山东大学

硕士学位论文

基于EFI的软件测试在TIANO中的应用与实现

姓名：胡可杨

申请学位级别：硕士

专业：软件工程

指导教师：贾智平

20071020

摘 要

随着现代信息技术的飞速发展，软件业竞争日趋激烈，软件系统日益复杂，对于软件功能、性能的要求不断提高，同时软件推出新版本的时间不断缩短。在这种情况下如何保证软件质量成为企业关注的重点。 仅仅依靠以密集劳动为特征的传统手工测试，已经不能满足快节奏软件开发和测试的需求。自动化为测试为此提供了成功解决方案。自动化测试是测试体系中新发展起来的一个分支，实施正确合理的自动化测试能够分担手工测试特别是回归测试的工作量，降低性能测试的难度，从而在保证软件质量的前提下，缩短测试周期，降低软件成本。

本论文的研究目的是利用软件手动或自动化测试技术测试基于 EFI（Extensible Firmware Interface）和 UEFI（Unified Extensible Firmware Interfac）规范（efi spec and uefi spec）的新型 BIOS--Tinao 软件。以基于 EFI /UEFI 计算机的预启动环境为背景，将 EFI 的驱动模式技术以及 EFI 的事件机制相结合。着重研究了可扩展（EFI）的平台架构包括：EFI 标准扩展接口的结构模式的分析，EFI 平台的启动过程，平台的管理过程，其中 EFI 特有的关键概念（协议）的介绍。

正在蓬勃兴起的软件管理技术，结合传统的软件测试原理，开发适合 EFI/UEFI 的自我认证测试系统(SCT)。它会测试每一个 EFI 规范中定义的启动时服务 Boot Service, 运行时服务 Runtime Service 和协议 Protocol 的实现，从而来验证这个实现是否符合规范本身。所有的测试用例都是 EFI 软件的驱动（Driver）,测试通过统一的接口和唯一的出错标识（GUID）管理和执行测试用例，并且获得统一格式的测试日志和测试报告。分析测试日志和报告，可以定位到软件出错位置并且修订 bug。

关键字：可扩展固件接口；自我认证测试系统；驱动-协议模型；黑盒测试；自动化测试

ABSTRACT

With modern information technology rapid developing, the software industry is becoming increasingly fierce and Software system is gradually complicated. On the request to the software functions, the functions improve unceasingly and the software debuts at the same time new edition time to shorten unceasingly. In such circumstances how ensure that software mass becomes the priority that enterprise shows solicitude for. Need depending on the tradition taking concentrated physical labors as characteristic to try on the side, already can not satisfy quick rhythm a software by hand developing and testing only.

Software testing is part of software development process, and plays a more and more import role in the software development.

In order to solve such kind of shortages of legacy BIOS, a new technology is coming. It is called EFI (Extensible Firmware Interface).

EFI provides Driver Model technology. This can make the binary codes from different vendors can work together. And the design of EFI is not related with the platform, so the codes for different platforms can be used.

SCT technique of black-box testing is introduced. The shortcomings of manual test are studied. And application of software test automation in black-box testing is proposed. A typical example of test automation is given.The test objects are analyzed, and the core script is given in the examples. It is proved by experiment that this software test adequacy measurement model provides a more objective and precise measurement of the fault detection capability of test sets.

Keyword: EFI, EFI Event, EFI Driver Model, Tiano, Black-box Test, Automation Test.

原 创 性 声 明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的科研成果。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律责任由本人承担。

论文作者签名：胡可杨 日 期：2007.10.20

关于学位论文使用授权的声明

本人同意学校保留或向国家有关部门或机构送交论文的印刷件和电子版，允许论文被查阅和借阅；本人授权山东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、缩印或其他复制手段保存论文和汇编本学位论文。

(保密论文在解密后应遵守此规定)

