

南京信息工程大学

硕士学位论文

基于UEFI技术的API性能分析与实现

姓名：倪兴荣

申请学位级别：硕士

专业：系统理论

指导教师：赵远东

20100501

摘 要

Unified Extensible Firmware Interface 是下一代计算机固件接口标准。UEFI 采用面向对象的方法进行设计, 用模块化的方法组织驱动及应用程序, 模块间以标准的协议进行通信。UEFI 旨在替代 BIOS, 成为能够利用高级语言以及现代软件工程方法的下一代固件接口。随着 UEFI BIOS 的进一步的推广, 越来越多的 OEM (厂商), 甚至个人, 都会在这个环境开发一些专用的软件。然而, 如何了解当前硬件平台的详细状况以及如何评估这些专用软件的执行性能, 目前没有现成的解决方案。

本文主要研究的是在 UEFI 规范下 BIOS 底层 API 的性能分析及其优化技术,并在 IA-32、X64 和 IA64 三种架构平台上进行分析并给出在 NT32 下的部分分析出结果, 利用了 EFI 技术在其内部的规范实现接口的调用, 性能的分析。

本文最终在 UEFI BIOS 的 IA-32 架构硬件平台上, 实现了性能分析模块对其他应用程序的分析, 并用工具将取得的结果以报告的形式打印出来。除此之外, 本文还给出了该项技术在 PEI、RUNTIME 等阶段和 IA-32 架构平台上的解决方案, 并给出实例。

关键词: UEFI, EDK, API, 固件

Abstract

Unified Extensible Firmware Interface is the next generation firmware interface standard. It is Object-Oriented and consists of drivers and applications that communicate with each other via standard protocols. UEFI is subject to replace legacy BIOS and becomes a firmware interface implemented with high level programming language and modern software engineering methodology.

This paper is the study of the UEFI specification BIOS low-level API under the Performance Analysis and optimization technology, and the IA-32, X64, and IA64 platforms, three kinds of structure analysis and the outcome of the use of EFI technology in their internal norms achieve the interface calls, performance analysis.

Eventually This UEFI BIOS for IA-32 architecture hardware platform, to achieve the performance analysis module for the analysis of other applications, and use tools to achieve the results to be displayed in the form of HTM pages, so that the result is more clear and intuitive. In addition, the paper also gives the technology in PEI, RUNTIME all stages and IA-32 architecture platform solutions.

Key words: UEFI, EDK, API, firmware

关键词汇及释义

UEFI	United Extensible Firmware Interface
HOB	Hand-Off Blocks
DXE	Driver Execution Environment
BDS	Boot Device Selection
VFR	Visual Forms Representation
HII	Human Interface Infrastructure Database
RVA	Relative Virtual Address
SMM	System Management Mode
EDK	EFI Development
EDKII	EFI Development Kit II
BIOS	Basic Input and Output System
SMBIOS	System Management BIOS
EFI	Extensible Firmware Interface
GUID	Globally Unique Identifier
IA-32	32-bit Intel architecture
IA-64	64-bit Intel architecture
PEI	Pre-EFI Initialization
PEIM	Pre-EFI Initialization Module
SEC	Security
IPL	Initial Program Load
DSC	Description file
CIS	Core Interfaces Specification
CSM	Compatibility Support Kit
Framework	Intel Platform Innovation Framework for EFI

论文独创性声明

本人声明所呈交的学位论文,是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外,论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名: 倪兴荣

签字日期 2010.06.10

关于论文使用授权的声明

本人授权南京信息工程大学拥有学位论文的部分使用权,即:学校有权按有关规定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,可以将学位论文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、汇编学位论文。

保密的学位论文在解密后也遵守此规定。

作者签名: 倪兴荣

签字日期 2010.06.10

导师 赵东

日期 2010.06.10

前言

基于UEFI^[1]技术的API性能分析设计与实现其主要信息来源于UEFI开源社区(www.tianocore.org)中,这个开源社区存在四个与UEFI BIOS相关的开源项目,分别为EDK(EFI Dev Kit)、EDKII、EFI Shell和EFI ToolKit。其中,EFI Shell是独立于EDK和EDKII之外的一个DXE阶段的应用环境。在该环境中,已经存在了一些基本的应用,比如进入退出目录、设置当前IP地址、编辑文本文件、复制移动文件等。但其中,却非常缺少一种对API的性能分析及其优化方案的支持功能。

EFI Shell是UEFI BIOS中一个非常强大的执行和调试环境。随着UEFI BIOS的进一步的推广,越来越多的OEM厂商,甚至个人,都会在这个环境开发一些专用的软件。然而,如何衡量和测评这些专用软件在UEFI BIOS实际硬件平台的执行性能,目前没有这方面的解决方案。同时,对于刚刚开始从事EFI Shell下应用程序开发的技术人员而言,如何快速的了解并掌握这项开发技术,大量的学习EFI Shell下自带的应用实例,显然是一个好的方法。那么,如何在EFI Shell下编写测试用例,使之能够完成对EFI BIOS内部模块的API进行分析,是一件挑战性的工作。

除此之外,由于EFI不仅仅可以方便地使用到服务器中,而且以后的发展趋势是可以嵌入到一些嵌入式平台的工程中,所以一旦支持嵌入式开发环境,就意味着EFI开发的库函数也要兼容在各种编译器编译出的可执行文件的有效性,也就是说,EFI开发的代码在各种编译器下都可以执行,增加代码的容错性等等。紧接着,本文还将提出如何在Linux环境中实现了EFI BIOS在实现阶段的调用方案,技术难度在于BIOS完成初始化之后将底层系统参数传递到操作系统后如何将这些参数提供给上次用户。本文所采用的方法是使用系统调用的方法实现了在操作系统中调用底层API的功能。

随着UEFI BIOS在世界范围内的逐步推广,对于UEFI BIOS下特定应用程序的需求增大和目前相对匮乏的情况存在必然的矛盾。UEFI BIOS下API功能的实现,可以从一定程度上提供UEFI环境下开发各种模块软件函数接口,因而选题具有一定的实用性。性能分析模块更是针对UEFI DXE阶段目前尚没有程序路径信息分析模块的情况,通过这样的研究,为其

他从事UEFI环境下应用程序开发者开发应用程序提供一种分析函数调用关系和函数执行时间的手段；此外，在UEFI BIOS执行环境中，通过NT32模拟环境实现对底层API性能分析的实例，更加直观的分析出UEFI BIOS底层功能函数性能。这三点使得选题具有一定的先进性。整个开发和设计过程，能够充分体现出在UEFI从事程序设计相对于传统BIOS环境下的优势。

本课题内容涉及以下几个方面：UEFI BIOS 背景知识的理解和熟悉，EDK（UEFI 开源开发框架）整体结构分析，在 PEI 阶段对字符串函数进行详细的分析得出结果，以及嵌入式系统下实时功能的设计和实现，性能分析功能的设计方案^[2]。

研究重点有两个。其一是，如何在 UEFI BIOS 这种新的应用环境中，对整个平台的底层 API 信息进行检测。这与之前存在的检测技术存在非常大的区别。UEFI BIOS 与传统的 BIOS 在整个系统的架构、信息存储机制、启动阶段划等方面截然不同。这就要求设计这样一款分析工具，需要对整个 UEFI BIOS 的完整的运行机制，有一个非常清晰的认识。

研究的第二个重点是，对 UEFI BIOS 在 RUNTIME 下直接运行的程序进行性能的分析，目的是在 BIOS 启动到 OS 后，在 OS 环境下对 BIOS 提供的 RUNTIME 服务进行分析检测。其一，UEFI BIOS 是一个新生的事物，在上面的扩展应用的实例还仅仅停留在能不能用上这个层次，没有直接可以借鉴的性能分析的资料；其二，直接在 OS 环境下对底层的 API 程序调用细节的难度是相当大的，需要对 OS 内核进行插入模块，使之能够在 BIOS 启动 OS 的过程中顺利的将底层的 API 传递到 OS 内核中供用户使用。这就需要对程序局部的细节进行比较透彻的研究，比如针对 IA-32、X64 和 IA-64 等特定架构平台上的函数的传参机制以及参数调用等。其三，如何在不影响或不破坏被测试程序或模块的整体结构的情况下，通过尽可能少的操作上加入自定义的测试代码，这又是一个比较有难度的问题。其四，由于编译完成的 UEFI BIOS 的固件代码是不包含函数名称信息的，存在的只是一些加载到内存中的指针地址。并且这些加载的地址，在不同的平台，甚至在同一平台，每次执行的地址都是完全不同的。如何让函数名称和实际执行的函数入口地址一一对应起来，这是一个必须仔细考虑的问题。

本文共分为五章。第三章着重从总体上对系统的总体架构情况进行阐述；结合 EFI 的框架的特殊性，搭建分析框架，并嵌入分析案例。具体来说，本论文的组织结构为：

第一章 绪论。本章首先阐述了 UEFI 背景概况和结构介绍。与传统 BIOS 的对比以及 UEFI 国内外概况和发展趋势，UEFI 的安全隐患和其局限性问题。

第二章 系统的架构实现与总体设计。首先，介绍了 Tiano 开发框架实现^[3]，包括环境

配置与在开发过程中总结出来的 DEBUG 调试技巧。然后，对 Tiano 开发框架的原理给出分析。

第三章 UEFI BIOS 在底层 API 性能分析设计实现。这一章中所做的工作是搭建一套测试框架，这个测试框架负责装载测试案例，之后又提出测试工具的概念，编译测试代码并分析出当前 API 功能上存在不足，结合原代码进行改进及优化，在性能上也给出分析代码、分析数据、Shell 下效果图。

第四章 主要实现在 Linux 下 RUNTIME 服务的实时调用，所做的工作是通过系统调用的方式实现内核和用户态的数据交换，获取 BIOS 传递到内核中的数据，编写出 shell 下的应用程序，获取底层时间服务。

第五章 课程总结与展望。

第一章 绪论

1.1UEFI 概况和技术特点

1.1.1UEFI 国内外概况和发展趋势

作为连接操作系统与硬件体系之间的桥梁,传统BIOS为PC的发展做出了重要贡献。BIOS是硬件与软件程序之间的一个转换器,或者说是接口(虽然它本身也只是一个程序),负责解决硬件的即时需求,并具体执行软件对硬件的操作要求。它负责操作系统执行前的初始化工作,包括检查系统配备和连接计算机内各种不同的硬件与操作系统等。作为由低级汇编语言写成的软件, BIOS以16位汇编代码、寄存器参数调用方式、静态链接,以及1MB以下内存固定编址的形式存在了很长一段时间。即使CPU经过了几番革新,但是加电启动的16位实模式却仍然保留了下来。尽管后来也发展出了“大实模式”,但后者只适用于系统固件本身。正是这一陈旧的运行方式,迫使Intel、AMD在开发新款CPU时,必须考虑加入导致系统性能大大降低的兼容模式。用户操作体验不佳、代码编写复杂等BIOS的缺点也导致设计者怨声载道。老旧的传统BIOS亟待一场新的革命, Intel已酝酿出一个革新方案——EFI(可扩展固件接口)计划。

2000年, Intel向业界展示BIOS的新一代接口程序EFI, 并在其安腾服务器平台上采用EFI技术。EFI是由Intel推出的一种在未来的电脑系统中替代BIOS的升级方案。传统的BIOS采用汇编语言编写, 面对BIOS的新需求明显力不从心。而新的EFI则更具优势, 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, 比BIOS更易于实现。另外, EFI驱动程序可以不由运行在CPU上的代码组成, 而是由EFI字节代码编写而成, 保证了在不同CPU架构上的兼容性。Intel将EFI定义为一个可扩展的、标准化的固件接口规范, 不同于传统BIOS的固定的、缺乏文档的、完全基于经验和晦涩约定的一个事实标准。

从核心来看, EFI很像一个被简化的操作系统, 介于硬件设备和高级操作系统之间。由于EFI内置了图像驱动功能, 能够提供一个高分辨率的彩色图形环境, 并且支持鼠标点击操

作,明显有别于传统BIOS单调的纯文本界面。与传统BIOS的另一显著不同点是,EFI使用全球最广泛的高级语言——C语言进行编写,摆脱了传统BIOS复杂的16位汇编语言代码编写方式。这意味着有更多工程师可参与EFI的开发工作,有利于平台创新快速发展。目前,EFI的应用已由服务器领域扩展至PC领域,苹果在其x86 PC机上已采用了EFI^[4]。与此同时,EFI技术向消费电子、家用设备领域的延伸也从未停止。例如,通过EFI技术,当计算机未进入操作系统前,就可接入互联网。

无疑,EFI在安腾服务器平台上的应用和它的前景,让更多人看到了EFI的魅力。2005年,在工业界达成共识的基础上,Intel将EFI规范交给了一个由微软、AMD、惠普等公司共同参与的工业联盟进行管理,并将实现该规范的核心代码开源于网站上。与此同时,EFI也正式更名为UEFI(统一可扩展固件接口)。UEFI联盟将负责开发、管理和推广UEFI规范。

UEFI联盟是由Intel、微软、惠普等厂商于2005年共同发起成立的国际组织。该组织约定期召开UEFI技术大会。目前联盟已有86家企业成员,这些企业成员共分为三级:推广者、贡献者和接受者,这三类成员在联盟中分别承担不同的权责和义务。UEFI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)和业界联络工作组(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。

目前,该联盟发布了两个最新规范——UEFI 2.3和新的PI(平台初始化规范)v1.2。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)提供的固件组件之间的互操作性。

在美国成功举办过两次UEFI技术大会后,2007年,UEFI技术大会首度移师中国南京。而该联盟常务执行副总裁魏东的宣讲,也让国内众多IT厂商更全面地了解了UEFI联盟及他们的职责。

从而,在世界范围内,掀起了一场轰轰烈烈的UEFI BIOS逐步取代传统BIOS的革命。

1.1.2 UEFI 对比与传统 BIOS 的优势

Basic Input/Output System 最初产生于 20 世纪 80 年代, 应用于 IBM 的 PC/XT 以及 PC/AT 系统。BIOS 被固化于主板上的 ROM 中, 如 EEPROM 等。PC 加电启动后首先运行 BIOS 代码, 由 BIOS 代码接管并负责 PC 硬件的识别, 检测以及初始化工作。BIOS 抽象出了不同 PC 平台的硬件组织, 使得应用程序能够运行于基于 X86 架构的多种 PC 之上。

由于硬件设备种类和功能的发展, 现代计算机系统中通常将 BIOS 代码分离在主板的 ROM 以及若干存在于硬件适配器的 Option Rom 中。主板 BIOS 包含基本的硬件识别, 检测代码, 并负责加载 Option Rom 中的特定硬件的驱动代码。这些 Option Rom 中的代码可以由硬件生产厂商自行开发。

由于传统 BIOS 采用了软件抽象的方法向应用程序提供了统一的访问系统资源的接口, 所以被广泛的采用。但是随着硬件的发展, 即便是使用 Option Rom 的方法, 传统 BIOS 也开始面临一系列问题。

第一, 传统 BIOS 定义了一组与 OS 无关的接口, 但它却不是平台独立的, 它依赖于中断来组织 BIOS 的软件结构, 也因此依赖于 X86 的中断模型。也就是说, 它的软件接口并不是完全硬件独立的。

第二, 传统 BIOS 运行于实模式 (Real Mode) 下, 寻址能力为 1MB。虽然理论上和技术上可以在实模式下拥有更大的寻址空间, 但这些技术往往采用特殊技巧, 使得系统的耦合度大幅增加。Option ROM 的地址空间也局限于 1MB, 大大的限制了 Option ROM 中驱动程序的扩展能力。1985 年 Intel 发布了 32 位 CPU, 而目前的 CPU 总线已到达 64 位, 传统 BIOS 的寻址能力对此造成了严重的浪费。

第三, 传统 BIOS 采用汇编语言, 软件工程实施与管理的效率较低。工程师往往采用大量的汇编技巧, 扩展性和可维护性较差。然而硬件的发展要求 BIOS 具有较高的扩展性和可维护性。

在此背景下, Intel 开发了 Extensible Firmware Interface 以替代 BIOS。新一代 BIOS 的体系规范统一可扩展固件接口 UEFI 已成为国际开放标准。UEFI 规范为操作系统与平台

固件的接口定义了一个新的模型，此接口由数据表组成，包括了平台相关信息，以及对操作系统和载入程序都有效的启动和实时服务请求。它们一起为启动操作系统和预导入应用程序的运行定义了先进的、完整的环境。UEFI 与传统的 BIOS 相比较而言，其最主要的优点在于具有规范化、模块化与可扩展的特点。UEFI BIOS 采用了 C 语言编写，能够在启动时实现丰富的功能扩展。它相当于一个简单的操作系统，在开机程序进入 DXE 阶段，使用者就可选择执行 UEFI Shell 作为用户交互接口，以在引导进入操作系统之前的阶段实现网络应用、信息检测、软硬件调试等扩展功能。但是，UEFI BIOS 离正式取代传统的 BIOS 仍有一段相当长的路要走。基于 DXE 阶段 UEFI Shell 开发的应用程序仍然局限于小规模群体中，如何推广这种发展前景巨大的底层应用开发环境，其意义是显而易见的。

1.1.1 UEFI 的安全隐患和其局限性

凡事都有两面性，UEFI 技术也不例外。它在为我们提供模块化、结构层次清楚、网络应用、高级语言开发、易于实现等优点的同时，也带来了一些安全隐患^[5]，比如：

1. UEFI 由于在 DXE 阶段就引入了完整的网络应用，比如 TCP、UDP、MTFTP 等支持，这一方面为我们提供了调试网络设备的便利，同时也为下一步的网络安全留下了隐患；
2. UEFI 良好的扩展性使其更多的接口保留出来，同时其核心代码部分开源使得黑客无需反汇编代码，仅通过阅读核心源代码的方法就有可能找出其中的漏洞和缺陷。
3. UEFI 由于采用高级语言开发，相比之前的汇编语言，降低了开发门槛，同时提供了丰富的调试接口，尤其是 DXE 阶段 UEFI Shell 这样的调试接口，非常有可能引起新一轮针对 BIOS SMM (System Management Mode) 的攻击^{[6][7][8][9]}。
4. UEFI 作为一种新型的技术，其复杂的阶段划分，每个阶段都有一个单独的控制核心，比如 PEI 与 DXE 阶段均由相应的分发器负责。对于简单计算机平台，尤其是对于嵌入式设备而言，这么多阶段与阶段之间的衔接必然会在启动速度上不如传统的 BIOS 或 Bootloader。

