

分类号_____

学校代码 **10487**

学号 **M201276096**

密级_____

华中科技大学

硕士学位论文

UEFI 的异常中断处理驱动的 设计与实现

学位申请人 陈书仪

学 科 专 业：软件工程

指 导 教 师：黄立群 副教授

答 辩 日 期：2015.1.13

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

Design and Implementation of Error Handling UEFI Driver

Candidate : Chen Shuyi

Major : Software Engineering

Supervisor : Assoc. Prof. Huang Liqun

Huazhong University of Science and Technology

Wuhan 430074, P. R. China

January, 2015

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

学位论文授权使用授权书

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

UEFI(Unified Extensible Firmware Interface)被认为是替代传统的 BIOS 的下一代计算机固件接口标准。UEFI 采用了面向对象的设计思想,通过使用模块化设计思路来组织驱动和应用程序的,并且采用标准协议 Protocol 来实现模块之间的通信。UEFI 相比之前的 BISO 是可以采用高级计算机语言如 C 并通过现代的软件工程方法来设计接口。随着 UEFI 得到越来越多的认可,被更多厂商认同并采用,UEFI 的应用和驱动也有了更多的使用场景,但是 UEFI 自动错误中断处理机制的不完善,影响了 UEFI 应用的使用体检。

在 UEFI Shell 的自动化测试过程中,常常遇到异常,而 UEFI 自带的错误处理机制是当应用程序运行发生异常时,调用异常中断处理函数,该函数会让程序停止并且机器也会停止,无法跳转。在异常处理结束之后,机器将不能进行工作,因此需要重新开机并重头开始运行自动测试程序的脚本。这样的异常中断处理机制会浪费大量的时间以及机器成本。因此需要设计实现一种新的异常处理机制,即实现一个异常中断处理驱动。该驱动只需要在程序执行之前,加载该异常中断处理驱动,当遇见程序错误时,就会调用驱动的接口,跳出当前程序。机器可以接收指令输入,可以进行其他操作。

这个异常中断处理 Driver 相比 UEFI 自带异常中断处理有如下优势:(1)利于数据的保存,规避了数据的易失性。由于自带异常处理机制生效之后,需要强制重新启动机器,会导致数据丢失。而本文的异常处理驱动运行之后,是跳出异常程序,并不需要重新启动机器,之前程序运行得到的数据就可以被保存下来,不会发生数据丢失。(2)节约时间和机器成本。由于机器也不需要重启,程序也不需要重新运行,便节省了时间和机器成本。

关键词: 统一可扩展固件接口 异常中断处理 驱动 应用

Abstract

UEFI (Unified Extensible Firmware Interface) is considered to be an alternative to traditional BIOS firmware interface standard for the next generation of computer. UEFI use object-oriented to design and modular design ideas to organize drivers and applications, use standard protocol to communicate between modules, it can be used high-level language and modern software engineering method to design the interface of the program. With UEFI gets more and more recognition, UEFI applications and drive more usage scenarios. But UEFI automatic error interrupt processing mechanism is imperfect, affected the UEFI application using medical.

The UEFI original error handling mechanism is calls abort processing function when run a application which have problem, the function will stop program then the machine will stop, can't jump. At the end of the exception handling the machine will not be able to work, so you need to restart the machine and you need to run the program from start to end again. So there is a requirement to design and implement a new exception handling mechanism, and realizes the exception interrupt handling. Just before execution the program, loading the exception interrupt handling drive at the first time, when meet the error of the running program, the code would call the driver interface, then jump out of the current program.

This implemented abnormal interrupt handling by UEFI bring extremely interrupt processing has the following advantages:(1) Conducive to the preservation of the data. Because the own exception handling mechanism need forced reboot the machine, this can lead to loss of data. But the run the exception handling drive after operation, this article is out of abnormal process, do not need to reboot the machine, before the program runs the data can be preserved, data loss will not occur. (2)To save time and machine costs.

Key words: UEFI Error Handling Driver Application

目 录

摘 要.....	I
Abstract.....	II
1 绪论	
1.1 研究背景与意义	(1)
1.2 国内外研究现状	(4)
1.3 论文研究内容与结构	(6)
2 相关技术分析	
2.1 UEFI Application 和 UEFI Driver 简介	(8)
2.2 UEFI 层次结构介绍.....	(11)
2.3 UEFI 协议介绍.....	(11)
2.4 EFI 系统表.....	(13)
2.5 UEFI Shell	(14)
2.6 虚拟平台 NT32	(15)
2.7 本章小结.....	(15)
3 UEFI 的异常处理 Driver 的需求分析与总体框架设计	
3.1 UEFI Driver 开发环境介绍	(16)
3.2 UEFI 的异常处理 Driver 的功能需求分析	(18)
3.3 UEFI Driver 的模型分析与开发流程分析	(23)
3.4 UEFI Driver 的总体结构设计	(19)
3.5 本章总结.....	(24)
4 UEFI 的异常处理 Driver 的详细设计与实现	
4.1 Driver 的 Inf 文件设计	(25)

华中科技大学硕士学位论文

4.2	Driver 的框架接口设计与实现.....	(27)
4.3	Driver 的功能实现接口的设计与实现.....	(35)
4.4	本章小结.....	(37)
5	Driver 在 UEFI Shell 自动测试中的应用	
5.1	UEFI Shell 自动测试介绍	(38)
5.2	搭建运行及测试环境	(45)
5.3	编译运行以及验证	(45)
5.4	本章小结.....	(49)
6	总结与展望	
6.1	全文总结.....	(50)
6.2	展望.....	(51)
	致 谢.....	(52)
	参考文献.....	(53)

1 绪论

1.1 研究背景与意义

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

1.1.1 传统 BIOS 介绍

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

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

（1）对 MBR 的依赖

在引导系统的过程中，传统的 BIOS 要先找到 MBR，然后把 MBR 中的引导指令加载到内存中^[4]。但是对于 MBR 的过分依赖，使得传统 BIOS 下的磁盘主分区不能超 4 个。然而电子技术的快速发展，使得大容量磁盘的应用变得越来越普及，在计算机系统中，传统 BIOS 对 MBR 的依赖严重制约了 PC 的发展。

（2）平台依赖

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

在处理器进入 32 位时代后，新的处理器兼容原有的 16 位的运行方式，甚至在处理器发展到 64 位的时代，处理器在加电启动的时候，还会使用 16 位的实模式^[6]。使得 BIOS 的根本性质没有得到任何改变，严重影响启动时间和运行速度，16 位的运行工作环境是其最为致命的缺点。

（3）ROM 容量限制

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

（4）汇编语言实现

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

（5）安全性问题

安全性问题在 BIOS 的发展过程中越来越引起重视，首先 BIOS 缺乏自身的安全防护功能，一些病毒或黑客利用 BIOS 芯片刷新机制，去破坏 BIOS 代码或添加危害系统安全的恶意代码，由于恶意代码处于系统底层，非常不易检测，给系统造成严重的安全威胁^[8]。此外，由于 BIOS 未考虑配置的安全问题，使得远程攻击者可利用本地计算机 BIOS 中的配置漏洞，通过网络来获取系统的控制权限，从而实现对本地计算机的远程控制和数据存取，而 BIOS 的配置漏洞深入系统底层，远程攻击者甚至可以在本地计算机关机的情况下开启本地计算机并进行存储访问^[9]。

1.1.2 UEFI 介绍

自第一套 BIOS 程序问世以来, BIOS 就以寄存器参数调用、16 位汇编代码、静态链接, 以及 1MB 以下内存固定编址的形式存在了几十年^[10]。BIOS 在外观上的落后、功能上的羸弱、安全上的薄弱、性能上的不足, 都严重制约着它的进一步发展^[11]。虽然目前的 BIOS 仍能够实现基本的功能, 但 PC 要进步就必须寻求更高更好的技术, 相应地原有 BIOS 的发展也就近乎走到了尽头。

EFI (Extensible Firmware Interface) 最初由 Intel 针对安腾处理器平台推出, 并在 HP、Microsoft 和 Phoenix 的合作下产生。尽管 EFI 是为安腾处理器开发的, 但 Intel 在设计 EFI 时没有定位于只适用于安腾处理器, 从而使得后来对 IA32、XScale, 以及 UEFI 的 x64 和 ARM 的绑定都成为可能^[12]。初期 EFI 一直用于 Intel 的安腾处理器系统 (IA64), 即 Intel 64 位服务器上, 所以很多 PC 用户甚至完全不知道它的存在。直到 2003 年, Intel 宣布 EFI 将在 IA32 架构上实现, 并在 IDF (Intel Developer Forum) 上展示了 EFI 的风采, EFI 才逐渐被大众所了解^[13]。EFI 从提出之初就具有足够的前瞻性, 随着 Intel 的积极推广, EFI 得到了越来越多的 PC 生产厂商和 BIOS 提供商的支持。2005 年, 包括 Intel、AMD 在内的一些 PC 生产厂家成立了统一 EFI (Unified Extensible Firmware Interface, UEFI) 论坛, 以共同推动和开发适用于各平台的 PC 固件标准。

本质上 UEFI 是一套崭新的 BIOS 接口, 并不对应具体的 BIOS 实现。通常而言, 遵照 UEFI 接口规范实现的 BIOS 称为 UEFI BIOS, 相应地将不遵循 UEFI 接口规范的早期 BIOS 系统称为传统 BIOS (Legacy BIOS)^[14]。

目前, UEFI 凭借自身的技术特点, 可以广泛运用于嵌入式应用、网络电脑、网络客户端电脑等产品, 其应用已由服务器领域扩展至 PC 领域。与此同时, UEFI 技术向消费电子、家用设备领域的延伸也从未停止。

1.1.4 UEFI 架构分析

做为一个开放的业界标准接口, UEFI 在系统固件和操作系统之间定义了一个抽象的编程接口, 就像操作系统的 API 一样。由于传统 BIOS 的种种弊端, 使得英特

尔抛弃了对传统 BIOS 的依赖，设计了 UEFI 的实现方案，并为这一新设计取名 Platform Innovation Framework for EFI，简称 Framework。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^[22]。

大体上，UEFI BIOS 的层次结构由以下 4 层组成：

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

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

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

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

1.2 国内外研究现状

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

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

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

为了统一 EFI 标准和规范，由英特尔、微软、惠普等全球著名的计算机软、硬件厂商于 2005 年发起成立了国际 UEFI 联盟。UEFI 是指 Unified Extensible Firmware Interface，中文意思是统一可扩展固件接口。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，就像操作系统的 API 接口一样^[25]。UEFI 规范并没有限定这个编程接口的具体实现，标准的 UEFI 接口可以有多种不同架构的实现，它们对外都表现为相同的接口。英特尔对 EFI 的实现定义了 Framework，UEFI 联盟基于 Framework 定义了 PI（平台初始化，Platform Initialization）规范；PI 规范建立了固件内部接口架构以及固件和平台硬件间接口，而后者使得平台硬件驱动程序具有模块化和互操作性^[26]。随着 UEFI 和 PI 规范的 Framework 实现，英特尔完成了取代传统 BIOS 的使命，并使得业界超越了传统 BIOS。

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

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

UEFI 的发展，带来了 BIOS 技术上翻天覆地的变化。对个人电脑来说，优势体现不是很明显。但是对于企业用户来说，优势巨大。比如，传统的 BIOS 只支持 4

个主分区，当工作需要 100 个主分区时，传统的 BIOS 就遇到瓶颈了。但是 UEFI 支持 128 个主分区，UEFI 的研究和应用潜力巨大^[31]。

目前国内外对于 UEFI 的研究，一方面停留在通用性的层面，即设计的 UEFI BIOS 能够满足加载不同的操作系统的要求^[32]。UEFI 的通用性设计，能够最大限度的兼容不同的操作系统^[33]。但是它同样带来了一个问题，无法结合特定的操作系统发挥出操作系统在加载前和加载后的优势。在启动过程中，如果相关硬件驱动发生问题，无法加载操作系统的时候，这系统的排错和纠错的能力也大大减弱^[34]。

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

1.3 论文研究内容与结构

在 UEFI Shell 的自动化测试过程中,常常遇到异常,而 UEFI 自带的错误处理机制是当应用程序运行发生异常时,调用异常中断处理函数,该函数会让程序停止并且机器也会停止,无法跳转。在异常处理结束之后,机器将不能进行工作,因此需要重新开机并重头开始运行自动测试程序的脚本。这样的异常中断处理机制会浪费大量的时间以及机器成本。因此需要设计实现一种新的异常处理机制,即实现一个异常中断处理驱动。本课题设计并实现了这种新的异常中断处理驱动。这个驱动只需要在程序执行之前,加载该异常中断处理驱动,当遇见程序错误时,就会调用驱动的接口,跳出当前程序。机器可以接收指令输入,可以进行其他操作。

本课题实现的异常中断处理 Driver 相比 UEFI 自带异常中断处理有如下优势:(1) 利于数据的保存,规避了数据的易失性。由于自带异常处理机制生效之后,需要强制重新启动机器,会导致数据丢失。而本文的异常处理驱动运行之后,是跳出异常程序,并不需要重新启动机器,之前程序运行得到的数据就可以被保存下来,不会发生数据丢失。(2)节约时间和机器成本。由于机器也不需要重启,程序也不需要重新运行,便节省了时间和机器成本。

全文一共分为六章

华中科技大学硕士学位论文

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

第二章介绍 UEFI Application 和 UEFI Driver 的基本概念,UEFI Driver 设计实现时使用到的一些关键性技术和分析,以及 UEFI Shell 和虚拟平台的介绍。

