Firmware Interface)^[2]。

统一可扩展固件接口是业界制定的关于旧的 BIOS 技术的升级版本，它为操作系统和平台固件的交互设计了一种新的模型，这个模型由多个特定意义的数据表组成，包括了一系列平台相关信息以及能有效的启动操作系统和对载入程序进行实时服务请求。它们集中为启动操作系统和预导入应用程序的运行定义完整的环境。在 UEFI 的开源社区中，与 UEFI BIOS 相关的开源项目有四类，分别是 EDK、EDKII、EFI Shell 和 EFI Tool Kit^[3]。其中，EDKII 是新一代的开源 EFI BIOS 的发布框架，包含一系列的开发示例和大量的底层库函数。UEFI 较之传统 BIOS 的启动机制优势更明显，它通过采用模块化、动态链接和 C 语言风格的参数堆栈传递方式的形式构建的系统，实现方式更为灵活^[4]。

EDKII 是一个开源的 UEFI BIOS 的发布框架，它里面包含了各种具有依赖关系的功能模块、描述文件和底层库函数^[5]。EDKII 作为新一代 BIOS 研究中的一部份，开发者在对此框架进行研究开发时，更需要在工作平台上运行和测试，因此固件生成工具可以解决为开发者自动创建适用于不同架构平台的固件。而基于 EDKII 框架

的开发，最常见的是在 Windows 和 Linux 工作平台下开发，开发完成后需要在不同的平台下进行测试和实现，此固件生成工具方便为开发人员提供一个方便操作和配置的固件生成环境。

1.2 研究目的和意义

随着 EFI Framework 的开放性和可移植性以及 UEFI 论坛的发展吸引了越来越多的开发者投入到新一代 BIOS 的研究中。EDKII 作为新一代 BIOS 研究的一部份，开发者为了研究和开发，更需要在工作平台上运行和测试，而该工具就可以解决 EDKII 没有固件生成工具的现状。本文的实现语言采用 Python，在执行 Python 程序之前无须编译 Python 代码，这大大提高了固件生成工具在跨平台运行的操作性和移植性。

课题的研究内容是实现一个能运行在 Windows 平台和 Linux 平台上，可为不同的架构如 IA32，X64 等创建固件，为用户提供一个易于操作，兼容性较好且具有良好的可扩展性的固件生成工具。该固件生成工具最终的实现集成了编译，链接，封装的各个功能，为开发人员节约了时间成本，大大提高了工作效率，使其更加受到业界的支持，从而加速 UEFI BIOS 在业界内的推广及应用。

1.3 国内外研究现状和发展趋势

1.3.1 国内外研究现状

Intel 在 2000 年首先发布了 EFI1.0，以此作为下一代 BIOS 技术的技术规范，为全新类型的 PC 固件的体系结构、接口和服务提出的建议标准。UEFI 中图形化的硬件设置界面主要目的是为了提供一组在 OS 加载之前在所有平台上一致的、正确指定的启动服务，被看做是有近 20 多年历史的传统 BIOS 的继任者。因为硬件发展迅速，传统 BIOS 成为发展缓慢，并且对一些新的硬件产品兼容性也越来越差，所以传统 BIOS 技术也渐渐地成为了制约计算机性能提升的瓶颈。为了解决这个问题，设计一个新的 BIOS 被逐渐提出。UEFI 属于开源项目，目前版本为 2.1。2004 年，由于越来越多的开发者们发现了 EFI 的优势，于是开发技术联盟成立了 UEFI 论坛，而 Intel 也将 EFI 核心技术开源化，使得 UEFI 产权归属于整个 UEFI 联盟，从此，每一个版

本的 UEFI BIOS 技术标准都由 UEFI 联盟发布。该联盟发布了两个最新的规范：UEFI2.3 和新的平台初始化规范 PI1.2。前者是用来定义系统固件与操作系统或其他高级软件(包括固件驱动程序)之间的接口，后者用来保证由不同企业如芯片厂商，固件开发商和维护固件代码的组织等提供固件组件之间的相互操作性。联盟的工作小组包括规范工作组(USWG)、测试工作组(UTWG)、平台初始化工作组(PIWG)及业界联络工作组(ICWG)。四个工作组通过对行业进行大量、多样化的教育和推广，促使业界尽快认识和采用 UEFI 标准^[6,7]。

目前已经有部分 PC 制造商开始在嵌入式装置上使用 UEFI，等到主板厂商开始顺应潮流后，就会快速普及。约在 2011 年，UEFI 就会开始普及，而眼下就是新旧时代的交替点。UEFI 将是近 3 年的趋势，到时候对于 PC 的利用以及维护都将步入一个新的时代。从发展趋势来看，业界将 BIOS 转移至 UEFI 的趋势已经不可避免，BIOS 即将被抛弃^[8,9]。UEFI 将会用更加友好的界面系统来代替传统 BIOS 的文字界面。UEFI 对网络协议一定有更佳的支持，因此上层管理者可更轻松的进行远程管理。至于对一般消费者而言，UEFI 的最大优点就是启动速度提升。目前 BIOS 的启动速度约需 25~30 秒才能进入操作系统加载画面，但 UEFI 仅需几秒。早在 2008 年，微星就有相关的 UEFI 产品推出，但最终因为成本过高而放弃。如今，用户的需求增强，存储技术也有了很大的进步，使得 UEFI 的实施工作变得更加简单，成本降低。随着 Intel 6 系列芯片组的来临，不少厂商都已经陆续曝光自家的 UEFI BIOS 的开机画面，UEFI BIOS 取代传统的 BIOS 是大势所趋。相信更人性化，更立体的 BIOS 设计将会很快呈现在我们面前^[10-12]。

国内的研究 BIOS 的公司较少，其中名气较大的是南京百敖软件有限公司，该公司成立至今已有 8 年的时间。该公司在掌握 UEFI BIOS 核心技术的基础上，对原型系统进行开发，实现了 UEFI BIOS 系统从无到有的突破。该公司与 Intel TIANO 项目组有着深度的合作关系。

1.3.2 发展趋势

当前情况下，越来越多的开发者和硬件开发公司将目光转入到 UEFI BIOS 开发。

而基于 EDKII 框架的开发和测试也迫切需要一种快速、稳定的固件生成工具作为载体来对工作平台进行功能测试，对一个能够在多种平台下对 EDKII 框架构建成为固件的工具的需求就十分迫切。而此固件生成工具的基本宗旨是要方便开发者，因此它要兼顾到操作简便、扩展性、兼容性和稳定性良好等要求，同时对开发者基于 EDKII 的开发和构建固件的阶段进行解耦，缩短开发时间，提高开发效率，进而加速业界对 UEFI BIOS 的应用，同时这也是本工程设计的价值所在^[13,14]。

业界将 BIOS 转移至 UEFI 的趋势已经不可避免。以 IBM 为例，早在 2007 年 IBM 就启动了基于 EDK 核心开发的 UEFI 试点项目，2009 年推出满足 UEFI2.1 和 PI1.1 规范的 X86 产品 Xeon*5500 服务器，到 2010 年，推出了基于 EDKII UEFI 核心的 Xeon6500/7500^[15]。IBM 在 UEFI 的项目开发主要致力于更简易的平台配置和管理，简化错误处理机制，用户可以通过高级设置对机器进行远程配置，通过网络对固件进行更新，超越传统 BIOS 的性能，允许挂载多个网络适配器，用 64 位模式来启动，而传统 BIOS 只有 16 位^[16]。UEFI 相对于传统的 BIOS 能支持大于 2T 的磁盘存储设备。在 2006 年开始采用 UEFI 的 PC 机还非常少，但到 2009 年底采用 UEFI 的 PC 系统全球大概有 3 万台 PC 和服务器的，其中一半已经采用 UEFI 和核心的 BIOS。现在仅 IBM 就已向客户发货超过 100 万的 UEFI 标准的 IBM x 系统的服务器^[17]。

在前期的详细调研和工程的需求分析和概要设计阶段的进行过程中，根据对 UEFI BIOS 的运行机制和复杂度分析发现，UEFI BIOS 类似于一个运行在硬件底层的操作系统。要实现该固件生成工具的功能，在技术上有诸多难点。首先，目前国内外对 UEFI BIOS 和 EDKII 框架的研究大部分处于源码设计阶段，而构建固件的工具的设计基本处于小作坊模式，比如特定的项目组根据自己的开发平台设计了一个便捷性和兼容性都较差的工具，只能运行在自己的项目环境中，耦合度很高，难以进行复用和维护，一旦更换某个硬件或不同的架构平台，可能整个工具的构建逻辑就要更改，而在硬件更新频繁的今天，这显然提高了开发成本，降低了开发效率。而且开发者还要对源码和工具进行繁琐的链接、调试和编译等工作，降低了开发进度^[18,19]。例如小组决定使用耦合度高，维护性差的固件生成工具，原始开发成本估算量为 2 人月，由于硬件厂商对定制的 UEFI BIOS 的功能的更新周期为 4 到 5 个月，

所以软件生命周期估算为 4 个月，而软件生命周期内的维护成本总共约为 1 人月，所以开发成本约为 3 个月，因为软件生命周期只有 4 个月，所以这种固件生成工具的开发成本与使用价值比较低，不适合在接下来 UEFI 快速发展的阶段使用。而本工程项目的开发成本约为 8 个月，而只要 UEFI BIOS 的发布框架 EDKII 不更新到 EDKIII(架构上的更新，至少 2 年内不会)，软件就一直处于其生命周期中，所以工程的开发价值较高。而本固件生成工具的设计难点就是要在一个缺乏相应的技术资料和项目经验的前提下完成，开发瓶颈在于如何对构建环境和硬件环境进行兼容，这就涉及到对固件生成工具的内部功能实现，例如如何生成 AutoGen 文件、PCD 机制的最优化设计、如何根据规范设计生成固件的文件结构都需要详细的研究分析。

1.4 本文的主要研究内容

课题的主要研究内容有：首先分析 UEFI BIOS 的背景和工作机制；研究 EDKII(符合 UEFI 规范的一种开源的开发框架)，对其结构和内部文件进行详细的分析；然后阐述了固件生成工具的整体设计思想和运行逻辑，最后对固件生成工具系统的三个重要功能模块：自动生成模块、编译模块、固件生成模块进行设计和实现。在实现功能的层次上本课题主要要解决以下两点内容：

1) 为了提高固件生成工具的通用性、可操作性和可移植性，该固件生成工具需要能够具备以下几点功能：(1) 工具能够跨平台运行，包括 Windows 和 Linux 系统。

(2) 为了能够使用户能够最方便的进行构建操作，系统采用建立编译链的方案，这就需要为每个平台自己的编译工具进行相关配置。

2) 如何去设计针对不同固件平台架构、不同构建目标的固件文件的实现逻辑。固件平台的架构有 IA32 和 X64 两种，EDKII 的源码中有针对不同平台架构的描述信息和配置文件，而外部输入也可以对某些文件内容重新定义。如何设计一个最优化的解析定义文件和配置文件的逻辑，对整个固件生成工具系统至关重要，是保证固件生成工具正常运行的第一步。

本论文的内容结构如下：

第一章 绪论。本章重点介绍了该课题的选题背景和研究意义，并从国内外的研

究现状和发展趋势来说明课题研究内容的先进性和实用性，最后阶段分析了本课题的研究内容和重点难点部分。

第二章 相关技术分析。首先阐述了 UEFI BIOS 的概念、规范和启动流程；然后介绍了一种符合 UEFI 规范的开源框架 EDKII 的内部结构和模块功能。

第三章 固件生成工具的总体设计。本章介绍的内容是整个固件生成工具系统的第一步，包括系统结构的设计；生成编译链文件、中间文件和 makefile 文件的设计；PCD 机制的设计，为接下来的阶段提供逻辑结构清晰的预处理文件。

第四章 固件生成工具的详细设计。本章详细介绍了固件生成工具三大核心模块的设计。

第五章 固件生成工具的系统实现。本章主要介绍了系统的开发环境和各功能模块的实现过程。

第六章 系统测试。本章主要介绍了对该固件生成工具进行的功能测试的情况。由于固件生成工具可以跨平台运行，同时可以生成针对 IA32 和 X64 平台架构的固件，所以选择 Windows7 64 位和 Ubuntu 9.04 32 位系统为测试平台，以 EDKII 包中的 NT32Pkg 和 UnixPkg 为例构建固件，并通过 Intel Tiano Test Team 组对固件文件进行功能测试，根据测试结果验证构建系统的正确性、稳定性和可移植性。

第七章 总结与展望。对本课题的研究过程和完成内容进行分析，并总结出课题研究成果的优点与不足之处，最后对课题进行展望，提出下一步可以研究的目标和方向。

2 相关技术分析

本章首先对 UEFI BIOS (Unified Extensible Firmware Interface) 的概念和启动流程进行了详细的阐述; 然后介绍了基于 UEFI 规范的开源框架 EDKII 的内部结构和模块功能。

2.1 UEFI BIOS 框架分析

UEFI 的英文全称是 Unified Extensible Firmware Interface, 中文意思是统一可扩展固件接口^[20,21]。EFI 把现代计算机的软件架构概念引入到固件程序设计, 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。

为了统一 EFI 的标准和规范, 由英特尔、微软、惠普等全球著名的计算机软件、硬件厂商于 2005 年发起成立了国际 UEFI 联盟^[22,23]。UEFI 作为一个开放的业界标准接口, 它在平台固件 (firmware) 和操作系统之间定义了一个抽象的编程接口, 并像操作系统的 API 接口一样, 如图 2.1 所示。

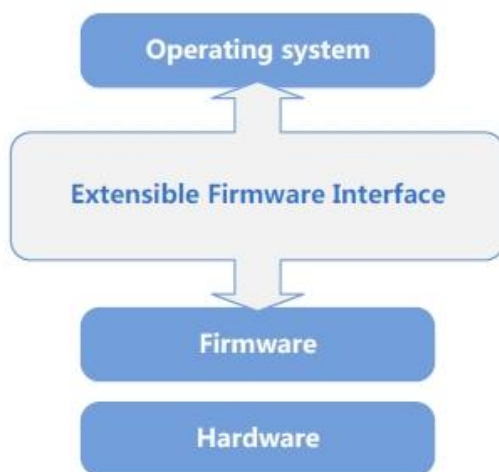


图 2.1 UEFI 功能示意

UEFI 规范并没有限定这个编程接口的具体实现, 标准的 UEFI 接口可以有許多不同架构的实现, 它们对外表现为相同的接口。英特尔对 EFI 的实现定义了 Framework,

UEFI 联盟基于 Framework 定义了 PI (Platform Initialization, 平台初始化) 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^[25,26]。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的 BIOS^[26-28]。

2.1.1 UEFI BIOS的特点

UEFI BIOS (以下简称为 UEFI) 的特点大体有如下几点:

- (1) 定义了操作系统和平台固件之间的可拓展接口;
- (2) UEFI 定义的固件接口并不针对某些特定的处理器架构, 适用范围更广;
- (3) 提供的启动环境更加强大;
- (4) 为独立软件开发商和制造商提供了一个清晰的预启动编程环境;
- (5) 模块化, 依易拓展;
- (6) UEFI 是一个跨平台的规范, 并且符合 UEFI 规范的固件非常易于维护;
- (7) 可以适用定义好的 UEFI API 来抽象标准子系统;
- (8) 对于非标准的总线和驱动, 可以创建符合驱动的 PI 规范;
- (9) UEFI 标准可以提供足够文件的预启动功能, 可以快速启动系统。

UEFI 和传统 BIOS 相比具有明显的优势, 它采用模块化、动态链接和 C 语言风格的参数堆栈传递方式的形式构建系统, 实现方式更灵活; UEFI 的固件接口规范更加标准, 可拓展性更强。同时, UEFI 的驱动程序是由 UEFI 字节代码编写而成, 这样保证了它在不同的 CPU 架构上的兼容性^[29,30]。

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 最大的变革之处在于开机后 UEFI 初始化时, 不仅检测硬件设备, 还可同时加载硬件的驱动程序, 而并不需要通过操作系统来加载, 这就免去了重装操作系统后需要重新安装驱动的复杂工作量。

UEFI 相对传统的 BIOS 而言, 有可视化操作、拓展性强、兼容性强以及 C 语言编写等优势。它能带给用户图形化界面的直观感受, 和一个初级操作系统相似, 不仅能支持鼠标基础操作, 还可以进行游戏、媒体播放等操作。不仅如此, 它还可以完成管

理文件、上网等功能。据英特尔软件与解决方案事业部中国首席研发官梁兆柱介绍，未来的 UEFI 功能还会更强大，并将在新一代 MID 和 HTPC 中发挥作用^[31,32]。

UEFI 由于自身的技术特点，在嵌入式应用、网络客户端电脑等产品中被广泛运用。同时，在电子消费、家用设备等领域，UEFI 技术也在不断延伸。^[33,34]

无论从技术水平还是市场反应看，UEFI BIOS 取代传统 BIOS 的时机已经成熟，它已成为公认的、可靠、稳定的新一代 BIOS 标准。

2.12 UEFI BIOS 的层次结构

做为一个开放的业界标准接口，在系统固件和操作系统之间，UEFI 定义了一个类似于操作系统 API 的抽象编程接口。标准的 UEFI 接口可以有很多种不同架构的具体实现方式，但是这些接口对外的表现形式是相同的^[37]。

英特尔公司设计了全新的 UEFI 实现方案，用来代替传统的 BIOS，同时开源了部分和硬件无关的代码，在公司内部称为 Tiano^[38,39]。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^[40,41]。

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 2.2 所示。

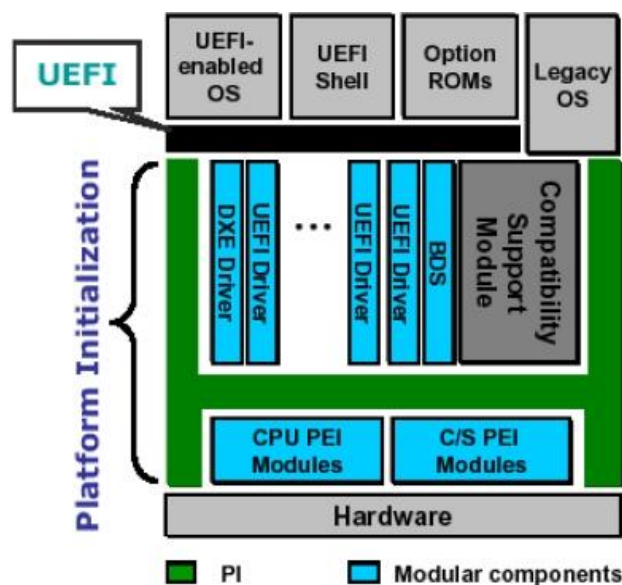


图 2.2 UEFI BIOS 的层次结构

华中科技大学硕士学位论文

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。这里说明一下，PI 是实现 UEFI 的平台基础，而 UEFI 是 PI 这一平台实现的价值集中体现。大体上，UEFI BIOS 的层次结构由以下 4 层组成：