1.2 新一代固件架构及 UEFI 的结构分析

1.2.1 新一代固件架构

EFI 是操作系统与平台固件之间的软件接口。该规范定义的接口包括平台信息的数据表和启动时以及启动后的服务。启动服务包括对不同设备，总线文件服务上支持的文本和图形控制台以及运行的服务。EFI 还定义了一个小型的命令行处理程序，在不直接启动进入操作系统的时候，用户可以选择进入 EFI Shell 的命令行处理程序。

EFI 启动管理器被用来装在操作系统，而不再需要专门的启动装载机机制辅助。EFI 的扩展可以从连接在计算机上，任何不易丢失的存储设备中装载。Framework 是一种固件的架构，它是 EFI 固件接口的一种实现，用来完全替代传统的 BIOS^[10]。整个 Framework 是层次化和模块化的，并且由 C 语言来实现，结构图如图 1.1 所示：

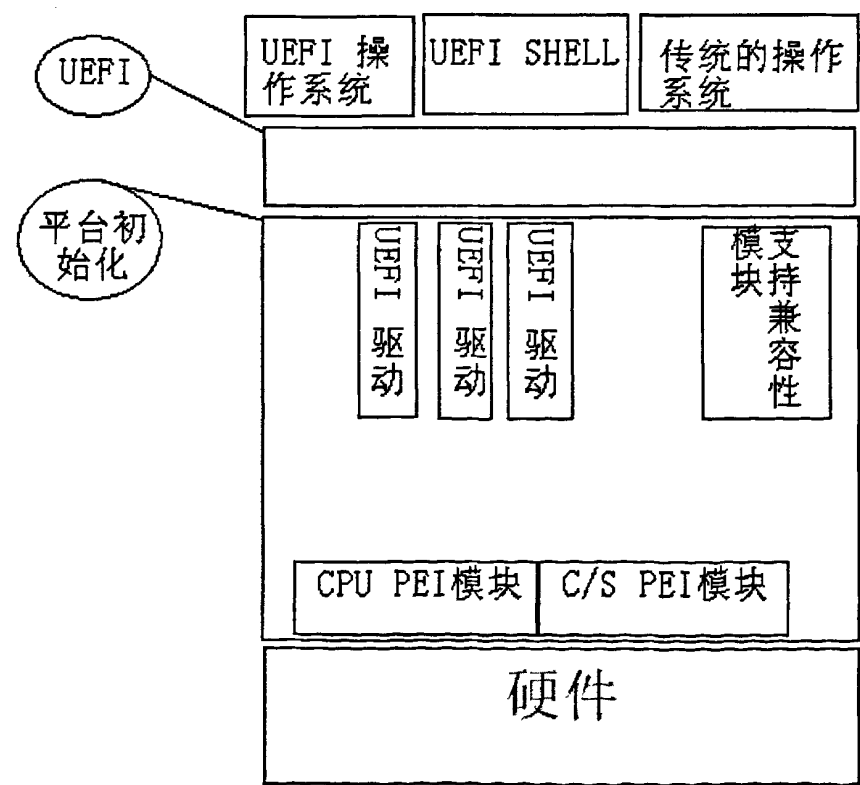


图 1.1 UEFI 的框架结构图

UEFI 并不直接对硬件进行操作,而是通过调用 SAL 和 PAL 操作硬件。UEFI 的引导服务(Boot Service)和运行时服务(Runtime Service)都调用 SAL 和 PAL 的相应功能。UEFI 还提供一些调试系统的工具软件,属于 UEFI 的 API 调用。总的来说,UEFI 在 Firmware (平台固件)模型里起到承上启下、屏蔽硬件层物理特性的作用。

UEFI 与其他模块的关系如图 1.2 所示,系统在经历了 PAL 和 SAL 模块后,会运行一些必须的服务,以便能将 EFI BIOS 完成的系统参数传递给 OS,这里有运行时服务和引导服务。同时也会有一些 EFI API 和 OS loader,在第四章的时候会介绍如何在 OS 底下分析这些 EFI API。

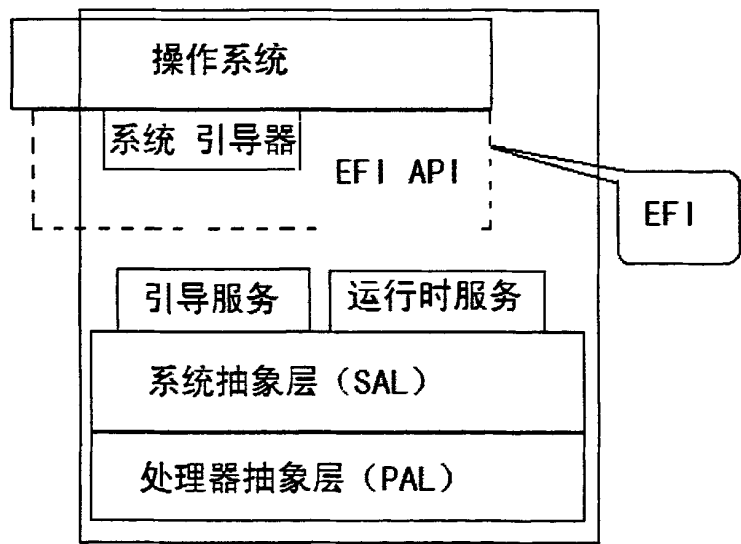


图 1.2 UEFI 与其他模块的关系

1.2.2UEFI 的阶段分析

不同于传统的 BIOS 实现, Tiano 基于现代软件体系设计的思想,按阶段来初始化平台,将操作系统的启动过程主要分为 4 个主要的阶段: SEC、PEI、DXE、和 BDS^[11],其运行机理如图 1.3 所示。

第一个阶段是安全阶段 (Security, SEC), 是上电后最先执行的步骤。SEC 支持检查系统执行最先的操作码,确保选择的平台固件映像没有被破坏。这个阶段通常需要硬件

支持，新一代 CPU 和芯片组支持这些功能。体系提供了扩展接口来支持可能在今后产品中被添加进的功能。

因为选择了高级语言 C 作为 UEFI 的开发一语言，所以这个阶段还有一个设计任务，就是找到一块初始化过的内存，使得 C 代码可以在给定的系统上实现执行。所以，接下来的 PEI 和 DXE 阶段的划分就是基于这个原则。

PrePEI 阶段，主要负责做尽可能少的工作以便寻找和初始化内存，很多芯片组和其他组件的初始化一直延续到驱动程序执行环境 DXE (Driver Execution Environment) 起来并运行后，才开始进行。最早的 PEI 阶段的代码倾向于用适合机器的汇编代码编写。一旦发现内存，PEI 阶段就准备状态信息来描述平台的资源图，并且将其初始化，然后跳到 DXE 阶段。PEI 到 DXE 阶段的转化是一个单向的过程，一旦为 DXE 阶段的初始化装入程序完成后，PEI 阶段代码将不可用，这时 DXE 成为一个设备齐全的操作环境。

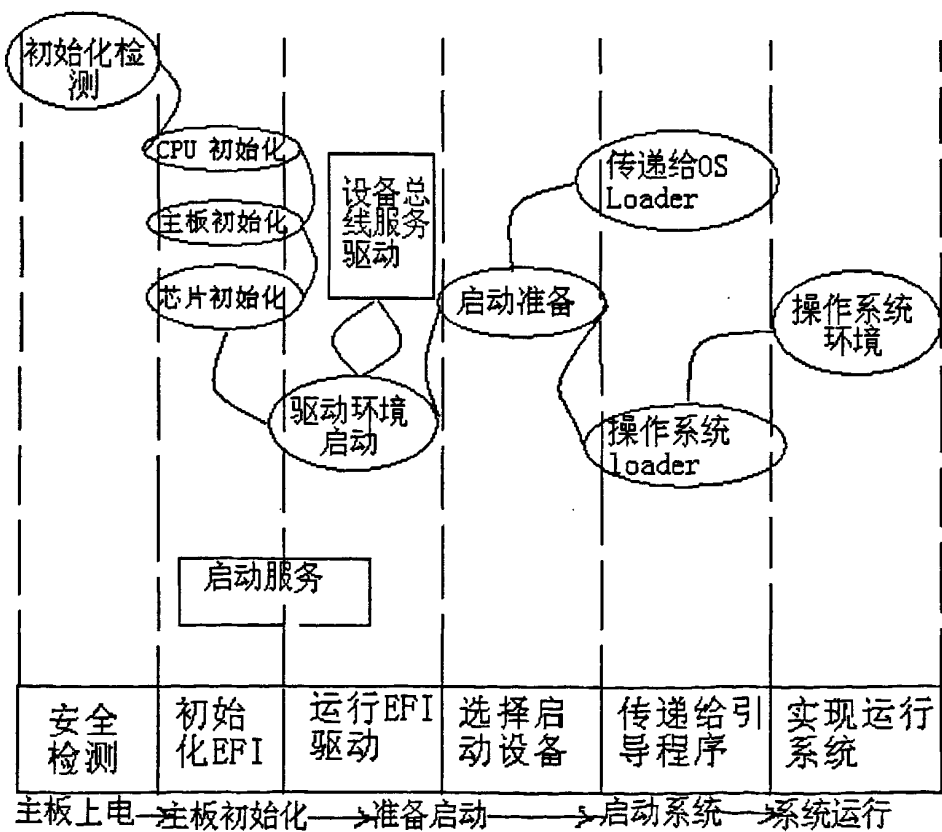


图 1.3 UEFI/Tiano 系统框架

。DXE 阶段是大多数的系统初始化执行的一个阶段。在 DXE 阶段有几个组件，包括 DXE 核、DXE 分派程序和一组 DXE 的驱动程序^[12]。DXE 核产生一组启动服务，即运行时服务和 DXE 服务。DXE 分派程序负责按照正确的顺序发现和运行 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，同时为系统服务、控制台设备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，并且提供启动一个操作系统所需的服务。

启动设备选择 BDS (Boot Device Selection) 阶段与 DXE 阶段一起工作，创建一个控制台，并且尝试从可用的启动设备来启动操作系统。BDS 是控制权交给操作系统之前的最后一个阶段。在 BDS 阶段，可以向最终用户展示用户界面，可以被 OEM 修改用来定制 DXE 适应的系统。

1.5 本章小结

本章对 UEFI BIOS 的目前国内外概况和发展趋势进行了简要的描述，明确提出了 UEFI 较传统 BIOS 的优缺点。接着，对 UEFI 的整体框架和启动过程阶段的划分进行了较为深入的分析。在此基础上，对本论文研究内容的选题、研究意义、先进性和论文的整体结构情况进行了论述。

第二章 Tiano 系统框架及实现原理

2.1Tiano 系统框架

Tiano 采用模块化设计，并基于现代软件体系设计思想，将 Tiano 框架流程主要划分为 SEC、PEI、DXE、BDS、TSL、RT 和 AL 等 7 个阶段，其运行机理如图 2.1 所示：

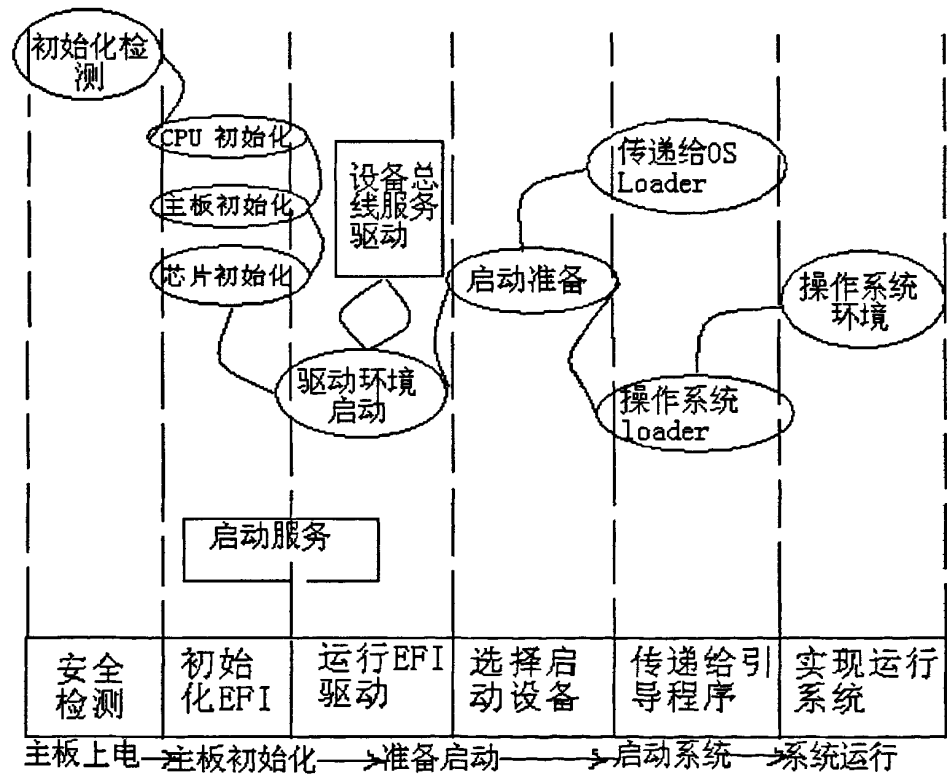


图 2.1 UEFI/Tiano 系统框架

SEC 阶段，让 UEFI 重新获得系统控制权。

2.2 Tiano 实现原理

2.2.1 开发总体框架介绍

EDKII (EFI Development Kit) 是一个开源的 EFI BIOS 的发布框架, 其中包含一系列的开发示例和大量基本的底层库函数, 开源部分涉及 NT32、Unix 和 Duet 三种平台架构。除了开源核心代码, EDK 同时也是一个开发、调试和测试 EFI 驱动程序的综合的平台。

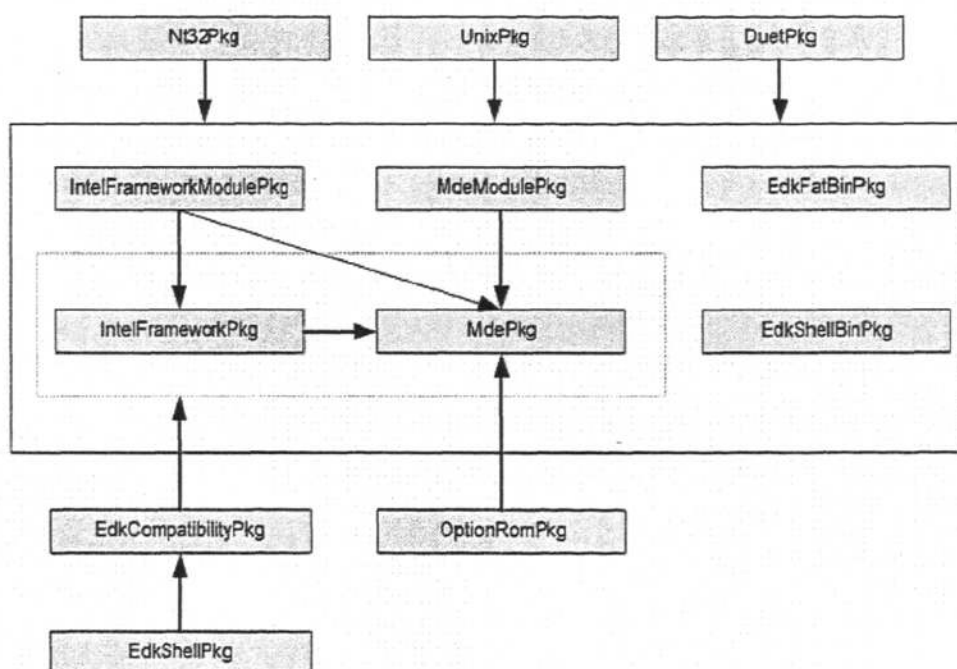


图 2.2 EDKII 主要模块依赖关系图

近年来, 针对 EFI 客户的反馈信息, Intel 又在 EDK 的基础上进行了功能扩展, 出现了一个新的功能更为强大的开发平台 EDKII。EDKII 在开发库、类机制以及 PCD 机制方面进行了扩展, 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的模块。

本文所作的研究就是在 UEFI BIOS 最新的开发平台架构 EDKII 上进行的, 其各模块之间存在紧密的依赖关系, 比如 MdeModulePkg 使用的协议的原型在 MdePkg 中定义, 因此, MdeModulePkg 依赖于 MdePkg。EDKII 主要模块的依赖关系如图 2.2 所示。

从依赖关系图中可以看出,要编译 OptionRomPkg,只要有 MdePkg 和 BaseTools 即可;要编译 EdkCompatibiliryPkg 只要有 MdePkg、IntelFrameworkPkg 和 BaseTools 即可;要编译 EdkShellPkg,则 EdkCompatibilityPkg 和 BaseTools 是必须的;要编译 NT32、Unix、Duet 模拟器平台,必须依赖 MdePkg、MdeModulePkg、IntelFrameworkPkg、IntelFrameworkModulePkg、EdkFatBinPkg、EdkShellBinPkg 和 BaseTools。

EDKII 主要模块的功能分析如下:

BaseTools: 提供二进制编译工具集和编译环境配置文件。

Conf: 作为产生编译环境信息、编译器参数和编译目标定义的保存路径,在配置操作完成后,该路径下会产生三个配置文件。

MdePkg: 包含 EDKII 各个平台的基本的底层库函数、协议和工业标准, IA-32 X64 IA-64 等各种平台架构的 UEFI BIOS 都可以在各自开发模块中引用该文件下的相应库函数,极大的简化了开发的难度。

MdeModulePkg: 提供一组跨平台的模块,包含 MdePkg 底层库的应用模块示例。

EdkShellPkg: 提供了 EFI Shell 下应用程序的开发环境,同样也是一个跨平台的开发环境。

EdkFatBinPkg: 提供了针对不同 CPU 架构的原始的 FAT 驱动。由于 FAT 的版权问题,所以这里只提供了 .efi 的执行文件,没有源代码。

Nt32Pkg: 在 Microsoft 操作系统下一个可加载 32 位模拟器,提供 UEFI Runtime 的支持环境。

UnixPkg: 在 Linux 操作系统下可加载运行的 32 位的模拟器,提供 UEFI Runtime 的支持环境。

DuetPkg: 提供基于传统 BIOS 的 Runtime 环境的支持库,可以对传统的 BIOS 进行改造,在实际的硬件平台上构造出实际的 UEFI BIOS 工作环境。