第三章是根据中断处理 Driver 的工作原理,明确系统的需求并进行总体驱动设计。

第四章是 UEFI 的错误中断处理 Driver 的设计与实现。介绍了 Driver 的功能接口和框架接口的设计和实现。

第五章是驱动在 UEFI Shell 自动化测试脚本中的应用。介绍了 UEFI Shell 自动化测试框架,已经测试用例的设计。并通过测试框架的运行验证了本课题设计的驱动异常中断处理驱动的可行性。

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

2 相关技术分析

UEFI 为用户提供了一个交互环境: UEFI shell 用户可以通过 UEFIshell 来导入自己编写的特定的 Application 和 Driver。同时,在系统出现问题,用户可以通过 shell 对平台进行诊断和调试。

2.1 UEFI Application 和 UEFI Driver 简介

2.1.1 UEFI Application 的概念

UEFI Application(应用程序)是一种 EFI Image 的类型, EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION 执行时,当 Image 退出或从入口点返回时这种 Image 时会自动卸载。UEFI Application 既可以被 UEFI OS Loader(引导管理器)加载,也可以被其它 UEFI Application 所调用来加载^[36]。它的加载是没有什么条件的,但它的卸载却是有条件的,只有当 Application 从入口点返回时,或者 Application 调用了引导服务 Exit()的时候,UEFI Application 才会被卸载。在卸载之后会将控制权返回到上一级即调用了该 Application 的上一级 UEFI Application 或是其他 UEFI 组件。UEFI Application 和 UEFI Drivers 被加载到内存的类型是有区别的,UEFI Application 会在需要被调用加载的时候作相应的调整,UEFI Application 加载完成以后,控制权就转到应用程序的入口点,这点跟 UEFI Application 卸载时也是很像的。

2.1.2 UEFI Drivers 的概念

UEFI Drivers 可以通过 UEFI OS Loader(引导加载器)或 UEFI 固件或 UEFI Application(应用程序)加载调用^[38]。与加载 UEFI Application 时类似,加载 Driver 的时候, Firmware(固件)会分配足够的内存来保留这个 UEFI Driver 的 Image(映像),接着会把该 Driver 的入口拷贝到已经分配好的内存中。当有 UEFI Application 或是 UEFI 组件加载调用这个 Driver 的操作完成时,才会将控制权转到 Driver 的入口点。如果 Driver 从入口点返回了 EFI_SUCCESS 状态,则表示加载成功了。但是

如果返回错误代码，Driver 就会从内存中卸载，并且将控制权返回给调用该 Driver 的 UEFI Application 或是 UEFI 组件。

UEFI Driver 有两种，一种是 UEFI Driver Model Driver（启动服务时驱动），另一种是 UEFI Non Driver Model Driver（运行程序时驱动）^[39]。它们两者之间，唯一的区别是 UEFI Non Driver Model Driver 在 OS Loader 调用 ExitBootServices（）之后就失效了，这个 Driver 将会被卸载，并且这个 Driver 所占用的内存资源也会被释放，而 UEFI Driver Model Driver 将不会被关闭仍然有效。

UEFI Drivers 可分为两大类：UEFI Driver Model Driver 和 UEFI Non Driver Model Driver.具体分类细节如图 2-1 所示。

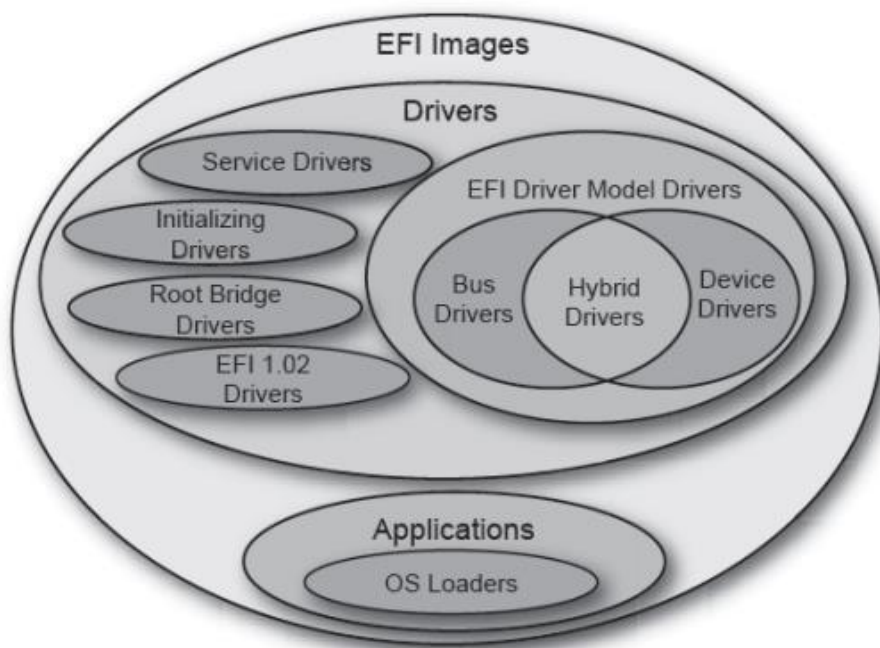


图 2-1 UEFI Driver 分类图

1) 服务驱动

这种驱动会将一个或多个协议加载在一个或多个服务器句柄上，并如果加载成功则在它的入口会返回 EFI_SUCCESS 状态。

2) 初始化驱动

这种驱动只会进行一些初始化操作，不会产生任何句柄，也不会增加任何协议

到句柄数据库，并且返回结果的状态，如 `EFI_SUCCESS` 或失败。所以这种驱动执行完后就会立即从系统内存中卸载。

3) 桥接器驱动

这种驱动会产生一个或多个控制句柄，而且包含一个设备路径协议和一个对芯片跟总线提供的 I/O 资源软件方式抽象出来的协议。

1) UEFI 驱动模型驱动

(1) 总线驱动

这种驱动会在句柄数据库中产生一个或多个驱动句柄或驱动映像句柄，并在该句柄上安装一个或多个驱动绑定协议(Driver Binding Protocol)的实例。这种驱动在调用协议的 `Start()` 函数时会产生新的子句柄，而且会在新的子句柄上增加额外的输入/输出协议。

(2) 设备驱动

与总线驱动有区别，设备驱动不会产生新的子句柄，只会在现有已存在的子句柄上增加额外的输入/输出协议。

(3) 混合型驱动

同时具有总线驱动和设备驱动的特征，既会在现有已存在的句柄上增加输入/输出协议，也会产生新的子句柄。

2.1.3 Application 与 Driver 的区别

每个程序必有入口点，入口点其实是一个接口，它就是主函数，编译器会去识别，让系统能够调用函数功能，UEFI 映像都是可移植可执行格式的，由 PE 映像头的 `Sub-system` 值来区别应用程序和驱动程序^[40]。

Application 和 Driver 本质没什么区别，都是一段程序代码，只是他们的侧重点有所不同而已。Driver 主要提供某一种特别服务。Application 除了提供特殊的服务外还需要和用户交互^[41]。并且 Application 可以直接被执行，但 Driver 是不可以直接被执行的，它需要被调用，它可以被脚本或应用程序调用。

2.2 UEFI 层次结构介绍

大体上, UEFI BIOS 的层次结构由以下 4 层组成^[42]:

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

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

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

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

2.3 UEFI 协议介绍

2.3.1 PCI I/O 协议

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

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

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

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

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

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

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

2.3.2 磁盘 I/O 协议

磁盘I/O协议用于将块I/O协议的块访问抽象为一种更加通用的偏移量长度协议。这一固件负责把该协议添加到任何在系统中还没有一个磁盘I/O协议的块I/O接口上去。文件系统和其他磁盘访问代码使用该磁盘I/O协议^[44]。

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

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

- (1) UEFI FAT12、FAT16 和FAT32 文件系统类型。
- (2) 传统的主引导分区记录块。
- (3) 扩展分区记录块。
- (4) El Torito逻辑块设备。

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

2.3.3 块 I/O 协议

块I/O协议用于抽象大容量存储设备,以允许运行在UEFI启动服务环境中的代码在不了解设备具体类型或者不了解管理设备的控制器情况下可以访问它们。已定义

了对大容量存储设备以块层次进行数据读写，以及在UEFI启动服务环境中进行设备管理的功能。

2.3.4 简单文件系统协议

简单文件系统协议允许运行在UEFI启动服务环境里的代码对一个设备进行基于文件的访问。简单文件系统协议用于打开一个设备卷，并且返回一个UEFI文件句柄，该句柄提供了访问设备卷文件的接口。这个协议与大多数协议有一点不同，除了简单文件系统对设备进行分层以外，还有一个辅助协议作用于该设备^[45]。

2.3.5 UEFI 文件协议

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

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

2.4 EFI 系统表

EFI 系统表是 UEFI 中最重要的数据结构。EFI 系统表的指针作为入口函数的一部分被传递到每一个驱动程序和应用程序。从这个数据结构中，可执行的 UEFI 映像就能访问系统配置信息和足够的 UEFI 服务^[46]，这些服务包括以下几个方面：

- (1) UEFI 启动服务 (UEFI Boot Services)
- (2) UEFI 运行时服务 (UEFI Runtime Services)
- (3) 协议服务 (Protocol Services)

UEFI 启动服务和 UEFI 运行时服务的访问分别通过 EFI 启动服务表和 EFI 运行时服务表来进行。这两个表都是 EFI 系统表里的数据段。每张表所能提供的服务数量和种类对于 UEFI 规范的每个修订版本来说是固定的。UEFI 启动服务和 UEFI 运行时服务在 UEFI 规范中定义。

协议服务是一组相关的功能和数据字段，它们被命名为全局唯一标识符 GUID，并被定义为统计意义上唯一的实体。通常，协议服务用于为诸如控制台、磁盘和网络这样一类的设备提供软件抽象，但是它们可被用于扩展平台可用的普通服务的数量。协议是 UEFI 固件扩展功能的机制。UEFI 规范定义了超过 30 种不同的协议，各种各样的 UEFI 固件的实现，UEFI 驱动程序可以产生更多的协议来扩展平台的功能。

2.5 UEFI Shell

UEFI Shell 是一个简单的、交互式的命令行环境，可以加载 UEFI 驱动、可以启动 UEFI 应用并且还可以引导操作系统^[47]。而且，更重要的是，它还提供了许多非常实用的基本命令，可以进行对文件和系统环境的管理操作等。UEFI Shell 是一种很重要的开发工具。UEFI Shell 的运行界面如图 2-3 所示。

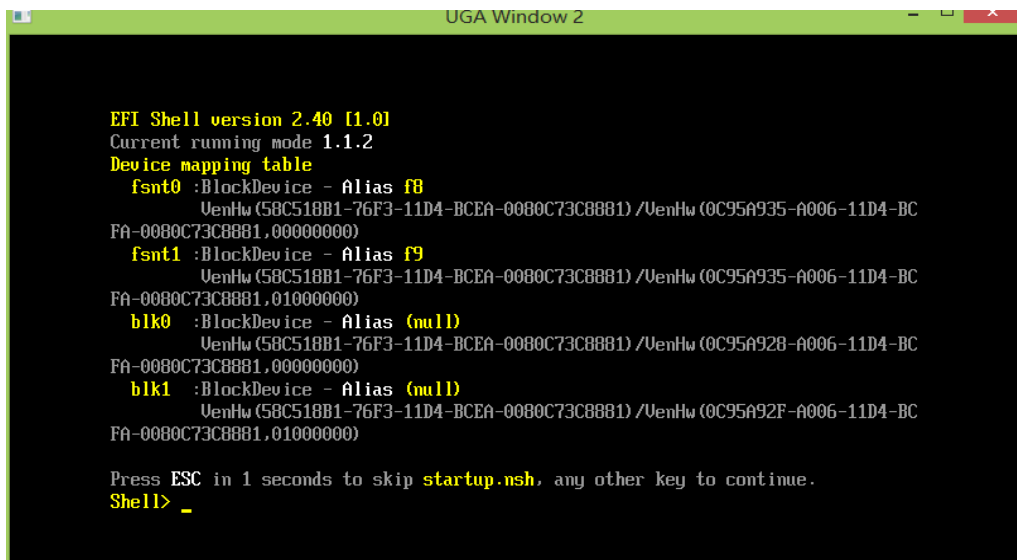


图 2-3 在 NT32 虚拟平台下 UEFI Shell 运行效果图

实际上 UEFI Shell 本质上也是一种 UEFI 应用，如果 Shell 本身的二进制代码被编译在固件中，那么就可以直接从固件加载 UEFI Shell；如果固件中没有编译进 UEFI Shell，那么可以从外接的媒介（软盘、USB、光盘）加载。

如果经常在 shell 下面做很多日常的工作，这时候有批处理文件会比较方便。为了方便开发人员，Shell 中也有批处理（脚本）文件，以 nsh 后缀名结尾的文本文件。Startup.nsh 是种特殊的脚本，它在 Shell 启动以后自动加载执行。

2.6 虚拟平台 NT32

NT32 是一个指令执行就像用户模式的进程一样的 32 位 Microsoft Windows System 虚拟的框架平台。这是一个“软”平台，该平台的功能是通过抽象 Win32 服务来实现。