UEFI 初始化准备（Pre-EFI Initialization, PEI）层，把启动的硬件资源信息传递给固件驱动执行层。

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。

UEFI 接口层，提供了符合 UEFI 规范的各种协议和接口，接口是指固件对外的接口。

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。

仔细观察上面的 UEFI BIOS 的层次结构，我们可以看出以下几点：

（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。基于 Green H 架构（实际上是 PEI 基础和 DXE 基础），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 接口层非常“薄”。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，它的意思是统一可扩展固件接口，是新一代的 BIOS，用于启动服务，具有良好的灵活性和兼容性。

（5）Framework 不仅实现了 PI 架构，还实现了 UEFI 架构。换句话说，英特尔设计了 UEFI 完整的实现方案，使得业界超越了传统 BIOS；

(6) UEFI 应用层具有非常有趣的特点。比如，可以在操作系统启动之前支持某些底层应用，甚至可以直接应用 UEFI 来完成控制任务而不需要操作系统，这在传统 BIOS 时代是非常困难的。应用 UEFI，可以在操作系统启动之前就欣赏一部好莱坞大片，或者运行一些嵌入式应用。使用 UEFI 进行嵌入式设计，具有启动快速、代码容量小、廉价等优势。通过 UEFI，甚至还可实现初级的操作系统功能，完成如上网、文件管理等功能。

2.1.3 启动流程

UEFI 启动流程如图 2.3 所示。按照时间先后顺序，该流程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。

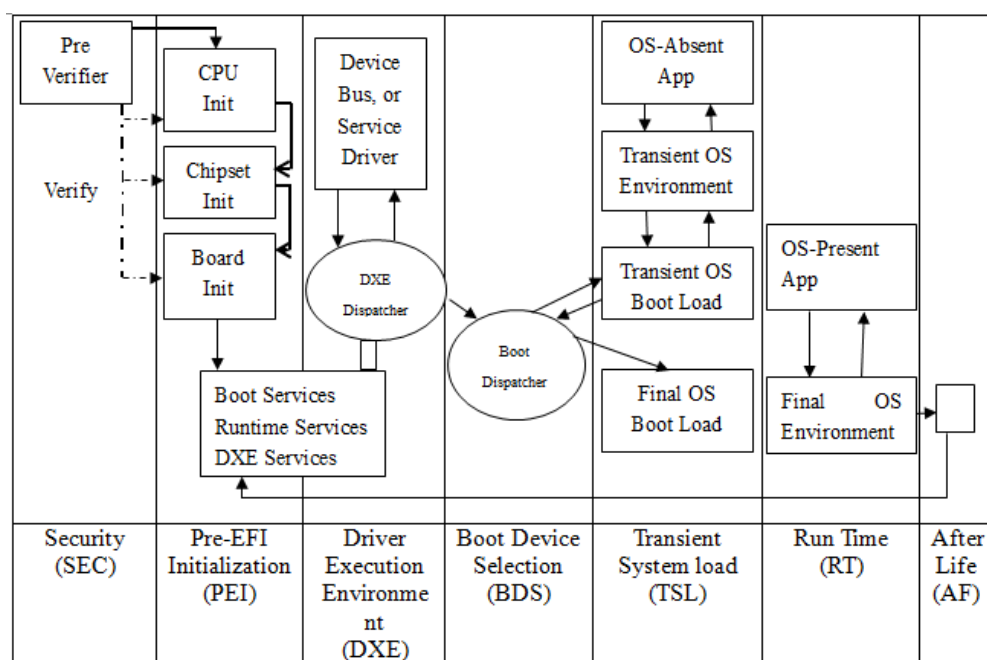


图 2.3 UEFI BIOS 启动流程图

下面将分各个阶段进行介绍。

（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）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 阶段包括被称为 PEIM（PEI 模块）的专用驱动程序和基础代码。这些 PEIM 用以定制 PEI 阶段对平台的具体操作，而基础代码负责按顺序调度并执行这些 PEIM。

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。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。

（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（permanent memory），使得 DXE 阶段可以被加载和执行。在 PEI 阶段末尾的系统状态通过被称为传递块（HOBs）的不依赖于位置的数据结构列表传递给 DXE 阶段。DXE 阶段包含下列组件：DXE 基础、DXE 调度程序和 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。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。

（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。

UEFI 启动管理器还是一个固件策略引擎，这个引擎可以通过修改已定义的全局 NVRAM 变量来进行配置。引导管理器通过全局 NVRAM 变量所拟定好的顺序来尝试加载 EFI 驱动和 EFI 应用程序(包括 EFI OS 引导加载器)。为了能够正常引导，平台固件必须遵循这一全局 NVRAM 变量定义的特定引导顺序。平台固件也可以从引导顺序列表中添加额外的引导选项或移除无效的引导选项。

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。

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。

（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。

（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。

（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。

2.2 EDKII 开发框架介绍

2.2.1 EDKII 框架介绍

EDKII 全称是 EFI Development Kit，它是 UEFI BIOS 发布的一个开源框架，包含了各种相互依赖的模块和基本函数库。它是 EDK 的升级版本，解决了用户在使用 EDK 时的反馈信息，它在 PCD 机制和模块依赖规则进行了较大的优化，开发人员可以利用 EDKII 进行相应的开发、测试^[42-45]。EDKII 主要模块的依赖关系如图

2.4 所示。

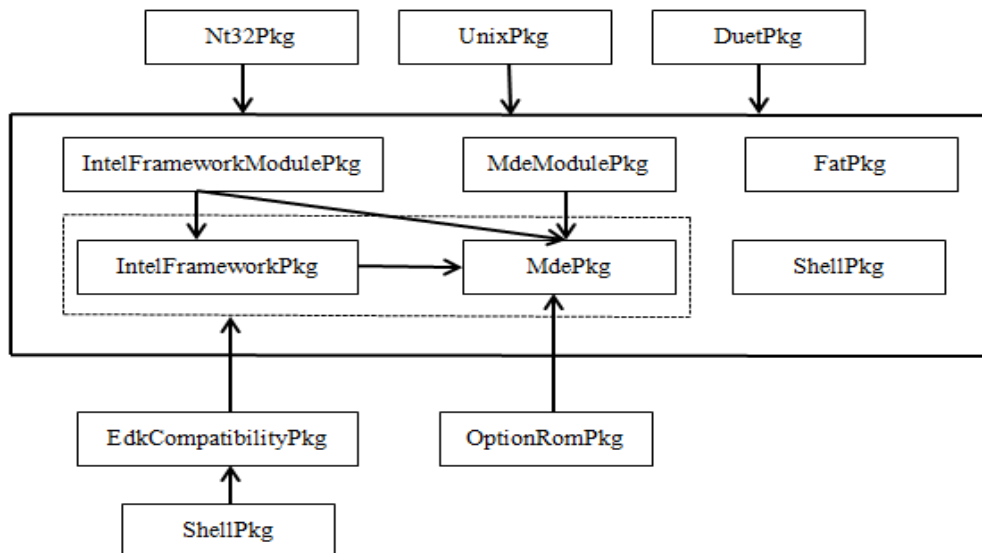


图 2.4 EDKII 主要模块依赖关系图

EDKII 目录下的主要包和模块的描述如下：

CONF: 该目录下存放了在 AutoGen 阶段需要的 3 个关键配置文件：target.txt、tools_def.txt、build_rule.txt。这些配置文件定义了编译环境、编译目标和工具链的配置。

MdePkg: EDKII 源码中包含了 IA32 和 X64 等固件平台依赖的底层库函数、协议和标准，在开发基于不同平台架构的固件模块时可以引用包内特定的库函数和功能模块。

MdeModulePkg: 该包下提供了一组基于 UEFI 规范的跨平台模块，同时提供了基于开发环境的底层库函数和相关的开发示例。

IntelFrameworkPkg: 该包内声明了基于 Intel 新开发框架下的协议、GUIDs 和相关数据结构的声明。

EdkShellPkg: 该包内为开发人员提供了 EFI Shell 应用程序的开发环境。

EdkFatBinPkg: 该包提供了针对不同 CPU 架构的二进制的 FAT 驱动，类似于 fat.efi 形式的可执行文件。

EdkCompatibilityPkg: 该包主要提供了传统 BIOS 在 UEFI 规范下的库函数和协议的兼容性。

Nt32Pkg: 该包提供了在 Windows 操作系统下对 UEFI BIOS 运行时的配置环境。

UnixPkg: 该包提供了在 Linux 操作系统下对 UEFI BIOS 运行时的配置环境。

以上每一个模块包都有相似的目录结构，例如：

Include\ --存放每个包公共文件的头文件；
Ia32\ --存放支持 IA32 架构的内部头文件；
X64\ --存放支持 X64 架构的内部头文件；
Ipfi\ --存放支持 IA64 架构的内部头文件；
Ebc\ --存放支持 EBC 架构的内部头文件；
Uefi\ --存放支持 Uefi2.3 规范的公共头文件；
Pi\ --存放支持 PI1.2 规范的公共头文件；
Protocol\ --存放各种协议的公共头文件；
Ppi\ --存放各种 PPIs 规范的公共头文件；
Guid\ --存放各种 GUID 规范公共头文件；
IndustryStandard\ --存放工业规范标准的公共头文件
Library\ --存放每个包中库函数的头文件

2.2.2 EDKII 的相关文件类型

EDKII 采用模块化、层次化的概念对源代码和各种文件按照依赖关系和规则进行封装。针对不同的平台架构、各个模块需要生成的中间文件类型和最终的构建固件目标，需要在 EDKII 源码中设定相应的配置文件和元数据文件。这些文件声明了有效包、模块之间的依赖关系、中间文件的生成规则和变量定义，例如平台配置数据库 PCD(Platform Configuration Database)和全局唯一标识符 GUID(Globally Unique Identifier)。

平台配置数据库 PCD 库，它存储了模块的各种配置信息，将模块参数化得以最大化的对模块复用。EDKII 在 PCD 机制方面进行了大量的优化。

全局唯一标识符 GUID，它为每个包、模块、中间文件、配置文件等实体设定 128 位的唯一标识符。

EDKII 内的包和模块描述文件称为元数据文件，主要有四种类型：平台描述文

件 (DSC), 包声明文件 (DEC), 模块信息文件 (INF), 闪存声明文件 (FDF), 下面介绍这四种文件的内部结构和部分变量的定义。

(1) 平台描述文件 (DSC):

平台描述文件 DSC, 固件生成工具可以根据平台描述文件的内容从 EDKII 源码中获取对生成固件有效的源码和文件。每个包内会有一个 DSC 文件, 定义了这个包中所有的库(Library)、组件(Component)和模块(Module)的描述信息, DSC 文件中有如下几种字段:

[Header]//DSC 的头文件

[Defines]//平台的配置信息

[SkuIds] //处理 PCD 的不同的方法, 以数值形式表示

[Libraries]//库信息定义

[LibraryClasses]//库实例定义

[Pcds]//PCD 设置

[Components]//声明 EDK 组件和模块

[UserExtensions]//扩展功能; 用户自定义信息

(2) 包声明文件 (DEC)

包声明文件主要是声明当前包中的有效变量和描述信息: 包括 Includes, GUIDs, Protocols, PPIs 和 PCDs, 每个包必须有一个 DEC 文件。包内的所有模块可以获得 DEC 内定义的内容, 但是没有修改权限。DEC 文件中的字段和字段定义与 DSC 文件内相同。

(3) 模块信息文件 (INF)

每个包内的子目录也被定义为一个 EDKII 模块, 如果某个 EDKII 模块中还有子模块, 则只定义一个 INF 文件放置在父模块的根目录下。INF 文件主要定义了源文件与其他模块和文件之间的依赖关系。构建过程中解析 INF 文件的目的是找到当前文件所依赖的其他文件信息。

(4) 闪存声明文件 (FDF)

闪存声明文件定义了可烧入闪存设备的固件的结构和封装规则, 是生成固件逻

辑的指引性文件。文件内主要包括了 FD 和 FV 字段定义了固件生成阶段用到的定义和规则，将解析 DSC 文件生成的二进制文件按照一定规则封装入固件文件。FDF 文件中的字段定义如下：

[Header]//FDF 头文件

[Defines]//FDF 文件的配置信息

[FD]//定义了固件镜像文件的生成规则

[FV]//定义了中间文件封装入固件文件的规则

[Rules]//用户可以自定义固件镜像文件的生成规则

[OptionRom]//创建 PCI Option ROM 映像

[UserExtensions]//用户自定义值

2.3 本章小结

本章对 UEFI BIOS 框架进行了简要介绍，包括它的特点和层次结构均进行了简单的阐述；紧接着对 EDKII 开发框架进行了介绍，以及该框架下包内文件类型进行了解释，主要是为下文介绍系统的设计与实现作理论铺垫，方便读者学习和理解。

3 固件生成工具总体设计

系统的设计必须满足用户的需求。因此，开始设计之前，必须对用户的需求进行详细的分析。

3.1 系统需求分析

本课题要设计的固件生成工具主要有两个需要实现的用户需求：第一，固件生成工具需要在不同的平台上生成固件(本课题以 Windows 系统和 Linux 系统为目标)，同时用户在不同的平台下构建的动作也要相同(以“Build”为命令)；第二，固件生成工具需要能够为不同的平台架构目标而生成相匹配的固件，这就需要构建系统分析源码包中的平台配置文件和模块配置文件，生成目标架构平台的配置文件。为了完成这两个目标，需要在构建系统的每个阶段设计不同定义的文件类型。

3.2 系统结构设计

固件生成工具系统划分为三个主要的构建阶段，下图是固件生成工具系统的构建流程图，描述了在每个模块需要用到的文件类型，和每个模块之间的依赖关系。

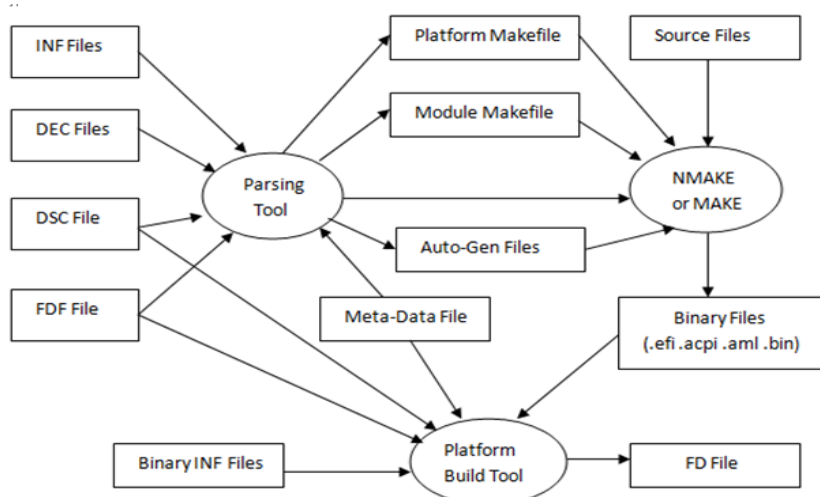


图 3.1 固件生成工具系统流程图

在自动生成阶段：固件生成工具类似一个解析工具，解析所有需要的元数据 meta-data 文件，UCS-2LE 编码文件和 VFR 文件，生成一些 C 源文件（AutoGen.c、

AutoGen.h) 和 makefile 文件。

在编译阶段：根据不同的平台调用 NMAKE 或 MAKE 编译工具处理上个阶段生成的源文件和 makefile 文件，生成符合 UEFI 规范的 PE32/PE32+/COFF 几种镜像文件，其中包括 ACPI 机器语言文件(ACPI machine language file, *.aml)、ACPI 表文件(ACPI table file, *.acpi)、实模式可执行文件(real mode executable file, *.com)和微码二进制文件(microcode binary file, *.bin)。

在固件生成阶段：获得上个阶段生成的 EFI 格式的文件 (.efi .acpi .aml .bin 后缀名文件)，并根据 FDF 文件和源文件定义的相关规则生成固件镜像文件(.fd 文件)。

根据固件生成工具要实现的需求，首先要分析目标配置文件、工具链配置文件和规则配置文件，这样才能保证最终的固件符合最初规范，并且工具能够运行在不同的平台上；其次要分析编译系统的配置文件和四种类型的描述文件：平台描述文件(DSC)，包声明文件(DEC)，闪存设备描述文件(FDF)、模块信息文件(INF)进行分析，通过调用 AutoGenC 和 AutoGenMake 功能模块根据描述文件自动生成 AutoGen.h，AutoGen.c 和相关的 makefile 文件再调用 MAKE 工具生成 EFI 格式的镜像文件；最后，根据固件文件系统的定义和配置，生成最终的固件。

3.3 系统功能模块设计

根据固件生成工具的系统流程，将自动生成阶段、编译阶段和固件生成阶段这三个阶段分别对应自动生成模块、编译模块和固件生成模块这三个模块，并将每个模块下的具体功能细分，构成整个系统功能模块，如图 3.2 所示：

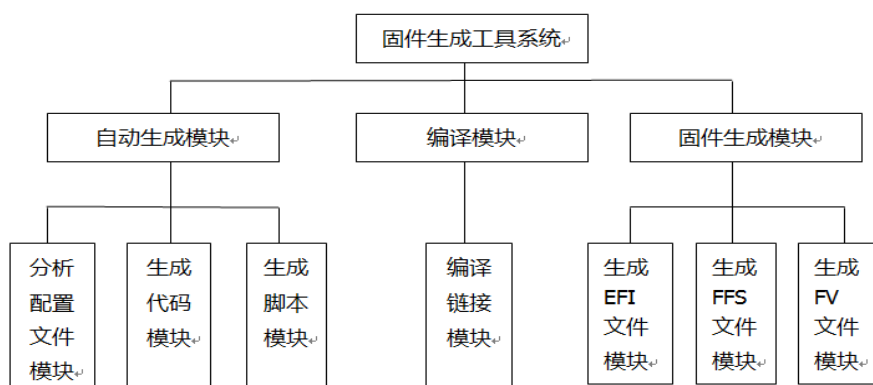


图 3.2 系统功能模块图

3.3.1 自动生成模块

自动生成模块是固件生成工具的最初阶段，需要获得三种配置文件，分别为：`target.txt`（定义了最终生成目标固件的配置），`tools_def.txt`（定义了构建使用工具的配置文件）和 `build_rule.txt`（定义了构建规则）。系统通过分析 `target.txt` 获取构建过程中需要的变量值，包括构建系统类型、目标平台架构、工具链的配置和构建规则；通过分析 `tools_def.txt` 根据用户指定的构建目标选择合适的外部工具和编译工具构造编译链；通过分析 `build_rule.txt` 获得用户指定的编译规则。如果有外部输入与这三个文件中的定义冲突，则外部输入的优先级最高，可以覆盖文件中的定义。解析完这三类配置文件后，构建系统开始解析 `INF`、`DEC`、`DSC` 和 `FDF` 文件中的内容。