OptionRomPkg: 包含针对不同 CPU 架构的编译 PCI 兼容镜像的实例程序。

EdkCompatibilityPkg: 提供针对传统 BIOS 和 EDK 定义的库和协议的兼容性支持,保证 EDKII 对 EDK 的全功能支持。

以上每一个文件包都有类似的结构，例如 MdePkg 有如下的子文件夹：

- Include\ —公共文件头文件；
- Ia32\ —支持 IA-32 架构的内部专有头文件；
- X64\ —支持 X64 架构的内部专有头文件；
- Ipf\ —支持 IA-64 架构的内部专有头文件；
- Ebc\ —支持 EBC 架构的内部专有头文件；
- Uefi\ —支持 Uefi2.3 规范的内部专有头文件；
- Pi\ —支持 PI1.2 规范的内部专有头文件；
- Protocol\ —包含各种协议规范的专有头文件；
- Ppi\ —包含各种 PPIs 规范的专有头文件；
- Guid\ —包含各种 GUIDs 规范的专有头文件；
- IndustryStandard\ —工业规范标准公共头文件
- Library\ —包含 MDE 库文件头文件
- Library\ —MDE 库原型

2.2.2 工具安装

1) Microsoft Windows 2000 or Microsoft Windows XP32 操作系统

2) 用来作为 Intel 32 位体系结构平台开发的：

- Microsoft Visual Studio .NET 2003 Enterprise (7.1)

3) 用作 Intel 安腾处理器开发的：

- Microsoft Windows Server 2003 Driver Development Kit (Driver Developer Kit), build 3790

4) ITP 工具：

ITP 全称(In Target Probe)，Linux 下我们用 GDB 或者内核调试工具 KGDB 进行应用程序或内核程序的调试，EFI 下我们使用 Intel 公司生产的 ITP 工具进行调试。

2.2.2.1 EDK 下载及配置

- 1) 通过 TortoiseSVN 客户端在本地更新到指定的 EDKII 开发版本。
- 2) 进入 C:\EDKII\EdkShellPkg 开发路径下, 修改 EdkShellPkg.dsc 配置文件, 将设定宏路径 “DEFINE EDK_SHELL_DIR”, 添加开发所用到的库文件, 比如如果要增加对图形界面语言 VFR 的支持, 就要添加在 [Libraries] 中添加:
“EdkCompatibilityPkg/Foundation/Library/Dxe/EfiIfrSupportLib/EfiIfrSupportLib.inf” 然后在 [Components] 中添加所开发模块的索引文件, 最后根据具体情况在 [BuildOptions] 部分修改编译器对工程的编译选项。
- 3) 准备工作完成后, 就可以开始编译整个工程了, 进入 Visual Studio 2005 command Prompt, 进入开发路径, 运行 edksetup 自动设置默认的开发配置; 然后输入:
“build -a IA32 -a X64 -a IPF -p EdkShellPkg\EdkShellPkg.dsc”。
- 4) 如果程序没有错误, 就得到能够在 EFI Shell 下运行的 .efi 的执行程序了。

2.2.2.2 DEBUG 调试方法

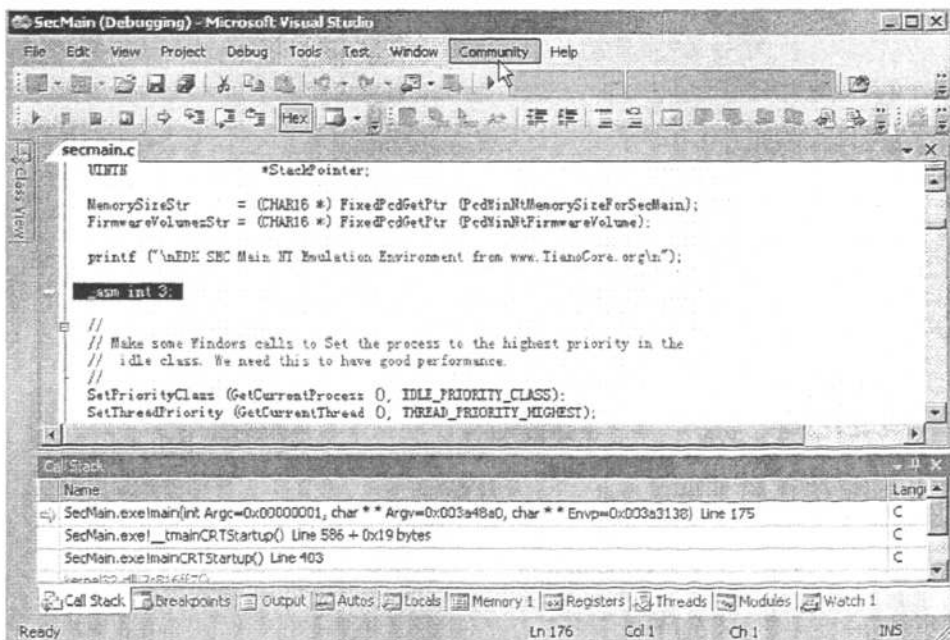


图 2.3 NT32 模拟器调试环境

在程序开发过程中，调试是至关重要的一个方面。就 EDKII 而言，有两种 DEBUG 的方法和技巧。

第一种方法比较容易掌握，针对 IA-32 的程序，可以直接在 NT32 模拟器中通过 Microsoft Visual Studio 2005 Enterprise 进行。

首先，在需要 debug 的代码中插入”_asm int 3;”。_asm 表示汇编指令，int 3 表示一个中断号，然后，重新编译代码，并在 NT32 模拟器中运行。当程序执行到需要调试的地方时，会触发一个应用程序的异常，选择“Cancel”和“New Instance of Visual Studio 2005”选项，这时候就进入了 DEBUG 环境。如图 2.3 所示。

第二种方法相对而言较难掌握，是通过 Intel ITP 工具直接对目标机的 CPU 进行硬件 Debug。这种调试方法非常通用，可以对 IA-32、X64 和 IA-64 架构的硬件平台进行单步跟踪、查看内部寄存器、分析硬件端口信息等高级操作。

具体方法是在需要 DEBUG 的程序中，加入一段死循环代码，比如：

```
volatile unsigned int Index;  
for (Index = 0; Index == 0;);
```

程序执行到这个地方会陷入死循环，这时候打开 ITP 硬件调试工具，输入 halt 命令，将程序暂停住；然后输入 loadthis 指令，根据 debug 文件的路径信息加载本地盘符的源代码。这里一般要求在本地编译，如果远程调试，要把远程编译目录映射成当前计算机的盘符。

由于临时变量一般存在 eax 中，所以，当前这个死循环一直在判断 eax 寄存器的数值。这时，在 ITP 工具命令行中输入”eax=1”，就能够跳出该循环，进入了需要调试的程序。查看当前内存内容的调试现场截图，如图 2.4 所示。

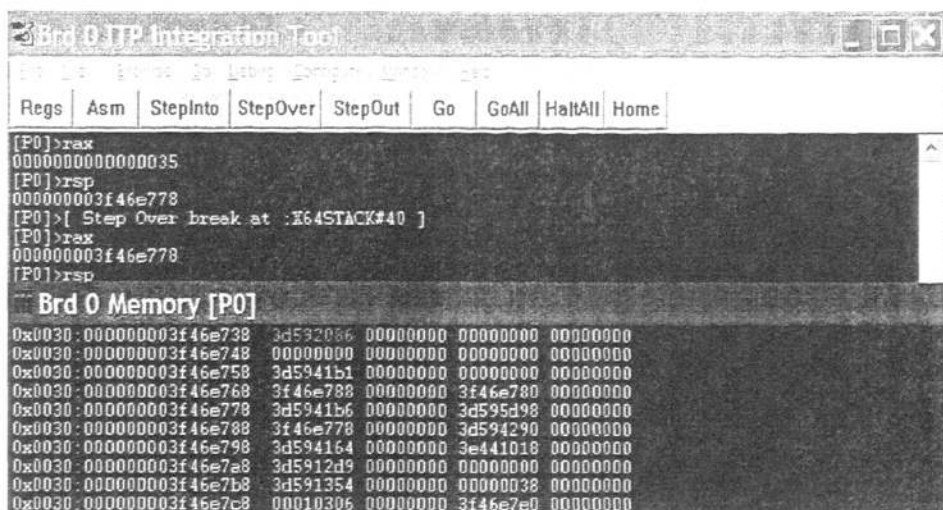


图 2.4 Intel ITP 调试现场截图

2.2.3 系统运行环境

首先，在 X64 架构的目标机中，通过 Intel ITP 工具将特定版本的 EFI BIOS 的 Image 烧写到 Flash 中；重启目标机，进入 EFI Shell 命令行下，转化到工具所在目录，直接运行开发程序；通过串口或目标机屏幕观察记录程序运行情况；如果程序发生异常，通过 ITP 硬件工具进行在线调试。

2.2.4 Hello world 程序在 EDK 下的实现

2.2.4.1 指定程序平台

这个程序主要让大家知道如何在 EDK 下实现一个应用程序的编译^[19]，通过这个例子，可以了解 EDK 下对应用程序的源程序是如何处理的。

首先我们必须选择这个应用程序将跑在一个什么架构的平台上面，为了演示方便，我们选用 IA32 模拟平台也就是 NT32，这是一个用 VS2005 编译器生成的一个模拟的 32 位的 EFI

BIOS运行环境。

在C:\EDK\Sample\Platform\Nt32\Build\Nt32.dsc 的[Libraries.Platform]部分 添加
[Libraries.Platform]Other\Maintained\Application\Shell\Library\EfiShellLib.inf
这是将shell运行环境加载到这个平台里面，也就是为helloworld程序提供一个运行的场所。其次将Other\Maintained\Application\Shell\HelloWorld\HelloWorld.inf

FV=NULL 加到[Components]部分，注意应用程序的位置，在dsc文件中限制了程序模块的位置。在这里需要解释的是HelloWorld.inf文件 这个文件相当于一个Makefile文件，但不等同于Makefile。这个HelloWorld.inf定义了程序的入口函数，GUID，源代码、支持的编译器等等。

2.2.4.2 编译 Hello world 程序

在程序完成之后，在shell下输入命令：

```
Cd C:\EDK\Sample\Platform\Nt32\Build
```

设置环境变量路径：

```
C:\EDK\Sample\Platform\Nt32\Build>Set EDK_SOURCE=C:\EDK
```

```
Build:
```

```
C:\EDK\Sample\Platform\Nt32\Build>nmake
```

程序的二进制代码将出现在下列路径，根据机器的路径而定：

```
C:\EDK\Sample\Platform\Nt32\Build\IA32\HelloWorld.efi
```

2.2.4.3 执行 Hello world 程序

编译成功后在C:\EDK\Sample\Platform\Nt32Build目录中就会看到一个HelloWorld.efi的文件，这就是一个在32位平台上的一个二进制执行代码，运行的命令很简单，首先要将这个模拟器Secmain.exe打开，运行进入Shell不用加任何参数，在C:\EDK\Sample\Platform\Nt32Build目录中直接打HelloWorld.efi，按回车即可，如图2.5

所示:

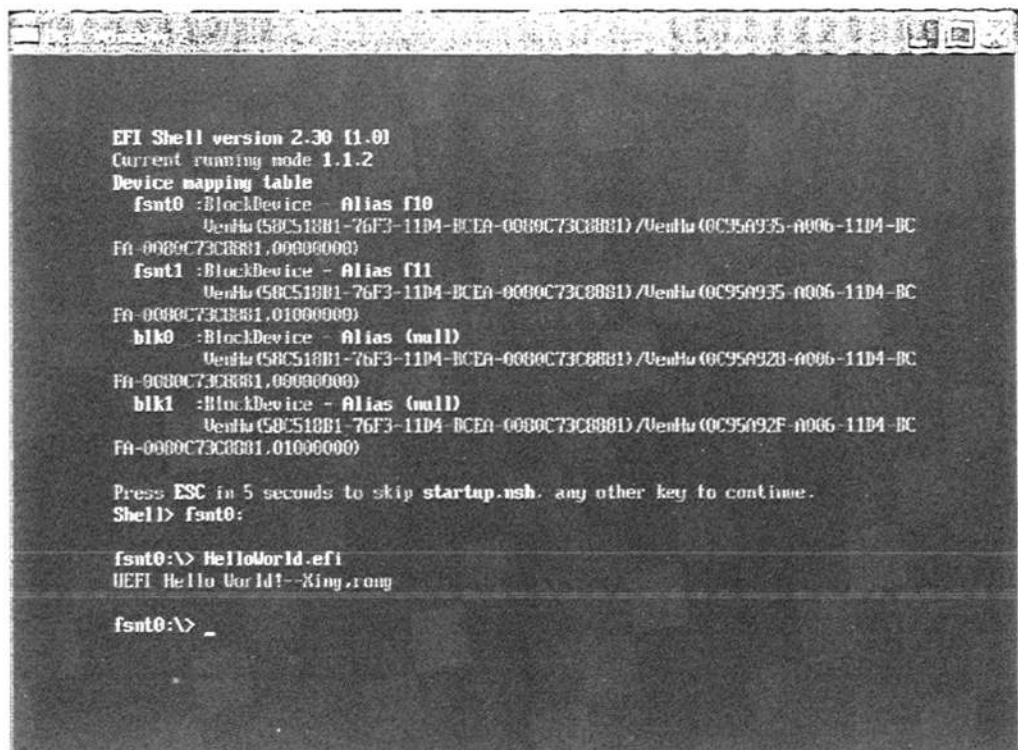


图 2.5 Hello world 现场截图

2.2.4.4 Hello world 程序源代码分析

Hello World在很多新的语言里面都会出现,在UEFI下Hello World的实现流程如图2.6所示:

```

33  EFI_STATUS
34  EFIAPI
35  UefiMain (
36      IN EFI_HANDLE      ImageHandle,
37      IN EFI_SYSTEM_TABLE *SystemTable
38  )
39  {
40      UINT32 Index;
41
42      Index = 0;
43
44      //
45      // Three PCD type (FeatureFlag, UINT32 and String) are used as the sample.
46      //
47      if (FeaturePcdGet (PcdHelloWorldPrintEnable)) {
48          for (Index = 0; Index < PcdGet32 (PcdHelloWorldPrintTimes); Index++) {
49              //
50              // Use UefiLib Print API to print string to UEFI console
51              //
52              Print ((CHAR16*)PcdGetPtr (PcdHelloWorldPrintString));
53          }
54      }
55
56      return EFI_SUCCESS;
57  }

```

图 2.6 Hello world 源代码

这是开始UefiMain(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)是应用程序的入口，首先定义一个32位的整形变量，并赋初值为0，这里使用一个for循环，其实没什么特殊意义因为首先这个PcdHelloWorldPrintEnable在platform里面永远是true的一个GUID，相当于C语言里面定义的一个宏，EFI里面喜欢将一个需要用到的数据初值定义在Platform的MdeModulePkg.dec文件里面，

原因是在EDK下用到string必须用PCD调用的方式来实现，EDK毕竟是属于底层的系统，和我们在用VC++有一定的区别，没有库可以调用，如图2.7所示：

```
1 //
2 // HelloWorldStrings.uni
3 //
4 // String resources for HelloWorld application
5 //
6 /=#
7 #langdef eng "English"
8 #langdef fra "Français"
9 #string STR_HELLOWORLD #language eng "Hello, world!\n"
10 #language fra "Hello, world!\n"
11 #string STR_GOODBYE #language eng "Goodbye, cruel world!\n"
12 #language fra "Goodbye, cruel world!\n"
13 Composing an EFI Shell Application
14 Version 0.91 June 27, 2005 22
15 #string STR_SHELLENV_GNC_COMMAND_NOT_SUPPORT
16 #language eng "%hs: This command is not supported in the \n"
17 "underlying EFI version, required EFI version \n"
18 "should be %hd.%hd or above\n\n"
19 #language fra "%hs: This command is not supported in the \n"
20 "underlying EFI version, required EFI
21
```

图 2.7 Hello world 依赖代码

也可以在 MdeModulePkg.dsc 里面将这个初值修改, 如果 MdeModulePkg.dsc 未做修改, 那么 HelloWorld.c 文件将用 MdeModulePkg.dec 的初值。这样无论是驱动还是应用程序想用的时候无须再定义, 只需要用 PCD 里面的函数调用就可以了, 其次 for 循环的 PcdHelloWorldPrintTimes 在 MdeModulePkg.dec 文件里面被定义为 1, 也就是说这个循环只运行一次。如果你想运行多次, 你需要在 MdeModulePkg.dec 文件里面将这个 PCD 的值改大, 最后 Print ((CHAR16*)PcdGetPtr (PcdHelloWorldPrintString)); 语句中首先 PcdGetPtr 函数获得 PcdHelloWorldPrintString 的值, 这个值也同样被定义在 MdeModulePkg.dec 中, 得到这个值后将它强制转换成 16 位的 char 型输出。这里还要说明一点的是这个 HelloWorld 所用到的 PCD 不一定必须放在 MdeModulePkg.dec 这个文件中, 也可以放在 Nt32Pkg.dec 中。

2.3 本章小节

本章首先对UEFI BIOS最新推出的开发框架EDKII进行了详细的分析。接着,根据实际开发过程,详尽地介绍在EDKII上进行软件开发的流程和方法,尤其是对程序设计过程,并结合Hello world入门程序讲解EDK下是如何编译出应用程序的。这样结合框架,实例分析,可以加深对Tiano及EDK的了解。

第三章 分析系统的总体设计及结构

本章将主要讨论UEFIBIOS下底层API性能分析的实现方法。针对新型的UEFI BIOS 模块功能机制，上一章节介绍了一些应用程序的编写，根据UEFI的工作原理工作。采用在EDK下嵌入一个模块，这个模块负责分析这些API的性能，根据这些函数提供的功能实现其各个功能和性能上的分析。

3.1 EDK 下的 MdePkg 模块的介绍

MdePkg全称Module Development Environment Package. 它是EDK II (EDK 2)可以调用的API库的集合，这些库在BIOS的启动的过程中提供一些服务，同时BIOS在进入系统后也可以提供Runtime调用的服务。启动BIOS时候提供BootService服务。这个功能模块向BIOS提供一些基本的函数功能，同时OS厂商也可以利用这个模块进行内核模块的开发。

3.1 MDE 里面的类库介绍

3.1.1 MDE 里面的类库介绍

下面这张表格如下图 3.1 主要介绍了 MDE 底层主要的一些 API 的类库，在这些类库下有很多 API,例如：Base Library 有很多 Stringcopy, Stringcmp 等等基层的 API,BIOS 在进入操作系统后可以调用这些 API，基于这些 API，用户可以开发一些 UEFI 相关的一些应用软件或工具。