NT32 平台不需要知晓硬件细节，而是使用操作系统的应用程序接口来实现对硬件平台的抽象。它作为正常启动进程的一部分，最终以启动应用程序的方式启动固件仿真平台。对多数开发者来说，创建和执行这个 NT32 仿真环境变得非常简单，不过是在一个标准的硬件平台上启动一个熟知的操作系统，然后象创建和执行普通的应用程序一样来启动这个仿真环境。

2.7 本章小结

本章主要介绍了 UEFI Application 和 UEFI Driver 的概念，并对比了 UEFI Application 与 UEFI Driver 的区别。介绍了 UEFI 的层次结构。在最后对运行 UEFI Driver 的 Shell 环境和 NT32 虚拟平台进行了介绍。

3 UEFI 的异常处理 Driver 的需求分析与总体设计

3.1 UEFI Driver 开发环境介绍

开发 UEFI 所需的工具包 EDKII (EFI Developer Kit) 可在 <http://www.tianocore.org/> 下载, 由于 BIOS 是系统底层软件, 直接在真实环境中开发与调试相对困难。UEFI EDK 工具包提供了一组仿真器和模拟器, 方便进行程序的快速开发与验证。本文以 Nt32 仿真器和 DUET 模拟器为主要的模拟开发环境来示例一些 UEFI 驱动和应用的开发。可以直接通过下载 EDK 开发工具包来获取 Nt32 以及 DUET 的程序和源代码。其中 NT32 的源码会在 AllPackagesDev\Nt32Pkg 文件夹下。DUET 的源码在 AllPackagesDev\DuetPkg 文件夹下。开发 UEFI 可以在不同平台下进行, 使用任何文本编辑器和 C/C++ 编译器。

搭建开发环境需要以下步骤:

1) 安装开发工具包。

在 Windows 下进行 UEFI 开发, 需要 Microsoft Windows 2000 或 Windows XP 以上版本操作系统。如果开发 Intel IA-32 平台的应用, 需要 Microsoft Visual Studio .NET 2003 企业版以上 (VC 编译器的版本 7 以上)。如果开发安腾™处理器家族的应用, 还需要 Microsoft Windows Driver Development Kit (DDK), build 3790 以上版本。本实验采用了 Visual Studio 2008 来进行开发。

2) 下载源代码。

即从网上下载 EDKII 的源码。本实验把源码下载到了桌面。文件夹目录为 C:\Users\tiano\Desktop\AllPackagesDev。EDKII 中各个代码包 Package 间有十分紧密的联系。

EDKII 包的结构图如 3-1 图所示。

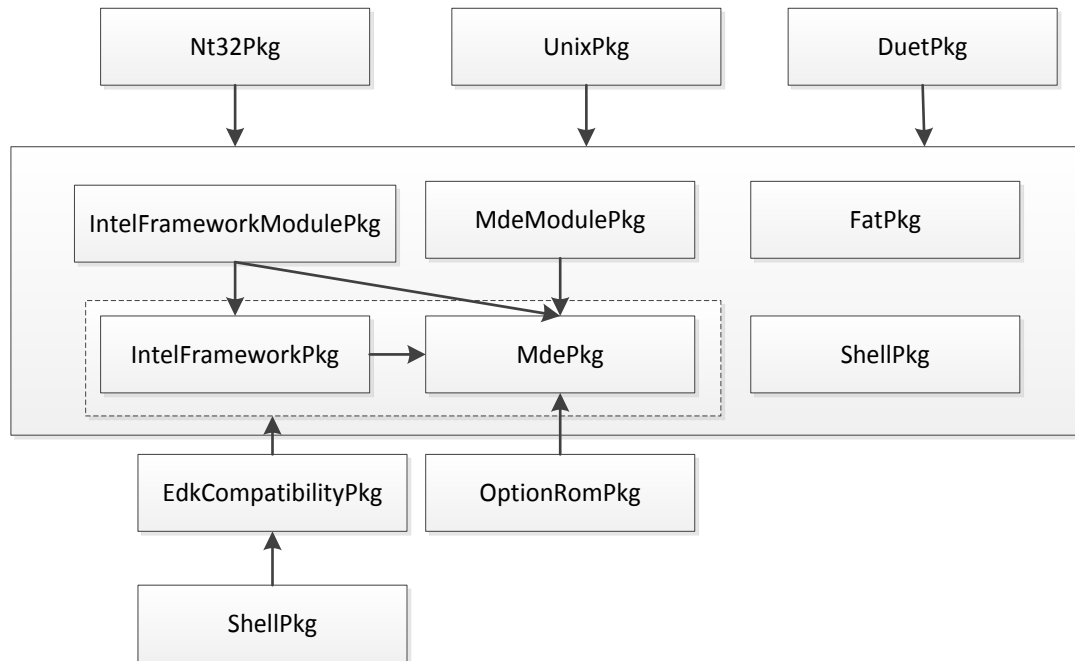


图 3-1 EDKII 包的结构图

其中MdePkg这个Package尤为重要，它里面涵括了EDKII环境下IA32、X64、IPF和EBC平台的底层库函数、工业标准和协议。而ShellPkg中包含的则是UEFI Shell的源代码。

3) 编译运行 Nt32 仿真环境

进行 Nt32 模拟，需要做以下几个步骤：

(1) 运行“Visual Studio 2008 Prompt Command”窗口。有两种方法可以开启。

a 可以从【开始】|【所有程序】|【Microsoft Visual Studio 2008】|【Visual Studio .Net Tools】|【Visual Studio 2008 Prompt Command】这样打开。

b 打开 Windows 命令行环境 (cmd.exe)，【开始】|【运行】，输入 cmd.exe 并回车，然后在命令行环境进入 Visual Studio 安装目录，找到 vcvars32.bat 批处理文件并运行。

(2) cd C C:\Users\tiano\Desktop\AllPackagesDev

(3) 在命令行下添加 EDK_SOURCE 环境变量，设置为 C:\Users\tiano\Desktop\AllPackagesDev。或者直接运行 EdkSetup.bat 脚本，即会自动

设置好当前运行环境。

(4) 编译 Nt32 虚拟环境。在窗口中输入 `build -p Nt32Pkg/Nt32Pkg.dsc -a IA32 -t VS2008x86`。编译成功后，将会生成 `C:\Users\tiano\Desktop\AllPackagesDev\Build\NT32IA32\DEBUG_VS2008x86` 文件夹。

(5) 编译结束后，”build run”命令，就运行 Nt32 仿真器。Nt32 默认会启动到 EFI Shell。

3.2 UEFI 的异常处理 Driver 的功能需求分析

在 EDK II 的代码包中，在一个 Application 运行时发生异常的情况下，自动调用的 `DEBUG_ASSERT` 方法，是让整个程序停下来，然后系统就挂住，无法进行任何操作。此时只能强制关闭系统并重新启动，这种处理方式会造成一些负面效果，如：数据丢失，强制关机重启后，之前得到的数据可能没有进行保存操作，从而丢失；浪费时间，由于是强制关闭后重启，需要重新运行这个 Application。该自带异常处理机制如图 3-2 所示。

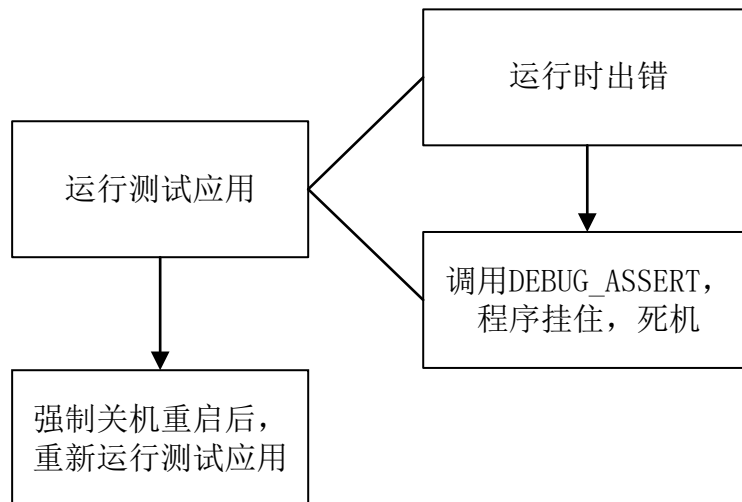


图 3-2 UEFI 自带异常处理流程

基于上面的原因，重新设计一种异常处理方式。首先设置异常发生后跳转到的代码位置，然后当运行的程序发生错误，调用 `DEBUG_ASSERT` 时，`ASSERT` 调用

跳转函数，跳出错误代码段，跳至之前设置的位置，从而使程序能够继续执行下去。本课题设计的异常处理机制如图 3-3 所示。

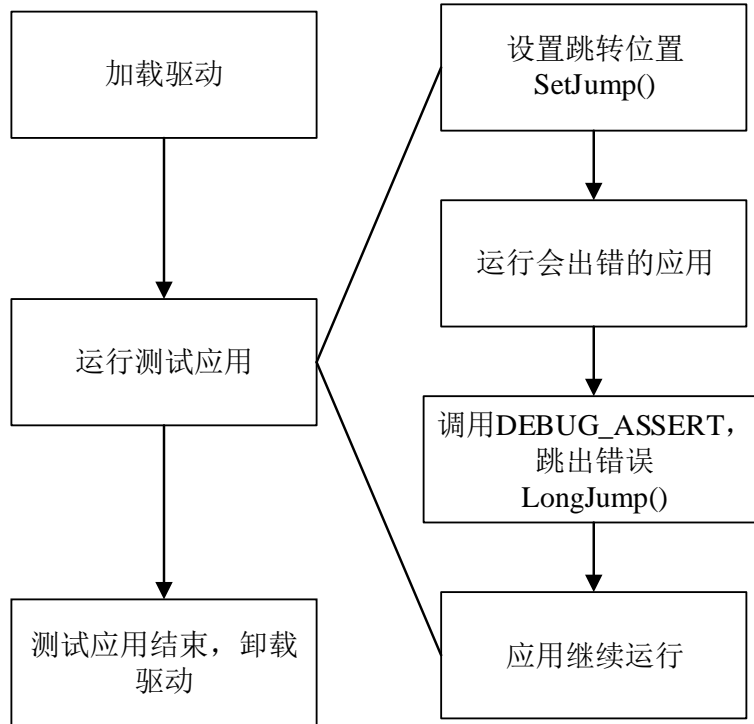


图 3-3 装载异常处理 Driver 后的异常处理流程

在 UEFI Shell 测试过程中,常常要运行 nsh 脚本来对 Shell 的各种命令进行检测。这样的测试脚本中往往包含了大量的 Shell 命令,当运行到某一个命令,发生错误时,Shell 被挂住,之前运行的数据都无法得到保持,并且之前运行的时候都被浪费了。这时候唯一的解决办法就是重启 Shell。这是一种成本过高且不合理的异常处理方式。装载 Driver 后,再碰到错误时,shell 不会挂住,而是会跳出当前脚本或当前错误,继续执行后面的命令。

3.3 UEFI Driver 的模型分析与开发流程分析

3.3.1 句柄和协议

UEFI 驱动程序模型使用句柄代表设备,每个设备对应有自己的句柄,句柄由一个或多个协议组成^[45]。协议是一个以 128 bit 的全局唯一标识符 GUID (Globally

Unique Identifier) 命名的结构体, 是一些指针和数据结构体或者规范定义的接口函数指针的集合, 协议代表设备提供的一类服务, 服务的具体功能在设备驱动程序(以下简称驱动)中实现。开发者首先找到指定设备句柄上挂载的指定协议, 再通过协议提供的接口访问设备驱动中实现服务的功能函数, 对设备进行操作。

图 3-4 显示了由 UEFI 驱动程序产生的句柄数据库中的一个单一的句柄和协议。一个 UEFI 驱动程序可以产生一个或多个协议, 这取决于驱动程序的复杂程度。这个协议由一个 GUID 和一个协议接口结构组成。协议接口结构本身仅仅包括协议函数的指针。通常, 产生一个协议接口的 UEFI 驱动程序还需要维护额外的私有数据段。协议函数实际上包含在 UEFI 驱动程序里。

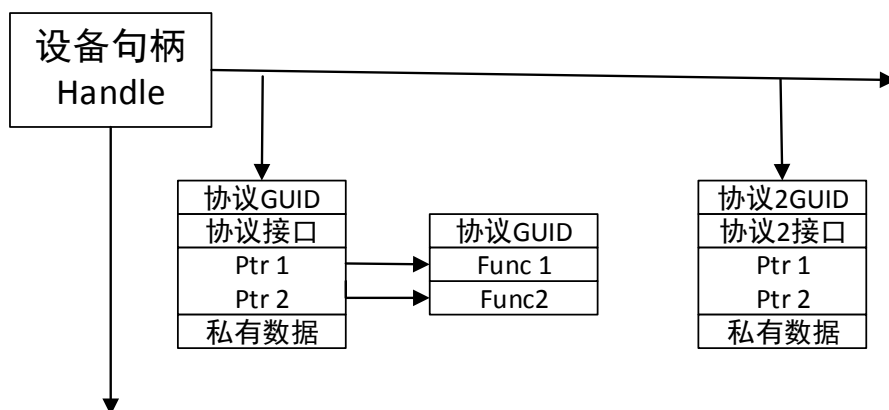


图 3-4 句柄协议结构图

1) Handel Database 句柄数据库包括了句柄和协议。

句柄数据库是一个由基于 UEFI 的固件来维护的对象的中心资料储存库。句柄数据库是一个 UEFI 句柄列表, 而每一个 UEFI 句柄通过一个唯一的句柄号来识别, 这个句柄号通过系统固件来维护。一个句柄号给句柄数据库的入口提供一个“关键字”。句柄数据库的每一个入口都是一个或多个协议的集合。由 GUID 来命名的被安装在一个 UEFI 句柄上的协议的类型决定了句柄的类型。一个 UEFI 句柄可以描述以下组件:

(1) 象 EFI 驱动程序和 EFI 应用程序这样一类的可执行映像

(2) 象网络控制器和硬盘驱动器 (hard drive partitions) 这样一类的设备

(3) 象 EFI 还原和 EBC 虚拟机这样一类的 EFI 服务

2) UEFI 的可扩展性在很大程度上依赖于协议。UEFI 驱动程序往往会与 UEFI 协议混淆。虽然它们的关系很密切,但是它们确实是不同的。一个 UEFI 驱动程序是一个安装了各种协议的多种句柄来完成工作的可执行映像。

UEFI 协议是一组由函数指针和数据结构块,或者由规范定义的 API 构成的集合。至少,规范必须定义一个 GUID。这个 GUID 号是协议真正的名字;例如 LocateProtocol 这样的启动服务使用这个 GUID 在句柄数据库中寻找所需要的协议。协议通常包括一套例程和/或数据结构,称之为协议接口结构。接下来的代码序列是来自 UEFI 规范的一个协议定义的例子。请注意它是怎样定义两个函数和一个数据段的。

3) 标签 GUID

一个协议可能仅由一个 GUID 组成。在这种情况下, GUID 被称为标签 GUID (tag GUID)。这种协议非常有用,例如用某种方法把一个设备句柄标记为特定的,或允许其它 UEFI 映像通过查询系统中协议的 GUID 所附属的设备句柄就能很容易地找到设备句柄。EDK 就使用 HOT_PLUG_DEVICE_GUID 去标记设备句柄,这个设备句柄描述诸如 USB 这样的来自于热插拔总线的设备。

3.3.2 驱动程序模型执行流程分析

UEFI 驱动程序模型是一种用于简化设备驱动设计和执行的机制,遵循驱动程序模型规范的 UEFI 驱动的可执行镜像大小会得到有效的减小。UEFI 驱动模型的执行流程如图 3-5 所示。

驱动程序模型采用 UEFI 驱动载入、连接的形式来进行硬件的辨识、控制及系统资源掌控。在 DXE 阶段,系统调用引导服务的 LoadImage () 函数将驱动镜像文件加载到内存中,调用 StartImage () 函数执行驱动的入口函数来启动驱动。遵循模型规范的设备驱动在入口函数的初始化中不涉及任何硬件操作,仅仅实现驱动绑定协议 (Driver Binding Protocol), 协议包含 3 个接口函数: Support ()、Start () 和

Stop ()。

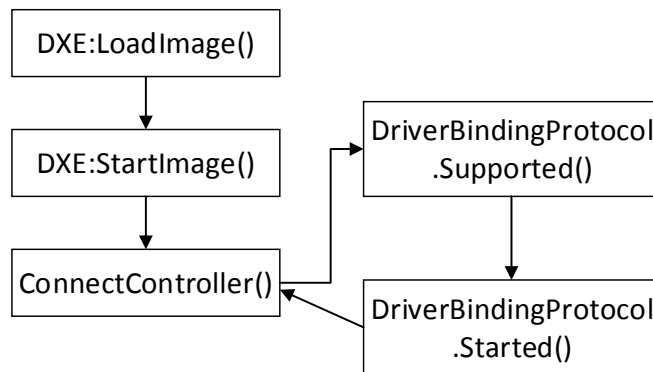


图 3-5 UEFI 驱动程序模型执行流程

(1)supported ():这个接口用于检测 Driver 是否支持相应的硬件,检查与 Driver 相关的控制器句柄是否是这个硬件设备相符合的句柄。检查通过就表明有某个设备 Driver 能处理控制器句柄所代表的硬件设备。检查成功时则返回 EFI_SUCCESS 状态。否则表明找改硬件设备不到对应的 Driver,返回 EFI_UNSUPPORTED 状态,。

(2) start ():这个接口用来启动加载 Driver 程序。这样硬件设备就可以挂上系统,然后硬件设备可以开始工作。该服务接口将通过驱动来打开 I/O Protocol 并安装设备的 I/O 抽象协议到句柄上去。

(3) stop ():这个接口用来卸载驱动程序,它其实就是 start () 的反过程,它将 start () 接口加载的驱动卸载并释放其创建的各种内容资源,把 start()打开的协议关掉。关闭协议的顺序应该和 start () 中打开协议的顺序相反。

要注意,编写 UEFI 驱动的要点是:

- (1) 接口点的时候不要去处理硬件(驱动模型的要求)
- (2) 复杂的 I/O 操作放到 Start () 和 Stop () 中实现
- (3) Start () 和 Stop () 的操作互相对应,如

InstallProtocolInterface ()	UninstallProtocolInterface ()
OpenProtocol ()	CloseProtocol
AllocatePages ()	FreePages ()
AllocatePool ()	FreePool ()

(4) Entry () 和 Unload () 互相对应。

3.3.3 UEFI Driver 开发流程分析

UEFI BIOS 主要使用 C/C++ 语言开发, 具有规范化、模块化的优点, 不仅解决了传统 BIOS 使用汇编开发的弊端, 功能也超出了 BIOS 的范围, 扩展延伸为一个强大的微型操作平台。UEFI BIOS 可以支持多种操作系统引导之前的底层应用, 承担起配置、调试、管理、网络等多种扩展功能。

UEFI 驱动基本的开发流程包括 (UEFI 应用的开发流程也类似):

(1) 建立 UEFI 开发环境, 包括配置好 Visual Studio, 设置系统环境变量, 下载 EDK 和使用 nmake 编译 Nt32/Duet 仿真环境等。

(2) 分析 UEFI 驱动的类型, 实现其中私有数据结构 (Private Data Structure, PDS), PDS 中包括: 驱动的签名, 控制器或者子驱动的句柄, 消费的协议接口, 产生的协议接口以及私有的数据域和服务等。

(3) 实现驱动提供的协议, 添加对应的包含文件, 检查该协议的实例是否已经加入到 PDS 中, 并发布这个协议, 实现 Start () 和 Stop () 等协议。

(4) nmake 编译这个 UEFI 驱动

(5) 在 UEFI Shell 中运行这个编译好的驱动, 可以使用一些 UEFI Shell 命令帮助调试, 比如 load、map、drivers、dh 等。为了调试方便, 可以在 Nt32 仿真环境中运行编译好的驱动。

(6) 最后可以将调试成功的 UEFI 驱动烧入主板的 BIOS flash 芯片。

3.4 UEFI Driver 的总体结构设计

UEFI Driver 的功能实现主要包括两块, 一部分是硬件相关的部分, 这部分实现了功能接口, 用于驱动硬件设备, 为用户提供服务, 以 Protocol 的形式出现, 真正的实现了跳转功能; 另一部分是框架接口, 需要实现 Driver Binding Protocol, 主要是其 Start, Stop 两个接口, 这两个接口实现了驱动的安装与卸载。因此针对上面两部分接口, 设计了两个 Protocol 协议。其中一个是 TslInitInterface Protocol, 这个 Protocol 对应的是框架接口部分内容; 另一个是 DebugTestLibJumpBuffer Protocol, 实现的是

功能接口部分。异常处理 Driver 的的总体结构设计如图 3-6 所示。

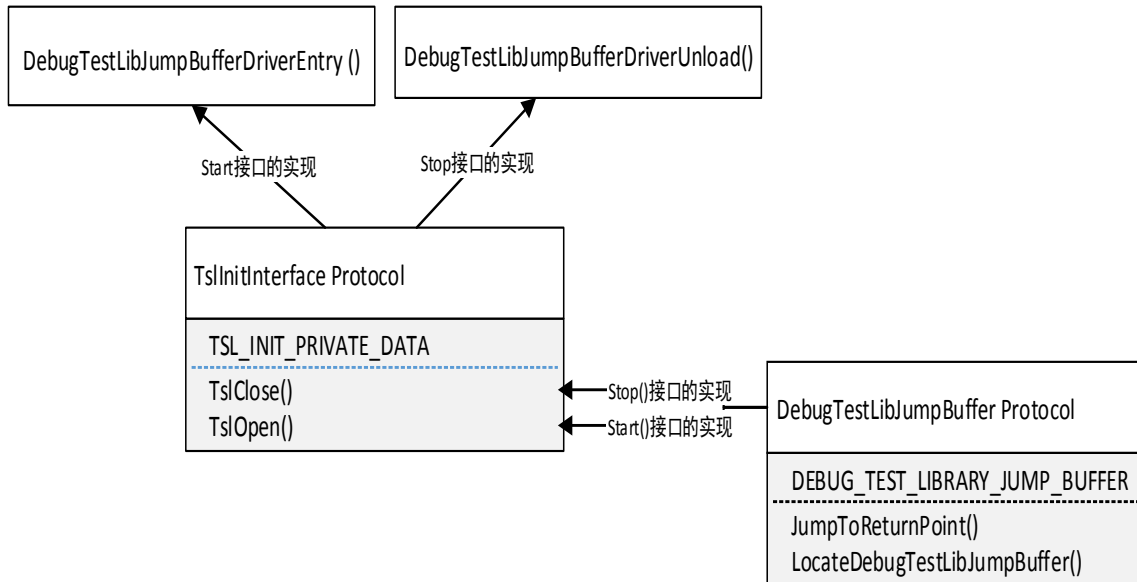


图 3-6 异常处理 Driver 的的总体结构设计图

3.5 本章总结

本章介绍了 Driver 的开发环境和运行环境。对异常处理 Driver 的功能进行了需求分析并进行了总体设计。分析了一个 UEFIDriver 的开发流程进行了分析，大概的介绍了 UEFI Driver 模型的一些概念。

4 UEFI 的异常处理 Driver 的详细设计与实现

4.1 Driver 的 Inf 文件设计

INF 文件全名为 EDK II Module Information File, 它描述了这个 Driver 的属性及依赖的其他文件等信息。在 Build 的时候, 一定需要这个文件, 它帮助自动生成 makefile 的配置文件, 在 inf 文件中主要包括以下部分:

- (1) Driver 的基本信息和入口函数信息, [Defines]部分
- (2) 库目录的路径, [Packages]部分
- (3) 源文件的列表, [sources]部分
- (4) 库目录的路径, [libraryClasses]部分
- (5) Protocol 的列表, [Protocols]部分

以下是各部分的介绍:

- (1) [Defines]部分是必须要存在的。这个部分定义了 Build 阶段会用到的变量。

[Defines]

```
INF_VERSION           = 0x00010005
BASE_NAME             = DxeDebugTestLibJumpBuffer
FILE_GUID             = 21BBF956-56B4-4318-B037-45F1AE4C4726
MODULE_TYPE           = DXE_DRIVER //这里的定义表明这是个 DXE 阶段执行的 Driver
VERSION_STRING        = 1.0
EDK_RELEASE_VERSION   = 0x00090000
EFI_SPECIFICATION_VERSION = 0x00020000
//ENTRY_POINT 定义了入口函数是 DebugTestLibJumpBufferDriverEntry ( ) .
ENTRY_POINT           = DebugTestLibJumpBufferDriverEntry
UNLOAD_IMAGE          = DebugTestLibJumpBufferDriverUnload
```

- (2) [sources]部分包括了 UEFI 目录中的.c 和.h 文件的文件名。这个部分中的

文件将是 Driver 功能的实现文件。

[Sources.common]

DebugTestLibJumpBuffer.c

CommonHeader.h

(3) [Packages]部分列出了 Driver 会用到的所有的声明文件。当这个部分中列出了任意文件时, 那 MdePkg/MdePkg.dec 也必须被包括。因为 MdePkg 包含 EDK II 构建系统所需的编译信息。

[Packages]

MdePkg/MdePkg.dec

ShellTestPkg/ShellTestPkg.dec

(4) [libraries]部分包括 UEFI 驱动需要链接的库列表。

[LibraryClasses]

UefiBootServicesTableLib

UefiDriverEntryPoint

MemoryAllocationLib

DebugLib

BaseMemoryLib

BaseLib

(5) [Protocols]部分列出了模块开发中会用到的 Protocol, 每个 Protocol 都有一个全球唯一的 12 位 guid 值标识, 都在 DEC 文件中进行了定义。

[Protocols]

gEfiTslInitInterfaceGuid # PROTOCOL

ALWAYS_CONSUMED

gEfiDebugTestLibJumpBufferProtocolGuid # PROTOCOL

ALWAYS_CONSUMED

4.2 Driver 的框架接口设计与实现

主要是在加载 Driver 时，初始化 Driver 用到。包括四个函数：TslOpen（），TslClose（），DebugTestLibJumpBufferDriverEntry（），DebugTestLibJumpBufferDriverUnload（）。

（1）TslOpen（）函数用于申请变量内存（该变量用于存储跳转点的信息）和 DebugTestLibJumpBuffer Protocol 的 Start 接口的实现。

（2）TslClose（）函数用于释放申请的变量空间和 DebugTestLibJumpBuffer Protocol 的 stop 接口的实现。

（3）DebugTestLibJumpBufferDriverEntry（）函数是入口函数。加载 Driver 时，这个函数相当于 C++ 中 main 函数的功能。函数实现 TslInitInterface Protocol 的 Start 接口，安装 TslInitInterface Protocol 到某个 Handle，这个 TslInitInterface Protocol 实例会常驻内存，用于驱动的安装和卸载。