论文作者签名：胡可杨 导师签名：[illegible] 日 期：2007.10.20

第 1 章 绪论

1.1 项目背景

基本输入输出系统(Basic Input Output System，BIOS)，它的主要功能就是加载操作系统。BIOS 是一组固化在计算机主板 ROM 芯片内的软件程序，通常为 128k 或 256k，在系统引导时加载到系统的固定内存中，最终完成对操作系统的加载。因为最初这段代码是固化到 ROM 中，所以也称为固件(Firmware)。传统 BIOS 发展到现在，技术已经趋于成熟，但也日益暴露出一些不足，主要有：不利于引入新技术。因为设计的时间比较早，所以现在有很多新技术，和当时的技术完全是不同的架构。若要在 BIOS 中加入这些新的特性，就必须使 BIOS 的结构变得越来越复杂，开发和维护的成本也日益升高。与新技术不兼容，使得传统 BIOS 已经越来越不适应当今计算机的发展。随着 BIOS 的结构变得越来越复杂，使得 BIOS 的体积不可避免的也变得越来越大。但是，平台上 ROM 的容量有限，为解决此矛盾，人们把 BIOS 的许多组件，比如驱动程序分布到可选 ROM 上。虽然缓解了 BIOS 的容量问题，但是却增加了 BIOS 和可选 ROM 之间的藕合度，使得两者之间越来越相互依赖，给开发和维护带来诸多问题。

传统的 BIOS 只给用户一个界面，让用户可以对平台进行一些配置，但不提供对硬件的诊断和测试手段。这一点虽然对大多数用户用处不大，但是对专门从事硬件或者底层软件开发，以及大型主机的调试管理人员来说，却是非常必要的功能。因为在操作系统崩溃的时候，可能需要诊断硬件的故障，传统的 BIOS 对此无能为力。

传统的 BIOS 基本上都是用汇编语言编写的，受到汇编语言的局限，开发人员只能使用有限的资源，实现的功能也非常有限。主要用汇编语言编写而成的传统 BIOS，在开发和维护上费用高昂。因为汇编语言是伪机器指令，不像其他高级语言那样贴近人的自然思维，代码在结构化上也比其他高级语言差，所以汇编的代码更加难懂，开发和维护费用更加高昂。

相对于 BIOS 技术的止步不前，芯片的架构日益发展。传统的 BIOS 是在

16 位的实模下实现的这使得它不能被用于如以 Itanium 或 Xscale 架构为平台的机器上。

为了解决这些弊端Intel 推出了一种用来替代现有的传统BIOS 的全新的技术—EFI Extensible Firmware Interface EFI 全称是可扩展固件接口 EFI。正如它的名字一样，EFI 不是一个具体的软件，而是在操作系统与平台固件（platform firmware）之间的一套完整的接口规范。EFI 定义了许多重要的数据结构以及系统服务，如果完全实现了这些数据结构与系统服务，也就相当于实现了一个真正的 BIOS 核心。

其实早在 2000 年就开始研发加入其中的还有 Adaptec AMI ATI 惠普 LSI 微软 PowerQuest 等公司首个版本为 0.92 到现在已经发展到 1.10 版本。EFI 首先有 Intel 发起，受到全球业界相应成立 UEFI（Unified Extensible Firmware）统一可扩展接口论坛，这一国际组织负责开发、管理和促进 UEFI 规范。该论坛包括:AMI，Insyde，Phoenix，Apple，Dell，HP，IBM，Lenovo，AMD，Intel，Micorosoft。该框架的基础代码约 50 万行已经开源。原来最初 EFI 是被用于 64 位的 Itanium 平台拥有更高度模组化的特性由于 EFIBIOS 采用的高级语言-- C 语言是现在编程人员熟悉的平台因此可以和许多驱动程序共用开发资源这样就可以减少开发成本缩短开发时间 PC 厂商也可以用外挂程序来丰富自己的产品增加个性体现 EFI 它带来了传统的 BIOS 所不具备的优点 EFI/Tiano 的可扩展性使得厂商可以轻易的根据自身的需要来增加他们平台的各种特性其次它推出了一种称为驱动模式的新技术使用这种技术可以使硬件设备的供应商大大降低他们的负担驱动模式有点类似于 Windows 的 WDM 硬件设备驱动程序的开发商可以用 C 语言来开发他们产品的驱动而且不用为不同的平台开发不同的版本而且还支持二进制的链接 EFI 相对于传统的 BIOS 来说具有更快的启动速度和更小的体积 EFI 的另外一个重要的特性是它是一个 pre-boot 的环境在进入操作系统之前在 EFI 的环境下用户就可以运行多种程序来实现很多功能操作。