`DSC` 文件定义了在整个构建阶段需要用到文件信息；`FDF` 文件中定义了 `flash` 内部结构和一些 `PCD` 的信息，自动生成阶段将这些信息转换成源文件和 `makefile` 文件。

3.3.2 编译模块

编译模块的主要目标是生成二进制 `EFI` 文件。首先通过第三方工具、编译工具和链接工具生成标准的 `PE32/PE32+/COFF` 文件，然后通过 `EDKII` 提供的 `GenFw` 工具将标准的镜像文件修改为 `EFI_IMAGE_SECTION_HEADER` 结构的 `EFI` 文件。`GenFw` 工具根据 `INF` 文件中 `ModuleType` 声明对不同的 `EFI_SECTION` 类型设置对齐偏移值。

这个阶段的主要实现是根据不同的平台选择 `NMAKE` 或 `GMAKE` 固件生成工具来完成。在上一阶段生成的 `makefile` 文件会指定编译器、链接器、汇编器和 `GenFw` 工具来生成下一阶段需要的文件。

3.3.3 固件生成模块

固件生成模块的主要目标生成最终固件的镜像文件。根据 `FDF` 文件和 `DSC` 文件中的信息将编译阶段的 `EFI` 文件按照指定的规则和结构生成最后的固件文件。首先要将 `EFI` 文件封装进 `EFI` 字段内，和其他有效的文件根据指定的规则生成固件文件系统（`FFS`），然后生成固件卷文件（`FV`），最后生成固

件镜像文件（FD）。

该功能模块有以下几个主要步骤：

- （1）解析 FDF 文件和 DSC 文件，确定构建规则和固件镜像文件的结构。
- （2）根据 FDF 的文件中定义的规则，将 EFI 文件和其他有效文件装载如固件文件系统。
- （3）当所有的 FFS 文件生成之后，将它们封装到 FV 文件后，FV 文件根据配置文件生成 FD 文件。

3.4 平台配置数据库设计

UEFI 规范的 PCD（平台配置数据库，Platform Configuration Database）类型有五种。FLAG_PCD 相当于一个宏开关，通过它的值可以设置当前宏的状态为激活还是关闭，只有 TRUE 和 FALSE 两种值；FIX_AT_BUILD_PCD 是编译时确定值的 PCD 类型，值存储在代码段中；PATCH_IN_MODULE_PCD 这种 PCD 的值是存储数据段中，而非代码段；DYNAMIC_PCD 和 DYNAMICEX_PCD 这两种 PCD 主要是为了控制模块间的访问时间而设置的。

PCD 是一个平台配置信息的数据库，它存储了平台组件的各种信息，用户可以直接修改数据库来更新各种信息，而不需要在繁琐的各种配置文件内修改，可以提高平台的可移植性。PCD 的组成内容包括 GUID 值，C 名字，配置信息、偏移量和一些默认值。这些内容定义了平台组件的各种配置。

在固件生成工具运行阶段，需要获取 PCD 的信息。DEC，INF，DSC 文件中都保存有 PCD 信息。不同类型文件中的 PCD 定义了不同类别的配置。由于 PCD 的类型有多种，所以定义 PCD 时候要保证 PCD 的类型和值是互相对应的，即 FLAG_PCD 类型的 PCD 不能给它定义成 TRUE\FALSE 之外的类型。DEC 文件主要是定义 PCD 的 GUID 值，C 名字，令牌，偏移量和默认值。INF 文件主要定义了 PCD 令牌的用法，在模块内可对 PCD 的默认值和偏移量做定义，并且这些值只能在模块内使用。DSC 是平台级的描述文件，这个文件详细的描述了编译某个指定的平台所需要的内容。在 DSC 中定义的 PCD 值和偏移量是整个平台可使用的。在 DSC 中指定的 PCD 的值的优先级是高于 DEC 和 INF 的，如果 DSC 中没

有定义它的值，那就取 INF 中的 PCD 值，如果这两个文件都没有，取 DEC 文件中定义的初始值。

图 3.3 表示了 PATCH_IN_MODULE_PCD 类型在构建系统中的工作流程图：

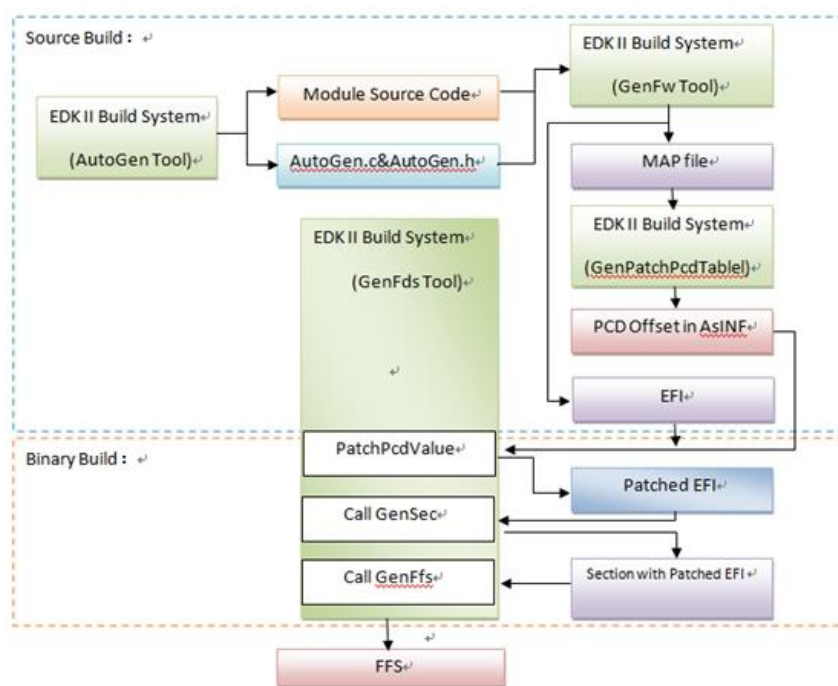


图 3.3 PCD 机制工作流程图

由上图可以看出，在源文件构建阶段，当每个模块构建完成后并且创建了映射关系文件后，固件生成工具会将 GenPcdPatchTable 当作一个 Python 类来调用；GenPcdPatchTable 工具将解析当前目录下的 MAP 文件来获取存储在数据段中 PCD 的值。在二进制文件构建阶段，GenFds 工具会读取数据段中 PCD 的偏移量和原值，同时在 FDF 和 DSC 文件中读取新的值，如果两个值不相同，将会调用 PatchPcdValue 工具将新值写入文件。在 GenFfs 操作之前有 GenFds 工具来调用。

3.5 文件类型设计

3.5.1 中间文件的类型设计

固件生成工具对源码进行编译过程中会有许多中间文件产生，为了区分不同类型的文件，需要专门定义扩展名，方便系统和外部工具的调用。表 3.1-表 3.5 列出了

华中科技大学硕士学位论文

每个阶段所定义的扩展名表。

表 3.1 自动生成文件阶段调用和生成文件的扩展名

Extension	Description
.c, .cpp	C code files
.h	C header files
.asm	32 and 64-bit Windows assembly files
.s	32 and 64-bit GCC assembly files
.S	IPF GCC and Windows assembly files
.i	IPF Assembly include files
.vfr	Visual Forms Representation files
.uni	Unicode (UCS-2) files
.dxs	Dependency Expression files (deprecated)
.asl	C formatted ACPI code files – these files are processed independent from the C code files
.asi	ACPI Header Files
.aslc	C formatted ACPI table files - these files are processed independent from the C code files
.txt	Microcode text files
.bin	Microcode binary files
.bmp	Logo files used in the ImageGen stage
.ui	Unicode User Interface files
.ver	Unicode Version files

表 3.2 编译阶段中间文件扩展名

Extension	Description
.obj	Object files generated by \$(MAKE) stage
.lib	Static Linked files generated by \$(MAKE) stage
.dll	Dynamically Linked files generated by \$(MAKE) stage
.aml	ACPI code files generated by \$(MAKE) stage
.i, .iii	Trim and C Pre-Processor output files
.bin	Microcode files

表 3.3 编译阶段输出文件扩展名

Extension	Description
.efi	Non UEFI Applications, DXE Drivers, DXE Runtime Drivers, DXE SAL Drivers have the Subsystem type field of the DOS/TE header set to EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION , EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER , EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER and EFI_IMAGE_SUBSYSTEM_SAL_RUNTIME_DRIVER respectively. For a Security Module, the Subsystem type is set to EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER . For PEI_CORE , DXE_CORE , PEIM , DXE_SMM_DRIVER , UEFI_APPLICATION , UEFI_DRIVER , the Subsystem type is set to EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER .
.acpi	ASL or IASL compiled ACPI tables
.depex	Compiled dependency sections
.mcb	Microcode Binary files

表 3.4 固件生成阶段中间文件的扩展名

Input Extension	Output Extension	Description
.efi	.pe32	EFI_SECTION_PE32
.pe32, .ui, .ver	.com	EFI_SECTION_COMPRESSION
.ui	.ui	EFI_SECTION_USER_INTERFACE
.depex	.dpx	EFI_SECTION_PEI_DEPEX or EFI_SECTION_DXE_DEPEX
.tmp, .sec	.guided	EFI_SECTION_GUID_DEFINED
.ver	.ver	EFI_SECTION_VERSION
.acpi, .aml, .bin, .bmp	.raw	EFI_SECTION_RAW
.com, .dpx, .guided, .pe32, .ui, .ver	.ffs	FFS file images
.ffs	.fv	Firmware Volume Image files
.fv	.sec	
.txt	.mcb	Microcode Binary File generated from the Microcode text files

表 3.5 固件生成阶段生成文件的扩展名

Input Extensions	Output Extension	Description
.fv, .mcb	.fd	Firmware Device Images
.efi, .pe32	.rom	UEFI PCI OptionROM Images

3.5.2 固件镜像文件的设计

固件镜像文件最终会烧入固件存储设备(Firmware Device)，固件存储设备一般指具有非易失性存储功能的设备。通常使用闪存芯片(Flash)来作为烧入的载体。固件卷文件(FV)是一个二进制格式文件系统，代表了符合闪存设备存储格式的固件镜像文件，可以存入闪存设备之中。

固件卷可以当作是逻辑固件设备，用于存放数据和代码。固件文件系统是为每个单独的固件卷设置的。段是存储数据的最小单位，多个 EFI 段文件组成了固件文件，多个固件文件组成了固件卷。

针对不同的架构平台，闪存内的布局结构也不一样。在 BIOS 启动阶段中，不同的阶段会调用不同段下的文件。例如 SEC 和 PEI 阶段会调用“RECOVERY”段下的代码。为了使闪存结构最优化，可以对某些段进行压缩存储。例如对“MAIN”段进行压缩就可以采用 UEFI 规范的 EFI 压缩定义的标准。闪存的其他区域可用于保留非易失性数据存储，容错的空间和其他用户自定义的产品的数据。NT32 仿真平台可以

模拟 UEFI BIOS 的运行环境，主要用于开发人员在平台上做 UEFI 应用程序开发和调试，降低设备成本。图 3.4 所示，NT32 平台仿真环境中包含的虚拟闪存设备的逻辑布局。

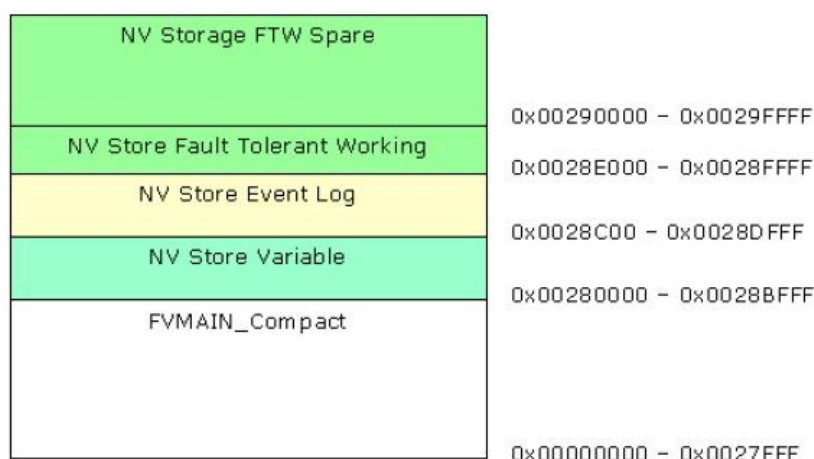


图 3.4 NT32 闪存芯片布局

图 3.5 代表了一种典型的 IA32/X64 闪存设备的布局，其中 SEC 和 PEI 代码位于 FV 的 Recovery 段，其余的驱动或接口被放置在 GUIDed 封装段内并被指定为 FVMAIN_Compact。

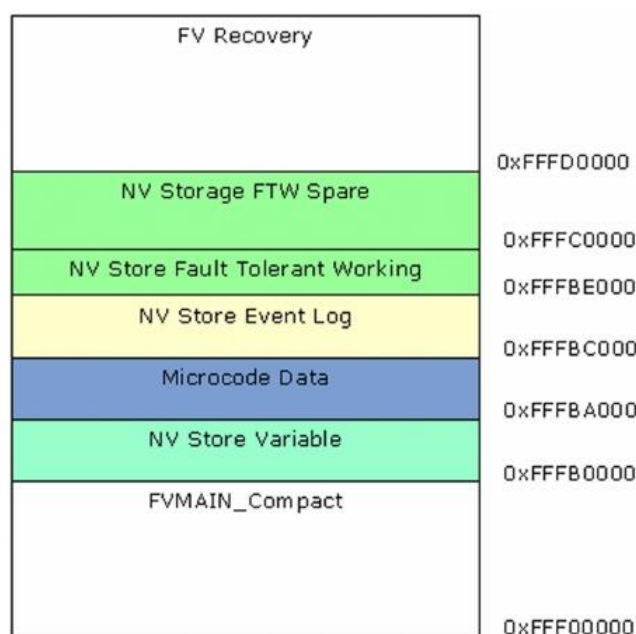


图 3.5 典型的 IA32/X64 闪存芯片布局

3.6 本章小结

本章详细介绍了固件生成工具系统的总体结构设计思路和系统每个功能模块的工作流程和依赖关系的初步设计，然后介绍了平台配置数据库 PCD 的设计，最后对固件卷结构、文件扩展名进行了相关定义。

4 固件生成工具详细设计

4.1 自动生成模块的设计

4.1.1 自动生成模块的功能分析

在自动生成文件阶段，构建系统首先搜索并分析在 CONF 包下的三种配置文件。第一步要分析的是 target.txt 文件。系统会将 target.txt 中的有用信息保存在内存中，如果有外部输入改变了 target.txt 的内容，则外部输入的优先级更高，覆盖了 target.txt 内的配置信息。第二步构建系统需要分析 DSC 文件，通常情况下 DSC 文件路径会在 target.txt 内定义或者由外部输入指定，如果没有找到则在当前目录下搜索。如果固件生成工具在 target.txt 内没有找到 DSC 文件路径，并且也没有外部输入指定参数，那么将会在当前目录下搜索 DSC 文件。如果固件生成工具在当前目录下找到模块描述文件（INF），则会为这个模块编译，而不是创建整个平台的镜像文件。

图 4.1 所示自动生成阶段系统的工作流程：

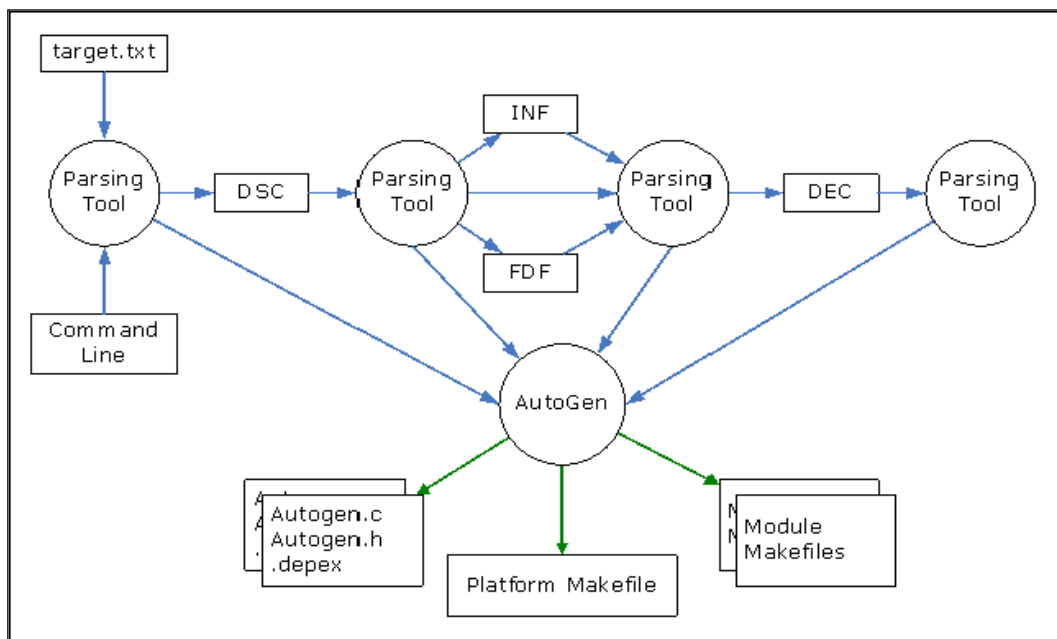


图 4.1 自动生成阶段工作流程图

当构建系统分析出要生成的目标和生成逻辑，接下来就要分析平台描述文件（DSC）。通过平台描述文件可以得到目标平台的模块描述文件（INF）。

通过分析模块描述文件，固件生成工具可以得到模块文件所依赖的包申明文件（DEC）。通过以上工作流程，构建系统可以获得所有的目标镜像文件需要的相关模块和包。

该阶段的最后一步，系统会为每个有效模块生成一些中间文件，包括：AutoGen.h，AutoGen.c，depex 和 makefile 文件。

只要在 DSC 文件中有定义的 INF 文件，构建系统都会为它生成一个模块级的 makefile。但是如果创建的是一个平台级的文件时，就会生成顶级的 makefile 文件。