（4）DebugTestLibJumpBufferDriverUnload（）函数实现了 TslInitInterface Protocol 的 Stop 接口，把 TslInitInterface Protocol 实例从 Handle 上卸载，即实现了驱动的卸载。

UEFI 中 Protocol 的概念可以理解为 C++ 中的类。TslOpen（）和 TslClose（）可以理解为 TslInitInterface Protocol 的私有函数。

1) TslInitInterface Protocol 的私有数据结构的实现：

```
typedef struct {  
    UINT32 Signature;  
    EFI_HANDLE ImageHandle;  
    EFI_TSL_INIT_INTERFACE TslInit;  
} TSL_INIT_PRIVATE_DATA;
```

TSL_INIT_PRIVATE_DATA 数据结构中包括了句柄变量，以及下面这个数据结构 EFI_TSL_INIT_INTERFACE 的变量实例。

```
typedef struct {  
    UINT64 Revision;
```

```
EFI_GUID                                LibraryGuid;
EFI_TSL_OPEN                            Open;
EFI_TSL_CLOSE                           Close;
}EFI_TSL_INIT_INTERFACE;
```

EFI_TSL_INIT_INTERFACE 数据结构中包含了 DebugTestLibJumpBuffer Protocol 的 start(),stop()接口的函数指针和对应的 GUID 指针。

2) DebugTestLibJumpBufferDriverEntry () 函数

这个函数相当于 C++中 main()函数的概念。当 Driver 加载时，就会找到这个接口，这个接口初始化了 TslInitInterface Protocol，实现了数据初始化及协议绑定，是入口函数，具体实现如图 4-1 所示。

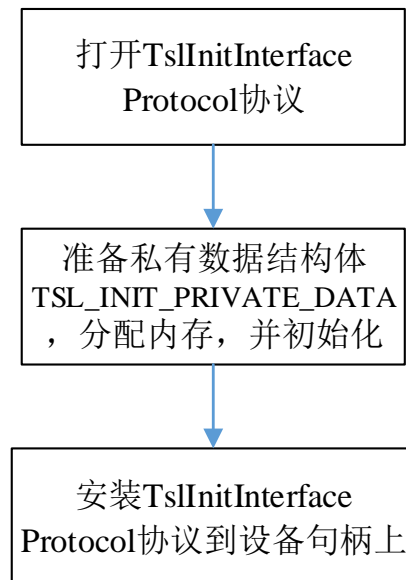


图 4-1 DebugTestLibJumpBufferDriverEntry () 函数的执行流程

(1) 使用函数 OpenProtocol () 打开所有需要的 TslInitInterface Protocol，OpenProtocol()函数是 UEFI 自己封装的函数接口，有特定的参数顺序，参数个数和特定的返回值。

```
Status = gBS->OpenProtocol (
    ImageHandle,
```

```
&gEfiTslInitInterfaceGuid,  
NULL,  
ImageHandle,  
NULL,  
EFI_OPEN_PROTOCOL_TEST_PROTOCOL  
);
```

(2) 如果 `OpenProtocol()` 返回错误, `OpenProtocol()` 出错时会返回 `EFI_ERROR` 状态。

```
if (!EFI_ERROR (Status)) {  
    return EFI_ALREADY_STARTED;  
}
```

(3) 分配并初始化要用到的数据结构, 这些数据结构包括驱动 `Protocols` 及其它相关的私有数据结构。如果分配资源时发生错误, 则关闭所有已打开的 `Protocols`, 释放已经得到的资源, 返回 `EFI_OUT_OF_RESOURCES`。

```
// Initialize the TslInit private data  
  
Private = (TSL_INIT_PRIVATE_DATA *) AllocateAlignedZeroPool (sizeof  
(TSL_INIT_PRIVATE_DATA), 16); //申请内存空间  
  
if (Private == NULL) { //如果申请内存空间失败, 则返回 error 状态  
    return EFI_OUT_OF_RESOURCES;  
}  
  
//为私有数据赋值  
  
Private->Signature = TSL_INIT_PRIVATE_DATA_SIGNATURE;  
Private->ImageHandle = ImageHandle;  
Private->TslInit.Revision = 0x10000;  
CopyGuid ( &Private->TslInit.LibraryGuid,  
&gEfiDebugTestLibJumpBufferProtocolGuid );
```

Private->TslInit.Open = TslOpen;

Private->TslInit.Close = TslClose;

(4) 用 InstallMultipleProtocolInterfaces () 安装驱动协议到 ControllerHandle。
返回安装的结果。

```
// Install TslInit protocol
```

```
return gBS->InstallMultipleProtocolInterfaces (
```

```
&ImageHandle, //找到指定的 control handle
```

```
&gEfiTslInitInterfaceGuid, //通过 Guid 值来确定 protocol
```

```
&(Private->TslInit),
```

```
NULL
```

```
);
```

3) DebugTestLibJumpBufferDriverUnload () 函数

这个函数实现了 TslInitInterface Protocol 的 stop()接口,即内存资源的释放以及协议的卸载,从而实现了驱动的卸载。具体的实现如图 4-2 所示。

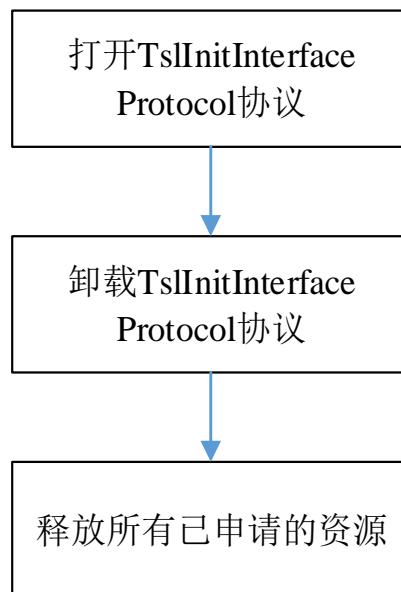


图 4-2 DebugTestLibJumpBufferDriverUnload () 函数的执行流程

(1) 使用函数 OpenProtocol () 打开所需要的 TslInitInterface Protocols。

```
// Open the TslInit protocol  
Status = gBS->OpenProtocol (   
ImageHandle,           //指定句柄  
&gEfiTslInitInterfaceGuid,  //指定的 GUID 值  
(VOID **) &TslInit,  
ImageHandle,  
NULL,  
EFI_OPEN_PROTOCOL_GET_PROTOCOL    //打开句柄的方式  
);
```

(2) 用 UninstallMultipleProtocolInterfaces () Uninstall 卸载所安装的 Protocols

```
if (!EFI_ERROR (Status)) {  
// Uninstall TslInit protocol  
Status = gBS->UninstallMultipleProtocolInterfaces (   
ImageHandle,           //需要卸载的句柄  
&gEfiTslInitInterfaceGuid,  //同过 GUID 值找到所要卸载的协议  
TslInit,  
NULL  
);  
if (EFI_ERROR (Status)) {    //如果卸载失败，则返回失败的错误原因  
return Status;  
}
```

(3) 释放所有已申请的资源。

```
Private = TSL_INIT_PRIVATE_DATA_FROM_THIS (TslInit);  
FreeAlignedPool (Private);
```

4) TslOpen () 函数的实现

这个函数实现了 DebugTestLibJumpBuffer Protocol 的绑定以及数据的内存分配，具体实现如图 4-3 所示。

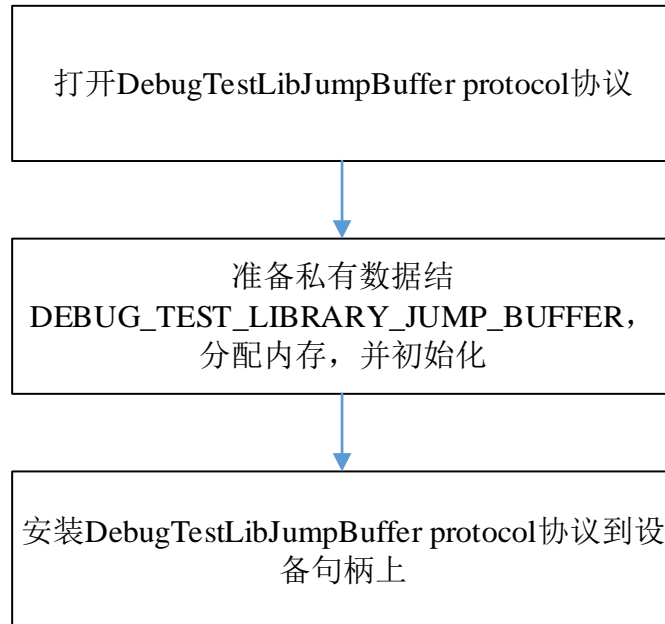


图 4-3 TslOpen () 函数的执行流程

(1) 使用函数 OpenProtocol () 打开 DebugTestLibJumpBuffer protocol. 如果出错则返回 EFI_ALREADY_STARTED, 表明协议已经启动装载了。DebugTestLibJumpBuffer protocol 实现了功能接口 LocateDebugTestLibJumpBuffer () 函数和 JumpToReturnPoint () 函数。

```
if (*LibHandle != NULL) {  
    Status = gBS->OpenProtocol (  
        *LibHandle,  
        &gEfiDebugTestLibJumpBufferProtocolGuid,  
        NULL,  
        TslPrivate->ImageHandle,  
        NULL,  
        EFI_OPEN_PROTOCOL_TEST_PROTOCOL
```

);

TslPrivate 是 DebugTestLibJumpBufferDriverEntry () 中实现的 TslInitInterface Protocols 的私有数据结构的实例。

(2) 分配并初始化要用到的数据结构, 这些数据结构包括驱动 DebugTestLibJumpBuffer protocol 及其它相关的私有数据结构。

如果分配资源时发生错误, 则关闭所有已打开的 DebugTestLibJumpBuffer protocol, 释放已经得到的资源, 返回 EFI_OUT_OF_RESOURCES。

```
DebugTestLibJumpBuffer = AllocateAlignedRuntimeZeroPool ( sizeof  
(DEBUG_TEST_LIBRARY_JUMP_BUFFER), 16);  
if (DebugTestLibJumpBuffer == NULL) {  
    return EFI_OUT_OF_RESOURCES;  
}
```

DebugTestLibJumpBuffer protocol 的私有变量 DEBUG_TEST_LIBRARY_JUMP_BUFFER 有四种类型, 分别是 IA32, IPF, X64, EBC.

(3) 用 InstallMultipleProtocolInterfaces() 安装 DebugTestLibJumpBuffer protocol 到 ControllerHandle。返回安装的结果。

```
return gBS->InstallMultipleProtocolInterfaces (  
    LibHandle,  
    &gEfiDebugTestLibJumpBufferProtocolGuid,  
    DebugTestLibJumpBuffer,  
    NULL  
);
```

5) TslClose () 函数

TslClose () 函数实现了内存资源的释放以及 DebugTsetLibJumpBuffer Protocol 的卸载。

具体的实现如图 4-4 所示。

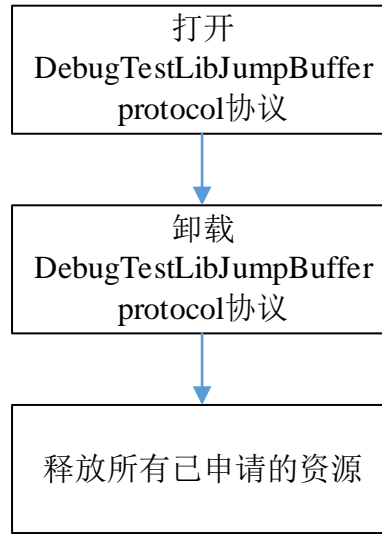


图 4-4 TslClose () 函数的执行流程

(1) 使用函数 OpenProtocol () 打开所需要的 DebugTestLibJumpBuffer protocol。

// Open the DebugTestLibJumpBuffer protocol to perform the supported test.

```
Status = gBS->OpenProtocol (
    LibHandle,
    &gEfiDebugTestLibJumpBufferProtocolGuid,
    (VOID **) &DebugTestLibJumpBuffer,
    TslPrivate->ImageHandle,
    NULL,
    EFI_OPEN_PROTOCOL_GET_PROTOCOL
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

(2) 用 UninstallMultipleProtocolInterfaces () 来卸载所安装的 DebugTestLibJumpBuffer protocol.

// Uninstall TslInit protocol

```
Status = gBS->UninstallMultipleProtocolInterfaces (
```

```
LibHandle,  
&gEfiDebugTestLibJumpBufferProtocolGuid,  
DebugTestLibJumpBuffer,  
NULL  
);  
if (EFI_ERROR (Status)) {  
    return Status;  
}
```

(3) 释放所有已申请的资源。

```
if (DebugTestLibJumpBuffer != NULL) {  
    FreeAlignedPool ((VOID *) DebugTestLibJumpBuffer);  
    DebugTestLibJumpBuffer = NULL;  
}
```

4.3 Driver 的功能实现接口的设计与实现

测试 Application 会调用 Driver 提供的接口来实现异常中断处理。主要包括两个函数接口：LocateDebugTestLibJumpBuffer（）函数和 JumpToReturnPoint（）函数。LocateDebugTestLibJumpBuffer（）函数用于读取跳转所需要的内容。

JumpToReturnPoint（）函数是跳转函数。即恢复保存的 CPU 的信息，实现跳转。

1) LocateDebugTestLibJumpBuffer（）函数的实现

LocateDebugTestLibJumpBuffer（）函数读取了 DebugTestLibJumpBuffer protocol 中的私有变量 DEBUG_TEST_LIBRARY_JUMP_BUFFER 的信息，DEBUG_TEST_LIBRARY_JUMP_BUFFER 即保存了跳转所需要的内容。