名称	描述
Base Library	提供的字符串函数，链表函数，数学函数，同步功能，和

名称	描述
	CPU 架构的具体职能
Base Memory Library	提供复制内存, 内存清零, 和 GUID 的功能。
Cache Maintenance Library	提供以维持指令和数据高速缓存的服务。
CPU Library	提供 CPU 架构功能函数。
Debug Library	提供把调试消息发送到调试输出设备的服务。
Device Path Library	提供库函数来构造和解析 UEFI 的设备路径。
DXE Core Entry Point Library	DXE core 的模块入口。
DXE Services Library	提供简化 DXE 驱动的实现的功能。这些功能可以帮助获取 FFS 文件中数据
DXE Services Table Library	提供 DXE 服务表的返回指针的服务。只适用于 DXE 模块类型。
ExtractGuidedSection Library	该库提供生成 GUID 数据功能的服务。
HOB Library	提供构造很擦除内存 HOB 列表服务, 创造和解析滚刀。仅用于 PEI 和 DXE 模块。
I/O Library	提供访问 I / O 端口和将 MMIO 寄存器服务。
Memory Allocation Library	提供各种类型的存储器的内存分配区的服务。
PAL Library	提供 PAL 调用服务。
PCD Library	提供获取和设置平台配置数据的服务。
PCI CF8 Library	提供通过 I / O 端口 0xCF8 和 0xCFC 进入 PCI 参数空间的服务。

名称	描述
PCI Express Library	提供通过 MMIO PCI Express 窗口进入 PCI 参数空间的服务
PCI Library	提供来访问 PCI 配置空间的服务。
PCI Segment Library	提供通过一个具有多个 PCI 段数的平台进入 PCI 参数空间的服务。
PE/COFF Entry Point Library	提供从 PE/COFF 映像获取 PE / COFF 入口点指针的服务。
PE/COFF Loader Library	提供装载和重分配一个 PE/COFF 映像的服务
PEI Core Entry Point Library	函数库的入口模块。
PEI Services Library	实现所有 PEI 服务的功能的一个库。
PEI Services Table Pointer Library	提供获取 PEI 服务表指针的服务。
PEIM Entry Point Library	PEIM 模块入口点库。
Performance Library	提供记录执行时间和恢复时间的服务。
Post Code Library	提供将错误程序信息发送到 POST 卡上的服务。
Print Library	提供将格式化字符打印至串缓冲区。并且支持所有的 Unicode 和 ASCII 字符串格式。
Register Table Library (TBD)	建设提供服务和处理登记表, 其中可能包含的 I / O, 将 MMIO, PCI 配置, 内存和海洋科学研究活动。
Report Status Code Library	提供记录程序运行的状态的服务
Resource Publication	提供检测系统资源的服务。

名称	描述
Library	
SAL Library	提供可以调用系统抽象层 SAL 接口的服务
Serial Port Library	提供了三种常见的串行 I / O 端口功能的服务
SMBUS Library	提供库函数来访问设备。此类函数库必须在 SMBUS 控制器手测中被说明。
Synchronization Library	提供同步功能。
Timer Library	提供校准延迟和性能计数器的服务。
UEFI Application Entry Point Library	UEFI 应用重新入口库的模块。
UEFI Boot Services Table Library	提供获取 EFI 启动服务的返回指针的服务，只适用于 DXE 阶段和 UEFI 模块类型。
UEFI Decompress Library	提供解压缩一个使用 UEFI 的解压缩算法的缓冲区的服务。
UEFI Driver Entry Point Library	UEFI 驱动、DXE 驱动、DXE Runtime 驱动和 DXE SMM 驱动的入口程序模块。
UEFI Library	实现统一 UEFI 操作功能的库函数。只适用于 DXE 及 UEFI 模块类型。
UEFI Runtime Library	实现 UEFI 的运行时服务的功能。只适用于 DXE 及 UEFI 模块类型。
UEFI Runtime Services Table Library	提供返回 EFI 运行时服务表指针的服务。只适用于 DXE 及 UEFI 模块类型。
UEFI SCSI Library	实现提交 SCSI 命令功能。

名称	描述
UEFI USB Library	实现大部分 USB 的 API 支持的 USB 1.1 规范和在 USB 1.1 定义的一些标准。

图 3.1 MdeLib 库列举

3.1.2 单元库介绍

一个单元库（library instance）包括一个或多个单元库类。单元库的实现可以随意占用一个或多个其他库类，也可以选择产生的构造或析构函数。另一个目的是为一个特殊系列的模块设计了一个链接。这样可以将他们联系在一起。如果库实例被声明为一个，那么可以与任何类型的模块连接。

软件工程师不可能要用一个函数才去开发这个函数，事先是需要对这个函数库进行分析，并开发出一套函数库来为上层提供服务。在 MDE 模块中主要利用 PCD 入口来实现函数间的调用。也就是说没一个 API 的调用时通过对应的 PCD 值来获取参数的。

3.2 系统的总体设计及结构

3.2.1 分析模块简介

该系统支持 IA32, X64, IA64 系统架构的编译器，而且它可以在系统起来的六个阶段进行测试。例如“BASE”模块在 PEI, DXE, SMM, Runtime, PEI Core and Dxe Core 阶段都可以分析。就那 PEI 阶段为例。这个系统结构可以概括为下面几点：

首先是“Proxy Driver”，它是把一个单元库封装到一个独立的模块中。这个模块包括了一些待测试的 API，没有“Driver”的意义，只是提供一个临时 API 的活动场所。他的是实现可以参考 EFI 下如何实现 Driver 的相关文档^[14]。

然后执行这个“Proxy Driver”的入口函数去插入这个库的接口函数到这个模块中从

而让测试程序对单元库进行测试。如下图。

接着在这个模块中执行测试程序并得到测试结果。整个测试流程如下图 3.2 所示：

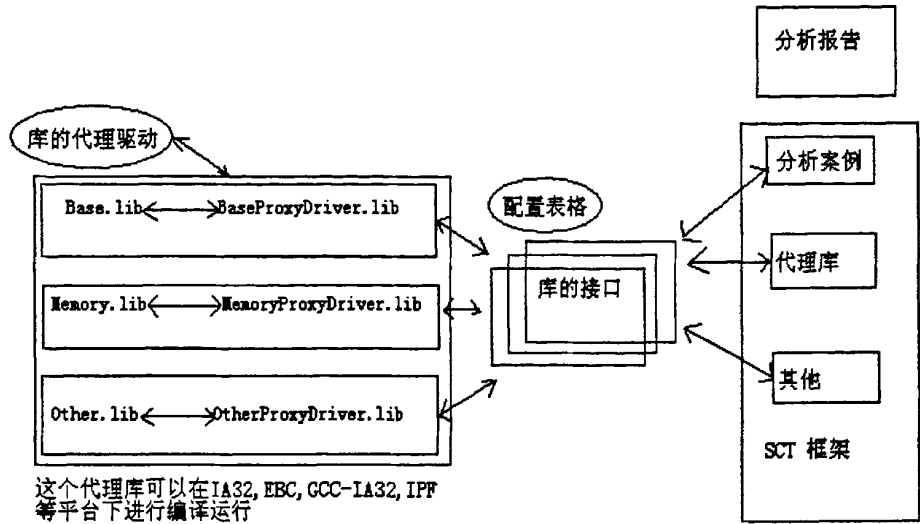


图 3.2 分析框架原理图 1

代理驱动将 MDE 中的单元库抽象出来，通过测试代码获取抽象出来的地址，这两个驱动相当于建立两个场所，提供分析案例活动的空间，分析程序获取 Lib Proxy Driver 转换过来的 Interface，调用这个 API 并对它输入相应的参数，从而获取函数的返回值，跟 MDElib 中相应的函数功能实现，判断该程序是否完成相应的功能，是否有功能或性能上面的不足。通过黑盒和白盒等测试思想分析出其性能。

- 1) GenTestPkg.bat 文件是用来生成输出二进制代码的测试包，这个和架构不一样。
- 2) Cleanall.bat 文件是用来删除代理驱动和测试案例的软连接，同时删除测试源码包。
- 3) MdeTestPkg.spd 文件是说明文档。
- 4) MdeTestPkg_*.fpd 文件是描述这个测试源码包的，它概括了测试的架构和测试的案例支持的驱动和库连接，同时也标示一些特殊的案例。

5) RunNt32.bat 文件是一个执行 SCT 的脚本。脚本执行的前提是 NT32 模拟环境要先编译出来。

3.2.2 分析模块函数框架说明

下面将主要介绍 EDK II 开发测试框架的一些必须的单元库，提出整个测试库的测试模型，并结合实例分析 API 的功能和性能分析。如下图 3.3 所示：

名称	描述
Debug Test Library	这个函数是为了方便 debug，可以用这个函数设定的一个断点和跳转到保存的返回点，当程序死掉停住的时候可以用它跟踪调查 CPU 以便找到程序出错的位置。
Pei BB Proxy Main Library	Pei 阶段 proxy driver 的模块入口函数。
Pei BB Test Case Main Library	Pei 阶段的测试程序的模块入口地址。
Pei Core BB Proxy Main Library	Pei Core 阶段的测试程序入口模块代理驱动。
Pei Core BB Test Case Main Library	Pei Core 阶段的测试程序入口模块。
Pei Standard Test Library	提供记录在 Pei Core 和 Pei 阶段的测试结果信息。

图 3.3 分析框架的组成

在 PEI 阶段的所以模块叫 PEIM，英文全称是 Pre EFI Interface Module，这是模块是由三类驱动组成，第一种驱动是负责插如点调试用的，由于在 BIOS 启动阶段 CPU 执行任务的时候是单任务模式的，一旦进入死循环或死锁状态，程序便无法继续执行，同时程序员也无法介入调试，所以在这个模块中必须嵌入一个调试模块测试模块（Debug Test

Library)；有了这个模块，程序员便可以在这个程序死循环前打上标记，用 ITP 工具单步跟踪找到程序死锁的原因；第二种类型的模块分为两种，一种是 Pei Core BB Proxy Main Library，另一种是 Pei BB Proxy Main Library，Pei Core BB Proxy Main Library 这个驱动是为了加载一些特殊函数接口的驱动，为了能让大部分函数接口可以被传递给下一个阶段的模块使用。Pei Core BB Proxy Main Library 为 Pei BB Proxy Main Library 准备好一些基本的函数接口，然后 Pei BB Proxy Main Library 再去将大部分的 MDE Library 函数接口装载在这个驱动中，以便调用。第三类模块是 Pei Core BB Test Case Main Library 和 Pei BB Test Case Main Library 组成，其作用类似于 Pei Core BB Proxy Main Library 和 Pei BB Proxy Main Library，Pei Core BB Test Case Main Library 将一些必须的分析案例装载到驱动中后，Pei BB Test Case Main Library 就开始装载测试代码，以便在其它阶段使用。

3.2.3 Debug 调试模块

首先我们来看一下测试模块 (Debug Test Library) ^[15]是如何实现的；在 EDK II 开发测试环境中，调试测试库通过提供所定的返回指针返回 (ASSERT)。其目的提供一个在不同阶段，其功能一样的调试函数，也就是说这个调试函数在 EFIBIOS 启动过程中的任何一个阶段均可以对不同的模块通过 Set_return_point 设置程序返回点。例如 Pei 阶段，Dxe 阶段和 Runtime 阶段。这是更高层次的对 SetJump 和 LongJump 服务进行封装。函数返回点的实现是通过用宏调用 SetJump 去保存的 CPU 的信息从而得知共享内存中的数据。主要是通过设置 SetJumpFlag 参数来实现的，如果 SetJumpFlag 参数等于 0，意思是第一次调用 SetJump 成功；如果 SetJumpFlag 参数等于 1，说明参数从 JumpToReturnPoint 返回；如果 SetJumpFlag 参数等于 2，代表找不到共享内存中的参数数据。函数结构申明如图 3.4 所示：

```

54  VOID
55  EFIAPI
56  JumpToReturnPoint (
57      VOID
58  )
59  {
60      BASE_LIBRARY_JUMP_BUFFER      *DebugTestLibJumpBuffer;
61      BASE_LIBRARY_JUMP_BUFFER      *JumpBufferWithZero;

```

图 3.4 JumpToReturnPoint 原函数

这个相当于在系统中触发一次中断，CPU 将从用户模式切换到系统模式，这个时候 CPU 要做的事情就是保存现场，接着去中断向量表去找这个触发中断号所对应的事件的功能差不多。在 EDK 里面用 SET_RETURN_POINT 保存在一个已知的共享内存的 SetJumpFlag 的变量值。接着由 JumpToReturnPoint 函数找到下一个执行的函数。此功能搜索已知的共享内存去获取 CPU 的情况，然后调用 LongJump 服务，恢复返回值 1。从测试案例中获取测试参数 GetBbTestParametersFromTestCase 函数如图 3.5 所示：

```

7  EFI_STATUS
8  EFIAPI
9  GetBbTestParametersFromTestCase (
10     OUT EFI_GUID                **TestGuid,
11     OUT EFI_BB_TEST_ENTRY_FIELD **TestEntryField,
12     OUT EFI_BB_TEST_PROTOCOL_FIELD **TestProtocolField
13 );

```

图 3.5 GetBbTestParametersFromTestCase 原函数

这个函数功能是从分析函数中读取测试的库，测试的入口等信息，其中 TestGuid 是测试案例的 GUID，TestEntryField 是记录所测案例属于哪个库，TestProtocolField 是记录测试案例属于哪个协议。具体操作如图 3.6 所示：

```

280     Status = GetBbTestParametersFromTestCase (
281         &TestGuid,
282         &TestEntryField,
283         NULL // it is not useful in PEI phase.
284     );
285 白 if (EFI_ERROR (Status)) {
286     return Status;
287 }

```

图 3.6 如何使用 GetBbTestParametersFromTestCase 函数

3.2.4 Proxy 模块

Proxy 模块分 Pei Core BB Proxy Main Library 和 Pei BB Proxy Main Library 这两个库是在 PEI 阶段提供一个代理, 你可以把他们当做是一个房间, 在这个房间里面放着一些 MDE Library 的 API 接口, 这些接口是被重定向到这里的, 就相当于 C 语言的指针的概念。从这个角度上看这两个驱动都是为了映射 API 接口的, 没有什么区别。函数原型如图

3.7 和 3.8 所示:

```

32  EFI_STATUS
33  EFIAPI
34  PeiCoreBBProxyMain (
35      IN EFI_PEI_STARTUP_DESCRIPTOR *PeiStartupDescriptor,
36      IN VOID                        *OldCoreData
37  )
38  白{
39      EFI_STATUS Status;
40      void        *StartAddr;
41      VOID        *ProxyInterface;
42      UINTN       ProxyInterfaceLength;
43      EFI_GUID    TestGuid;
44
45      ProxyInterface = NULL;

```

图 3.7 Pei Core BB Proxy 函数原型

```

25  EFI_STATUS
26  EFI_API
27  PeiBBProxyMain (
28      IN      EFI_PEI_FILE_HANDLE      FileHandle,
29      IN CONST EFI_PEI_SERVICES        **PeiServices
30  )
31
32  {
33      EFI_STATUS  Status;
34      void        *StartAddr;
35      VOID        *ProxyInterface;
36      UINTN       ProxyInterfaceLength;
37      EFI_GUID     TestGuid;
38
39      ProxyInterface = NULL;

```

图 3.8 Pei BB Proxy 函数原型

Pei Core BB Proxy Main Library 其中的 PeiStartupDescriptor 是启动信息的指针，OldCoreData 指针是指向一个之前初始化用到的核心数据，通俗来说，这个函数需要瞻前顾后，需要知道参数传递进来的地址，也要负责初始化一下函数为 Pei BB Proxy Main Library 做准备。Pei Core BB Proxy Main Library 功能是 Pei Core BB Test Case Main Library driver 的入口地址，另一个是 GetBbTestParametersFromProxy，通过 proxy drivers 插入相应的测试的 GUID；最后一个 InitializeProxyInterface，这是负责初始化当前的测试程序接口的所有函数。在函数调用前都要去判断一下这个接口是否可调用，就是我们在用 malloc 函数一样，如果返回值是 NULL，则说明当前没有内存可以被分配。

3.2.5 Test 模块

Test 模块分 Pei Core BB Test Case Main Library 和 Pei BB Test Case Main Library，前面介绍了 Pei Core BB Proxy Main Library 和 Pei BB Proxy Main Library，这里介绍起来相对来说就比较简单了，你只要知道他们之间的区别就行了，很简单，一个是 proxy 类型的，一个 test 类型的，一个是提供源 API 的地址的，很明显另一个就是分析案例的地址，什么案例连接对应的 API 地址就要看它属于 Pei Core BB Proxy Main Library 还是

Pei BB Proxy Main Library。函数原型如图 3.9 和 3.10 所示:

```

245   EFI_STATUS
246   EFIAPI
247   PeiBBTestCaseMain (
248       IN EFI_PEI_FILE_HANDLE      FileHandle,
249       IN CONST EFI_PEI_SERVICES    **PeiServices
250   )
251
252   ␣{
253       EFI_STATUS      Status;
254       EFI_GUID         *TestGuid;
255       EFI_BB_TEST_ENTRY_FIELD *TestEntryField;
256       VOID             *TestInterface;

```

图 3.9 Pei BB TestCase 函数原型

```

155   EFI_STATUS
156   EFIAPI
157   PeiCoreBBTestCaseMain (
158       IN EFI_PEI_STARTUP_DESCRIPTOR *PeiStartupDescriptor,
159       IN VOID                        *OldCoreData
160   )
161   ␣{
162       EFI_STATUS      Status;
163       EFI_GUID         *TestGuid;
164       EFI_BB_TEST_ENTRY_FIELD *TestEntryField;
165       VOID             *TestInterface;

```

图 3.10 Pei Core BB TestCase 函数原型

如果我们在测试 Strcpy 函数的时候发现函数死循环了,这个时候我们就要采取在函数死循环之前设置断点,如何设置断点呢,我们可以考虑几个情况,BIOS 启动到 PEI 阶段后先是加载 PEIM,可以在程序进入死循环前任意一个地方设置断点,但是为了单步调试的快一些,尽量在一个离死循环程序附件的函数设置断点,那么毫无疑问最近的就是 Pei BB TestCase Main Library,如前面所说,我们在函数里面插入一个”_asm int 3”。如图 3.11 所示。然后,重新编译代码,并在 NT32 模拟器中运行。当程序执行到需要调试的地方时,会触发一个应用程序的异常,选择“Cancle”和“New Instance of Visual Studio 2005”

选项，这时候就进入了 DEBUG 环境

```
245 EFI_STATUS
246 EFIAPI
247 PeiBBTestCaseMain (
248     IN EFI_PEI_FILE_HANDLE      FileHandle,
249     IN CONST EFI_PEI_SERVICES   **PeiServices
250 )
251
252 {
253     EFI_STATUS      Status;
254     EFI_GUID        *TestGuid;
255     EFI_BB_TEST_ENTRY_FIELD *TestEntryField;
256     VOID            *TestInterface;
257
258     __asm int 3;
```