4.1.2 自动生成模块的详细设计

系统需要首先解析 target.txt 和 tools_def.txt 文件以找到目标的编译工具链，并将需要用到的工具链清单存入 makefile 中并确定要生成的模块。

构建系统会在配置文件 target.txt 中找到 TOOL_CHAIN_CONF 字段的值，这个值内以“name=value”的形式指定了在构建模块和平台上用到的所有外部工具的定义。该内容包括可执行文件的路径，可执行文件需要的动态库的路径和命令行参数。在外部输入和配置文件中的每一个外部工具都有可以被引用的标记名称。

构建系统解析 tools_def.txt 文件时需要扩展工具链中定义的宏和*标记。为了系统能够统一路径格式，扩展后的数据会放入专用的数据库中保存。如果在 DSC 或 INF 文件中使用一个工具的参数，构建系统就可以通过数据库查找对应工具的相关信息。

在 AutoGen 阶段的末尾会生成 makefile 文件，在这个脚本文件中包含<TOOLCODE>的宏和<TOOLCODE>_<FLAGS>的名字。例如，makefile 中有“GCC”和“GCC_FLAGS”宏，其中 GCC 表示 GCC 编译器，而 CB_FLAGS 只是内部定义的一个表示该编译器的变量参数。系统将动态库的路径作为系统环境变量的前缀，这样在 makefile 中需要使用的工具才能够被正确的调用。

在 target.txt 中有一个重要的字段 BUILD_RULE_CONF，它指定了从源文件生成中间文件，从中间文件生成最终的镜像文件 FV/ FD 的构建逻辑和规则。不同的文件扩展名决定了源文件和中间文件的类型。同一个类型的文件可以有多个扩展名，在

不同的阶段可能会根据不同的扩展名来被调用。

系统利用函数 RuleDatabase()解析 build_rule.txt 文件并将内容存入描述编译规则的数据库。数据库中有工具链字段，输入内容字段，输出内容字段和命令参数字段。

图 4.2 是解析数据文件流程图。

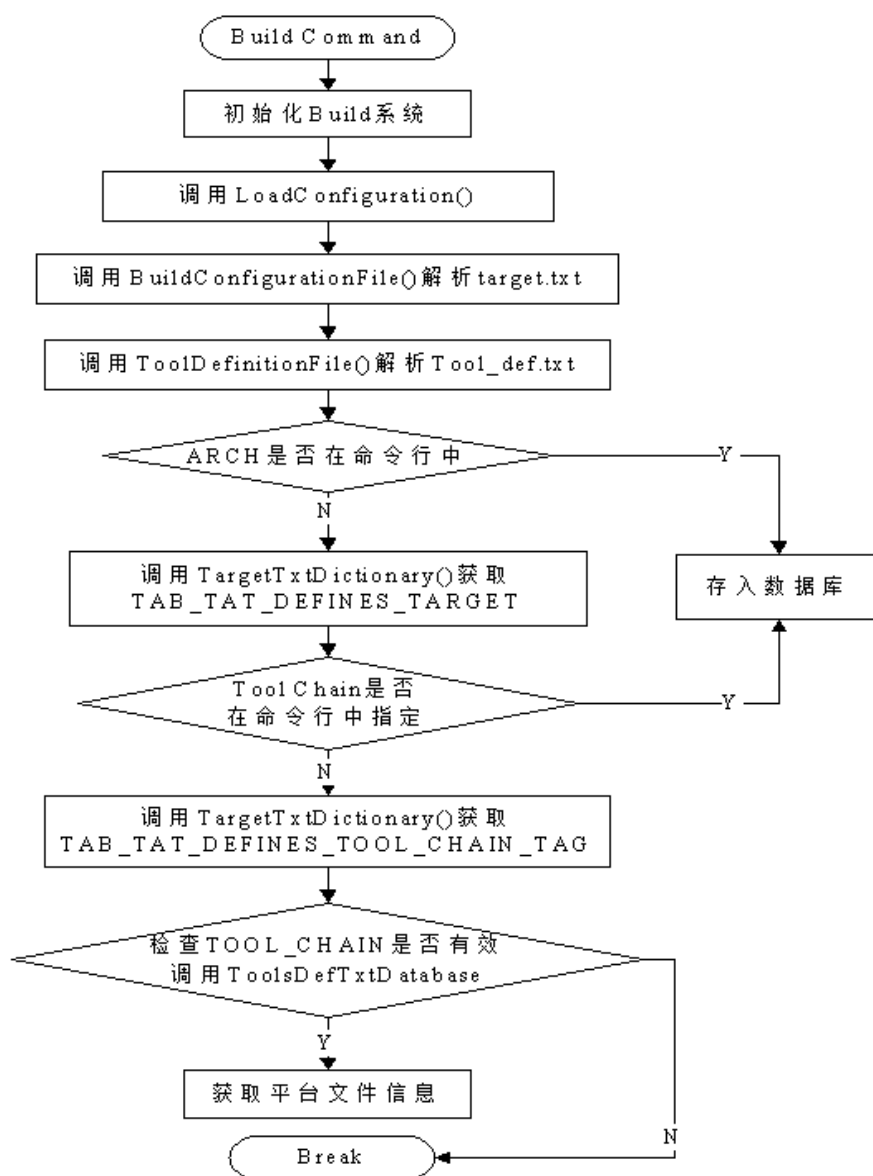


图 4.2 解析数据文件流程图

平台配置文件包括 DSC/INF/DEC/FDF 这四种类型文件，构建系统通过平台配置文件中的 GUID，Protocol，PPI，PCD 的参数来作为 C 文件和 makefile 文件的源数据。

首先，固件生成工具从 DSC 文件中收集库类与库实例之间的映射关系，整个平台的 PCD 配置数据，指定平台的模块描述信息链表和文件输出路径，FDF 文件名，指定平台构建的一些特殊操作。

接下来固件生成工具会解析所有的 INF 文件，包括库实例的 INF 文件。固件生成工具从 INF 文件中获取源文件表、库函数表、相关的包/协议/GUID/PPI 等表。

INF 文件解析结束后，固件生成工具将会检索所有相关的 DEC 文件的列表，然后解析 DEC 文件，从 DEC 文件中，固件生成工具可以得到相关的包/协议/GUID/PPI 的值和包内所有的默认 PCD 配置信息。

最后固件生成工具初步解析 FDF 文件，目的是获得一些模块可能用到的 PCD 信息，然后将这步获得的 PCD 值整合到 DSC 文件中。

解析 DSC 文件，编译系统就会从中收集库类和库实例，整个平台的 PCD 数据，INF 文件模块的列表，以及输出目录这些数据的映射。如果为某个平台编译文件的时候，它的 DSC 文件，有“FLASH_DEFINITION”字段且定义了闪存映像文件的布局描述(FDF)文件，这个 FDF 文件中部分信息也会在 AutoGen 阶段被工具解析。

在所有的 INF 文件被解析完成以后，系统检索所有在 INF 的[Packages]中出现的 package 文件，也就是 DEC 文件列表，然后解析它们。从 DEC 文件中，编译工具将获得的信息，包括文件夹的 GUID, Protocol, PPI 的值，以及 DEC 文件中的所有 PCD 的默认设置等。

最后，如果 DSC 的 FLASH_DEFINITION 定义了 FDF 文件，编译工具尝试解析 FDF 文件。当前阶段仅需要获取 FDF 文件中一些被模块使用到的 PCD 信息，并且将这些 PCD 的信息与合并到 DSC 文件中的组合。

EDKII INF 文件必须在[Defines]段中的 MODULE_TYPE 定义一个有效的名字通常是用来表示这个模块是与 EFI 的哪一个阶段相对应的。如果模块类型是无效的，编译工具就立即停止并给出相应的错误消息。

如果是编译 EDK 的 INF 文件必须在[Defines]段中的 COMPONENT_TYPE 定义一个有效的名字。如果无法识别组件类型，编译工具应该终止编译并给出相应的错误消息。

华中科技大学硕士学位论文

GUID 是文档化的标准接口，为了保证在运行过程中对所调用的值不产生冲突，使用 128 位的 GUID 与在 EDKII 中所有用到的协议(PROTOCOL)，预初始化 EFI 阶段模块与模块之间的接口(PPI)，平台配置数据库(PCD)以及其他一些变量的 C 名称相匹配，它们是一一对应的关系。

协议是指将对平台上的设备的操作封装成为接口。通过对协议的调用完成对设备的操作。每一个设备都有一个与之对应的协议。每个协议都有一个全局唯一的 GUID 值与之对应。

PPI 是在 PEI(Pre-EFI Initialization)阶段用来为 PEIM(Pre-EFI Initialization Module)和 PEIM 之间的接口，每个接口都有一个唯一的 GUID 值与之对应。

编译工具根据输出文件的要求，还需将翻译十进制版本的十六进制版本格式或者从十六进制到十进制的固件生成工具。对于输出的 C 文件，Define 声明语句中使用的版本值是十六进制。

由于固件生成工具可以生成多个平台下的固件文件，为了能够兼容不同的平台和不同的构建目标，需要设计不同的工具链。系统中引入宏，定义每个可能使用到的外部工具，这样如果需要修改某个工具的配置参数，只需要修改宏定义即可。外部工具主要包括不同平台下的编译器和汇编工具。

为了编译适用 IA32 和 X64 平台架构的固件需要用到以下编译器：

- (1) EDKII 默认了 Visual Studio2005 去编译 IA32 和 X64 平台架构的固件；
- (2) Intel C++ Compiler (ICC) 9.1；
- (3) Microsoft Windows Driver Development Kit (DDK) 2790.1830，为编译 16 位的汇编代码；
- (4) GCC Tool Chain；
- (5) 在 Linux 平台上运行需要软件工具包括 GNU C Compiler(GCC) 4.2.1，GNU Binutils 2.17.50，GNU Glib C 2.3.6，X11 7.2，SQLite 3.0，Python 2.5.2；
- (6) ACPI 编译器可选择 ACPI Components Architecture 20061109 或更高级的版本。

target.txt 文件是告诉构建系统源文件中有效的源码和配置信息，同时指定了

tools_def.txt 文件的地址, target.txt 文件中的 ACTIVE_PLATFORM 字段指定了默认的 DSC 文件; TARGET_ARCH 指定了在那种平台下进行编译; TOOL_CHAIN_CONF 指定 tool_def.txt 文件的路径; TOOL_CHAIN_TAG 指定了进行编译的工具链; MAX_CONCURRENT_THREAD_NUMBER 指定了可以容纳最多的线程个数; BUILD_RULE_CONF 与 build_rule.txt 文件指定了创建 makefile 文件的规则和目标。在 target.txt 文件中的 TARGET_ARCH, TOOL_CHAIN_TAG 等字段, 如果外部输入也同样指定了值, 则外部输入的优先级更高, 覆盖文件中原来的值。

build_rule.txt 文件指定了编译和链接有效模块的规则, 与 tools_def.txt 文件和 INF 及 DSC 文件的[BuildOptions]部分共同决定了生成的 makefile 文件的规范。要生成的文件有多种类型, 包括 C 代码, C 头文件, 汇编代码, VFR 文件, Obj 文件, 静态库, 动态库等所有生成固件的相关文件都分别指编译的规则。比如, 如果是编译 C 代码, 则输入的文件是*.C 或*.CC 和*.CPP, 输出文件则是*.obj 文件并通过路径宏 \$(OUTPUT_DIR)放到指定文件夹下。

4.2 编译模块的设计

4.2.1 编译模块的功能分析

这一阶段的功能目标是根据不同的平台调用 NMAKE 或 MAKE 编译工具和链接工具处理上个阶段生成的源文件和 makefile 文件, 生成符合 UEFI 规范的 PE32/PE32+/COFF 几种镜像文件, 然后通过 EDKII 提供的 GenFw 工具将标准的镜像文件修改为 EFI_IMAGE_SECTION_HEADER 结构的 EFI 文件。EFI 文件将在下一阶段封装在 EFI_SECTION_PE 类型的文件段内, 与其他 EFI 段文件组成 FFS 文件(固件文件系统)。

如图 4.1 所示, 从平台级别来分析本阶段的执行过程可以分为编译库模块(Library Module)阶段, 生成函数库, 可以被其他驱动调用; 编译非库模块(Non-Library Module)阶段, 生成 PEI driver、DXE driver 或普通可执行文件; 生成闪存镜像文件(Flash Image)阶段。

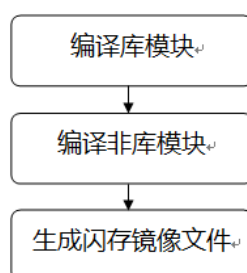


图 4.1 平台视角编译流程图

如图 4.2 所示，从模块级别来分析，此阶段可以分为预处理、编译，静态\动态链接，最终生成模块的镜像文件。

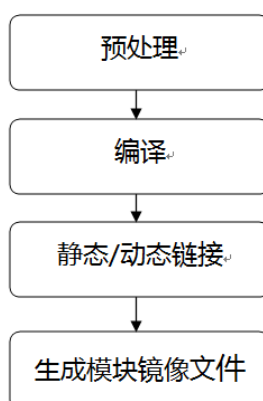


图 4.2 模块视角编译流程图

4.2.2 编译模块的详细设计

在 C 源代码被编译器编译之前，预处理器就需要对源程序中的宏定义进行处理。预处理是 C 编译器内部过程，在后台进行，并不需要显示的调用而是自动的进行。一般的 C 编译器都将预处理，汇编，编译和连接过程集成到一起。但是如果一些非 C 文件的编译器并不自动的进行预处理，那就需要一个独立的预处理阶段。为了使用 C 头文件中定义的宏，在汇编文件中使用“#include”指令，在被编译器调用之前，需要有一个独立的预处理过程完成宏替换。例如 VFR 文件，ASL 文件和 DXS 文件为了能用宏，它们要有预处理过程。

同时，需要一个额外的修整阶段来对预处理之后的汇编文件，VFR 文件和 DXS 文件进行格式化操作，删除多余的内容。

VFR (Visual Forms Representation)是目前应用在 UEFI BIOS 下 SetUp 环境变量设

置的一种全新的界面开发语言。通过 VFR 编译器对该语言和一些 Unicode Code 文件的前期处理会产生 IFR 文件，比如：IFR 图表文件，字符串文件，字体图标文件。这三种符合 C 语言规范的文件可以嵌入 SetUp 环境的 EFI Driver 的源程序中，通过 C 语言编译器进行编译就可以得到能够在 SetUp 环境中产生的特定图形页面的 EFI Driver。

在编译阶段，系统调用编译工具将源代码编译成目标代码，系统只负责调用相应文件类型的编译工具，并不负责编译过程的执行。编译器把一个源程序翻译成目标程序，它的工作过程由五个阶段组成：词法分析、语法分析、语义的检查及中间代码的生成、代码优化、目标代码生成。例如，系统调用 GCC（Linux 平台）或者 C 编译器来编译 C 文件和汇编文件，再由 as 汇编器将中间代码转换为目标文件；系统调用 VfrCompiler 工具将 VFR 文件转换成 C 和头文件，再使用适合的编译器编译；调用 ASL 编译器将 ASL 文件转换为 ACPI 机器语言文件。

静态链接用于所有模块的 C 文件，如果模块中包含 C 文件，那么该模块将要经过静态链接步骤，如果模块内不包含 C 文件，则跳过静态链接。如果是库模块，则将目标文件链接到静态库文件中，如果模块中不含有 C 文件，则不需要此步骤。

对于非库模块，静态链接并不是必须的。但是为了更好的优化，微软的工具链 (MSFT) 仍然会使用静态链接。

在静态链接中产生的静态库文件与其他有依赖关系的库模块的静态库文件将被链接 (DLINK) 到 *.dll 文件中。从 aslc 文件生成的目标文件将被直接链接到 *.dll 文件中。从实模式的汇编文件生成目标文件通过实模式连接器 (ASMLINK) 链接 *.com 文件。(Intel corporation. 2006-7-13)。

如果非库模块中包含有 C 文件、ASLC 文件或 ASM16 文件，则需要动态链接阶段，这些类型的文件会在 INF 文件中声明。

4.3 固件生成模块的设计

固件生成阶段是固件生成工具系统的最后一个阶段，在该阶段：系统首先解析

FDF 文件和 DSC 文件部分内容，得到构建规则和固件文件结构，然后将编译阶段生成的 EFI 文件转换成为 EFI 段文件，根据配置信息将 EFI 段文件封装成为固件文件系统文件 FFS，再转换成固件卷文件 FV 和固件镜像文件 FD。

4.3.1 固件生成模块的功能分析

一旦所有模块的 EFI 镜像文件生成之后，就进入了构建系统的最后一个阶段，这个阶段将生成最后的目标固件文件。这一阶段将用到 FDF 文件和部分 DSC 文件中的内容来生成最终的固件镜像文件。这一阶段将处理各种 EFI 规则的文件，使他们转换成 EFI_SECTION 类型的文件并根据规则生成不同的固件文件系统（FFS）、固件卷文件（FV）和最终的固件镜像文件（FD），在 FDF 文件和少部分的 DSC 文件定义了生成的是何种类型的文件。