```
// Locate JumpBuffer for saving long jump context from Protocol.  
Status = gBS->LocateProtocol (  
    &gEfiDebugTestLibJumpBufferProtocolGuid,  
    NULL,
```

```
(VOID **) &DebugTestLibJump  
);  
if (EFI_ERROR (Status)) {  
    return NULL;  
}  
return DebugTestLibJump;
```

2) JumpToReturnPoint () 函数

2) JumpToReturnPoint () 函数调用了 LongJump () 接口从而实现了跳转出错代码段。具体的实现如图 4-5 所示。

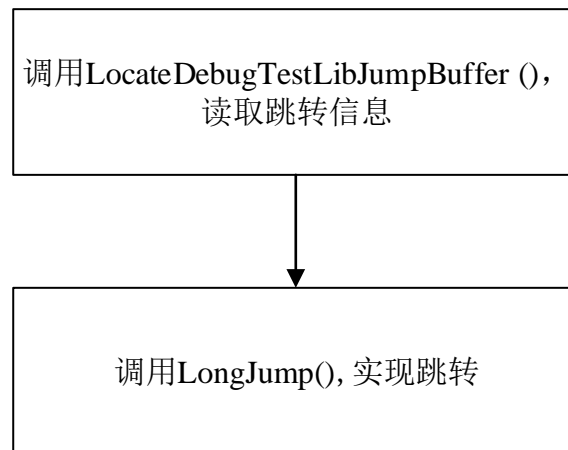


图 4-5 JumpToReturnPoint () 函数的执行流程

(1) 通过 LocateDebugTestLibJumpBuffer () 函数读取保存的跳转信息。

```
// Get Jump Buffer Context.  
DebugTestLibJumpBuffer = ( BASE_LIBRARY_JUMP_BUFFER * )  
LocateDebugTestLibJumpBuffer ();  
if (DebugTestLibJumpBuffer == NULL) {  
    return ;  
}  
  
(2) 调用 UEFI 内部函数 LongJump () 函数，进行跳转。  
LongJump (DebugTestLibJumpBuffer, 1);
```

4.4 本章小结

本章主要是对 Driver 设计及实现进行了分析介绍。介绍了 Driver 的 Inf 文件。具体说明了 Driver 的功能接口和框架接口的设计与实现流程及具体实现方法。本章还对重要代码进行了详细的解释。

5 Driver 在 UEFI Shell 自动测试中的应用

5.1 UEFI Shell 自动测试介绍

UEFI Shell 是一个类似于 Windows Dos 的用户交互平台。用户可以往窗口里输入命令，从而得到平台反馈的一些相关信息。UEFI Shell 有几十条命令，每条命令都有不同的语法规则，因此针对 UEFI Shell 的测试，设计了一千多个基本测试用例。要运行这么多个测试用例，并比对测试结果，采用纯手工的测试方法是及其浪费人力和物力的。

为了实现对 UEFI Shell 的高效率测试，设计了一套自动化测试的方法。测试脚本运行时，会依次运行各个测试用例。但是用例的设计不一定一次性就能都完善成功，或者当 Shell 有一定错误时，脚本运行就会发生异常。如运行到第 800 个测试用例时，自动化测试脚本发生异常中断，此时若是按照 UEFI 自带的异常中断处理机制，测试脚本就会停下来，机器死机。这时候只能强制重启机器。这样会造成多方面的负面影响：(1)数据遗失。由于是强制关机，之前得到的测试结果肯定没有得到妥善的保存，之前的测试就作废了。(2)浪费时间。假设跑一个跑一个用例需要 1 分钟，800 个用例就是 10 多个小时，之前十多个小时也就浪费了，这对测试进度的影响是巨大的。(3)浪费资源。一般测试时，机器资源是很紧张的，由于这次测试的异常中断，机器就白跑了。

为了解决自带异常中断机制导致的一系列问题，在自动测试过程中会调用本课题设计实现的异常中断处理驱动。本课题设计实现的异常中断处理驱动会在碰到异常中断时，跳出异常代码段，这样脚本会自动结束运行。这种情况下，机器不会死机。这样，只需要在修改异常的测试用例，或跳过这个导致异常的用例，继续运行剩下的 200 个测试用例即可。这样大大提高了测试的效率与可行性。

5.1.1 自动化测试框架介绍

首先进入 UEFI Shell，运行配置环境脚本。配置脚本实现了 UEFI Shell 自动化测试的脚本的环境配置，如要运行的用例个数，或指定用例。

然后运行自动化测试脚本。测试脚本会调用已编译好的测试应用。测试应用会去调用已存在于特定文件夹下的用例文件。在某个用例不完善导致的异常中断，或是 UEFI Shell 本身的问题导致的异常中断时，会调用本课题设计的异常中断处理驱动。

接下来调用结果 比对脚本。结果比对脚本，会将运行测试用例得到的现实结果与之前保存的预期结果进行比较。如果现实结果与预期结果一致，那么这个测试点就通过了。如果不一致，则说明这个测试点的 UEFI Shell 有问题或者是这个测试用例有问题。

最后会把运行的结果记录下来，生成测试报告。测试报告会包含测试用例的编号、测试结果和最后的测试通过率。

如图 5-1 为自动化测试框架执行顺序图。

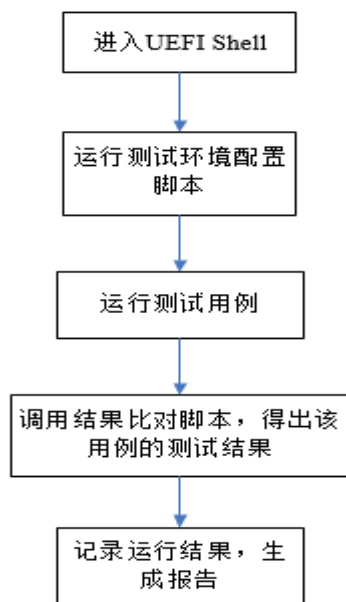


图 5-1 自动化测试框架执行流程

5.1.2 测试用例的设计

UEFI Shell 所有的命令都可以通过输入命令名来调用命令提示符。而外部命令，它们必须放在在一个文件系统中，因此要运行它们的用户需要至少有一个映射文件系统把这些外部命令文件系统下。

如图 5-2 是 UEFI Shell 的命令图。

```
Shell> help
? - Displays the EFI Shell command list or verbose command help
alias - Displays, creates, or deletes EFI Shell aliases
attrib - Displays or changes the attributes of files or directories
cd - Displays or changes the current directory
cls - Clears standard output and optionally changes background color
comp - Compares the contents of two files
connect - Connects one or more EFI drivers to a device
cp - Copies one or more files or directories to another location
date - Displays or changes the current system date
dblk - Displays one or more blocks from a block device
devices - Displays the list of devices managed by EFI drivers
devtree - Displays the EFI Driver Model compliant device tree
dh - Displays EFI handle information
disconnect - Disconnects one or more EFI drivers from a device
dnmem - Displays the contents of memory
dmpstore - Displays all EFI NVRAM variables
drivers - Displays the EFI driver list
drvcfg - Invokes the Driver Configuration Protocol
drvdiag - Invokes the Driver Diagnostics Protocol
echo - Controls batch file command echoing or displays a message
edit - Full screen editor for ASCII or UNICODE files
eficompress - Compress a file
efidecompress - Decompress a file
err - Displays or changes the error level
exit - Exits the EFI Shell environment
for - Executes commands for each item in a set of items
goto - Forces batch file execution to jump to specified location
guid - Displays all registered EFI GUIDs
help - Displays the EFI Shell command list or verbose command help
hexedit - Full screen hex editor
if - Executes commands in specified conditions
ifconfig - Modify the default IP address of UEFI network stack
ipconfig - Displays or modifies the current IP configuration
```

图 5-2 UEFI Shell 命令图

如 cd 命令，即跳入某个文件夹。如果 cd 后跟着某一个文件夹名字或路径，如果该文件夹名或路径，则跳转至该文件夹。如果该文件夹名或路径不存在，则显示错误代码及错误信息，说明该文件夹或路径不存在。如果 cd 后跟特殊字如“.”，“..”，“\”，

会实现特殊的功能，如跳转到根目录或上级目录等。假设某个用例为”cd fs0:”，现实运行结果是 fs0:\>。将预期运行结果 fs0:\>保存在一个文件中。然后将现实运行结果与预期运行结果进行比对，将比对结果保存。

5.1.3 自动化测试应用的设计与实现

1) 测试 Application 的设计

在测试 Application 的 inf 文件中指定入口函数 ShellCEntry ()。

然后 ShellCEntry () 函数调用 ShellAppMain () 函数。

ShellCEntry () 函数相当于 C++中 main 函数的概念。函数会判断当前的 shell 是 shell1.0 还是 shell2.0，然后根据 shell 的版本来给 ShellAppMain () 函数传入参数。

ShellAppMain () 函数首先会定位 Driver 的 protocol，然后调用 SET_RETURN_POINT () 函数来设置保存跳转的位置，接着就会执行有错误的代码，当发生错误异常时，会自动调用 DEBUG_ASSERT () 接口。DEBUG_ASSERT () 接口会通过 Driver 的 DebugTestLibJumpBuffer protocol 的 JumpToReturnPoint () 函数实现跳转。

2) 测试 Driver 的 Application 的实现

(1) ShellCEntry () 函数的实现

UEFI 基本 Application 的 main 函数的返回值类型为 EFI_STATUS；它有两个参数，*.efi 文件加载到内存后称为 Image，ImageHandle 用来描述、访问、控制此 Image。第二个参数是 SystemTable，它是我们的程序同 UEFI 内核打交道的桥梁，通过它我们可以使用 UEFI 提供的各种服务，如 Boot Services 和 Runtime Services。SystemTable 是 UEFI 内核中的一个全局结构体。函数实现流程如图 5-3 所示。

(1) 通过 gST->BootServices->OpenProtocol 启动测试 Application。

```
// Add for ShellLib Test
```

```
Status = gST->BootServices->OpenProtocol (ImageHandle,  
&gEfiShellParametersProtocolGuid,  
(VOID **) &EfiShellParametersProtocol,  
ImageHandle,
```



```
NULL,  
EFI_OPEN_PROTOCOL_GET_PROTOCOL  
);
```

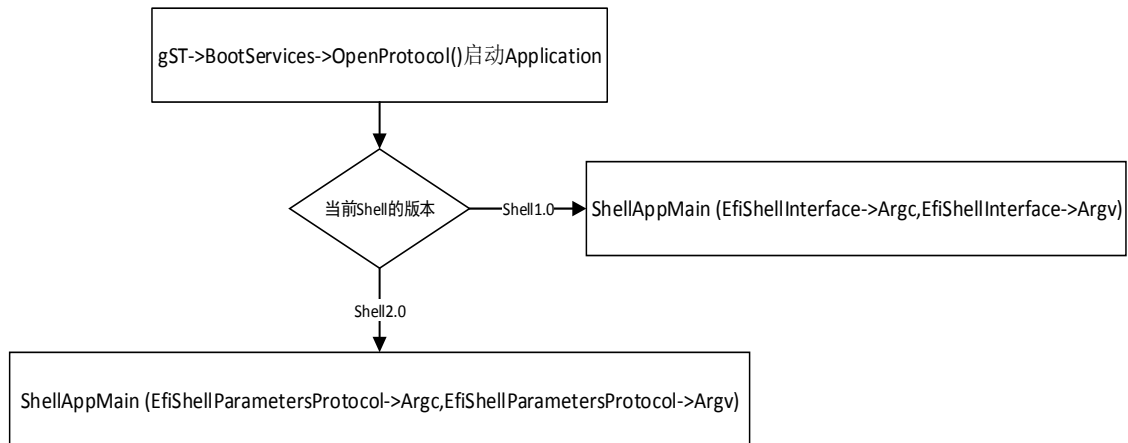


图 5-3 ShellCEntry () 函数的执行流程

(2) 判断当前 shell 的 Version, 给 ShellAppMain () 接口相对应的参数。

当用的是 shell1.0 时, ShellAppMain (

EfiShellInterface->Argc,

EfiShellInterface->Argv

);

当用的是 shell 2.0 时, ShellAppMain (

EfiShellParametersProtocol->Argc,

EfiShellParametersProtocol->Argv

);

3) ShellAppMain () 函数的实现

如图 5-4 介绍了 ShellAppMain () 函数的实现。

(1) 调用 LocateDebugTestLibTsl () 函数,将 Driver 加载

Status = LocateDebugTestLibTsl ();

(2)调用 SET_RETURN_POINT ()函数,设置 DebugTestLibJumpBuffer protocol

的私有变量 `DebugTestLibJumpBuffer`，即跳转信息。

`SET_RETURN_POINT (ReturnValue) ;`

(3) 执行一段会导致异常的代码，出现异常之后会调用 `DEBUG_ASSERT ()` 函数。`DEBUG_ASSERT()`函数会调用 `DebugTestLibJumpBuffer protocol` 的私有函数接口 `JumpToReturnPoint ()`，从而实现跳转。

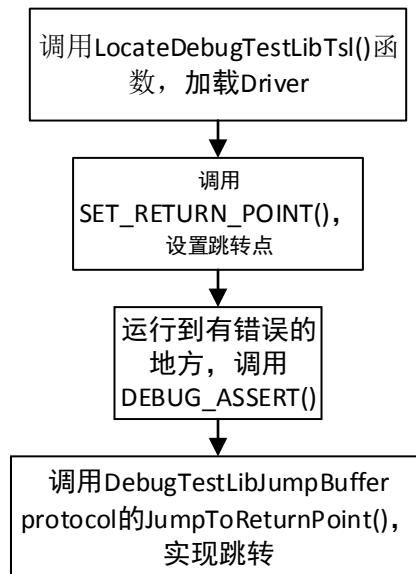


图 5-4 `ShellAppMain ()` 函数的执行流程

3) `LocateDebugTestLibTsl ()` 函数分析

下图 5-5 介绍了 `LocateDebugTestLibTsl ()` 函数的实现。

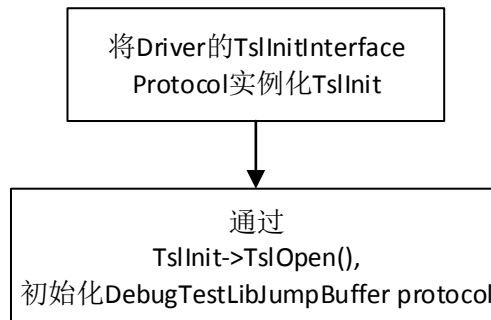


图 5-5 `LocateDebugTestLibTsl ()` 函数的执行流程

(1) 通过 gBS->LocateProtocol () 函数将 Driver 的 TslInitInterface Protocol 实例化。