1.2 国内外的发展现状

虽然早在 2000 年 EFI 的概念就已经被提出，但由于各种因素的制约一

直没有全面得到实施更糟糕的是从提出到实施的过程中。Intel 也没有对 EFI 进行重点的宣传和推广以至于人们遗忘了它。随着 Intel 逐渐向平台方案提供商——这一角色的靠近。EFI 这个话题又被提上了议程而根据 Intel 的路线图. 在桌面市场上。从 Intel 915 芯片组开始.市场上已经有主板使用了 EFI. 到 945 芯片普及时。超过 50% 的主板要安装 EFI. 而在 965 芯片组. 更是要 100 %使用最新的 EFI。

2005 年 9 月，Intel、 AMD、 American Megatrends、 戴尔、惠普、IBM、Insyde、 微软和凤凰科技等重量级重量级企业共同成立“UEFI（Unified EFI）论坛”。这家非赢利性的机构从因特尔手中接过 EFI 的开发和标准制定权，EFI 也因此更名为 UEFI。

UEFI 抱括硬件控制和 OS 软件两大模块，前者是一套标准化的硬件连接协议。任何版本相同的 UEFI，硬件控制部分的内容就完全相同 后者则是提供给厂商用 c 语言撰写扩展功能的开放接口，这也是厂商可以自主定义的部分。作为 UEFI 的发起者.英特尔承担最关键的工作，它撰写了代号为“Tinao”的 EFI 功能核心框架. 其他厂商都是在此基础上挂接各自编写的功能模块。英特尔最初对 EFI 的推行非常乐观。它计划在 2005 年时就在企业领域完全实现 EFI 支持。到 2006 年则实现桌面、移动和服务器平台的全线迁移。这个计划当然没能成功施行。英特尔后来主导建立了 UEFI 论坛进行共同推广。从目前的情形来看。UEFI 有望在今年中期通过 Santa Rosa 平台进入到移动领域。并随后朝向桌面领域扩展预计 2008 年。UEFI 产品会逐渐成为市场主流。

1.3 本文的主要工作

本文主要介绍测试 BIOS 软件是否符合 EFI 标准软件测试流程、标准、所使用到的管理软件，工具等等。以 EFI 预启动环境为背景，将 EFI 系统、EFI 驱动模式和 EFI 事件机制相结合。

本文主要研究了：

(1) 可扩展性（EFI）的平台架构

EFI 标准扩展接口的结构模式的分析，EFI 平台的启动过程，平台的管理过程，其中 EFI 特有的关键概念（协议）的介绍。

(2) EFI 软件测试流程

由于 EFI 标准架构的独特性，应当设计符合 EFI 架构的测试用例。EFI/Tiano 也是软件，所以其测试用例的编写也必须符合一般软件的测试用例的标准和方法。

(3) SCT 系统的详细设计

详细研究在 EFI/SCT 的详细架构和测试的实现过程。

1.4 论文的组织结构

论文第一章绪论主要介绍了 BIOS 软件的现状，发展。

第二章主要论述了 EFI 的基本架构，启动器管理器、服务、协议和驱动模型作用，通过这四大组件，EFI 抽象出一个通用的引导环境，可以在不同的硬件环境下正确引导平台。