图 3.11 如何在 Pei BB TestCase 函数中设置断点

重新编译后程序便不能正常运行如图 3.12

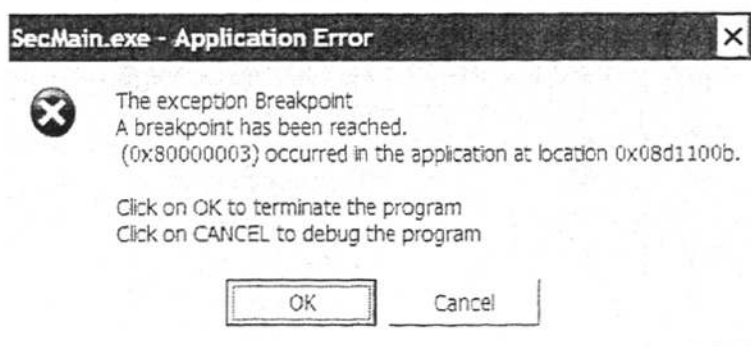


图 3.12 SecMain 进入死循环

点 Cancel 进入 VS2005 按 F9 进行调试。如图 3.13 所示：

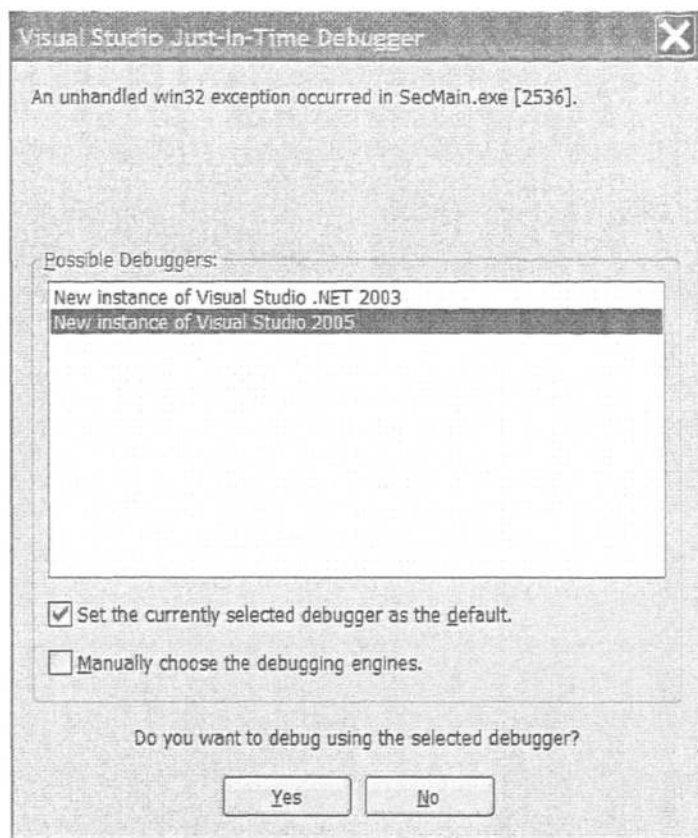


图 3.13 选择是否调试窗口

接着选择 Yes 进入调试窗口，在 VS2005 里面可以跟踪源代码是如何走的。并在 CPU 出现循环的时候查看出问题的代码。分析出死循环的原因所在。

3.3 NT32 下具体 API 分析举例

3.3.1 如何在 NT32 上搭建测试环境

PEI 是整个 BIOS 启动过程中的一个阶段，这个阶段主要是为了下一个阶段初始化环境用的，在这个阶段里面需要对 MDE 提供的 API 进行分析，如果在这个阶段中 API 没有实现其功能这对下面的工作带来不可想象的灾难，例如基本 base 函数不能实现 copy 功能，内

存不能传递信息到下一个阶段，等等这都对后期的驱动初始化环境带来灾难。测试框架大体设计如图 3.14 所示：

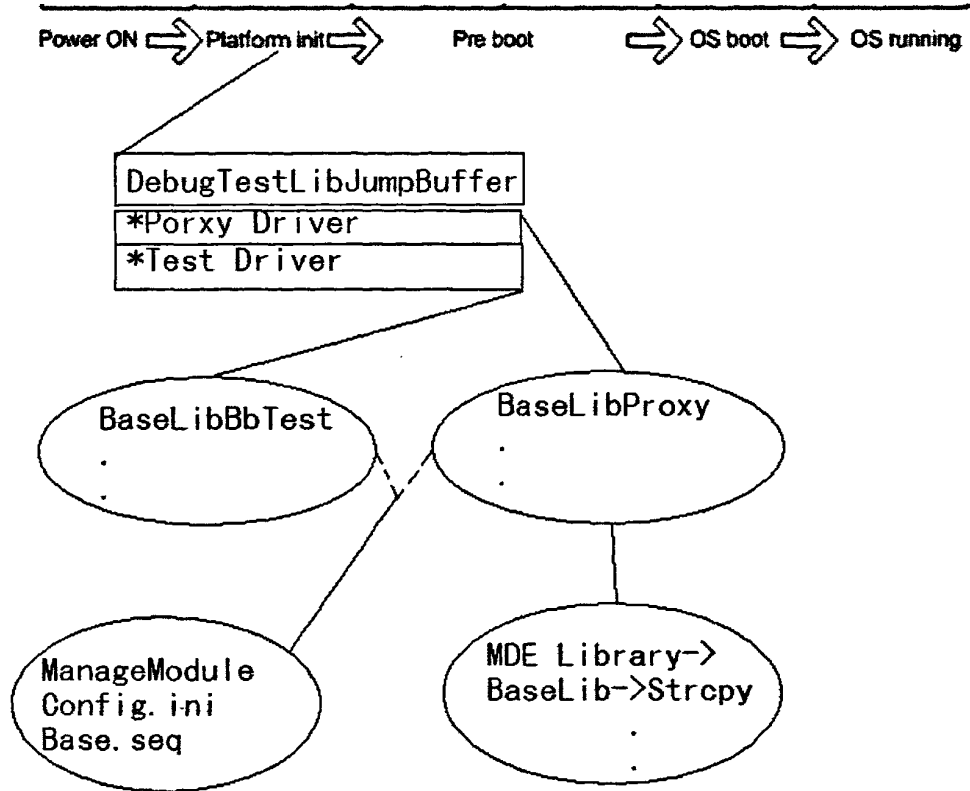


图 3.14 整个分析代码的工作流程

在 BIOS 上电后，首先进行安全检查，接着进行一些初始化，在 PI (Platform Initiation) 阶段，整个阶段包括 PEI 阶段，PEI 阶段需要做的事情就是初始化 MDE 函数接口，图中描述出程序的大致框架，通过嵌入一个模块完成性能分析，详细说明将在下文作出一一说明。

我们选定在 NT32 上面模拟出整个分析过程，NT32 是用 VS2005 为函数库，编译成一个模拟 BIOS 的环境，任何一台装有 VS2005 版的 PC 机都可以用来模拟仿真 EFI BIOS 的环境，NT32 源代码框架图如图 3.15 所示：

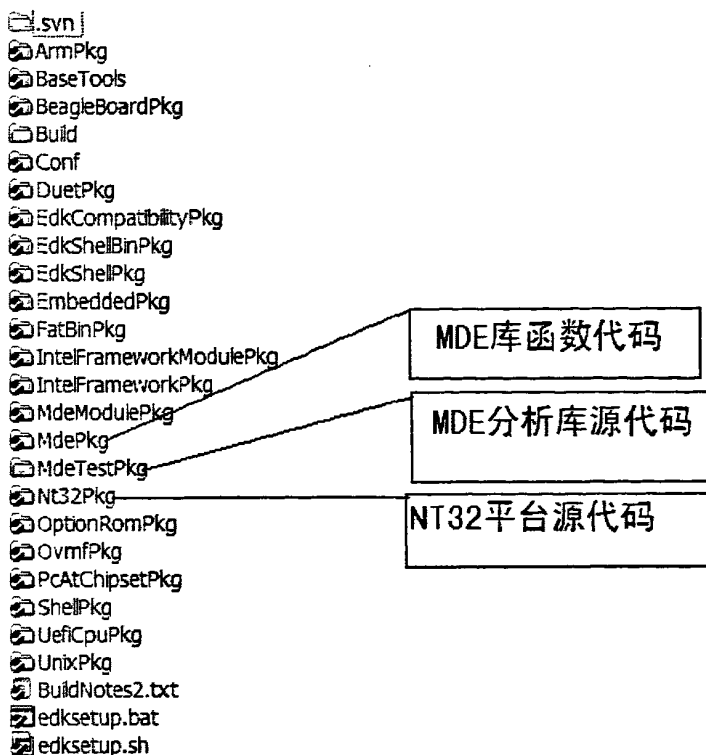


图 3.15 NT32 平台源代码文件

这里主要说明图中标示的几个文件夹，Mdepkg 就是底层 API 的函数库的集合，MdeTestPkg 是分析函数库的集合，Nt32Pkg 是模拟环境的代码库。其他有的是其他平台的代码库集合，例如 ArmPkg、BeagleBoardPkg 等等。我们要在 Nt32Pkg 下实现 API 的性能分析，就必须在 Nt32Pkg 嵌入测试 MdeTestPkg 代码，这些代码且支持在 32 位模拟环境下编译。Nt32Pkg 文件夹下有两个重要文件，一个是 Nt32Pkg.dsc 文件，还有一个是 Nt32Pkg.fdf 文件，Nt32Pkg.dsc 文件是制定 Nt32 平台编译出来的二进制代码，Nt32Pkg.fdf 是将编译出来的二进制代码进行封装，换句话说就是将编译出来的二进制代码重新组合成一个映像文件。

3.3.2 将分析库嵌入 NT32 平台

那么首先我们要将 MdeTestPkg 测试代码放到 Nt32Pkg.dsc 文件中，如下图 3.16 所示：

首先要介绍一下图中每个文件都是由一个文件名加上一串位置, 然后最后由一个*.inf 的文件结尾, 这里的 inf 文件的意识有点类似于嵌入式软件编程里面用到的一个 Makefile 是有类似的地方的, 首先, Makefile 中会定义一些可执行文件时由哪些源代码文件编译而成, 这里主要说明一个编译的关系, 就是说由什么编译成什么的一个概念。其次, 会定义一个最终生产的可执行文件。最后会声明一些编译时用到的编译命令。这是书写 Makefile 要了解的地方, 这里的*.inf 文件里面也会制定一些生成的可执行文件, 还有一些编译关系, 依赖关系等等。了解了这个*.inf 文件后来看一下下图是如何实现的。首先, 会有编译一 MdeTestPkg/TestInfrastructure/Pei/DebugTestLibJumpBuffer/DebugTestLibJumpBuffer.inf 这个就是前面所说的调试模块, 也就是为了分析进入死循环的时候调试所用到的, 接着就会连接到这个调试模块需要连接到的库, 这里连接的是

DebugTestLib|MdeTestPkg/Library/PeiDebugTestLib/PeiDebugTestLib.inf 文件。接着, 就是分析代理库的实现, MdeTestPkg/Proxy/BaseLibProxy/Pei/BaseLibProxy.inf, 要编译这个模块需要连接到许多的库, 这个库也是需要测试的, 所以最初的测试需要测试一些最基本的库接口, 如果最基本的库接口都不能工作, 那么对下面的库的分析带来不便。我们来一一分析, 这里 PcdLib|MdePkg/Library/PeiPcdLib/PeiPcdLib.inf 是一个平台需要用到的数据库, 将一些数据定义成一个标识, 后来的库接口无需定义新的标识, 可以直接用 Pcd 里面的数据。前面 HelloWorld 的实现也用到了这个 Pcd 的概念。然后连接模块 PrintLib|MdePkg/Library/BasePrintLib/BasePrintLib.inf 是用来打印日志的, 分析代码会调用这里模块里面的打印接口, 如果没有他, 调用就会失败。连接

IoLib|MdePkg/Library/BaseIoLib/BaseIoLib.inf 是为了调用一些基本 IO 输入输出的函数的, 简单的分析库可以不用到它。TimeLib|MdePkg/Library/BaseTimeLibNullTemplate/BaseTimeLibNullTemplate.inf 是时间模块, 打印日志的同时会输出时间信息。

PeiBBproxyMainLib|MdePkg/Library/PeiBBproxyMainLib/PeiBBproxyMainLib.inf 这个模块是比较重要的模块也是分析的重要模块, Pei 是指 EFI BIOS 启动的阶段, BB 是 BlackBox 简称, 后面 ProxyMainLib 是指代理主要库, 也就是将需要分析的接口重定向到这个地方。PeiBBConfigurationTableLib|MdePkg/Library/PeiBBConfigurationTableLib/PeiBBConf

figurationTableLib.inf 这个是一些分析接口的配置信息模块。

HobLib|MdePkg/Library/PeiHobLib/PeiHobLib.inf 这个 HobLib 是内存传递信息的一个空间模块，这个模块将一些内存信息传递到这里，告诉当前内存是一个什么情况。

PeiServiceLib|MdePkg/Library/PeiServiceLib/PeiServiceLib.inf 和

PeiServiceTablePointerLib|MdePkg/Library/PeiServiceTablePointerLib/PeiServiceTablePointerLib.inf 两个模块是在 PEI 阶段申请服务的一个必要步骤，我们的分析是在这个阶段的，必须要申请一个服务，有了这个服务我们才可以创建这个驱动，才能装载分析接口，对 API 分析。

MemoryAllocationLib|MdePkg/Library/PeiMemoryAllocationLib/PeiMemoryAllocationLib.inf 模块是用来从空闲内存中申请内存存放分析数据。

PeimEntryPointLib|MdePkg/Library/PeimEntryPointLib/PeimEntryPointLib.inf 模块是具体的模块存放分析模块的这里的 PEIM 模块就是最终存放调试模块，测试模块，代理分析模块的。DebugLib|MdePkg/Library/DebugLib/DebugLib.inf 这里的模块和调试模块不一样，之前那个是在 PEI 阶段调试的，这个是一个专门为分析阶段设置的调试模块。

BaseMemoryLib|MdePkg/Library/PeiBaseMemoryLib/PeiBaseMemoryLib.inf 是为 PEI 阶段提供对内存操作接口的一个模块。BaseLib|MdePkg/Library/BaseLib/BaseLib.inf 是一个基本的函数调用的模块，这个模块比较通用，他包括一些字符串拷贝函数接口，链表函数接口等等。CpuLib|MdePkg/Library/BaseCpuLib/BaseCpuLib.inf 是提供对 CPU 接口访问的模块。一般很少用。

DebugTestLib|MdeTestPkg/Library/PeiDebugTestLib/PeiDebugTestLib.inf 这个和前面一样。最后的 MdeTestPkg/TestCase/BaseLib/Pei/BaseLibBbTest.inf 是整个分析库所用到的函数接口，里面的模块和代理模块一样。

```

386 MdeTestPkg/TestInfrastructure/Pei/DebugTestLibJumpBuffer/DebugTestLibJumpBuffer.inf
387 <LibraryClasses>
388     DebugTestLib/MdeTestPkg/Library/PeiDebugTestLib/PeiDebugTestLib.inf
389 }
390 MdeTestPkg/Proxy/BaseLibProxy/Pei/BaseLibProxy.inf
391 <LibraryClasses>
392     PcdLib/MdePkg/Library/PeiPcdLib/PeiPcdLib.inf
393     PrintLib/MdePkg/Library/BasePrintLib/BasePrintLib.inf
394     IoLib/MdePkg/Library/BaseIoLibIntrinsic/BaseIoLibIntrinsic.inf
395     TimerLib/MdePkg/Library/BaseTimerLibNullTemplate/BaseTimerLibNullTemplate.inf
396     PeiBbProxy/MainLib/MdeTestPkg/Library/PeiBbProxyMainLib/PeiBbProxyMainLib.inf
397     PeiBbConfigurationTableLib/MdeTestPkg/Library/PeiBbConfigurationTableLib/PeiBbConfigurationTableLib.inf
398     HobLib/MdePkg/Library/PeiHobLib/PeiHobLib.inf
399     PeiServicesLib/MdePkg/Library/PeiServicesLib/PeiServicesLib.inf
400     PeiServicesTablePointerLib/MdePkg/Library/PeiServicesTablePointerLib/PeiServicesTablePointerLib.inf
401     MemoryAllocationLib/MdePkg/Library/PeiMemoryAllocationLib/PeiMemoryAllocationLib.inf
402     PeimEntryPoint/MdePkg/Library/PeimEntryPoint/PeimEntryPoint.inf
403     DebugLib/MdeTestPkg/Library/BaseDebugLibForTest/BaseDebugLibForTest.inf
404     BaseMemoryLib/MdePkg/Library/PeiMemoryLib/PeiMemoryLib.inf
405     BaseLib/MdePkg/Library/BaseLib/BaseLib.inf
406     CpuLib/MdePkg/Library/BaseCpuLib/BaseCpuLib.inf
407     DebugTestLib/MdeTestPkg/Library/PeiDebugTestLib/PeiDebugTestLib.inf
408 }
409 MdeTestPkg/TestCase/BaseLib/Pei/BaseLibBbTest.inf
410 <LibraryClasses>
411     PcdLib/MdePkg/Library/PeiPcdLib/PeiPcdLib.inf
412     PrintLib/MdePkg/Library/BasePrintLib/BasePrintLib.inf
413     PeiBbTestCaseMainLib/MdeTestPkg/Library/PeiBbTestCaseMainLib/PeiBbTestCaseMainLib.inf
414     PeiBbConfigurationTableLib/MdeTestPkg/Library/PeiBbConfigurationTableLib/PeiBbConfigurationTableLib.inf
415     PeiServicesTablePointerLib/MdePkg/Library/PeiServicesTablePointerLib/PeiServicesTablePointerLib.inf
416     PeiStandardTestLib/MdeTestPkg/Library/PeiStandardTestLib/PeiStandardTestLib.inf
417     PeiServicesLib/MdePkg/Library/PeiServicesLib/PeiServicesLib.inf
418     PrintLib/MdePkg/Library/BasePrintLib/BasePrintLib.inf
419     HobLib/MdePkg/Library/PeiHobLib/PeiHobLib.inf
420     PeimEntryPoint/MdePkg/Library/PeimEntryPoint/PeimEntryPoint.inf
421     BaseMemoryLib/MdePkg/Library/PeiMemoryLib/PeiMemoryLib.inf
422     MemoryAllocationLib/MdePkg/Library/PeiMemoryAllocationLib/PeiMemoryAllocationLib.inf
423     DebugTestLib/MdeTestPkg/Library/PeiDebugTestLib/PeiDebugTestLib.inf
424     DebugLib/MdeTestPkg/Library/BaseDebugLibForTest/BaseDebugLibForTest.inf
425     BaseLib/MdePkg/Library/BaseLib/BaseLib.inf
426 }

```

调试库

代理驱动

分析程序函数入口

图 3.16 将分析源代码嵌入 NT32 平台 (1)