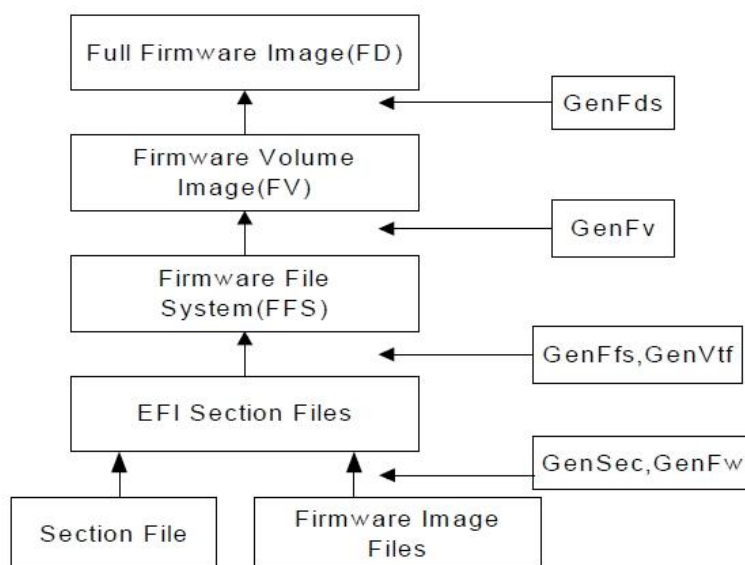


图 4.3 固件生成阶段模块图

图中的右边是调用的工具模块来生成各阶段的文件，包括有 GenSec、GenFw、GenFfs、GenVtf、GenFv 和 GenFds 工具。

GenSec: 此工具是用于将 PE32/PE32+/COFF 格式文件或二进制文件转换成符合 UEFI 规范的 EFI_SECTION 类型的文件。

GenFw 工具将由工具链生成的 PE32/PE32+/COFF 格式的映像文件转换成基于 INF 文件中定义的 UEFI 固件镜像文件。

GenFfs 工具是用于为固件卷生成固件文件系统 FFS，如何生成 FFS 文件的规则在闪存描述文件 FDF 中指定。

GenFv 工具根据模块内的 INF 文件按照规则把 FFS 文件生成 FV 镜像文件。

GenFds 工具根据模块内的 INF 文件按照规则把 FV 文件生成 FD 镜像文件

4.32 固件生成模块的详细设计

最终生成的固件文件需要存放在物理载体中，这个载体通常是一个闪存（Flash）芯片，这个闪存芯片可以称为固件设备，而固件设备的结构层根据不同的目标平台架构而设计。

固件文件 FD 的内部组织结构如图 4.4 所示。

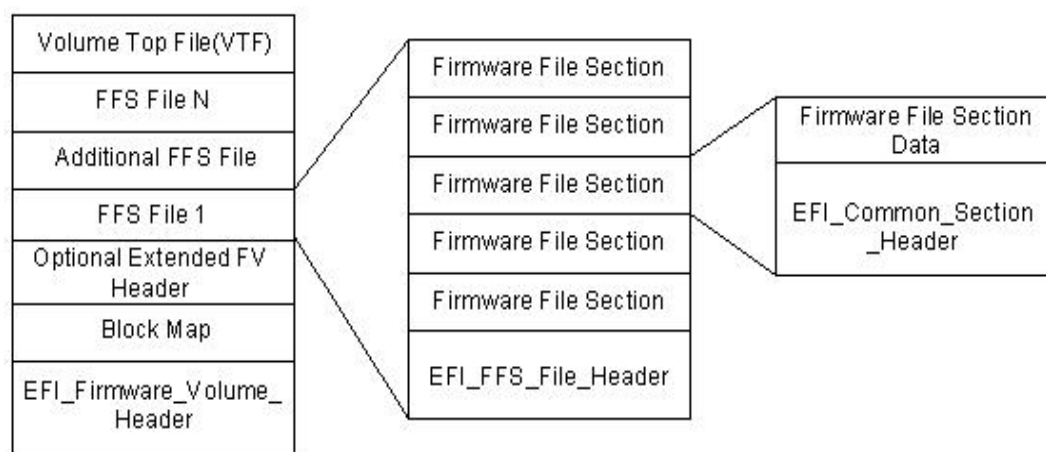


图 4.4 固件卷组织结构图

一个最终的固件镜像文件(FD)主要包括固件卷文件(FV)部分，代码数据段，日志段等内容。固件卷文件中封装了目标平台上的所有需要的功能模块。多个固件卷组成一个最终固件文件。固件卷的结构可以看作是由首部和数据部组成，同时固件卷的内容又是由一个或者多个的固件文件系统封装组成。固件文件系统的结构是由首部，数据区和尾部组成，同时内容上是由一个或多个 EFI 段封装而成，段里面又可包含多个段。而需要的功能模块的文件数据就存放在每个段的数据区中。

表 4.1 固件文件类型

固件文件类型	数值	描述
EFI_FV_FILETYPE_RAW	0x01	原始的二进制数据文件
EFI_FV_FILETYPE_FREEFORM	0x02	原始的段数据文件
EFI_FV_FILETYPE_SECURITY_CODE	0x03	SEC 阶段的平台文件
EFI_FV_FILETYPE_PEI_CORE	0x04	PEI 阶段的核心固件文件
EFI_FV_FILETYPE_DXE_CORE	0x05	DXE 阶段的核心固件文件
EFI_FV_FILETYPE_PEIM	0x06	PEI 阶段的驱动文件
EFI_FV_FILETYPE_DRIVER	0x07	DXE 阶段的驱动文件
EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER	0x08	PEI 阶段和 DXE 阶段都可以调用的驱动文件
EFI_FV_FILETYPE_APPLICATION	0x09	应用程序文件
EFI_FV_FILETYPE_SMM	0x0A	被加载到 SMRAM(系统管理内存)中的驱动文件
EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE	0x0B	固件卷文件
EFI_FV_FILETYPE_COMBINED_SMM_DXE	0x0C	DXE 阶段和 SMM 阶段都可以调用的驱动文件
EFI_FV_FILETYPE_SMM_CORE	0x0D	SMM 阶段的核心固件文件

表 4.2 固件文件段类型

段类型	数值	描述
EFI_SECTION_COMPRESSION	0x01	段中数据经过压缩
EFI_SECTION_GUID_DEFINED	0x02	段中数据为GUID
EFI_SECTION DISPOSABLE	0x03	段中数据为特殊用途
EFI_SECTION_PE32	0x10	存放PE32+可执行代码的段
EFI_SECTION_PIC	0x11	存放位置独立代码的段
EFI_SECTION_TE	0x12	存放TE可执行代码的段
EFI_SECTION_DXE_DEPEX	0x13	存放DXE依赖关系的段
EFI_SECTION_VERSION	0x14	存放版本号
EFI_SECTION_USER_INTERFACE	0x15	存放用户界面的段
EFI_SECTION_COMPATIBILITY16	0x16	存放16位兼容代码的段
EFI_SECTION_FIRMWARE_VOLUME_IMAGE	0x17	存放固件卷的段
EFI_SECTION_FREEFORM_SUBTYPE_GUID	0x18	存放GUID原始数据的段
EFI_SECTION_RAW	0x19	存放原始数据的段
EFI_SECTION_PEI_DEPEX	0x1b	存放PEI依赖关系的段
EFI_SECTION_SMM_DEPEX	0x1c	存放SMM依赖关系的段

表 4.1 描述 PI 规范中定义的固件卷的文件类型。固件卷服务接口会识别特定的固件卷文件类型，比如，EFI_FV_FILETYPE_DRIVER 表示的是 DXE 阶段的驱动文件，当运行在 DXE 阶段时，DXE 阶段的核心固件文件会识别并加载执行。

表 4.2 所示描述了固件文件段类型，例如其中 `EFL_SECTION_PE32` 类型的文件是 EFI 文件。在 UEFI BIOS 的启动过程中，被封装的固件卷文件都会被 EFI Core 解析得到通过分析后固件中的 `EFL_SECTION_PE32` 段内存储的 EFI 文件就会被加载执行。

4.4 本章小结

本章主要介绍了固件生成工具各模块的功能分析和详细设计。其中在自动生成模块重点介绍了如何设计固件生成系统的配置文件，怎样去分析源文件，包括配置文件和 `EDKII` 中的平台信息文件，如何自动生成中间文件；在编译模块详细描述了编译的过程，怎样处理那些源文件使之逐渐变成符合 UEFI 规范的映像文件，编译系统严格按照文件依赖关系的先后顺序来执行生成文件；在固件生成模块主要介绍了固件生成模块的工作流程，然后分析了该阶段下生成的固件镜像文件的结构。

5 固件生成工具的实现

5.1 系统开发环境

固件生成工具的开发是基于 Windows 7 64 位操作系统系统。用 C 语言和 Python 语言作为主要的开发语言。利用 Eclipse 和 PyDev 插件作为开发工具。

操作系统：Microsoft Windows 7 64 位

相关软件：Microsoft Visual Studio 2008

Microsoft Windows Driver Development Kit(DDK) 2790.1830

Tortoise SVN

开发软件：Eclipse+PyDev

开发语言：Python2.5

5.2 系统功能实现

5.2.1 自动生成模块的实现

1) 生成依赖表达式文件

固件生成系统将按顺序生成中间文件，首先，考虑是否需要生成依赖表达式文件(*.depex)，这取决于模块的 INF 文件中是否含有[depex]段和段中的内容，其内容就是依赖关系表达式。在前面了解到，UEFI BIOS 运行分为几个阶段，其中 PEI 阶段和 DXE 阶段都分别有一个 PEI 分配器(Dispatcher)和 DXE 分配器，这两种分配器的工作任务相似，主要就是发现和根据依赖关系来安排执行 PEI 模块以及 DXE 驱动的顺序。依赖关系的原型就是在 INF 文件的[depex]段中所定义的内容。但不是所有的 INF 文件都有这部分的内容，这取决于这个模块的类型，比如，PEIM，DXE_DRIVER 等这些类型，它们就需要用到这种依赖关系。[depex]段中的内容可以从库继承过来，使用的是 GUID 的 C 名字，它支持不同的依赖关系的逻辑表达式。

实现原理是：首先，调用 GuidStructureStringToGuidString()函数在依赖关系表达式中的 GUID C 名字转换为 C 结构格式的值，然后再将整个表达式字符串通过

GetPostfixNotation()转换成逆波兰式。这样操作的目的是在保存其内容到文件之后，先将逆波兰式中的所有操作符和 GUID 值都转换成二进制值，用 ValidateOpcode()函数验证表达式的有效性，并将其存储在栈 stack()，在 python 里用列表来实现栈。最后就是调用 Generate()保存文件。

这个二进制依赖关系表达式文件被放到\$(MODULE_BUILD_DIR) / OUTPOUT 文件下，这个目录也是属于全局宏，它是指明该模块的输出目录，生成以扩展名是“.depex” 的文件。

如果有以下的逻辑表达式：

NOT (gEfiHiiDatabaseProtocolGuid AND gEfiHiiStringProtocolGuid) OR
gPcdProtocolGuid

通过转换之后在存储到文件是*.depex：

```
02          //表示入栈
72 c1 9f ef b2 a1 93 46 b3 27 6d 32 fc 41 60 42 // gEfiHiiDatabaseProtocolGuid
02          // 入栈
74 69 d9 0f aa 23 dc 4c b9 cb 98 d1 77 50 32 2a    // gEfiHiiStringProtocolGuid
03          // 与
05          // 非
02          // 入栈
06 40 b3 11 5b d8 0a 4d a2 90 d5 a5 71 31 0e f7    // gPcdProtocolGuid
04          // 或
08          // 结束
```

2) 使用 HII 字符串包

HII (Human Interface Infrastructure)是一种 UEFI BIOS 规范的交互机制，类似一种数据库。它定义了一些基本的管理用户的输入机制，提供 HII 相关协议、类型和功能定义，支持字符串和字符的管理、内部图形控件的索引以及对鼠标键盘输入的抽象翻译工作。HII 在显示接口的调用上支持如下功能：本地字符显示、本地图形显示、远程字符显示、远程图形显示、网络浏览器支持和为操作系统提供相关的图形

界面。HII 库内部通过 GUID 机制进行索引,通过查询 GUID 就可以把与其关联的特殊显示控件和特殊字符取出来进行显示。

供用户交互的 HII 包的内容是以 Unicode 格式保存在*.UNI 文件中。编译工具按一下流程将字符串信息转换成为 HII 字符串包类型的数据结构:

系统调用 LoadUniFile()函数载入 UNI 文件,通过调用 GetStringObject()函数从 UNI 文件得到字符串序列号,与序列号对应的字符串和使用的语言格式。语言格式包括 RFC_LANGUAGES、RFC4646、ISO_LANGUAGES、ISO639-2 等多种类型。

对于 EDK II 模块内的文件必须使用 RFC4646 语言代码,否则固件生成工具将停止并抛出警告信息。通过函数 GetLanguageCode(LangName, IsCompatibleMode, File)对其他标准的语言进行转换,并引用一个全局的变量来保存所有语言代码类型。

查找所有的源文件、包和模块,找出使用的字符串序列号。

在最后生成的 AutoGen.h 文件,字符串序列号将被宏来代替。

3) 生成 AutoGen.h 文件

在此阶段生成的 AutoGen.h 文件包括以下几个方面:

(1) 文件头

构建系统会为每一个有效的模块生成一个 AutoGen.h 的文件,而这个的文件头就是该模块内 INF 文件的 GUID 值,例如:

```
include_statement(AutoGen.h, “  
#ifndef _AUTOGENH_6987936E_ED34_44db_AE97_1FA5E4ED2116  
#define _AUTOGENH_6987936E_ED34_44db_AE97_1FA5E4ED2116”);
```

(2) 加载的头文件

通常情况只加载一个头文件,例如:

```
#include_statement(AutoGen.h, “#include <Base.h>”);
```

(3) PCD 的定义

对应于不同类型的模块(库模块或者非库模块),会生成两种不同的 PCD 代码非库模块部分要比库模块多加上一个对 PCD 类型的判断的方法

(4) HII 字符串包的定义

当.uni 文件被找到的时候才会去生成这些包。查找模块中所有用到的字符串变量, 并定义成宏。

4) 生成 AutoGen.c 文件

只有该模块的类型是非库模块时, 构建系统才会生成 AutoGen.c 文件, 和.h 文件一样包括文件头和 PCD 和 HII, 但会额外多出以下几个部分:

(1) 库的构造函数声明

在当前模块中 INF 文件的[Defines]段下有一个定义的 CONSTRUCTOR 字段。在 [LibraryClasses]段下定义了所需要的库类的名称, 而对应库实例的路径是定义在指定平台的 DSC 文件中, 每个库实例与库类是一一对应的。在 INF 的[Defines]段下面定义的 CONSTRUCTOR 主要是用来链接库实例的。构造函数声明部分主要分为定义构造函数, 处理构造函数, 和判断构造函数的状态。

```
If (CONSTRUCTOR defined in INF) {  
  //模块类型是 BASE,< CONSTRUCTOR>表示构造函数的名字, 对这个值定义  
  If (MODULE_TYPE == "BASE") {  
    include_statement(AutoGen.c, "  
    EFI_STATUS  
    EFIAPI  
    <CONSTRUCTOR> (VOID);  
    "); }  
  //判断这个函数的状态  
  If (CONSTRUCTOR defined in INF) {  
    If (MODULE_TYPE == "BASE") {  
      include_statement(AutoGen.c, "  
      EFI_STATUS Status;  
      Status = <CONSTRUCTOR>();  
      ASSERT_EFI_ERROR (Status);  
      ");}
```

(2) 库的析构函数声明

在库实例模块的 INF 文件[Defines]下定义 DESTRUCOTR 字段的值, 主要目的

是用来销毁库实例，同样分为三部分：定义析构函数，处理析构函数和判断函数状态。

(3) 模块入口点声明

如果 INF 的[Defines]下有 ENTRY_POINT 字段，则需要定义入口点声明：

```
//判断 ENTRY_POINT 是否存在
```

```
If (ENTRY_POINT defined in INF) {
```

```
If (MODULE_TYPE == 'PEI_CORE') {
```

(4) 全局变量定义

获得 INF 文件定义的 Guids, Protocols, Ppis, PCD 的值，例如定义 GUID 的定义：

```
//定义 INF 文件的列表
```

```
InfList = [];
```

```
add (ModuleInf, InfList);
```

```
foreach LibraryInstance {
```

```
add (LibraryInf, InfList);
```

```
foreach DependentLibraryInstance {
```

```
add (LibraryInf, InfList);
```

```
}}
```

```
//遍历 INF 列表，从上往下的执行，取出 GUIDs 的值
```

```
foreach INF in InfList {
```

```
If (“[Guids]” defined in INF) {
```

```
foreach GuidCName {
```

```
include_statement(AutoGen.c, “
```

```
GLOBAL_REMOVE_IF_UNREFERENCED EFI_GUID \
```

```
<GuidCName> = <GuidValue>;“);
```

```
}}
```

在每个模块的 AutoGen.c 文件生成之后。需要为特定的平台生成各自的 makeFile 文件。

5) 自动生成 MakeFiles

Makefile 文件由两种级别的文件组成：平台级 makefile 和模块级 makefile。指定的平台对应相应的 makefile，定义了对应的工具链和编译目标，其中编译目标分为调试类型和发布类型。平台级 makefile 可以引用模块级 makefile。固件生成工具调用编译工具（nmake 或 make），在平台级构建模式中，固件生成工具将编译工具作为平台级 makefile 的参数，平台级 makefile 调用编译工具对每个模块及 makefile 进行编译。

下一阶段用到的源文件的之间的调用关系、编译工具和链接工具的定义信息保存在 makefile 文件中。

（1）生成模块级的 makefile

模块级 makefile 是由两部分组成：宏定义和目标定义。DSC 文件中 OUTPUT_DIRECTORY 的确定了 makefile 的生成路径。模块级 makefile 的关键设计包括宏定义和编译目标两种。

① 宏定义类型主要有以下几种：

平台信息的宏定义： DSC 文件的[Defines]段定义了平台信息的宏。

模块信息的宏定义： INF 的[Defines]和 DSC 的[Components]，定义了模块信息的宏。

编译工具配置的宏定义： target.txt、外部输入和 DSC 文件的[Defines]根据优先级定义了最终的编译工具配置的宏。

工具路径的宏：编译当前模块需要的工具由 target.txt 中的 TOOL_CHAIN_CONF 的值来确定。

源文件和目标文件列表宏： 根据规则定义文件 build_rule.txt 文件中最后一条规则和 INF 文件[Sources]段的定义确定这“CODE_FILES”和“CODA_TARGET”宏。

```
INIT_TARGET = init
```

```
CODE_FILES = $(WORKSPACE)\MdeModulePkg\App\Hello\HelloWorld.c
```

```
CODA_TARGET = $(DEBUG_DIR)\$(MODULE_NAME).efi
```

② 目标定义主要有以下几种：

“All”目标，实际上就是指默认目标是“mbuild”。

“pbuild”目标，仅用来编译当前模块的源文件，总是被用在顶层的 makefile。

华中科技大学硕士学位论文

“mbuild”，在单个模块的编译模式下使用的。因为在单个模块模式下编译，顶层 makefile 文件不会被调用，如果当前模块在模块的 makefile 需要这时编译系统必须要重新对库进行编译。

“fds” 目标目标，用于告诉系统，当一个单一的模块已经被编译好以后，重新生成一个固件文件。这就需要调用的顶层 makefile 来做到这一点。

“init” 目标，初始化信息，为编译系统创建一个目录做为输出文件的根目录。

不同种类的目标的合集，这是用来表示要从源文件开始编译到目标文件，然后转化为最终的 lib 文件，EFI 文件或其他文件。

最后就是删除释放的操作：clean, cleanall, cleanlib。

构建系统按如上所示顺序将每个宏逐步写入 makefile，最终生成模块级的 makefile 文件。

(2) 生成平台级的 makefile

平台级 makefile 文件也是由宏和编译目标定义组成。

(3) 宏定义

包括了平台宏信息，编译配置信息和编译目录宏，与模块级 makefile 文件生成相似。

(4) 编译目标定义

“all” 目标，如果在 makefile 中引用了这个目标，决定平台的构建顺序，先要初始化，对库进行编译，再对模块，最后是生成固件。

```
all: init build_libraries build_modules build_fds
```