```
Status = gBS->LocateProtocol (
    &gEfiTslInitInterfaceGuid,
    NULL,
    (VOID **) &TslInit
);
```

(2) 通过调用 TslInitInterface Protocol 的实例 TslInit 的私有函数 TslOpen (), 来初始化 DebugTestLibJumpBuffer protocol。

```
// Open the support file
Status = TslInit->Open (
    TslInit,
    (EFI_HANDLE *) &gImageHandle,
    NULL
);
if (EFI_ERROR (Status)) {
    Print (L"LocateDebugTestLibTsl:TslInit->Open failed. \n");
    return Status;
}
return EFI_SUCCESS;;
}
```

4) SET_RETURN_POINT () 函数的代码分析

如图 5-6 介绍了 SET_RETURN_POINT () 函数的实现。

(1) 调用 DebugTestLibJumpBuffer protocol 的私有函数接口 Locate DebugTestLibJumpBuffer ()

```
VOID          *DebugTestLibJumpBuffer;

DebugTestLibJumpBuffer = LocateDebugTestLibJumpBuffer ( );
```

//为 DebugTestLibJumpBuffer 初始化。

(2) 调用 SetJump () 函数实现跳转内容保存

SetJumpFlag = SetJump (DebugTestLibJumpBuffer) ;

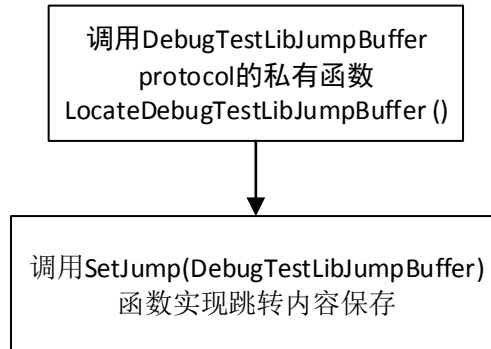


图 5-6 LocateDebugTestLibTsl () 函数的执行流程

5.2 搭建运行及测试环境

由于 UEFI 是系统底层软件，直接在真实环境中开发与调试相对困难，UEFI EDKII 工具包提供了一组仿真器和模拟器，方便进行程序的快捷开发与验证。本次采用的是 Nt32 仿真器。

进行 UEFI 驱动或应用开发可在不同的平台下进行，本次在 Windows 平台下进行实现和测试。采用了 Windows8 操作系统和 Visual Studio 编译器。

(1) 安装 Visual Studio 2008.

(2) 从网上下载 EDKII 的代码包 AllPackagesDev 后，将 AllPackagesDev 文件夹拷至桌面，路径为 C:\Users\tiano\Desktop\AllPackagesDev。

(3) 将代码文件夹 ShellTestPkg，拷入文件夹 AllPackagesDev 中

(4) 以管理员方式打开 Visual Studio 2008 Command Prompt 窗口

5.3 编译运行以及验证

5.3.1 编译环境配置

(1) 在窗口中输入 cdC:\Users\tiano\Desktop\AllPackagesDev

(2) 在窗口中输入 edksetup.bat.

edksetup.bat 是一个批处理文件。目的是根据当前采用了哪种平台和哪种编译工具，以及当前的工作目录等信息进行配置编译环境的工作。编译环境配置图如图 5-7 所示。

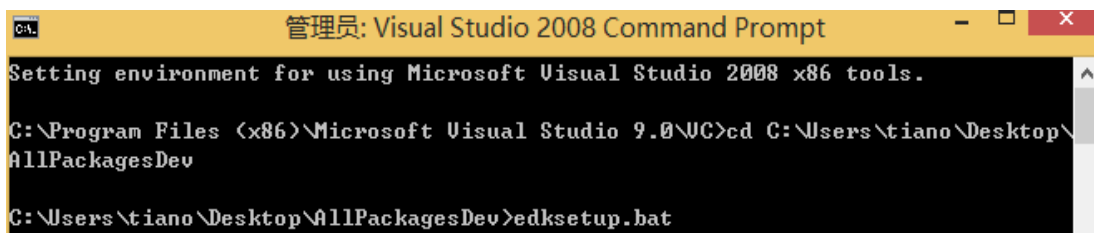


图 5-7 编译环境配置图

5.3.2 编译文件

在窗口输入 `build -p ShellTestPkg/ShellTestPkg.dsc -a IA32 -t VS2008x86` .编译结束后可在 `C:\Users\tiano\Desktop\AllPackagesDev\Build\ShellTestPkg\DEBUG_VS2008x86\IA32` 下找到文件 `DxeDebugTestLibJumpBuffer.efi` 和 `ShellLibTest.efi`。

5.3.3 在 Nt32 虚拟平台下运行

(1) 在窗口输入 `build run build run -t VS2008x86 -p Nt32pkg\Nt32pkg.dsc` 进入 Nt32 虚拟平台。会自动跳转到 UEFI Shell。UEFI Shell 的界面图如图 5-8 所示。

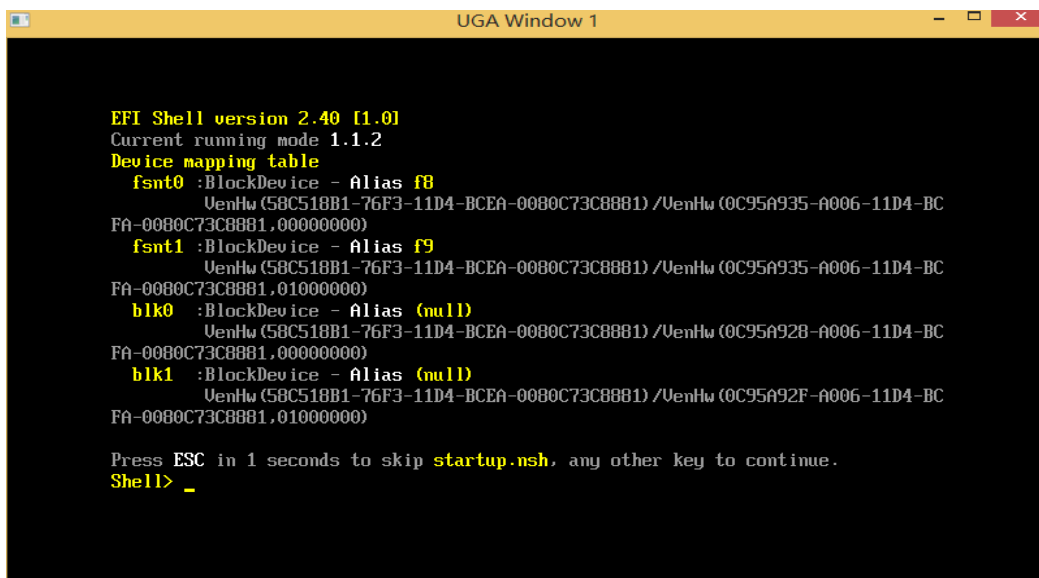


图 5-8 UEFI Shell

(2) 运行测试脚本

在窗口输入”fsnt0:” “cd ShellLibTest” “ShellLibTestAppTestScript-chenshuyi.nsh”
脚本的文件夹路径是：C:\Users\tiano\Desktop\AllPackagesDev
\Build\NT32IA32\DEBUG_VS2008x86\IA32\ShellLibTest。

输入命令后运行截图如图 5-9 所示。

```
Press ESC in 5 seconds to skip startup.nsh, any other key to continue.
Shell>
Shell> fsnt0:

fsnt0:\> cd ShellLibTest

fsnt0:\ShellLibTest> ls
Directory of: fsnt0:\ShellLibTest

12/13/14 03:32p <DIR>          0 .
12/13/14 03:32p <DIR>          0 ..
12/13/14 03:25p              6,304 DxeDebugTestLibJumpBuffer.efi
12/13/14 03:29p              262 ShellIsFileConfTest.log
12/13/14 03:29p              625 ShellIsFileFuncTest.log
12/13/14 03:25p          137,824 ShellLibTest.efi
12/13/14 03:28p              120 ShellLibTestAppTestScript-chenshuyi.nsh

5 File(s)      145,135 bytes
2 Dir(s)

fsnt0:\ShellLibTest> ShellLibTestAppTestScript-chenshuyi.nsh_
```

图 5-9 运行脚本截图

5.3.4 运行结果验证

(1) 当运行的 ShellLibTestAppTestScript-chenshuyi.nsh 脚本不调用
DxeDebugTestLibJumpBuffer.efi 这个本课题设计的错误中断处理 Driver。
ShellLibTestAppTestScript-chenshuyi.nsh 脚本的代码如图 5-10 所示。

```
*C:\Users\tiano\Desktop\AllPackagesDev\Build\NT
文件(F) 编辑(E) 搜索(S) 视图(V) 格式(M) 语言(L) 设置(T)
ShellLibTestAppTestScript-chenshuyi.nsh
1 #
2 # Automatic Test Script for ShellLib
3 #
4
5 ShellLibTest.efi ShellIsFile
6
7
```

图 5-10 ShellLibTestAppTestScript-chenshuyi.nsh 脚本的内容

运行结果：

不能跳出错误，平台挂掉。碰到这种情况，只能把平台关闭再重启，十分不方便。UEFI Shell 挂住的界面截图如图 5-11 所示。

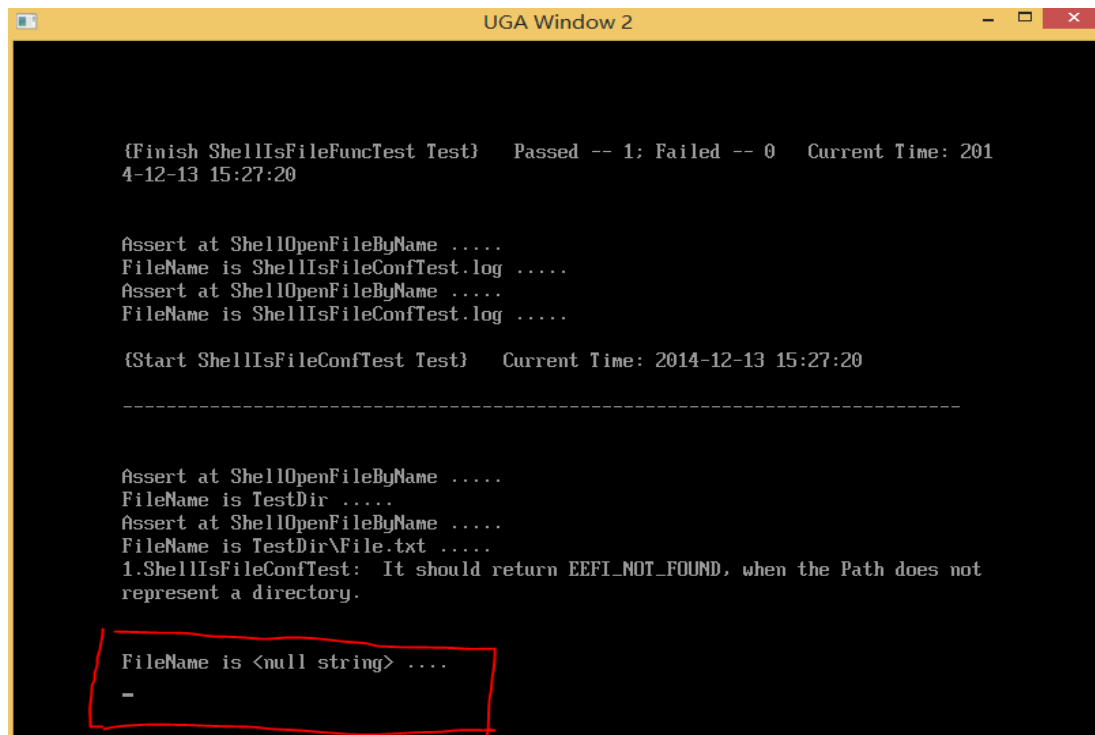


图 5-11 不加载驱动时运行错误截图

(2) 当运行的 `ShellLibTestAppTestScript-chenshuyi.nsh` 脚本调用 `DxeDebugTestLibJumpBuffer.efi` 这个本课题设计的错误中段 Driver 时，脚本代码如图 5-12 所示。

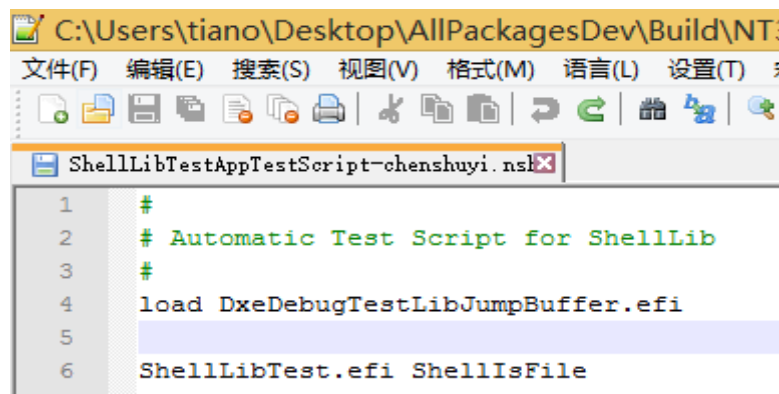
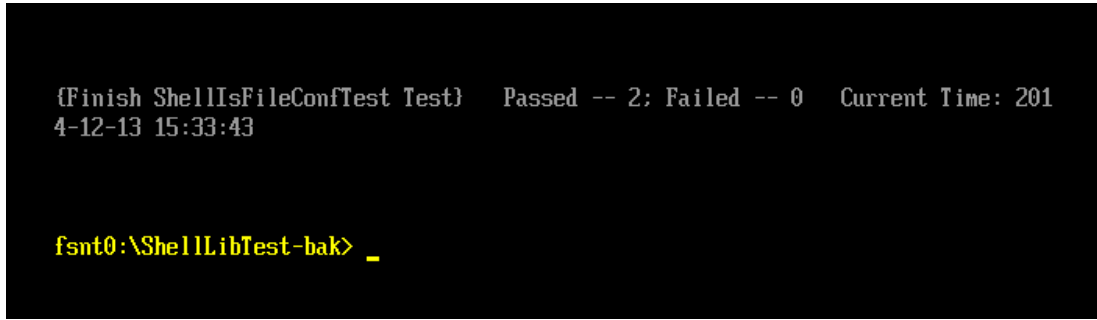


图 5-12 加载驱动脚脚本截图

运行结果：

碰到错误时，平台不会挂住停止，而会跳出当前代码段继续执行后面的代码，直到所有程序运行结束。下图 5-13 为成功跳出错误代码段的运行截图。