接下来要将编译好的二进制代码放入到映像文件中，以上所说的代理库和分析模块是嵌入到 Nt32Pkg.dsc 平台里面按照这个平台编译条件编译出的模块，却没有真正产生一个 FD 文件，FD 文件就是最终固化到 flash 上的可执行文件，下面将整个分析模块嵌入到 Nt32Pkg.fdf 里面，如图 3.17 所示，这里只要将调试模块，代理模块以及分析模块嵌入即可。同时这三个模块在编译的过程中再连接给需要的模块如图 3.16 所示。

```

179 #####For PEI Test#####
180 INF MdeTestPkg/TestInfrastructure/Pei/DebugTestLibJumpBuffer/DebugTestLibJumpBuffer.inf
181 INF MdeTestPkg/Proxy/BaseLibProxy/Pei/BaseLibProxy.inf
182 INF MdeTestPkg/TestCase/BaseLib/Pei/BaseLibBbTest.inf
183 #####

```

图 3.17 将分析源代码嵌入 NT32 平台 (2)

这里需要几个解释, Nt32Pkg.fdf 是将编译好的模块加入 image 当中, 我们插入了这个分析模块, 就需要将这个分析模块的写入 Nt32Pkg.fdf 里面, 以便于最终的 image 里面含有分析模块。

3.3.3 将分析库编译到 NT32 平台里面

下面就开始编译, 编译前首先要设置环境变量, 也就是要指定编译路径, 编译选项, 编译工具等等。 “build -p Nt32Pkg\Nt32Pkg.dsc -a IA32 -t MYTOOLS” 其中 ‘-p’ 是指定哪个平台的命令, p 的全称是 Platform 的意思, 后面 ‘-a’ 是指定在编译出来的二进制文件要在哪个架构上, a 的全程是 Architecture, ‘-t’ 意思是指定在哪个编译工具下编译, t 的全程是 tools 的意思。执行具体步骤如图 3.18 所示:

```

C:\ViS2005 - build -p Nt32Pkg\Nt32Pkg.dsc -a IA32 -t MYTOOLS
\SysInternals;c:\\OSHook\CPSTools\WinXPSupportTools;c:\Program Files\UltraEdit;c:
\Program Files\Microsoft SQL Server\90\Tools\bin\c:\WINDOWS\system32;c:\WINDOW
S;c:\WINDOWS\System32\Wbem;c:\OSHook\CPSTools;c:\OSHook\CPSTools\SysInternals;c:
\OSHook\CPSTools\WinXPSupportTools

    WORKSPACE           = C:\Q1Release
    EDK_TOOLS_PATH       = C:\Q1Release\BaseTools

C:\Q1Release>build -p Nt32Pkg\Nt32Pkg.dsc -a IA32 -t MYTOOLS
21:13:56, Mar. 20 2010 [Windows-XP-5.1.2600]

WORKSPACE               = c:\q1release
ECP_SOURCE               = c:\q1release\edkcompatibilitypkg
EDK_SOURCE               = c:\q1release\edkcompatibilitypkg
EFI_SOURCE               = c:\q1release\edkcompatibilitypkg
EDK_TOOLS_PATH           = c:\q1release\basetools

TARGET_ARCH              = IA32
TARGET                   = DEBUG
TOOL_CHAIN_TAG           = MYTOOLS

Active Platform          = c:\q1release\Nt32Pkg\Nt32Pkg.dsc
Flash Image Definition   = c:\q1release\Nt32Pkg\Nt32Pkg.fdf

Processing meta-data ...
  
```

图 3.18 编译 NT32 平台

分析程序的核心代码如图 3.19 所示:


```

396  EFI_STATUS
397  EFI_API
398  StrCpyBbTestEntry (
399      IN VOID                                *This,
400      IN VOID                                *ClientInterface,
401      IN EFI_TEST_LEVEL                    TestLevel,
402      IN EFI_HANDLE                        SupportHandle
403  )
404  {
405      EFI_TEST_ASSERTION                    AssertionType;
406      BASE_LIBRARY_INTERFACE                *BaseLibraryInterface;
407      UINTN                                CheckPoint;
408      UINTN                                Index;
409      CHAR16                                *DestinationArray[4];
410      CHAR16                                *SourceArray[4];
411      CHAR16                                *ReturnValue;
412      INTN                                  DestinationResult;
413      INTN                                  ReturnResult;
414
415      BaseLibraryInterface = (BASE_LIBRARY_INTERFACE *) ClientInterface;

```

图 3.19 分析案例的设计

EFI_STATUS 和 EFI_API 是 EDK 里面定义的一个宏, EFI_STATUS 宏的申明是这样的: typedef unsigned long EFI_STATUS; 表示以下函数的返回值的类型是一个无符号长整型, EFI_API 没有什么特殊的意义, 仅仅是为了让程序员知道这里将会实现一个函数, 就像路标一样, 告诉驾驶员前方到达什么地方的意思。在 API 入口处的 IN 还有 OUT 等等这样类似的宏的意思和 EFI_API 的用处是一样的, IN 表示这里是输入的参数, OUT 是函数将要输出的参数, 但是下面出现的 UINTN、CHAR16、INTN 等宏就不一样了, 他们是经过 typedef 将类型重新定义了一下。EDK 里面为了统一类型申明, 将 C 语言里面的类型全部重定义了一些, 这样为了方便。也就相当于申明一下不同类型的变量以便函数所使用。这里要说明两重要的地方, 第一个是 AssertionType 这个变量是 EFI_TEST_ASSERTION 定义的一种返回值, 顾名思义, 测试函数有两种返回值, 要么成功, 要么失败。当然如果返回不是这两个, 那就是函数定义错了, 或者是其他的原因。譬如, 函数实现问题又或者硬件错误。

函数实现如图 3.20 所示: 这里先用一个 for 循环将一些地址做 StrCpy, StrCmp 等接口调用, 这些地址都是事先设置好的, 调用后会返回一个值, 给 ReturnValue, ReturnResult 和 DestinationResult, 这三个变量不是 0 就是 1, 最后用 if 语句判断成功与否。

```

449 白 for ( Index=0; Index<CheckPoint; Index++ ) {
450      ReturnValue      = BaseLibraryInterface->StrCpy (
451          DestinationArray[Index],
452          SourceArray[Index]
453      );
454      ReturnResult      = BaseLibraryInterface->StrCmp(
455          ReturnValue,
456          SourceArray[Index]
457      );
458      DestinationResult = BaseLibraryInterface->StrCmp(
459          DestinationArray[Index],
460          SourceArray[Index]
461      );
462 白  if ( ( DestinationResult | ReturnResult ) != 0 ) {
463      AssertionType = EFI_TEST_ASSERTION_FAILED;
464  } else {
465      AssertionType = EFI_TEST_ASSERTION_PASSED;
466  }

```

图 3.20 判断调用结果是否为真

第二个是 BASE_LIBRARY_INTERFACE, 这个定义的*BaseLibraryInterface 指针是向 Proxy 里面 MDElib 里面的 API, 具体函数实现如图 3.21 所示:

```

90  BASE_LIBRARY_INTERFACE *Interface;
91
92  Interface      = (BASE_LIBRARY_INTERFACE *) *ProxyInterface;
93
94  //
95  //String functions
96  //
97
98  Interface->StrLen      = StrLen;
99  Interface->StrCmp      = StrCmp;
100 Interface->StrCmp      = StrCmp;
101 Interface->StrCpy      = StrCpy;
102 Interface->StrSize      = StrSize;
103 Interface->StrnCmp      = StrnCmp;

```

图 3.21 如何将 API 地址转换成代理

在函数 InitializeBaseLibraryInterface.c 里面将 baselib 里面的函数进行初始化指向一个地址, 在 BaseLibBbTestString.c 里面拿到这个地址就可以对这个 API 进行分析。如图 3.22 所示

```
1163 | LibRecordAssertion (  
1164 |     mStandardLibHandle,  
1165 |     AssertionType,  
1166 |     gBaseLibBbTestStringAssertionGuid001,  
1167 |     L"BaseLib_StrCopy - Destination is NULL",  
1168 |     L"%a:%d:It should be ASSERT() and ReturnValue should be 1 but it returns %d.",  
1169 |     __FILE__,  
1170 |     __LINE__,  
1171 |     AssertReturnValue  
1172 | );
```

图 3.22 打印日志函数

其中 LibRecordAssertion 函数相当于 C 语言里面的 print 函数，只不过 LibRecordAssertion 函数里面定义了更多的类型输出，它将一些输出进行了规范化。它将程序分析的结果定义在里面的 AssertionType，这是一个枚举变量，这个变量决定下面的结果输出时成功还是失败。其中还会输出 eLibBbTestFunctionAssertionGuid006 这个是测试程序的 GUID，前面已经介绍了 GUID 的概念了，没有什么特殊的意思，但在机器语言中只能通过查找 GUID 找到程序入口记录分析数据。ReturnValue 这是 Strcpy 函数原型里面的返回地址，在字符串完成 copy 功能之后会返回一个地址，通过获取这个地址我们就可以比较 strcpy 函数功能是否完成这个字符串复制功能。

3.3.4 测试工具设计与分析

在 shell 下实现 ManageModule 工具核心代码如图 3.23 所示。

```

98     Status = InitManageModule();
99     if (EFI_ERROR(Status)) {
100         Print(L"Init Fail!\n");
101         goto end;
102     }
103     Status = RecordUserSelection();
104     if (EFI_ALREADY_STARTED == Status) {
105         Status = GenerateReport();
106         if (EFI_ERROR(Status)) {
107             }
108     }
109     Status = SelectedItemAvailable(&ItemRecord);
110     if (EFI_NOT_FOUND == Status) {
111         NeedReset = FALSE;
112     }
113     if (NeedReset) {
114         BurnFd(ImageHandle, FdLocation[ItemRecord.ItemId]);
115         gST->RuntimeServices->ResetSystem (EfiResetCold, EFI_SUCCESS, 0, NULL);
116     }
117     //return Status;
118     end: Print(L"\n\n");
119     Status = Fremove(CHECK_LIST_FILE);
120     return Status;
121 }

```

图 3.23 ManageModule 工具源代码

函数首先初始化,接着从测试程序中读取需要分析的 GUID,然后通过获取的变量把这些 GUID 写入到 BIOS PEI 阶段的变量中,接着再次重新运行工具,这是测试程序就会得到刚刚写入的 GUID 完成测试的 API。其中 InitManageModule 是初始化工具 ManageModule,初始化成功程序继续运行,失败则打印初始化失败,接着程序退出。RecordUserSelection() 这个函数是去文件中拿一些需要测试的 API 的 GUID,也就是说用户需要写入想分析的 API 的 GUID, ManageModule 将这些 GUID 写入到 BIOS 中, BIOS 根据这些链接找到程序的入口地址,从而完成测试。整个测试流程需要运行三遍 ManageModule 工具。前两次上面已经叙述,第三遍是获取分析数据。首先分析日志将打印一串 GUID 数字,这个 GUID 记录了一次调用后输出的结果。为什么会有多次输出,原因是我们会对每个 API 传几个代表性的参数,这个也是黑盒分析需要注意的地方。接着打印出结果 PASS 还是 FAILURE,也就是这次测试点的结果,最后输出日志。如图 3.24 所示。

```

83 918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS|BaseLib_StrCpy - with two unicode strings :
84 c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:SourceString=,
85 DestinationString=, ReturnValue=
86
87 918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:FAILURE|BaseLib_StrCpy - with two unicode strings :
88 c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:SourceString=
89 abcdefg, DestinationString=abcdefg, ReturnValue=
90
91 918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS|BaseLib_StrCpy - with two unicode strings :
92 c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:SourceString=,
93 DestinationString=, ReturnValue=
94
95 918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:FAILURE|BaseLib_StrCpy - with two unicode strings :
96 c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:SourceString=
97 hijklmn, DestinationString=hijklmn, ReturnValue=

```

结果有两个输入的参数返回是失败的

图 3.24 分析结果 (1)

在 shell^{[16][17]}下执行测试过程的现场效果图如图 3.25 所示:

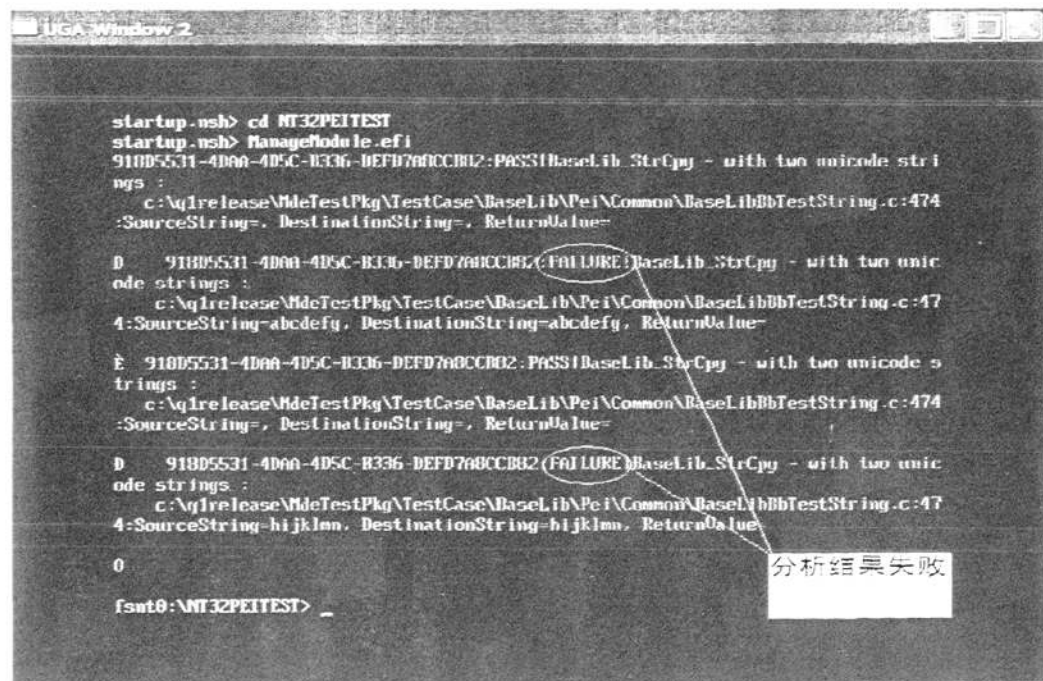


图 3.25 分析结果 (2)

这个时候我们需要查看原函数实现, 打开源代码如图 3.26 所示:

```

52  CHAR16 *
53  EFIAPI
54  StrCpy (
55      OUT      CHAR16          *Destination,
56      IN       CONST CHAR16    *Source
57  )
58  {
59
60      //
61      // Destination cannot be NULL
62      //
63      ASSERT (Destination != NULL);
64      ASSERT (((UINTN) Destination & BIT0) == 0);
65
66      //
67      // Destination and source cannot overlap
68      //
69      ASSERT ((UINTN) (Destination - Source) > StrLen (Source));
70      ASSERT ((UINTN) (Source - Destination) > StrLen (Source));
71
72  while (*Source != 0) {
73      *(Destination++) = *(Source++);
74  }
75  *Destination = 0;
76  return Destination;
77  }

```

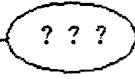


图 3.26 strcpy 源代码分析

函数实现是通过函数指针，返回一个正确的地址来实现的，函数内部定义了两个 char 类型的指针，Source 是输入的，Destination 是输出的，Source 指针被 CONST 修饰表示不可以被修改，这里是为了保护传进来的字符串的值。下面两宏是用来判断指针是否指向 NULL 的，如果是 NULL 程序异常退出，再下面两个宏是用来判断内存是否会出现重叠现象，如果两个地址之间的长度没有源字符串的长度大，就会引起内存覆盖，说白了就是目标字符串会覆盖掉原字符串，这是非常危险的，做完了这些安全检查后开始复制字符串，首先判断 while (*Source != 0)，当 *Source 不为 0 时，程序继续往下执行，当 *Source=0 时，程序跳到 *Destination = 0; 对 Destination 末尾赋值为 0，这是字符和字符串的重要区别，最

后在 return Destination 出错了，我们想一下，上面的 while 循环在做判断的时候，其内部会将 Destination++，也就是说最后 Destination 的地址里面的值是什么我们不得而知，所以我们要将 Destination 地址保存一份，以便下面程序返回，修改后的代码实现如图 3.27 所示：

```

52  CHAR16 *
53  EFIAPI
54  StrCpy (
55      OUT      CHAR16          *Destination,
56      IN       CONST CHAR16    *Source
57  )
58  {
59      CHAR16 *ReturnValue;
60
61      //
62      // Destination cannot be NULL
63      //
64      ASSERT (Destination != NULL);
65      ASSERT (((UINTN) Destination & BIT0) == 0);
66
67      //
68      // Destination and source cannot overlap
69      //
70      ASSERT ((UINTN) (Destination - Source) > StrLen (Source));
71      ASSERT ((UINTN) (Source - Destination) > StrLen (Source));
72
73      ReturnValue = Destination;
74      while (*Source != 0) {
75          *(Destination++) = *(Source++);
76      }
77      *Destination = 0;
78      return ReturnValue;
79  }

```

先定义一个指针
在下面将它指向目标字符串的首地址

返回目标字符串的首地址

图 3.27 strcpy 源代码优化

修改好了之后重新编译，如果仅仅修改源代码，而没有重新编译时不行的，因为程序代码一旦编译完成之后在内存中是不会随便改变的，所以必须重新编译。重新编译步骤和上面一样，重新测试数据如图 3.28 所示：

```

2  918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS\BaseLib_StrCpy - with two unicode strings :
3  c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:
4  SourceString=, DestinationString=, ReturnValue=
5
6  918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS\BaseLib_StrCpy - with two unicode strings :
7  c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:
8  SourceString=abcdefg, DestinationString=abcdefg, ReturnValue=abcdefg
9
10 918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS\BaseLib_StrCpy - with two unicode strings :
11 c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:
12 SourceString=, DestinationString=, ReturnValue=
13
14 918D5531-4DAA-4D5C-B336-DEFD7A8CCB82:PASS\BaseLib_StrCpy - with two unicode strings :
15 c:\qlrelease\MdeTestPkg\TestCase\BaseLib\Pei\Common\BaseLibBbTestString.c:474:
16 SourceString=hijklmn, DestinationString=hijklmn, ReturnValue=hijklmn

```

分析结果PASS

图 3.28 分析结果 (3)

在 shell 下运行的效果图如图 3.29:

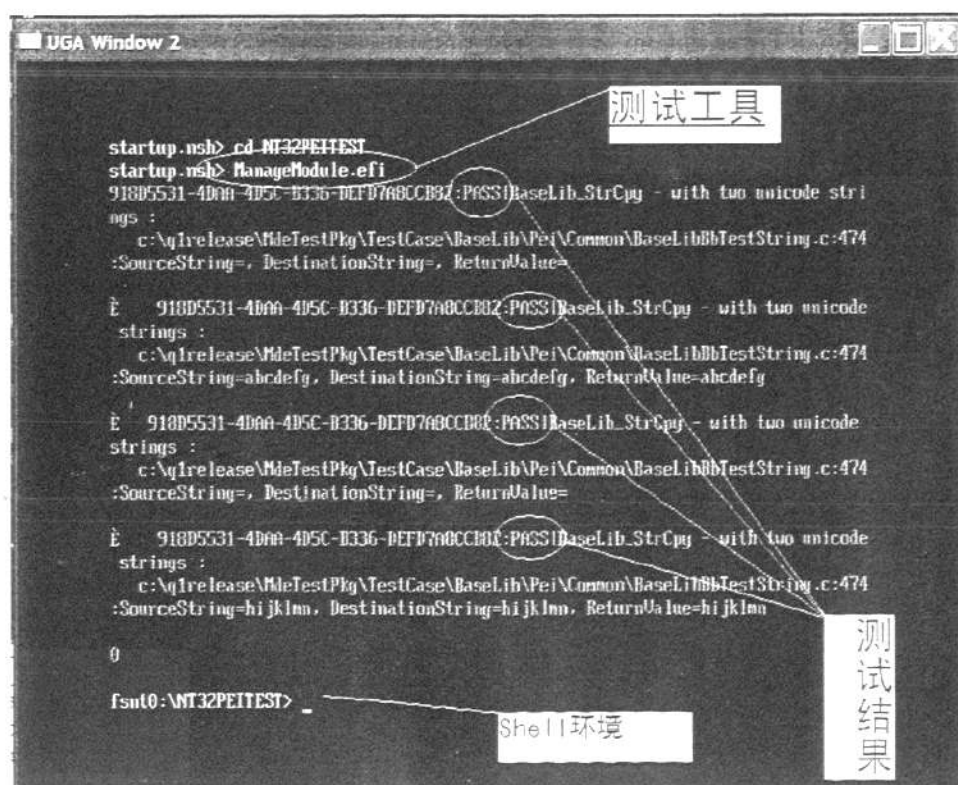


图 3.29 分析结果 (4)

以上是对 strcpy 进行功能分析, 下面来看看如何对其性能进行分析的, 性能分析是基于功能之上的测试分析, 说白了, 就是在完成功能的前提下再去考虑这个函数的鲁棒性。

这种考虑分两种情况，第一对 strcpy 函数输入一个错误的值，看它是否正确返回错误，第二种就是对函数实现的内部代码进行分析，这种情况下需要去读写 MDE Library 里面的 string.c 里面 strcpy 的实现，源代码实现如下：

函数实现是通过函数指针，返回一个正确的地址来实现的，函数内部定义了两个 char 类型的指针，Source 是输入的，Destination 是输出的，Source 指针被 CONST 修饰表示不可以被修改，这里是为了保护传进来的字符串的值。接着定义一个指针，由接下来的注释可知 Destination 不能为空，把握这条信息，我们可以在分析程序中将 Destination 赋一个空类型的指针。我们来看看分析程序是如何对性能进行分析的，如图 3.30 所示：

```

1259  EFI_STATUS
1259  EFIAPI
1260  StrCpyBbComformanceTestEntry (
1261      IN VOID                *This,
1262      IN VOID                *ClientInterface,
1263      IN EFI_TEST_LEVEL      TestLevel,
1264      IN EFI_HANDLE          SupportHandle
1265  )
1266  {
1267      EFI_TEST_ASSERTION      AssertionType;
1268      BASE_LIBRARY_INTERFACE  *BaseLibraryInterface;
1269      UINTN                   CheckPoint;
1270      UINTN                   Index;
1271      CHAR16                  *DestinationArray[4];
1272      CHAR16                  *SourceArray[4];
1273
1274      UINTN                   AssertReturnValue;
1275      UINT32                   MaxStringLength;
1276      CHAR16                  *SourceString;
1277      CHAR16                  *DesString;
1278      UINTN                   SourceLength;
1279      CHAR16                  *UnboundaryDestinationString;
1280      CHAR16                  *UnboundarySourceString;
1281      CHAR16                  *AnotherSourceArray[2];
1282      CHAR16                  *AnotherDestinationArray[2];

```

图 3.30 strcpy 性能分析源代码 (1)

函数申明和功能分析代码差不多，首先是接下来是分析的核心代码，如图 3.31 所示：

```

1348     SET_RETURN_POINT (AssertReturnValue);
1349
1350     if (AssertReturnValue == 0) {
1351         BaseLibraryInterface->StrCpy (NULL, L"abcd");
1352         AssertionType = EFI_TEST_ASSERTION_FAILED;
1353     } else if (AssertReturnValue == 1) {
1354         AssertionType = EFI_TEST_ASSERTION_PASSED;
1355     } else {
1356         //
1357         // Must never happen
1358         //
1359         AssertionType = EFI_TEST_ASSERTION_FAILED;
1360         CpuDeadLoop ();
1361     }
1362
1363     LibRecordAssertion (
1364         mStandardLibHandle,
1365         AssertionType,
1366         gBaseLibBbTestStringAssertionGuid001,
1367         L"BaseLib_StrCpy - Destination is NULL",
1368         L"%a:%d:It should be ASSERT() and ReturnValue should be 1 but it returns %d.",
1369         __FILE__,
1370         __LINE__,
1371         AssertReturnValue
1372     );

```

图 3.31 strcpy 性能分析源代码 (2)

这里和 strcpy 功能分析的区别就在于对这个函数输入的参数，在这里我们这里将字符串 ‘abcd’ 赋值给空类型 NULL，这个显然是不可以的，那么这就要看 MDE Libraray 里面的 strcpy 是如何做这件事了，这里就是性能分析的重要之处。如果函数能正确返回一个错误的值，那么就是说 strcpy 函数的性能在接受一个错误参数时是能够工作的。这里也可以说是函数的容错性能分析。性能分析结果图如图 3.32 所示：

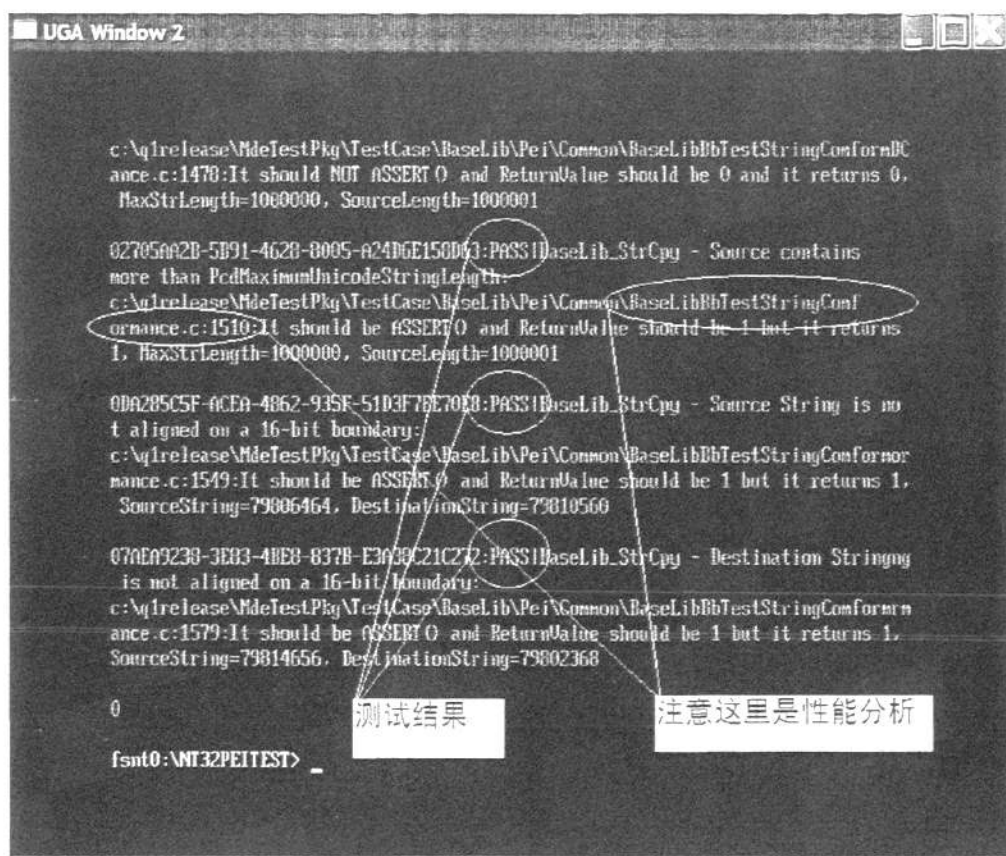


图 3.32 分析结果 (5)

如何将 GUID 写入到 BIOS 变量中, 以上介绍了 GUID 是用户自行写入到 BIOS 中的, 我们采取是用测试工具把 GUID 写入到 BIOS 的变量中, BIOS 重启的时候获取到变量里面的 GUID, 得知需要分析哪些数据。输入 GUID 的文件如图 3.33 所示:

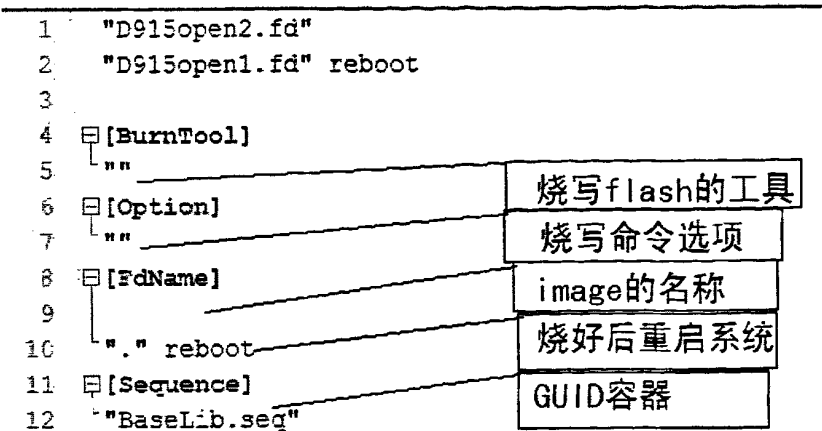


图 3.33 ManageModule 工具配置文件 Config.ini (1)

GUID 是通过*.seq 文件写入到 BIOS 中的，例如 Base.seq 文件的模型如图 3.34 所示：

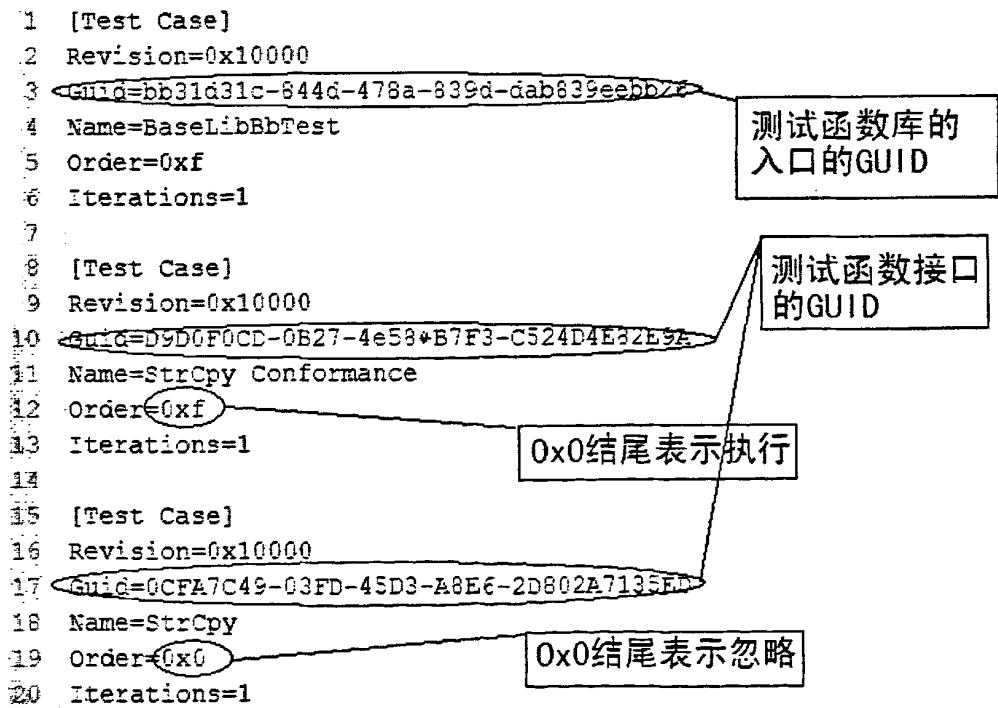


图 3.34 ManageModule 工具配置文件 Base.seq (2)

在这个 seq 文件中，对于 BIOS 来说最重要的是两个地方，一个是 Guid，一个是 Order，Order 是命令的意思，用户将 GUID 写入这个 seq 文件还不一定让程序去调用相应的测试程序，要通过 Order 的值来最终判段，换句话说程序先装载分析程序的 GUID，然后再与上 Order

的值，结果为 1，则分析；结果为 0，则丢弃。其他几个选择是给程序员看的。没什么特殊的说明。

3.4 本章小结

本章中首先介绍 MdePkg 模块在 EDK^{[18][19][20]}项目中的位置和作用，分析内部函数初始化的以及后期的作用，并结合实例讲解如何分析到内部某一个 API，并从功能和性能上对 API 进行内部分析，得出其功能和性能上的不足，并提出优化方案。同时给出在 NT32 下实现整个测试的流程，最后给出一些调试方法。

第四章 Linux 下实现 RUNTIME 的性能分析

4.1 性能分析模块的设计和实现简介

如何在 OS 底下实现 EFI BIOS 提供的实时服务^[21]，这个取决于 OS 环境是否提供这个接口，大家知道在 Windows 内核代码实现不是开源的，对于一个非微软内部的程序员是无法知道其内核模块是如何实现的，要想在用户态获取底层 API 是很困难的。而 Linux 就不一样，Linux 是一个极年轻的操作系统，它的诞生日期从 1991 年算起，迄今也就十几年，但是它的发展和成长却迅捷无比，成为操作系统领域中一匹名符其实的黑马。迄今为止，对 Linux 在全球范围内的装机台数的估计各有说法，最低的估计为 300 万，最高的估计数字为 900 万。而 1997 年，MacOS 的装机台数为 380 万，IBM OS/2 为 120 万，Windows NT 则为 700 多万。虽说 Linux 还无法与拥有一亿多用户的 Windows 相比，但是它确立自身地位和影响力所花费的时间却只有 Windows 的一半。作为一种 Unix 操作系统，Linux 的强大性能显然使得其它品牌的 Unix 黯然失色。有分析家认为，“Linux 的广泛普及已使其成为 Unix 市场上最具活力的一只新军。”甚至连 Unix 之父 Dennis Ritchie 也认为 Linux “确实不错”。有一些分析家甚至认为，在未来数年间，Linux 将成为 Windows XP 真正强有力的对手，也是唯一可以冲破微软垄断性文化圈的出路所在。Linux 的代码开发性也是我们选择 Linux 去实现实现服务的重要原因，实现框架大体如图 4.1 所示：

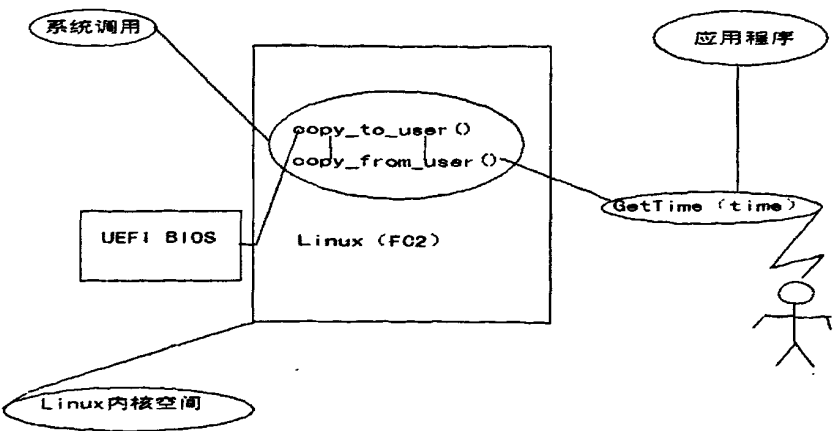


图 4.1 Runtime 阶段系统调用的实现架构

首先我们来看 Linux Kernel，在这里我们先实现一个驱动，这个驱动主要的功能是将底层 BIOS 的接口传递到内核空间，在驱动里面实现了一个内核和用户之间数据交换的功能，这个时候应用程序就可以间接的访问到 BIOS 提供的 API。

程序流程图如图 4.2 所示：

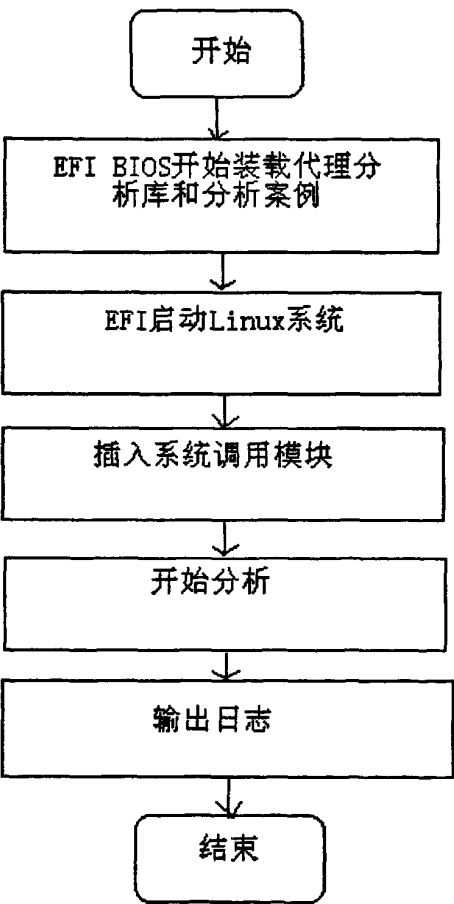


图 4.2 Runtime 阶段分析案例的实现架构

4.1.1 系统调用的设计与实现

4.1.1.1 什么系统调用

顾名思义，系统调用说的是操作系统提供给用户程序调用的一组“特殊”接口。用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务，比如用户可以通过文件系统相关的调用请求系统打开文件、关闭文件或读写文件，可以通过时钟相关的系统调用获得系统时间或设置系统时间等。

从逻辑上来说，系统调用可被看成是一个内核与用户空间程序交互的接口，它好比一个中间人，把用户进程的请求传达给内核，待内核把请求处理完毕后再将处理结果送回给用户空间。