“init” 目标，初始化工作，如创建目录，实现如下：

```
init:
```

```
-@echo Building ... $(PLATFORM_NAME) $(PLATFORM_VERSION) [IA32]
```

```
-@if not exist $(BUILD_DIR)\IA32 mkdir $(BUILD_DIR)\IA32
```

```
-@if not exist $(FV_DIR) mkdir $(FV_DIR)
```

“library” 目标，主要告诉编译工具，如果读到这个关键字就要编译所有的库。

“module” 目标，这是用来构建所有的模块和它们所依赖的库，实现机制类似于 “library” 目标。

“fds”目标，用来生成平台的固件。

删除释放的操作：clean, cleanall, cleanlib。

“run”目标，主要是用来执行仿真平台，比如 NT32 或 Unix 等。

5.2.2 编译模块的实现

每一个经过编译阶段的模块都会生成一个符合 UEFI 规范的镜像文件，包括 EFI 可执行镜像文件(.efi)、ACPI 机器语言文件(.aml)、实模式可执行文件(.com)和二进制文件(.bin)。其中 EFI 可执行文件是由含有 C 文件的非库模块产生的，它是经过 GenFw 工具在动态连接阶段生成的.dll 文件转换的。另外在工作目录的输出目录中还可能会有下面的一些文件：

- *.aml 文件在编译，汇编过程从 ASL 文件生成的。

- *.acpi 文件是由 Genfds 将*.dll 文件转换的。

- *.com 文件是由实模式连接器（ASMLINK）在动态链接过程中产生的。

- *.bin 文件是由 GenFw 工具将*.txt 文件转换而来的。

GenFw 工具是用来生成基于组件或由第三方工具链生成的 PE/PE32+/COFF 图像的 INF 文件列出的模块类型的 UEFI 固件映像文件，它收集在 Build 阶段的编译阶段生成的.dll 文件，并转换头文件创建出可执行的二进制文件(.efi file)。

5.2.3 固件生成模块的实现

1) FDF 元数据文件的解析

在自动生成阶段中对 DSC, INF, DEC 及 FDF 的部分解析，到固件生成阶段，系统才开始对 FDF 进行完全解析。

通过在命令行输入的“-f”操作指定特定的 FDF 文件，得到对固件各类组织信息。然后 GenFds 工具从 FDF 文件中获得 flash 镜像文件的组织信息。组成固件镜像文件的部分在 INF 文件和 FDF 文件的 FV 段声明出来。这些文件包括文件名、文件类型和其他的有效信息使得 GenFds 工具知道使用 FDF 文件中的哪一个构建规则来生成 FFS 文件。在命令行里使用“-p”选项来指定一个 DSC 文件，这个 DSC 文件声明了 GenFw 生成的镜像文件的输出目录。用“-a”选项来声明生成的镜像文件应

该支持哪一种平台架构。

在 FDF 文件的 FV 字段按照如下格式列出了需要用到 INF 文件路径：

```
INF LakeportX64Pkg/SecCore/SecCore.inf
INF MdeModulePkg/Core/Pei/PeiMain.inf
INF MdeModulePkg/Universal/PCD/Pei/Pcd.inf
INF IntelFrameworkModulePkg/Universal/StatusCode/Pei/PeiStatusCode.inf
INF IntelFrameworkModulePkg/Universal/VariablePei/VariablePei.inf
```

INF 文件中的信息指定了如何生成一个 FFS 文件。如果需要讲一个文件放入到固件镜像中，但是此文件并不是通过 INF 文件中描述的方式生成的，则可以使用文件声明直接指定该文件的相关操作，通过这一系列步骤就可以按规则生成指定的 FFS 文件。

通过 FDF 文件的 FV 字段中 INF 的声明，通知系统在基于各个 INF 模块的类型和内容用哪种 FV 的文件类型去创建 FFS 文件，如下 FV 中指定下面的 INF 文件：

```
DEFINE EDKMU = $(WORKSPACE)/EdkModulePkg/Universal
INF VERSION = "1.1" $(EDKMU)/GenericMemoryText/Dxe/NullMemoryTest.inf
```

系统将生成一个 EFI_FV_FILETYPE_DRIVER 的 FFS 文件，这个文件有三个段，它们的类型分别是 EFI_SECTION_PE32，EFI_SECTION_VERSION 和 EFI_SECTION_DXE_DEPEX。知道 INF 文件中的信息，才可确定如何生成一个 FFS。如果用户可自定义 FFS 指定一个覆盖规则：

```
INF Rule_Override = PICOMPRESSED Ich7Pkg/UhciPei/Ich7Uhci.inf
INF Rule_Override = PICOMPRESSED My/Bus/Pci/UhciPei/UhciPei.inf
```

由上面的例子，GenFds 工具将使用“PICOMPRESSED”规则产生的 FFS，而忽略那些被定义在 INF 文件中的正常处理规则。

如果要把一个文件放到固件映像中，但这个文件并不是用 INF 文件所提供的信息而生成的，比如，一个应该被放到 FV 文件中的微代码，可以使用文件声明(<FileStatements>)直接指定该文件。换言之，文件声明的内容是为了将所有的 EFI FFS 文件整合到一个平台固件。如下文件格式所示，首先，Fat.efi 文件被放置到一个 PE32 段内，然后放至固件类型是“DRIVER”的 FFS 文件中，并用 GUID 值命名。

FILE DRIVER = 961578FE-B6B7-44c3-AF35-6BC705CD2B1F {

SECTION PE32 = FatBinPkg/EnhancedFatDxe/X64/Fat.efi }

这些信息用于告诉系统哪项规则可用来产生 FFS 文件。作为一种可以为不同平台的生成固件的系统设定“-a”参数来指定为哪一个平台进行映像文件编译

2) 构建中间文件

中间文件包括二进制模块、EFI 段文件、先驱文件、子节点创建的 FFS 文件、封装字段文件。

(1) 解析二进制模块

二进制模块有两种方法插入到固件镜像文件中，一种方法是上面提到的文件声明，另一种方法是通过一个 INF 文件去描述这个二进制模块的动作，例如以下描述：

[Defines]

INF_VERSION = 0x00010005

BASE_NAME = Logo

FILE_GUID =7BB28B99-61BB-11D5-9A5D-0090273FC14D

MODULE_TYPE = USER_DEFINED

VERSION_STRING = 1.0

EDK_RELEASE_VERSION = 0x00020000

EFI_SPECIFICATION_VERSION = 0x00020000

[Binaries.common]

BIN|Logo.bmp|*

这个 INF 文件描述了二进制文件 Logo.bmp 将会包装进 Logo FFS 文件中。

(2) 创建 EFI 字段（子段文件）

EFI 段文件是由 GenSec 工具得到了 FDF 文件的中声明的内容就会生成 EFI 段文件。FDF 文件中声明的内容定义了 EFI 段文件需要包含的文件类型和内容。段文件在 FDF 内被分为两种类型，子段文件和父段文件（封装段文件）。

将标准的 PE32+/COFF 文件（.text，.debug，.reloc 和.data）添加 EFI 段头部生成一个标准的 EFI 映像文件。如图 5.1 所示

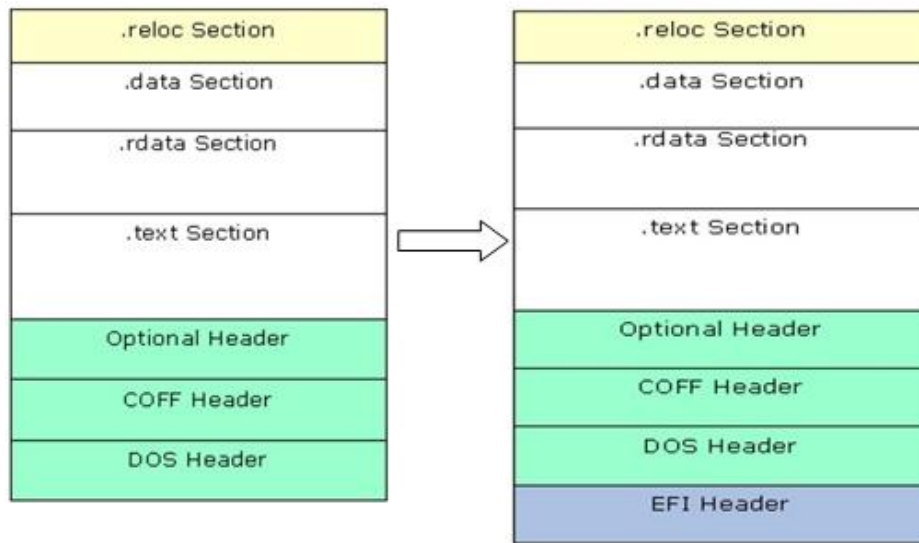


图 5.1 标准 PE32+/COFF 文件转变 EFI 段文件

用 `EFI_IMAGE_SECTION_HEADER` 来替代 PE32+文件的文本段和数据段的头部，数据结构的定义如下：

```
typedef struct {
    UINT8 Name [EFI_IMAGE_SIZEOF_SHORT_NAME];
    union {
        UINT32  Phy Address;//物理地址
        UINT32  Virtual Size;//虚拟长度
    } Misc;
    UINT32  Virtual Address;//虚拟地址
    UINT32  Size Of RawData;//每行数据长度
    UINT32  Pointer To RawData;
    UINT32  Pointer ToRelocations;
    UINT32  Pointer ToLinenumbers;
    UINT16  Number Of Relocations;
    UINT16  Number Of Linenumbers;
    UINT32  Characteristics;
} EFI_IMAGE_SECTION_HEADER;
```

GenSec 工具将 PE32+的段名添加到 EFI 段的名称中，其余的部分做出的修改如下所示：

```
Hdr->Misc.VirtualSize = Size;
Hdr->VirtualAddress = Offset;
Hdr->SizeOfRawData = Size;
Hdr->PointerToRawData = Offset;
Hdr->PointerToRelocations = 0;
Hdr->PointerToLinenumbers = 0;
Hdr->NumberOfRelocations = 0;
Hdr->NumberOfLinenumbers = 0;
Hdr->Characteristics = Flags;
```

文本段 (.text) 的标志值是由 `FI_IMAGE_SCN_CNT_CODE` , `EFI_IMAGE_SCN_MEM_EXECUTE` 和 `EFI_IMAGE_SCN_MEM_READ` 按位或运算所得。

数据段 (.data) 的标志值是由 `EFI_IMAGE_SCN_CNT_INITIALIZED_DATA` , `EFI_IMAGE_SCN_MEM_WRITE` 和 `EFI_IMAGE_SCN_MEM_READ` 按位或运算所得。

.reloc 段的值则是依据 `EFI_IMAGE_SCN_CNT_INITIALIZED_DATA` , `EFI_IMAGE_SCN_MEM_DISCARDABLE` 和 `EFI_IMAGE_SCN_MEM_READ` 按位或运算所得。

.debug 段的标志值是由 `EFI_IMAGE_SCN_CNT_INITIALIZED_DATA` , `EFI_IMAGE_SCN_MEM_DISCARDABLE` 和 `EFI_IMAGE_SCN_MEM_READ` 按位或运算所得。

当这些段值修改完成后, 将 `EFI_COMMON_SECTION_HEADER` 放在文件的头部。到此阶段一个子段文件创建完成。

(3) 创建封装 EFI 字段 (父段文件)

封装 EFI 段可以包含一个或多个子段或其他封装部分。封装 `EFI_SECTION` 段的类型有三种, 如表 5.1 所示, 列表中前两种类型会有扩展的头部信息。

表 5.1 封装的 `EFI_SECTION` 类型

段类型	值
<code>EFI_SECTION_COMPRESSION</code>	<code>0x01</code>
<code>EFI_SECTION_GUID_DEFINED</code>	<code>0x02</code>
<code>EFI_SECTION DISPOSABLE</code>	<code>0x03</code>

压缩文件的段使用 `EFI_SECTION_COMPRESSION` 做首部， `GUID` 定义的段使用 `EFI_SECTION_GUID_DEFINED` 做首部。这些段的长度是一个 24 位无符号整数。

如果是 `EFI_SECTION_COMPRESSION`，做为标准的压缩，压缩类型值必须设置为 `0x01`，如果是不压缩的文件 `0x00`。UEFI2.3 对 `EFI_SECTION_COMPRESSION` 的定义如下：

```
typedef struct {
    EFI_COMMON_SECTION_HEADER CommonHeader;
    //24 位无符号整数，解压后包括 EFI_COMMON_SECTION_HEADER
    UINT32 UncompressedLength;
    //0x01 表示标准压缩，0x00 表示没有压缩
    UINT8 CompressionType;
} EFI_COMPRESSION_SECTION;
```

`EFI_GUID_DEFINED_SECTION` 用于标准的非压缩，以 `GUID` 命名定义的段在 `EFI_COMMON_SECTION_HEADER` 下面。在这个 `GUID` 后面又有两个额外的 16 位无符号整数的参数，第一个是数据偏移量，包含以字节为单位从普通首部开始至数据的第一个字节之间的偏移量，一个属性参数是一个位代码，它声明了这个段内容的具体特征。这些前缀头部的数据文件其中可能包括标准 `PE32` 头部。

系统生成 `EFI` 段的文件只要有文件和文件类型的声明就可以对 `PE32+/COFF` 文件进行修改。除此，还可以通过 `FDF` 文件[Rule]下指定规则生成 `EFI` 段文件。

构建系统解析 `FDF` 文件，取得 `FD`，`FV` 和 `FFS`，通过这些信息可以知道需要用到的文件和构建 `EFI` 段的规则。

3) 创建 `FV` 镜像文件

当需要的 `FFS` 文件生成之后，下一步就是调用 `GenFv` 工具来将 `FFS` 文件组成一个固件卷文件（`FV`）。要创建一个的 `FV` 文件也是需要依据 `FDF` 文件，在 `FDF` 文件描述如何结合所有 `FFS` 文件。文件到不同的 `FV` 的文件，以及在 `FD` 文件内布局。在 `FDF` 中定义的每个 `FV` 部分都是用来构建一个 `FV` 的数据结构。`FV` 的头部则是依据 `UEFI2.3` 和 `PI1.2` 的规范 `GenFv` 工具需要知道将要生成的目标 `FV` 文件的属性和要绑定进去的 `FFS` 文件，而这个内容是在 `FDF` 文件中的 `FV` 字段中定义的。`FV` 字段如下示例：

```
typedef struct {
    UINT8 ZeroVector[16];
```

```
EFI_GUID FileSystemGuid;
UINT64 FvLength;
UINT32 Signature;
EFI_FVB_ATTRIBUTES_2 Attributes;
UINT16 HeaderLength;
UINT16 Checksum;
UINT16 ExtHeaderOffset;
UINT8 Reserved[1];
UINT8 Revision;
EFI_FV_BLOCK_MAP BlockMap[];
} EFI_FIRMWARE_VOLUME_HEADER;
```

开始的 16 字节(ZeroVector)设置为零。FV 的 FileSystemGuid 的值是 PI 规范定义的 GUID(EFI_FIRMWARE_FILE_SYSTEM2_GUID)，作为标识为 PI 兼容的固件卷。Signature 是 “_FVH”，并设置保留的字节等于零。PI 规范定义的 Revision 是设置值为 0x02。

由于 FFS 的文件要被添加到的 FV，FFS 的长度值也被加到 FvLength 域，FvLength 是一个完整的固件卷长度，它包括头部和扩展头部的信息。此外，一个 FFS 文件添加到的 FV，如果是 ROM 驱动运行，驱动程序的基地址在 FFS 文件里将被调整为物理位置的 ROM 的位置，也就是基地址与偏移的和。

属性(Attribute)被定义在 FDF 文件中。

HeaderLength 是 FV 首部的大小。

BlockMap 数组是 FFS 文件的映射，在块中给定 FFS 的长度和在 FV 中的每 FFS 文件块的大小，从第一个 FFS 文件开始。可以看成这是一个块的索引，不通过名字指定每个 FFS。如果需要一个扩展头，会立即放入 BlockMap 数组后面。

ExtHeaderOffset 是用于记录一个扩展头部的位 置。每块都会与 EFI_FVB2_ALIGNMENT 属性指定的最大值对齐。另外，它是允许使用可变块长度的设备，这样，每块条目都有 BlockMap[index].NumBlocks=1，而 BlockMap[index]。块长度根据 FFS 文件的大小而发生变化。如果不包括扩展头，ExtHeaderOffset 设置 为零。如果有一个扩展头，后面跟零个或多个可变长度的延伸

(EFI_FIRMWARE_VOLUME_EXT_ENTRY)项。

在加入 FDF 文件指定的最后一个 FFS 文件之前，FFS 文件必须要修改，以符合卷首文件规范(the Volume Top File specification)。在最后的 FFS 文件被添加，校验字段设置为零。

以上结构中 EFI_FVB_ATTRIBUTES 定义了 FV 文件的属性，EFI_FV_BLOCK_MAP BlockMap[]定义了所有需要用到的 FFS 文件的映射。首先调用 GenFds 工具解析 FDF 文件的 FV 字段，将有效内容写入 INF 文件中，然后通知 GenFv 工具根据解析出的 FV 文件的属性生成正确的目标 FV 文件。

综上所述，对于编译工具而言，需要两类信息来生成目标 FV 文件，一个是 FV 的属性，另一个是要放置到这个 FV 文件中的文件清单。

4) 创建完整固件文件

这是生成固件的最后一个阶段。生成 FD 镜像文件的规则是在 FDF 文件的 FD 字段中的<RegionLayout>来定义的。一般的闪存设备是一个 flash 芯片，目前绝大部分的闪存设备的块的数目都是可变的。当需要将一个固件文件烧入到 flash 芯片中时，在任何时间内必须将设备中所有的块一起烧入。闪存中一般有多个可变容量的块，它们的基址由偏移量与块的大小来确定。由于把一整块全部烧入是无法实现的，设备中最后的块在一些情况下会出现空白。所以所有的 FD 镜像文件的布局必须从块的起始点开始。为了确保闪存中是空白的，在烧入映像文件之前首先应该对块进行擦除。在烧入镜像之前。多个非易失性内存可以当作一个连续的空间，可也可以被当作是一个单一设备。区域必须定义升序排列，不能重叠。每个布局区域开始以 8 位十六进制偏移量。其次通过管道“|”，紧接着就是字符区域的大小，也是十六进制，以“0x”开始，FD 布局区域的格式如下：

Offset|Size

[TokenSpaceGuidCName.PcdOffsetCName|TokenSpaceGuidCName.PcdSizeCName]

[RegionType]

如果指定 RegionType，它就必须是 FV、DATA 或 FILE 中的一种。不指定 RegionType 意味着该地区以最初偏移值开始。未指定的区域类型通常是用于系统复位的事件日志。

5.3 本章小结

本章主要阐述了固件生成工具系统实现的开发环境，以及自动生成模块、编译模块和固件生成模块的具体实现过程。

6 系统测试

6.1 Windows7 64位平台上的系统测试

6.1.1 搭建 Windows 平台的系统环境

在 Windows 平台下选用 NT32Pkg 作为测试固件生成工具的实例。由于 NT32Pkg 是 EDKII 源代码中的示范实例，它的主要目的是提供给开发人员在平台上对 UEFI 的应用程序进行开发和调试，降低开发成本。同时 NT32 中的配置和源代码是完全按照标准规范设计，这就可以最大化的避免此次测试由于 EDKII 开发框架的源码文件而影响测试结果。

6.1.2 Windows7平台的测试环境

固件生成工具在 Windows7 64 位平台上所需的环境配置如下：

编译平台：Windows7 64 位。

相关软件：Python 2.5.4；VS 2008；Tortoise SVN。

装载固件平台架构：X64 架构。

功能测试：INTEL PSI TOOL and QA TEAM\TIANO TEST TEAM.