第三章主要介绍了 EFI 的测试原理及其软件测试流程，利用软件配置管理工具管理测试过程，bug 的管理流程和生存周期。

第四章详细设计了 SCT 自动化测试软件的架构，功能模块及其接口，黑盒测试用例编写。

第五章详细介绍基于 Tiano 的 SCT 的测试流程，以及测试报告的分析。然后举一个例子说明怎样根据 SCT 给出的测试结果来修正错误。

第 2 章 EFI 的架构

2.1 引言

EFI 是一个通用的可扩展的平台环境。EFI 能够通过接口实现屏蔽平台和固件特性。EFI 为标准的总线提供了统一的接口如 PCI、USB 和 SCSI 而且这个总线类型的列表会随着时间而增加使得将来的总线也可以被使用。可以使操作系统开发人员专注于开发操作系统，即使硬件系统有了变化，只要其固件的实现代码仍然符合 EFI 的规范，EFI 的标准接口依然有效，操作系统也不需要做任何改动[1]。

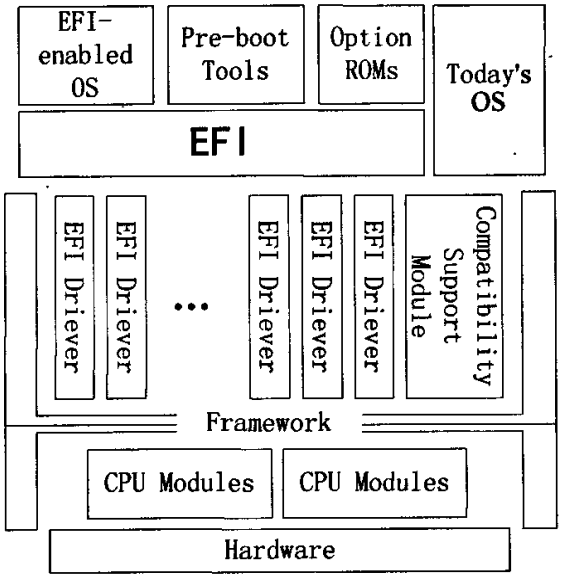


图 2-1 EFI 模式图

如图 2-1 所示[1]，EFI 并不直接对硬件进行操作，而是通过调用系统其他资源的抽象层（由主板开发商进行开发）和处理器抽象层（由处理器制造商负责开发）对硬件进行操作。EFI 的引导服务和运行时服务都是调用的这两个抽象层的相应功能。而调试系统的工具软件，则属于 EFI 应用程序(API)。总的来说，EFI 在 Intel 提出的这个固件模型里是承上启下、屏蔽硬件层物理特性的一层。EFI 是固件与操作系统进行交互的接口，可以在尚未引导操作系统

时或者操作系统崩溃时对系统进行调试。

软件测试的定义是什么?1983 年 IEEE 定义:使用人工或自动手段来运行或测定某个系统的过程，其目的在于检验它是否满足规定的需求或是弄清楚预期结果与实际结果之间的差别。这就非常明确地提出了软件测试以检验是否满足需求为目标[2]。1993 年 IEEE 给出了一个更加综合的定义:①将系统化的、规范的、可度量的方法应用于软件的开发、运行和维护的过程，即将工程化应用于软件中;②是对①中所述的方法的研究。它指出软件工程是一种层次化的技术。科学的测试是贯穿整个产品生命周期中的测试[3]。

2.2 启动过程

通过载入 EFI 驱动程序和载入 EFI 应用程序的映像的方法来扩展平台的固件。EFI 需要加载 EFI 驱动程序和 EFI 应用程序映像来扩展平台的固件功能。当驱动程序和应用程序被加载以后，这些程序才能访问的引导服务(Boot Service)和运行时服务(Run Time Sevice)[1]。