系统服务之所以需要通过系统调用提供给用户空间的根本原因是为了对系统“保护”，因为我们知道 Linux 的运行空间分为内核空间与用户空间，它们各自运行在不同的级别中，逻辑上相互隔离。所以用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用户用空间函数。比如我们熟悉的“hello world”程序（执行时）就是标准的用户空间进程，它使用的打印函数 `printf` 就属于用户空间函数，打印的字符“hello word”字符串也属于用户空间数据。

但是很多情况下，用户进程需要获得系统服务（调用系统程序），这时就必须利用系统提供给用户的“特殊”接口——系统调用了，它的特殊性主要在于规定了用户进程进入内核的具体位置；换句话说用户访问内核的路径是事先规定好的，只能从规定位置进入内核，而不准许肆意跳入内核。有了这样的陷入内核的统一访问路径限制才能保证内核安全无虞。我们可以形象地描述这种机制：作为一个游客，你可以买票要求进入野生动物园，但你必须老老实实的坐在观光车上，按照规定的路线观光游览。当然，不准下车，因为那样太危险，不是让你丢掉小命，就是让你吓坏了野生动物。

4.1.1.3 如何使用系统调用-函数实现

上面介绍了一下系统调用的原理，下面来看一下在 Linux (FC2) 下是如何用系统调用来将 EFI BIOS 底层的可以提供 RUNTIME SERVICE 的 API 的参数传递到用户空间，系统调用主要可以用一下函数：`access_ok()`，`copy_to_user()`，`copy_from_user`，`put_user`，`get_user`。

`access_ok()` 函数原型: `int access_ok(int type, unsigned long addr, unsigned long size)`
函数 `access_ok()` 用于检查指定地址是否可以访问。参数 `type` 为访问方式, 可以为 `VERIFY_READ` (可读), `VERIFY_WRITE` (可写)。`addr` 为要操作的地址, `size` 为要操作的空间大小 (以字节计算)。函数返回 1, 表示可以访问, 0 表示不可以访问。

`copy_to_user()` 和 `copy_from_user()` 函数原型: `unsigned long copy_to_user(void *to, const void *from, unsigned long len)`

`unsigned long copy_from_user(void *to, const void *from, unsigned long len)`

这两个函数用于内核空间与用户空间的数据交换。`copy_to_user()` 用于把数据从内核空间拷贝至用户空间, `copy_from_user()` 用于把数据从用户空间拷贝至内核空间。第一个参数 `to` 为目标地址, 第二个参数 `from` 为源地址, 第三个参数 `len` 为要拷贝的数据个数, 以字节计算。这两个函数在内部调用 `access_ok()` 进行地址检查。返回值为未能拷贝的字节数。`get_user()` 和 `put_user()` 函数原型: `int get_user(x, p)`、`int put_user(x, p)` 这是两个宏, 用于一个基本数据 (1, 2, 4 字节) 的拷贝。`get_user()` 用于把数据从用户空间拷贝至内核空间, `put_user()` 用于把数据从内核空间拷贝至用户空间。`x` 为内核空间的数据, `p` 为用户空间的指针。这两个宏会调用 `access_ok()` 进行地址检查。拷贝成功, 返回 0, 否则返回 `-EFAULT`。

在 `UEFIBIOS` 将 CPU 控制权交给 OS 的时候同时会将一个底层初始化成功的函数参数地址传递到系统内核当中, 所以这里我们主要到 `copy_to_user()` 和 `copy_from_user()` 函数, 用他们将一些底层的函数地址传递到用户空间, 同时也可以将用户的请求传递到内核空间, 为什么这么做, 上面已经阐述了一些, 在内核中提供一个特殊的系统调用途径, 这个限制的用户的请求, 同时也保护了内核中的其他模块不受干扰, 这个就是 `copy_to_user()` 和 `copy_from_user()` 产生的原因之一。实现部分核心代码 (`efiRT.c`) 如图 4.3 所示:

```

67     transfor_number=copy_from_user(&param, (void*)arg, sizeof(param));
68
69     if (param.Time == NULL) {
70         p1 = NULL;
71     } else {
72         p1 = &eft;
73         transfor_number=copy_from_user(p1, param.Time, sizeof(efi_time_t));
74     }
75
76     if (param.Cap == NULL) {
77         p2 = NULL;
78     } else {
79         p2 = &efc;
80         transfor_number=copy_from_user(p2, param.Cap, sizeof(efi_time_cap_t));
81     }
82
83     spin_lock_irqsave(&efi_rt_lock, flags);
84     param.Status = efi.get_time(p1, p2);
85     spin_unlock_irqrestore(&efi_rt_lock, flags);
86
87     transfor_number=copy_to_user((void*)arg, &param, sizeof(param));
88     if (p1 != NULL) {
89         transfor_number=copy_to_user(param.Time, p1, sizeof(efi_time_t));
90     }
91     if (p2 != NULL) {
92         transfor_number=copy_to_user(param.Cap, p2, sizeof(efi_time_cap_t));
93     }
94
95     _return 0;
96 }

```

图 4.3 Runtime 阶段系统调用的源代码分析

4.1.1.3 如何将系统调用嵌入到 Linux 内核当中

需要做几点解释, efiRT.c 经过 GCC 编译之后, 加载到 Linux 内核当中, 加载的步骤和内核加载驱动的过程一样, 首先需要建立一个设备节点, 使用命令 'mknod /dev/efiRT.ko c 231 0', 创建好节点之后 再使用命令 'insmod efiRT.ko' 之后就可以在 /dev 下看到一个 efiRT 的设备, 说明系统调用模块已经成功加载到内核当中, 上图中这个片段函数主要实现的将用户的请求获取 BIOS 时间的请求传到内核当中来, 接着将 BIOS 传到内核中的时间参数, 传递给用户空间。这个过程图形表示如图 4.4 所示:

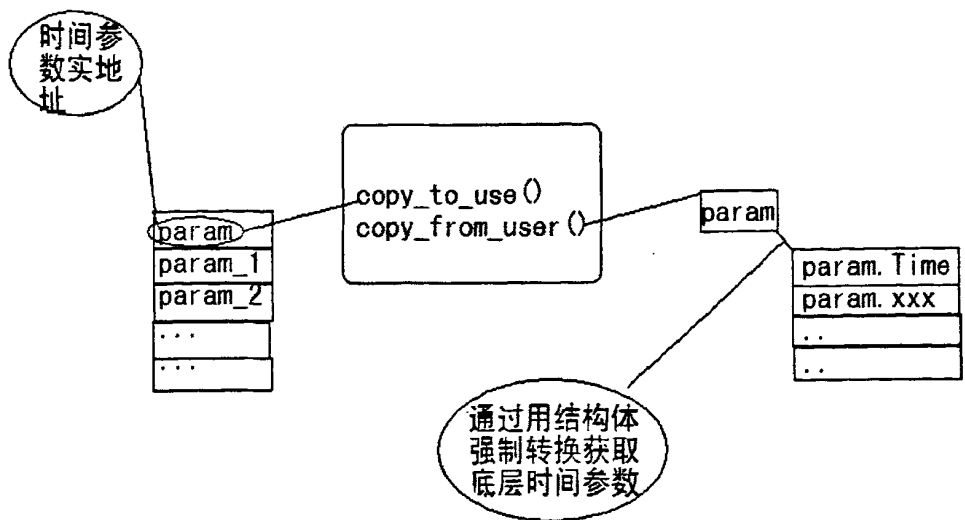


图 4.4 底层函数地址传递到上层

系统调用将时间参数传递给用户空间后，用户空间通过地址强制转换，便可以知道，这个地址下面的其他参数地址，前提是要指定这个结构体的原型，如果你不知道这个结构体的原型是无法获得其中的参数地址的，有的结构体原型，我们可以重新定义这个结构体变量，然后将得到的地址强制转换成一个结构体，并将其中的变量取出来赋值给新的变量，从而可以实现实时调用。

4.1.2 系统调用的应用程序设计与实现

4.1.2.1 编写应用程序

实现了 Linux 下系统调用的功能之后，完成应用程序的编写就相对来说简单一些，这个时候我已经知道系统内核态有一个函数接口是提供给用户使用的，我只需要编写一个调用函数去内核中寻找我想得到的函数。函数原型如图 4.5 所示，这里的首先用到一个函数接口 `gtRT->GetTime()`，这个接口是为了给实时操作系统准备的一个服务，这个接口返回值是 0 或者 1，下面使用这个接口为了获取当前系统时间，获取后用 `if` 语句判断 `Status` 来输出获取的时间，就是说如果 `gtRT->GetTime()` 获取成功，那么 `if` 语句就向下执行打印

函数 RecordAssertion 里面的参数, 如果失败, 程序则退出, 推出就打印失败, 由 EFI_ERROR 这个宏来输出错误。

```

66     Status = gRT->GetTime(
67         &OldTime,
68         NULL
69     );
70     if (EFI_ERROR(Status)) {
71         StandardLib->RecordAssertion (
72             EFI_TEST_ASSERTION_FAILED,
73             gTestGenericFailureGuid,
74             "RT.GetTime - Get time",
75             "%a:%d:Status - %r",
76             __FILE__,
77             __LINE__,
78             Status
79         )

```

图 4.5 Runtime 阶段分析案例的源代码 (1)

图 4.6 如下所示的具体含义是这样的, 将时间改变并输出结果。上图 4.5 中已经获取了 OldTime 的时间, 这里将 OldTime 时间赋给 Time, 注意这里的 OldTime 和 Time 不是一个变量而是一个结构体的首地址, 拿到这个首地址之后将可以获取结构体里面的成员值, 首先用 if 语句判断 Year 的值, 这里做了一个简单判断, 其思路是判断 Year 的值是否是 2010, 如果不是就改成 2010, 如果是测改成 2011, 这个设计是将新的时间用 gRT->SetTime() 设置到 Time 中进而判断 gRT->SetTime() 是否工作。接着打印结果, 输入日志。

```

84      //
85      // Change year
86      //
87      Time = OldTime;
88      if (Time.Year != 2010) {
89          Time.Year = 2010;
90      } else {
91          Time.Year = 2011;
92      }
93      Status = gRT->SetTime(
94          &Time
95      );
96      if (Status == EFI_SUCCESS) {
97          AssertionType = EFI_TEST_ASSERTION_PASSED;
98      } else {
99          AssertionType = EFI_TEST_ASSERTION_FAILED;
100     }
101     StandardLib->RecordAssertion (
102         AssertionType,
103         gTimeServicesInterfaceTestAssertionGuid003,
104         "RT.SetTime - Change year",
105         "%a:%d:Status - %r. Year %d",
106         __FILE__,
107         __LINE__,
108         Status,
109         OldTime.Year
110     );

```

图 4.6 Runtime 阶段分析案例的源代码 (2)

4.1.2.2 编译运行

编写好程序之后，接着 GCC 编译，编译过程首先在 shell 下使用命令 ‘gcc -c GetTimeFunc.c -o GetTimeFunc’，‘-c’ 是制定源程序，‘-o’ 是输出可执行文件，可执行文件的名字是 GetTimeFunc。编译完成后在编译的目录下会找到一个 GetTimeFunc 文件，这个时候通过将前面的系统调用模块加载到内核中之后，在 shell 下执行 ‘./GetTimeFunc’，接着就会出现下面的程序输出结果如图 4.7 所示：

```
1  -----
2  GetTime_Func
3  -----
4  Logfile: "Log/GetTime_Func.log"
5  Test Started: 03/04/2009  14:12:15
6  -----
7  RT.GetTime - Valid parameters -- PASS
8  7D06EE71-E5F5-4AF3-BC74-ECF471F8F612
9  GetTimeFunc.c:80:Status - Success, Time 03/04/2009  14:12:15
10 -----
11 RT.GetTime - Verify returned time -- PASS
12 93EF949C-241E-4365-B00B-0842F63F978C
13 GetTimeFunc.c:95:Status - Success
14 -----
15 GetTime_Func: [PASSED]
16   Passes..... 2
17   Warnings..... 0
18   Errors..... 0
19 -----
20 Logfile: "Log/GetTime_Func.log"
21 Test Finished: 03/04/2009  14:12:16
22 -----
```

图 4.7 Runtime 阶段分析结果

第五章 课题总结与展望

5.1 课题总结

本文基于目前最新的 EDKII/Tiano 体系架构,针对 Intel 主流平台架构提出了底层 API 检测和 PEI 阶段程序结构性性能分析的设计方法,并给出了具体的编码实现。

主要工作具体有以下几个方面:

1. 研究了 EDKII/Tiano 系统技术,了解了 UEFI/Tiano 系统的技术背景以及体系结构、运行机制、主要功能、存在问题等。

2. 根据 SMBIOS 工业规范、EFI Shell 下应用程序开发规范,设计完成一款 PEI 阶段的底层 API 性能分析测试模块;通过努力,成功的在 UEFI SHELL 下实现了 MDE Library^[22] 的分析。

3. 研究了程序结构性性能分析的基本方法,通过对整个 EDKII 的编译选项进行改进,使之支持特殊的编译选项;编译出能够在实际硬件平台运行的 efi 应用程序;在实际运行过程中,通过 ITP 硬件调试工具对 CPU 进行实时分析,成功解决了插入函数不稳定导致程序崩溃的实际问题。

4. 通过对 RUNTIME 阶段底层 API 的研究,实现了在 Linux (FC2) 下底层 API 的实时调用,所做工作是实现 Linux 内核中系统调用,并编写应用程序代码获取底层 TimeService 提供的准确时间。

5.2 研究展望

本文的内容紧紧围绕着 UEFI BIOS 的底层 API 的分析以及从 PEI 阶段到 RUNTIME 阶段分析其性能,这里面还有很多工作可以继续展开:

1. 对于在 PEI 阶段的分析是远远不够的,毕竟 BIOS 启动过程还有经历很多的阶段,要想这些模块正常工作就需要在不同的阶段对他进行各种角度的分析,譬如 DXE 阶段, SHELL 下的 API 状态等等,本文提出在 PEI 阶段阶段分析方法,但是如果 PEI 阶段没问题,

不代表以后的阶段的 API 也是没有问题的，所以还需要进一步进行完善。

2. 对底层 API 信息的检测，除了在 PEI 阶段进行测试，同时可以扩展到 DXE 阶段，也可以通过 UEFI BIOS 提供的 Runtime 的服务获取 System table 的起始虚拟地址来进行。

3. 通过对 RUNTIME 阶段调用底层 API 的研究是远远不够的，还需要对 ACPI 提供的接口做进一步的研究，没有 ACPI 对 OS 提供的 S3, S4 的支持是没有意义的。UEFI BIOS 的程序结构性能分析，目前仅仅在 Linux 内核中实现，在 Windows 系统目前还没有方法，因为 Windows 内核函数接口不得而知，需要微软支持方可获取正确的接口。只有通过不同的操作系统的测试分析，UEFI BIOS 的稳定性才能得到提高。

参考文献

- [1] 倪光南, UEFI BIOS是软件业的蓝海, UEFI技术大会, 2007年06月13日
- [2] 余志超, 朱泽民. 新一代BIOS—EFI、CSS BIOS技术研究, 科技信息, 2006(5)
- [3] 李振华. 于USBKEY的EFI可信引导的设计与实现, 硕士论文, 北京交通大学, 2008
- [4] 胡藉. 面向下一代PC体系结构的主板BIOS研究与实现, 硕士论文, 南京航空航天大学, 2005
- [5] 洪蕾. UEFI的颠覆之旅, 中国计算机报, 2007年07月09日
- [6] Framework Open Source Community, EFI_Shell_getting_Started_GuideVer0_31, Framework Open Source Community, June 27, 2005
- [7] Framework Open Source Community, EFI_Shell_Release_Notes0_91, Framework Open Source Community, June 29, 2005
- [8] Framework Open Source Community, Pre_EFI Initialization Core Interface, Framework Open Source Community, March 2008
- [9] Framework Open Source Community, Shared Architectural Elements, Framework Open Source Community, March 2008
- [10] UEFI Forum. May 2009. UEFI Specification Version2.3.
- [11] Vincent Zimmer. 2006. Beyond BIOS Intel corporation. 17-32, 143-146
- [12] 潘登等, 刘光明. EFI结构分析及Driver开发, 计算机工程与科学, 2006(02) .
- [13] 吴松青等, 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)
- [14] 石浚菁. EFI接口BIOS驱动体系的设计、实现与应用, 硕士论文, 南京航空航天大学, 2006
- [15] 姚颀文等, 谢康林, 石勇军, 曾文. 基于EFI驱动-协议模型自我认证测试系统, 计算机仿真, 2005(07)
- [16] 杨荣伟. 基于Intel多核平台的EFI/Tiano图形界面系统研究, 硕士论文, 上海交通大学

学, 2007

[17] 张朝华. 基于EFI/Tiano的协处理器模型的设计与实现, 硕士论文, 上海交通大学, 2007

[18] Intel corporation. August 2008. EDK II Extended INF File Specification Revision Version 1.1

[19] Intel corporation. May 20, 2007. EDK Flash Description File (FDF) Specification

[20] Intel corporation. Oct. 12, 2006. EDK II Module Development Environment Library Test Infrastructure

[21] Intel corporation. October 4, 2006. Intel Platform Innovation Framework for EFI Human Interface Infrastructure Specification

[22] Intel corporation. October 9, 2006. MDE Library Spec

作者在攻读硕士学位期间发表的论文

1. 赵远东 倪兴荣 曹平 “基于 CDMA2000 的移动视频监控系统”. 通信技术 2009 第九期 57-59 页.

致 谢

本课题在选题及研究过程中得到赵远东副教授的悉心指导。在三年的研究学习期间，赵老师为我提供学习和实习的条件，从论文的开题和撰写过程中赵老师提出了很多宝贵的意见和建议，帮助我开拓研究思路。赵老师严谨求实的态度，踏踏实实的精神，深深地感染了我。赵老师为人谦和，平易近人，从他身上我学到了不仅仅是专业知识，更多的是学习的方法和做人的态度。

其次感谢 Intel 亚太研发有限公司的 SSG 组的软件工程师王岩，在论文项目和实习项目中，王岩都给予了大量的技术支持。感谢王岩为我提供的硬件方便的研究工作。同时感谢实习项目组的顾晓刚、蒋文杰、刘江、梁晓彤以及同组的实习生王宇飞，其他项目组的实习生桂坤等等。在论文撰写的过程中，他们给予了我大力的支持。和他们讨论的过程中，学习到了很多知识，进一步对 UEFI BIOS 的了解和领会。感谢我同宿舍的同学陈超，杨旭对我学习、生活的关心和帮助。

最后，向我的父亲、母亲致谢，感谢他们这些年来对我的辛苦付出。在此，我要向他们深深地鞠上一躬。