6.1.3 Windows7平台的测试步骤

测试固件生成过程的步骤如下：

- (1) 通过 Tortoise SVN 工具从服务器上下载最新的 EDKII 源码包。
- (2) 将固件生成工具拷贝到 EDKII 的根目录。
- (3) 打开 VS2008 的命令行，将当前目录定位到 EDKII 的根目录，此目录即是 WORKSPACE 路径。如图 6.1 所示。

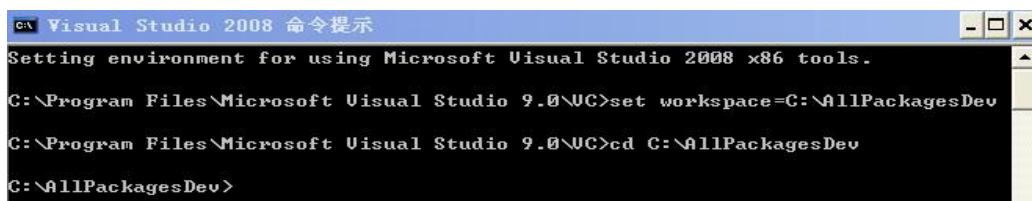
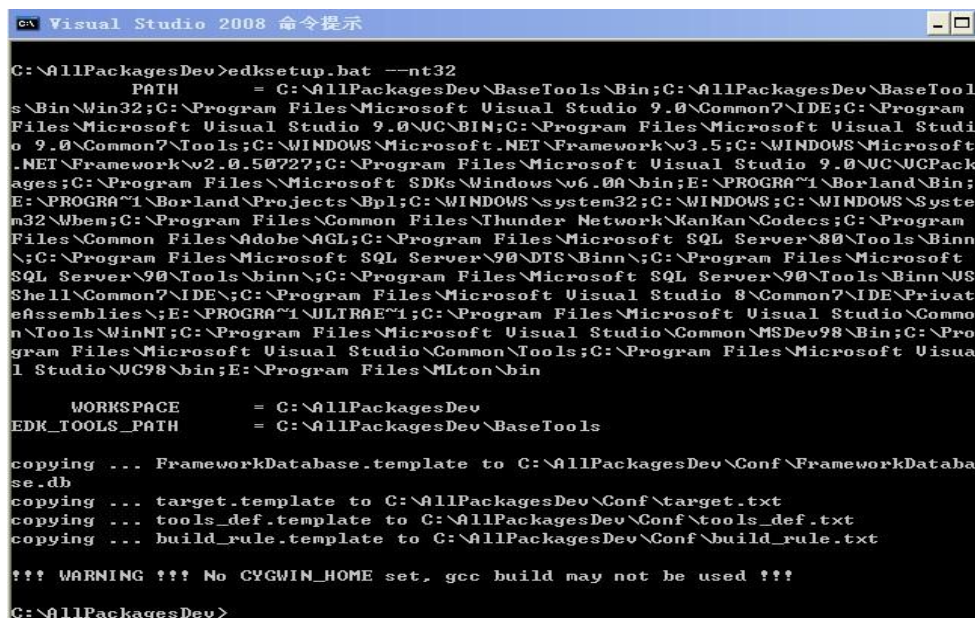


图 6.1 设置工作目录

华中科技大学硕士学位论文

(4) 运行 NT32 环境变量配置批处理文件 edksetup.bat, 设置构建系统需要的环境变量。如图 6.2 所示。



```
C:\AllPackagesDev>edksetup.bat --nt32
PATH = C:\AllPackagesDev\BaseTools\Bin;C:\AllPackagesDev\BaseTools\Bin\Win32;C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE;C:\Program Files\Microsoft Visual Studio 9.0\UC\BIN;C:\Program Files\Microsoft Visual Studio 9.0\Common7\Tools;C:\WINDOWS\Microsoft.NET\Framework\v3.5;C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727;C:\Program Files\Microsoft Visual Studio 9.0\UC\Packages;C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin;E:\PROGRAM~1\Borland\Bin;E:\PROGRAM~1\Borland\Projects\Bpl;C:\WINDOWS\system32;C:\WINDOWS\System32\Wbem;C:\Program Files\Common Files\Thunder Network\KanKan\Codecs;C:\Program Files\Common Files\Adobe\AGL;C:\Program Files\Microsoft SQL Server\80\Tools\Binn\;C:\Program Files\Microsoft SQL Server\90\DTs\Binn\;C:\Program Files\Microsoft SQL Server\90\Tools\Binn\;C:\Program Files\Microsoft SQL Server\90\Tools\Binn\Shell\Common7\IDE;C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\PrivateAssemblies\;E:\PROGRAM~1\ULTRA~1;C:\Program Files\Microsoft Visual Studio\Common\Tools\WinNT;C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin;C:\Program Files\Microsoft Visual Studio\Common\Tools;C:\Program Files\Microsoft Visual Studio\VC98\bin;E:\Program Files\MLton\bin

WORKSPACE = C:\AllPackagesDev
EDK_TOOLS_PATH = C:\AllPackagesDev\BaseTools


copying ... FrameworkDatabase.template to C:\AllPackagesDev\Conf\FrameworkDatabase.db
copying ... target.template to C:\AllPackagesDev\Conf\target.txt
copying ... tools_def.template to C:\AllPackagesDev\Conf\tools_def.txt
copying ... build_rule.template to C:\AllPackagesDev\Conf\build_rule.txt

!!! WARNING !!! No CYGWIN_HOME set, gcc build may not be used !!!

C:\AllPackagesDev>
```

图 6.2 设置环境变量

(5) 构建固件。用户只需要运行“build”命令, 构建系统第一步是解析三个数据配置文件, 为各模块生成一系列描述了依赖关系的文件, 之后进入自动化构建流程。图 6.3 所示是构建初始化阶段。



```
C:\AllPackagesDev>build
Build environment: Windows-32bit
Build start time: 18:58:25, Sep.14 2011

WORKSPACE = c:\allpackagesdev
ECP_SOURCE = c:\allpackagesdev\edkcompatibilitypkg
EDK_SOURCE = c:\allpackagesdev\edkcompatibilitypkg
EFI_SOURCE = c:\allpackagesdev\edkcompatibilitypkg
EDK_TOOLS_PATH = c:\allpackagesdev\basetools

TARGET_ARCH = IA32
TARGET = DEBUG
TOOL_CHAIN_TAG = MYTOOLS

Active Platform = c:\allpackagesdev\Nt32Pkg\Nt32Pkg.dsc
Flash Image Definition = c:\allpackagesdev\Nt32Pkg\Nt32Pkg.fdf

Processing meta-data ...
```

图 6.3 构建初始化

- (6) 生成固件镜像文件。
- (7) 通过烧片器将固件镜像文件烧入 flash 芯片。
- (8) 将 flash 芯片装载入配置有 UEFI BIOS 环境的硬件, 进入开机启动阶段, 如图 6.4 成功进入 BIOS 界面。

将中间输出文件和固件提交给测试小组，完整测试 BIOS 功能。

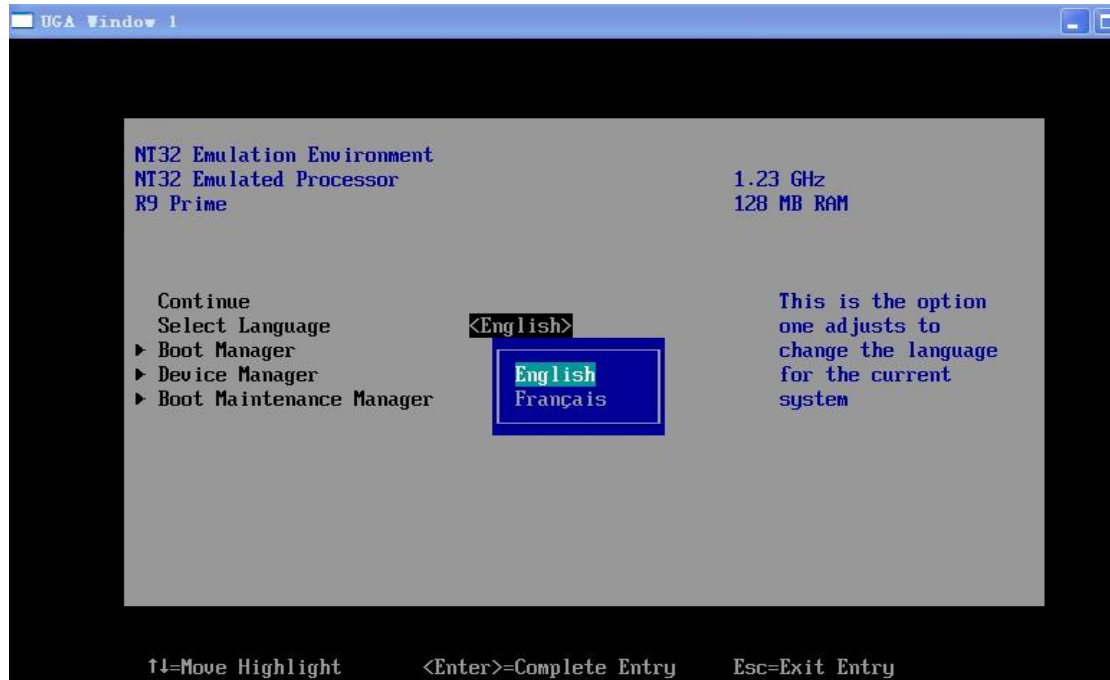


图 6.4 UEFI BIOS 界面

6.1.4 测试结果分析

EDKII 各模块和包之间有着复杂的依赖关系，构建目标架构为 X64 平台需要用到以下的 Package:

- MdePkg
- MdeModulePkg
- IntelFrameworkPkg
- IntelFrameworkModulePkg
- Nt32Pkg
- edkShellBinPkg
- EdkFatBinPkg

在 Nt32Pkg 文件夹下的 Nt32Pkg.fdf 文件中描述的固件设备映像文件，其内容根据 NT32 固件结构设置如下：

```
[FD.Nt32]
# FD basic definitions
BaseAddress = 0x0|gEfiNt32PkgTokenSpaceGuid.PcdWinNtFdBaseAddress
```

```
# The base address of the FLASH Device.
Size = 0x002a0000      # The size in bytes of the FLASH Device
ErasePolarity = 1
BlockSize = 0x10000
NumBlocks = 0x2a

#####
#####
# Following are lists of FD Region layout which correspond to the
# locations of different images within the flash device.
# Regions must be defined in ascending order and may not overlap.
#####
#####
0x00000000|0x00280000
FV = FvRecovery
0x00280000|0x0000c000
gEfiNt32PkgTokenSpaceGuid.PcdWinNtFlashNvStorageVariableBase|gEfiMdeMod
ulePkgTokenSpaceGuid.PcdFlashNvStorageVariableSize
# NV_VARIABLE_STORE
.....
0x0028c000|0x00002000
# NV_EVENT_LOG
gEfiNt32PkgTokenSpaceGuid.PcdWinNtFlashNvStorageEventLogBase|gEfiNt32Pkg
TokenSpaceGuid.PcdWinNtFlashNvStorageEventLogSize
.....
0x0028e000|0x00002000
gEfiNt32PkgTokenSpaceGuid.PcdWinNtFlashNvStorageFtwWorkingBase|gEfiMde
ModulePkgTokenSpaceGuid.PcdFlashNvStorageFtwWorkingSize
```

固件生成工具在构建固件时，通过 DEBUG 显示出来的实际所生成的固件结构，再与 FDF 文件中设置的各区域地址及其区域名相比较，检验生成内容的是否与设定的值一致，如图 6.5 所示，可判定经系统编译后生成的固件结构与其保持一致。

```

Generate Region at Offset 0x280000
Region Size = 0xC000
Region Name = DATA

Generate Region at Offset 0x28C000
Region Size = 0x2000
Region Name = None

Generate Region at Offset 0x28E000
Region Size = 0x2000
Region Name = DATA

Generate Region at Offset 0x290000
Region Size = 0x10000
Region Name = None

Generate Region at Offset 0x0
Region Size = 0x280000
Region Name = FU

Generate Region at Offset 0x280000
Region Size = 0xC000
Region Name = DATA

Generate Region at Offset 0x28C000
Region Size = 0x2000
Region Name = None

Generate Region at Offset 0x28E000
Region Size = 0x2000

```

图 6.5 生成的固件结构

接下来,通过验证生成的固件的内容来检验其内容正确性,对 FVRECOVERY.Fv 文件进行分析,用 UltraEdit 工具打开 FVRECOVERY.Fv,发现它的地址是从 0x00000000 到 0x00280000,这与由 FDF 文件中设定的值是一致的。由于文件的数据量比较大,现截取一小部分进行分析,如图 6.6 所示,根据 PI 规范定义的 EFI_FIRMWARE_VOLUME_HEADER 数据结构可知,从 0x00000000h 地址开始的 16 个字节按照约定被设置为零值,0x00000010h 地址开始的 16 个字节表示的是 File-SystemGuid 值,在固件生成系统的 PiFirmwareFileSystem.h 文件中是宏定义:

```

#define EFI_FIRMWARE_FILE_SYSTEM2_GUID { \
0x8c8ce578, 0x8a3d, 0x4f1c, {0x99, 0x35, 0x89, 0x61, 0x85, 0xc3, 0x2d, 0xd3}}

```

```

00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000010h: 78 E5 8C 8C 3D 8A 1C 4F 99 35 89 61 85 C3 2D D3 ;
00000020h: 00 00 28 00 00 00 00 00 5F 46 56 48 FF FE 0C 00 ;
00000030h: 48 00 76 F6 00 00 00 02 28 00 00 00 00 00 01 00 ;
00000040h: 00 00 00 00 00 00 00 00 0A CC 45 1B 6A 15 8A 42 ;
00000050h: AF 62 49 86 4D A0 E6 E6 88 AA 02 00 5C 00 00 F8 ;
00000060h: 44 00 00 19 4F DA 3A 9B 56 AE 24 4C 8D EA F0 3B ;
00000070h: 75 58 AE 50 42 04 61 A3 9F E6 F3 4D 82 CA 23 60 ;
00000080h: C4 03 1A 23 37 52 22 9D 01 FA 4C 46 A9 49 BA AB ;
00000090h: C0 2D 31 D0 BD D6 33 D2 B1 F1 5A 42 BF 45 5C AF ;
000000a0h: 2B 88 ED DC FF FF FF FF E7 0E 51 FC DC FF D4 11 ;
000000b0h: BD 41 00 80 C7 3C 88 81 36 AA 02 00 3C 00 00 F8 ;
000000c0h: 24 00 00 19 57 72 CF 80 AB 87 F9 47 A3 FE D5 0B ;
000000d0h: 76 D8 95 41 4A AB 4C 15 B5 52 CD 46 99 C3 43 68 ;
000000e0h: AB BA CF FD FF FF FF FF FF FF FF FF FF FF FF ;
000000f0h: FF FF FF FF FF FF FF FF 11 AA F0 00 00 0F 00 F8 ;

```

图 6.6 FVRECOVERY.Fv 部分内容

从 0x00000020h~0x00000023h, 是 FvLength 值 0x00280000, 紧跟着的 12 个字节是 Signature 和 FV 文件的 Attribute; 0x00000030h~0x00000031h, 存放的是 FV 文件的 HeaderLength, 接下来 2 个字节是 Checksum; 从 0x00000034h~0x00000047h 中间的内容描述的是当前使用的 FV 文件的版本, 每个块(Block)大小, 以及该 FV 文件有多少块。这些值在 NT32Pkg 文件中有定义, 但随着 UEFI/PI 规范的更新, 也是为了更好的维护 EDKII 框架中各类平台间的信息, 在固件生成系统中重新为 FV 文件的数据进行了定义, 所以生成的*.FV 文件中数据信息取的固件生成系统中的值。

FV 头部信息填充完整之后, 就是固件文件系统(FFS)文件的填充, 如下图 6.7 所示, 截取 FVRECOVERY.inf 部分文件内容。

```
[files]
EFI_FILE_NAME = c:\allpackagesdev\Build\NT32\DEBUG_MYT00LS\FV\Ffs\1B45CC0A-156A-428A-AF62-49864DA0E6E6FURECOVERY\1B45CC0A-156A-428A-AF62-49864DA0E6E6FURECOVERY.Ffs
EFI_FILE_NAME = c:\allpackagesdev\Build\NT32\DEBUG_MYT00LS\FV\Ffs\FC510EE7-FFDC-11D4-BD41-0080C73C8881FURECOVERY\FC510EE7-FFDC-11D4-BD41-0080C73C8881FURECOVERY.Ffs
```

图 6.7 FVRECOVERY.inf 部分内容

0x00000048h~0x00000057h, 填充的是第一个 FFS 的 GUID, 接下来的两个字节的內容“0x88 和 0xAA”是 FFS 的 INTEGRITY_CHECK 分别包含了 Header 和 File 的 Checksum; 0x0000005ah~0x0000005fh, 分别是 FFS 的 Type, Attribute, Size, State 的内容, 这样一个 FFS 头部信息就被填充完整。FFS 头部下面跟着的是 EFI Section 的头部信息, 从 0x00000060h~0x00000063h 填充的是这个 EFI Section 的 Size=0x48 和 Type=EFI_SECTION_RAW, 他们各占两个字节。在这个 EFI Section 下面存放的就是 Driver 的信息, 每个一个 Driver 都有自己的 GUID 做为对外的接口, 存放的顺序根据 NT32Pkg.fdf 文件中 APRIORI PEI 中的顺序来填充的, 如图 6.8 所示, 0x00000064h~0x000000a3h, 共存放了 4 个 Driver 的 GUID, 这表示, 当程序被执行运行时, 这些 Driver 将按顺序执行。0x000000a4h~0x000000a7h 是填充位。从 0x000000a8h 开始是下一个 EFI Section 的内容, 一个 EFI Section 接一个的直到 FVRECOVERY.Fv 这个文件被填充完整。通过对结果从结构和内容两方面的数据分析, 可以验证固件生成系统生成固件的准确性以及可执行性。

```
APRIORI PEI {  
  INF MdeModulePkg/Universal/PCD/Pei/Pcd.inf  
  INF MdeModulePkg/Universal/ReportStatusCodeRouter/Pei/ReportStatusCodeRouterPei.inf  
  INF MdeModulePkg/Universal/StatusCodeHandler/Pei/StatusCodeHandlerPei.inf  
  INF Nt32Pkg/WinNtOemHookStatusCodeHandlerPei/WinNtOemHookStatusCodeHandlerPei.inf  
}  
APRIORI DXE {  
  INF MdeModulePkg/Universal/PCD/Dxe/Pcd.inf  
  INF Nt32Pkg/MetronomeDxe/MetronomeDxe.inf  
}
```

图 6.8 NT32Pkg.fdf 部分内容

6.2 Linux 平台上的系统测试

6.2.1 搭建 Linux 平台上的系统环境

在 Linux 平台下选用 UnixPkg 作为测试固件生成工具的实例。和 NT32Pkg 一样，UnixPkg 也是 EDKII 源代码中的示范实例，这就可以最大化的避免此次测试由于 EDKII 开发框架的源码文件而影响测试结果。

6.2.2 Linux 平台的测试环境

固件生成工具在 Linux 平台上所需的环境配置如下：

编译平台：Ubuntu 9.04 32 位。

相关软件：Gcc 3.0；Python 2.5；Texinfo；Bison；Flex；Libmpfr；Libgmp。

在 Ubuntu 系统可以直接执行下面这一条命令：

```
sudo apt-get install build-essential texinfo bison flex libgmp3-dev libmpfr-dev
```

目标平台架构：IA32 架构。

功能测试：INTEL PSI TOOL and QA TEAM\TIANO TEST TEAM.