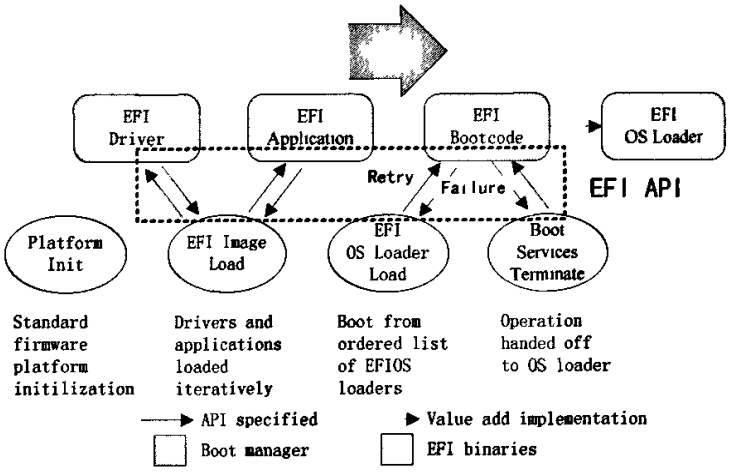


图 2-2 EFI 启动图

图2-2中说明了EFI 的启动过程：

第一步：开机对系统的固件进行标准的初始化。

第二步：加载EFI 的应用程序和驱动程序。通过对EFI 映像的加载完成了EFI 环境的建立，可以在其中完成所需的各种操作也可以为操作系统的启动做准备。OEM厂商也可以通过EFI 映像的加载来扩展他们的平台固件。

第三步：启动EFI 操作系统载入程序,开始EFI的启动代码(EFI Boot Sevice)。第四步:当EFI 操作系统载入程序准备好以后把EFI Boot Services 停掉,然后调用操作系统载入程序来启动操作系统.这里EFI 操作系统载入程序和操作系统载入程序是两个不同的应用程序,EFI 操作系统载入程序负责调用EFI 的启动代码为操作系统载入程序做好准备然后再由操作系统载入程序来继续载入操作系统。

如果介质中包含有 EFI OS Loader 或者 EFI 定义的系统分区(EFI-defined System Partition)，EFI 就可以从介质中引导系统。从块设备(block device)引导需要 EFI 定义的系统分区。EFI 不对分区的第一个扇区进行修改，所以可以在同一个分区引导传统的 Intel 架构以及 EFI 平台[3]。

2.3 启动器管理

EFI 的启动管理器负责从 EFI 的文件系统中的文件中或从 EFI 的映像载入服务中加载 EFI 的应用程序（包括操作系统第一阶段载入程序）和 EFI 驱动程序。EFI 定义了 NVRAM Nonvolatile RAM 变量用来指向要被加载的文件。这些变量中包含有在菜单中显示的字符串和直接传递给 EFI 应用程序的数据。启动管理器的主要工作就是完成把控制权从固件转到 OS。与 EFI 的功能是重叠的。所以，启动管理器在 EFI 中的地位不仅仅只是 EFI 的一个组件，它基本上贯穿了 EFI 的始终，涵盖了 EFI 的绝大部分范围。后文提到的内核，协议和驱动模型，都要由启动管理器管理[4]。

EFI 映像是EFI 中含有可执行代码的一类文件。EFI映像的最显著的特征是这个文件的前面几个字节定义了一个头结构用来定义可执行映像的编码。EFI 映像有包括EFI应用程序和EFI 驱动程序。最大的区别是EFI 应用程序在退出入口点时总是被卸载而EFI 驱动程序只有在收到错误代码时才卸载。一个EFI 映像按照PE32+的格式通过EFI Boot Service 的服务被加载入内存，PE32+载入程序会把映像的所有的段都放入内存。在映像被载入后，相应的修改程序会被执行然后控制权会交给被加载映像的入口点。

2.4 EFI 内核(EFI Core)及其映像

EFI 的内核是 EFI 提供的一些系统底层服务，是 EFI 对底层平台进行抽象的结果。EFI 内核提供两种服务:引导服务(Boot Service)和运行时服务(Runtime Service)[5]。

引导服务为已经加载的 EFI 映像抽象了一个通用的引导环境，采用全 64 位设计，对将来硬件技术的发展预留了一定的扩展空间。该抽象使平台和操作系统能够相对独立的发展。引导服务所提供的调用是系统在引导期间所需要的，在操作系统被引导之后失效(调用 ExitBootServiceso 引导服务)[6]。和引导服务不同，运行时服务能够在引导和操作系统运行的时候被调用。运行时服务能够完成物理内存地址到虚拟内存地址的转换，只有少数几个运行时服务始终运行在物理内存地址方式。因为运行时服务在 OS 加载以后也一直运行，所以，为了保证运行时服务对平台的性能不至于造成太大的影响，运行时服务的数量比较少，而且都很简单。