```
{Finish ShellIsFileConfTest Test} Passed -- 2; Failed -- 0 Current Time: 2014-12-13 15:33:43

fsnt0:\ShellLibTest-bak> _
```

图 5-13 加载驱动后运行截图

5.4 本章小结

本章详细的介绍了 UEFI Shell 的自动化测试框架。本章还介绍了 UEFI Shell 的自动化测试用例，以及对本课题设计的异常中断处理 Driver 的调用，以及测试用例程序的编译方法。通过对比分析 UEFI 自带异常中断处理的运行效果与本文设计的异常中断处理驱动加载后的运行效果，见证了本文驱动的优越性。本章还介绍了具体测试环境的搭建。

6 总结与展望

6.1 全文总结

UEFI 是用面向对象来设计，采用模块化设计思路来组织驱动和应用程序的，模块之间用标准协议通信，它是能够用高级语言和现代的软件工程方法设计的接口程序。而随着 UEFI 得到越来越多的认可，被更多厂商认同并采用，必然会有更多的人参与到 UEFI 驱动和 UEFI 应用程序的开发。但是 UEFI 自带的异常中断处理机制的不完善，为 UEFI 驱动和应用的开发带来了短板。

本文设计和实现了一种新的 UEFI 异常中断处理驱动，很好的解决了原有机制的完善。原始的 UEFI 异常中断处理机制会导致机器死机，这样会造成数据丢失，时间机器成本浪费等问题。而本文设计的这个新驱动，可以在碰到异常时，自动跳出当前异常，从而解决了上述问题。

本文主要完成了如下几个工作：

(1) 介绍了 UEFI 的起源，以及目前的发展前景。对比了 Bios 与 UEFI 的优劣势。从而对 UEFI 有了更深入的了解，对 UEFI 的广阔应用前景有了更进一步的认知。

(2) 由于 UEFI 是用面向对象来设计的，它采用了一种类似 C 的语法，因此驱动的代码实现及分析就显得尤为重要，因为这是大多数人不了解的一种语言规范，虽然它与 C 类似。而且由于 UEFI 驱动的设计与 Windows 驱动的实现逻辑有所不同，因此本文画了一些流程图来详细解释 UEFI 驱动的各个接口直接的调用关系。这样能让读者更清晰的了解本驱动的调用机制。

(3) 设计了一个完善的测试用例。本文采用了用于测试 UEFI Shell 的自动化测试脚本来验证本文的异常中断处理驱动的可用性。UEFI Shell 是一个类似 D o s 的命令行交互平台，因此它有几十种命令，而为了测试这些个命令设计了一千多个测试用例。通过这一千多个用例的运行，从多种角度验证了本课题实现的驱动。

6.2 展望

UEFI 的设计是 Intel 站在一个新的高度的思考。它首次采用了面对对象的思想融入了 UEFI 的开发中。因此 UEFI 还有很多很多地方等待着我的学习。本文设计的异常中断处理驱动，现在所能应用的异常场景还不完善，因此还需要将功能更进一步的完善。另外由于测试环境的限制，驱动并没有在很多的硬件平台上试验过，只在虚拟平台及部分硬件平台上跑过，因此将来需要考虑它在其他平台上的可移植性与兼容性。

致 谢

时光荏苒，岁月匆匆。转眼两年半的研究生生涯就要结束，而这段看似不长的岁月对我的人生却是有着巨大的影响。

首先我庆幸自己得以有机会能在华中科技大学这样一个高等学府得到深造，我因此得以有机会在这样一所治学严谨的高校里邂逅了众多优秀且风姿卓绝的才俊，更加因此而有幸成为黄利群老师的学生。黄老师朴素的生活作风，严谨的治学态度，渊博的学识见地，都让我钦慕不已。还记得研一的时候选修了老师的一门高级嵌入式的课程，我真真切切的感受到黄老师的知识内容以及教学水平的含金量。我真的十分感谢黄老师对我的教导与指引，他总是那么宽容和蔼，低调内敛，他的言传身教都让我获益匪浅，我对他的敬爱之情无法用文字言之一二。

其次，我还要感谢在过去的一年中，在 Intel 亚太研发中心实习时，各位同事给予我的巨大帮助。我要感谢刘江组长和令杰组长对我论文编写过程中给予的帮助，感谢你们在非常繁忙的工作时间中，还抽空为我讲解技术难点，技术细节。我要感谢组里同事们在平常工作生活中给我的指导关心，是你们让我在这短短的一年里，使我的能力得到了很大的提升。

最后我要真挚的感谢我的同学。十分感谢大家的陪伴与理解。一切尽在不言中。

参考文献

- [1] 张道华. 传统 BIOS 终结者——UEFI. 电脑爱好者, 2013(13): 72-73
- [2] 韩德强, 马骏, 张强. UEFI 驱动程序的研究与开发. 电子技术应用, 2014(5): 10-13
- [3] 裘文锋. UEFI 让系统安全从启动开始. 电脑迷, 2013(8): 44-45
- [4] Intel Corporation. Legacy BIOS and UEFI Boot Process. Intel Corporation SSG, March, 2008
- [5] 余超志, 朱泽民. 新一代 BIOS——EFI、CSSBIOS 技术研究. 科技信息, 2006(5): 14-15
- [6] 刘宗凡. 揭秘系统引导. 中国信息技术教育, 2014(13): 72-77
- [7] Intel Corporation. EDKII user manual Revision 1. 0. The United States: Intel Corporation, 2011: 10-12
- [8] 吴松青, 王典洪. 基于 UEFI 的 Application 和 Driver 的分析与开发. 计算机应用与软件, 2007(2): 98-100
- [9] 周洁, 谢智勇, 余涵等. 基于 UEFI 的国产计算机平台 BIOS 研究. 计算机工程, 2011(S1): 355-358
- [10] 王晓箴, 刘宝旭, 潘林. BIOS 恶意代码实现及其检测系统设计. 计算机工程, 2010(21): 17-18
- [11] 付思源, 刘功申, 李建华. 基于 UEFI 固件的恶意代码防范技术研究. 计算机工程, 2012(9): 117-120
- [12] 刘佳, 辛晓晨, 沈钢钢等. 基于 UEFI 的 Flash 更新的开发研究. 计算机工程与设计, 2011(1): 114-117
- [13] 胡藉. 面向下一代 PC 体系结构的主板 BIOS 研究与实现: [硕士学位论文]. 南京: 南京航空航天大学图书馆, 2005
- [14] Intel Corporation. EDK II Package Declaration (DEC) File Specification Revision.

华中科技大学硕士学位论文

Version 1. 24, 2013

- [15] 赵文华. 一种基于 BIOS 嵌入式网卡驱动的设计与实现: [硕士学位论文]. 燕山大学图书馆, 2006
- [16] Intel Corporation. EDK II Platform Description (DSC) File Specification Revision. Version 1. 24, 2013
- [17] Intel Corporation. EDK II Flash Description (FDF) File Specification Revision. Version 1. 24, 2013
- [18] 王晓箴, 于磊, 刘宝旭. 基于 EDK2 平台的数据备份与恢复技术. 中国计算机报, 2011(15): 262-264
- [19] 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): 156-158
- [20] 霍卫华. 基于 UEFI 的安全模块设计分析[J]. 信息安全与通信保密, 2008(7): 91-93
- [21] Intel Corporation. Driver Writer's Guide for UEFI 2. 0. Revision 0. 95, 2009: 9-51
- [22] Intel Corporation. EDKII Platform Configuration Database Infrastructure Description. Revision 0. 55, 2009: 11-13
- [23] 刘冬, 文伟平. EFI 及其安全性分析. 信息网络安全, 2009(5): 32-34
- [24] 黄海彬, 金晶. 基于 EFI 系统的多文件系统解决方案, 2010(6): 122-126
- [25] Intel Corporation. Pre-EFI Initialization (PEI) Overview. Intel Corporation SSG, March, 2008
- [26] Intel Corporation. Driver Execution Environment (DXE) Overview. Intel Corporation SSG, March, 2008
- [27] 张朝华. 基于 EFI/Tiano 的协处理器模型的设计与实现: [硕士学位论文]. 上海: 上海交通大学图书馆, 2007
- [28] 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
- [29] Intel Corporation. Intel UEFI Packaging Tool. Revision 040, 2013

华中科技大学硕士学位论文

- [30] Intel Corporation. EDK II Build Specification Revision. Version 1, 24, 2013
- [31] 李振华. 基于 USBKEY 的 EFI 可信引导的设计与实现: [硕士学位论文]. 北京: 北京交通大学图书馆, 2008
- [32] 万象. 基于 UEFI 系统的 LINUX 通用应用平台的设计与实现: [硕士学位论文]. 上海: 上海交通大学图书馆, 2012
- [33] Mark Lutz. Programming Python, 3rd Edition. O'Reilly, August, 2006
- [34] 杨荣伟. 基于 Intel 多核平台的 EFI/Tiano 图形界面系统研究: [硕士学位论文]. 上海: 上海交通大学图书馆, 2007
- [35] 邱忠乐. 基于 PC 主板下一代 BIOS 的研究与开发: [硕士学位论文]. 南京: 南京航空航天大学图书馆, 2005
- [36] 吴广, 何宗键. 基于 UEFI Shell 的 Pre-OS Application 的开发与研究. 科技信息, 2010(1): 15-17
- [37] Intel, HP, Microsoft, Advanced Configuration and Power Interface Specification. Revision 4.0, 2009: 5-496
- [38] 邢卓媛. 基于 UEFI 的网络协议栈的研究与改进: [硕士学位论文]. 上海: 上海交通大学图书馆, 2011
- [39] 曾颖明, 谢小权. 基于 UEFI 的可信 Tiano 设计与研究. 计算机工程与设计, 2009(11): 2645-2648
- [40] 任超. EFI 技术的实现与应用研究: [硕士学位论文]. 上海: 上海交通大学图书馆, 2005
- [41] 王兴欣. 基于 EFI 和多核体系的软件运用架构: [硕士学位论文]. 成都: 电子科技大学图书馆, 2011
- [42] 唐文彬, 陈熹, 陈嘉勇. UEFI Bootkit 模型与分析. 计算机科学, 2012(4): 8-10
- [43] 陈文钦. BIOS 研发技术剖析. 第 1 版. 北京: 清华大学出版社, 2001: 30-44
- [44] 周伟东. 基于 EFI BIOS 的计算机网络接入认证系统的设计与实现: [硕士学位论文]. 西安: 西安电子科技大学图书馆, 2008
- [45] 高云岭, 庄克良, 丁守芳. UEFI+BIOS 全局配置数据库的设计与实现. 物联网技术, 2014(10): 52-54

- [46] Intel Corporation. UEFI Driver Development Driver Development Training, Adding Code Lab. Intel Corporation SSG, March, 2008
- [47] Intel Corporation. Supplemental OS-BIOS interface-ACPI Interface. Intel Corporation SSG, March, 2008