6.2.3 Linux 平台的测试步骤

(1) 由于固件生成工具源码有一部分的代码是 C 代码，需要对这一部分进行编译：

编译指令：make -f GNUmakefile。

(2) 为固件生成工具目录和 WORKSPACE 目录建立一个软链接：

```
ln -s /home/usr/BaseTools /home/usr/EDKIIWorkspace/Conf/BaseToolsSource
```

(3) 运行：edksetup.sh，目的也是设置环境变量。

(4) 构建固件：在 Linux 系统上构建 UnixPkg 的固件，需要使用如下几个指令：

“-p”参数指定平台信息描述文件；“-a”指定要为哪个平台创建固件文件；“-t”指定使用的工具链。其中命令行的定义优先级高于 target.txt 文件里的，只有当命令行没有指定的时候，系统才会去读取 target.txt 里面相应的信息。构建指令：

```
build -p UnixPkg\UnixPkg.dsc -a IA32 -t ELFGCC
```

(5) 创建完成后，将固件镜像文件烧入 flash 芯片，将 flash 芯片装载入配置有 UEFI BIOS 环境的硬件，进入开机启动阶段。

(6) 将中间输出文件和固件提交给测试小组，完整测试 BIOS 功能。

6.2.4 测试结果分析

Unix 平台与 NT32 平台类似，它是运行在类 Unix 系统上，如各种 Unix 和 Linux 系统。Unix 的仿真平台加载器 SecMain 模拟类 Unix 操作系统。

在 /home/MyNewR9Pkg/Build/Unix/DEBUG_ELFGCC/FV 这个目录下找到 FVRECOVERY.Fv 的文件，如图 6.9 所示，其内容分析与 NT32 的分析方式类似，从中 0x00000000 到 0x00470000 是 FV 文件的基本信息和 Header 信息。

```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000010h: 78 E5 8C 8C 3D 8A 1C 4F 99 35 89 61 85 C3 2D D3 ;
00000020h: 00 00 28 00 00 00 00 00 00 00 5F 46 56 48 FF FE 07 00 ;
00000030h: 48 00 7B F6 00 00 00 00 02 28 00 00 00 00 01 00 ;
00000040h: 00 00 00 00 00 00 00 00 0A CC 45 1B 6A 15 8A 42 ;
00000050h: AF 62 49 86 4D A0 E6 E6 A8 5A 02 00 3C 00 00 F8 ;
00000060h: 24 00 00 19 4F DA 3A 9B 56 AE 24 4C 8D EA F0 3B ;
00000070h: 75 58 AE 50 3A F5 C0 1E E0 FD 76 45 8F 25 7A 1A ;
00000080h: 41 0F 58 EB FF FF FF FF E7 0E 51 FC DC FF D4 11 ;
00000090h: BD 41 00 80 C7 3C 88 81 26 5A 02 00 4C 00 00 F8 ;
000000a0h: 34 00 00 19 57 72 CF 80 AB 87 F9 47 A3 FE D5 0B ;
000000b0h: 76 D8 95 41 1B 0A DE FE A2 BC 9F 4A BB 2B D9 FD ;
000000c0h: 7D EC 2E 9F FE F6 48 F3 85 89 DB 11 B4 C3 00 40 ;
000000d0h: D0 2B 18 35 FF FF FF FF FF FF FF FF FF FF FF ;
```

图 6.9 FVRECOVERY.Fv 部分内容

0x00000048h~0x00000057h，填充的是第一个 FFS 的 GUID，接下来两个字节的“0xA8 和 0x5A”是 FFS 的 INTEGRITY_CHECK 的值，从 0x0000005ah 到 0x0000005fh 分别描述了这一个 FFS 的 Type=0x03，Size=0x3C，State=0x07，在这填充的 State 的值是一个经过系统取反后的值“0xF8”，这样一个 FFS 头部信息就被填充完整。一个 FFS 文件可包含一个或多个 EFI Section，EFI Section 的头部信息就是存放在 0x00000060h~0x00000063h 描述了 Size=0x24 和 Type 的值是 0x19，系统中定义的 0x19 对应 Type=EFI_SECTION_RAW。0x00000064h~0x00000083h 存放 EFI

Section 中包含的 2 个 Driver 的 GUID, Driver 的内容就 UnixPkg.fdf 中 APRIORI PEI 中指明的两个 Driver。通过对结果从结构和内容两方面的数据分析, 得以验证固件生成系统生成固件的准确性以及可执行性。

6.3 本章小结

本章主要是测试固件生成工具的可执行性和生成的固件的正确性, 分别选择了两个主要应用平台 Windows7 和 Ubuntu9.04, 利用开源的 NT32Pkg 和 UnixPkg 来为 X64 和 IA32 的目标平台构建固件并给出中间生成文件的分析, 并且通过组内的测试小组验证了生成固件的正确性, 通过这两个部分的测试和验证来证明固件生成工具的正确性和实用性。

7 总结与展望

7.1 全文总结

随着计算机硬件和体系结构的不断发展，人们对计算机性能的追求越来越高。为了替代传统 BIOS，业界制定出了下一代 BIOS 固件技术的国际标准 UEFI(Unified Extensible Firmware Interface): 统一可扩展固件接口。EDKII 是一个开源的 UEFI BIOS 的发布框架，它里面包含了各种具有依赖关系的功能模块、描述文件和底层库函数。

本文的研究从目前最新的 EDKII 体系结构入手，结合 Intel 主流架构 IA32 和 X64，了解 UEFI/PI 业界标准，深入分析 EFI 固件文件系统的逻辑结构和 EFI 文件格式，根据不同的应用平台，提出了基于 EDKII 的固件生成工具的设计方案，并给出具体的实现方法。本文完成了以下几个主要的工作：

(1) 详细的介绍了 UEFI 的背景和发展现状，研究了 UEFI BIOS 的运行机制、体系结构和工作流程。然后详细分析了 EDKII 开源框架的内部结构和功能模块。根据 UEFI/PI 的业界规范和 EDKII 的开发框架规范，固件设备的逻辑结构，设计完成一款可跨平台操作的固件生成工具。

(2) 为了提高固件生成工具的通用性、可操作性和可移植性，该固件生成工具需要能够跨平台运行，包括 Windows 和 Linux 系统；为了能够使用户能够最方便的进行构建操作，系统采用建立编译链的方案，这就需要为每个平台自己的编译工具进行相关配置。为固件生成工具构造了三个配置文件，分别详细定义了要编译什么、怎样编译、用什么编译，以便于更好的满足用户的需求。

(3) 设计出针对不同固件平台架构、不同构建目标的固件文件的实现逻辑。固件平台的架构有 IA32 和 X64 两种，EDKII 的源码中有针对不同平台架构的描述信息和配置文件，而外部输入也可以对某些文件内容重新定义。针对这些问题设计了一个最优化的解析定义文件和配置文件的逻辑，对整个固件生成工具系统至关重要，保证了固件生成工具正常运行的第一步。

(4) 确定了固件生成工具的三个主要功能模块：自动生成模块、编译模块和固

件生成模块。自动生成模块负责提取配置信息、构建目标和构建规则并生成编译链文件、中间文件和 `makefile` 文件；编译模块负责调用编译工具、链接工具和第三方工具将上一阶段输出的文件生成符合 UEFI 规范的 EFI 文件；最后固件生成模块根据构建目标和规则生成最终的固件镜像文件。

(5) 为了测试固件生成工具的可执行性和生成的固件的正确性，分别选择了两个主要应用平台 Windows7 和 Ubuntu9.04, 利用开源的 NT32Pkg 和 UnixPkg 来为 X64 和 IA32 的目标平台构建固件并给出中间生成文件的分析，并且通过组内的测试小组验证了生成固件的正确性，通过这两个部分的测试和验证来证明固件生成工具的正确性和实用性。

本文最终完成了预期目标，实现了一个跨平台运行的、可为 IA32 和 X64 两种目标架构生成固件文件的固件生成工具。

7.2 展望

本文研究的固件生成工具能够在 Windows 和 Linux 下运行，能为 IA32 和 X64 平台架构构建固件镜像文件。根据一年多来对课题的研究和分析发现本课题还有更远的发展前景：

(1) 固件生成工具最终生成的是固件镜像文件。在一些开发调试阶段可能还会需要其他的镜像文件或者中间文件，这仍需要在后续的开发过程做进一步的实现。

(2) 固件生成工具能在 Windows 和 Linux 操作平台下运行，仅支持 IA32、X64 两类主流的平台架构，今后可以进一步研究如何支持其他操作系统，例如 MAC OS 等，同时，如何构建其他非主流架构平台下的固件是值得深入研究的方向。

致 谢

光阴如箭，岁月如梭，转眼间研究生生活已经过去两年多了。在这两年多的时间里，我不仅学习了专业课程方面的相关知识，同时在外实习的一年工作经验，极大的锻炼了自己的适应能力和抗压能力。这一过程中，有很多人对我帮助过，指导过。在此，我表示深深的感谢。

首先，我要感谢的是我的导师，肖来元教授。感谢他这两年多来，一直对我学习和生活上的关心和帮助。肖老师治学严谨，工作繁忙。但总不忘叮嘱我们时刻保持学习的心态。同时，每个月让我发给他实习期间的总结，并能给予深刻的评价。这些对于我后期的论文工作都起到了很大的帮助。在此，我表示深深的感谢！

其次，我想感谢的是我在上海英特尔实习期间的领导、同事和同学。第一次进入到这么大的企业中实习，开始遇到了很多的困难，感谢他们在这个过程中对我提供的帮助和关怀。特别是我的辅导人朱哥，在技术上给予了我很多的指导，而且能耐心的讲解给我，帮我梳理整个项目的脉络，对我论文框架的搭建起到了很大的帮助。同时还有我实验室的同学，和他们在一起我很快乐。感谢一路上有他们的陪伴，使我的研究生生活更加充实和美满。

最后，我想感谢我的家人。谢谢他们一直在背后默默的关心和支持，使我不断前进。

参考文献

- [1] 曾颖明. 基于 UEFI 的可信 Tiano 设计与研究. 计算机工程与设计 2009, 30(11): 23-24
- [2] 洪蕾. UEFI 的颠覆之旅. 中国计算机报, 2007(5): 1-3
- [3] Intel Corporation. Intel 64 and IA-32 Architectures Software Develop's Manual. The United States: Intel Corporation, 2009: 10-778
- [4] 倪光南. UEFI BIOS 是软件业的蓝海. UEFI 技术大会, 2007: 6-13
- [5] 余超志, 朱泽民. 新一代 BIOS_EFI_CSSBIOS 技术研究. 科技信息(学术版), 2006(5): 14-15
- [6] 潘登. EFI 结构分析与 Driver 开发: [学士学位论文]. 湖南: 国防科技大学图书馆, 2004
- [7] 邢卓媛. 基于 UEFI 的网络协议栈的研究与改进: [学士学位论文]. 上海: 华东师范大学图书馆, 2011
- [8] 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
- [9] 欧文. Intel 汇编语言程序设计. 第 5 版. 温玉杰译. 第 1 版. 北京: 电子工业出版社, 2008: 14-138
- [10] 丘恩. Python 核心编程. 中文第 2 版. 宋吉广译. 第 1 版. 北京: 人民邮电出版社, 2008: 20-265
- [11] 孙广磊. 征服 Python: 语言基础与典型应用. 第 1 版. 北京: 人民邮电出版社, 2007: 11-25
- [12] Intel Corporation, EFI 1. 1 Driver Model Draft, May, 2001
- [13] Intel Corporation, EFI Developer Kit(EDK)Getting Started Guide
- [14] Morales López A. Validation of ESCAP-CD as an instrument of measure for the evaluation of the quality of life in prostatic cancer. Actas Urologicas Espanolas, 2002, 26(5): 15-17
- [15] Intel Corporation, EFI How To Guide Draft, March, 2004

华中科技大学硕士学位论文

- [16] Intel Corporation, EFI Shell Developer's Guide, June, 2005
- [17] Intel Corporation. 2008-10. EDKII user manual Revision 0. 5: 10-12
- [18] 庄克良, 高云岭, 纪向尚. UEFI BIOS 在复杂嵌入式系统中的可应用性的研究. 舰船电子工程, 2009(12): 1-10
- [19] Intel Corporation. 2004-03-12. Setup Design Guide. Version 0. 7: 20-80
- [20] Fu-Chan Wei. Bilateral High Upper Limb Replantation in a Child. Plastic and Reconstructive Surgery, 2004, 113(6): 23-25
- [21] Intel Corporation. 2009-09-03. Driver Writer's Guide for UEFI 2. 0. Revision 0. 95: 9-51.
- [22] Etter J F. Development and validation of the Attitudes Towards Smoking Scale (ATS-18). Addiction, 2000, 95(4): 27-29
- [23] Intel Corporation. 2006-10-04. Intel Platform Innovation Framework for EFI Human Interface Infrastructure Specification. : 12-23
- [24] Intel Corporation. 2006-10-09. MDE Library Spec. Version 0. 2: 10-30
- [25] Gong L, Zhao Y, Liao JH. Research on the Application Security Isolation Model. China Communications . 2010
- [26] Intel Corporation. 2008-08. EDK II Extended INF File Specification Revision. Version 1.1: 11-15
- [27] Intel Corporation. 2010-05. EDKII Flash Description File (FDF) Specification. Version 1. 22
- [28] Intel Corporation. 2010-05. EDKII Platform Description File (DSC) Specification. Version 1. 22
- [29] Intel Corporation. 2011-06. EDKII Package Declaration File (DEC) Specification. Version 1. 23
- [30] 刘洪宇. EFI 接棒_传统 BIOS 将功成身退. 中国计算机报, 2008(7): 1-3
- [31] Intel Corporation. 2011-06. EDKII Module Information File (INF) Specification. Version 1. 23
- [32] 霍卫华. 基于 UEFI 的安全模块设计分析. 信息安全与通信保密, 2008(7): 1-30
- [33] Intel Corporation. 2006-10-12. EDK II Module Development Environment Library Test Infrastructure: 385-389

- [34] M. Tim Jones. 2005-06. Visualize function calls with Graphviz [EB/OL]. <http://www.ibm.com/developerworks/linux/library/l-graphviz>
- [35] Microsoft corporation. 2008. Visual C++ MSDN [EB/OL]. [http://msdn.microsoft.com/zh-cn/library/c63a9b7h\(VS.80\).aspx](http://msdn.microsoft.com/zh-cn/library/c63a9b7h(VS.80).aspx)
- [36] Microsoft Corporation. 2008. Visual C++ MSDN [EB/OL]. [http://msdn.microsoft.com/zh-cn/library/cc438637\(VS.71\).aspx](http://msdn.microsoft.com/zh-cn/library/cc438637(VS.71).aspx)
- [37] PC Online. 2011-11-11. 引领业界技术潮流 华硕图形化 UEFI BIOS 解析
- [38] Tool Interface Standards (TIS). 1999-08. Portable Formats Specification. Version 1. 1. Executable and Linkable Format (ELF)
- [39] 石浚菁. EFI 接口 BIOS 驱动体系的设计、实现与应用: [学士学位论文]. 南京: 南京航空航天大学图书馆, 2006
- [40] UEFI Forum. 2009-05. UEFI Specification. Version 2. 3 [EB/OL]. <http://www.uefi.org/specs/>
- [41] UEFI Forum. 2008-09. UEFI Specification Version 2. 2: 1387-1658 [EB/OL]. <http://www.uefi.org/specs/>
- [42] Vincent Zimmer. 2006. Beyond BIOS. Intel corporation: 17-32, 143-146
- [43] 张朝华. 基于 EFI/Tiano 的协处理器模型的设计与实现. 上海交通大学学报, 2007(3): 13-16
- [44] 胡藉. 面向下一代 PC 体系结构的主板 BIOS 研究与实现. 南京航空航天大学学报, 2005(8): 91-94
- [45] 张孝亭. Intel Xeon-E5 多处理器的 UEFI 固件驱动开发: [学士学位论文]. 南京: 南京邮电大学图书馆, 2013