2.5 协议(Protocol)

协议[4]是 EFI 对平台各个设备(device)的抽象，对设备的操作，都是通过相应协议提供的接口完成的。对于任何 一个设备，要能被 EFI 使用，首先必须支持相应的协议，然后在使用前注册（安装）该协议，才能正常使用.使用完毕以后，也应该卸载该协议。每个协议都有一个全局唯一的 GUID 与之对应，并且有一个数据结构，结构中既有属于该协议的私有数据，也有该协议提供的服务函数。这些函数都是以函数指针的形式存在的[7]，使用时，要在安装协议的时候把这些函数指针指向具体的函数，服务函数才能实现具体的功能。这提供一个灵活的机制，使得为设备更换驱动程序的操作更加简单，从而简化了驱动程序开发与测试，更新的步骤。

2.6 EFI 驱动模型(EFI Driver Model)

管理硬件的实现模块就是驱动。驱动实现的方法通过协议的接口被调用。设计EFI驱动模型时为了使设备驱动程序的设计和实现更为简单,生成的更小的可执行代码,所以一些复杂的机制被放到了总线驱动程序中和通用的固件服务中。符合EFI 驱动模式的驱动程序必须能够被简单的实现简单的维护。它

应该允许驱动程序的开发人员把注意力集中在被开发的设备上而不用考虑平台的策略或者平台的管理等问题[8]。

计算机系统的架构可以被看做一个或多个处理器被连接到一个或多个的核心芯片上。EFI 驱动模式只定义了在核心芯片上产生的I/O 总线的集合以及这些总线的子设备，没有指定特定处理器或核心芯片。这些子设备可以是硬件设备也可以是另外的I/O 总线。

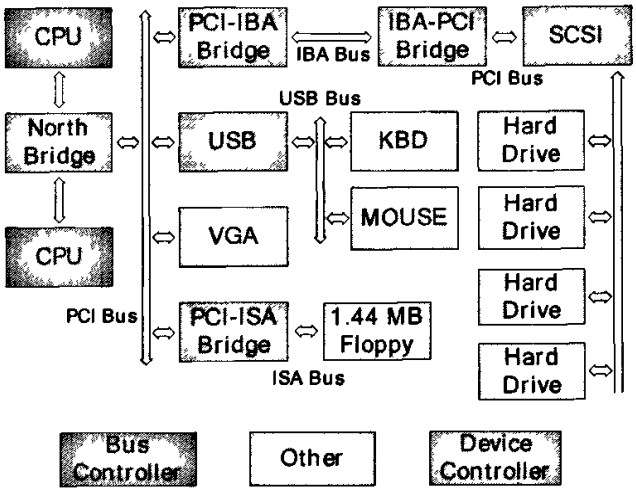
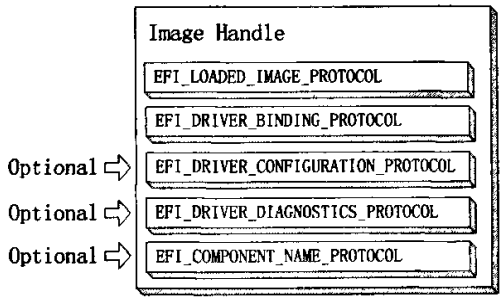


图 2-3 服务器体系结构

如图 2-3 所示。因此系统可以看作一棵由总线和硬件设备组成的树结构。树的根部是核心芯片，负责产生一个或多个 I/O 总线，树结构的叶结点是产生某种 I/O 操作的周边设备。这些周边设备可以是键盘显示器磁盘或网络等，那些不是叶子的节点是由总线控制器组成。这些总线负责在硬件设备和总线之间或者在不同类型的总线之间传递数据[9]。

符合 EFI 驱动模式的设备驱动程序需要在加载这个驱动程序的映像句柄上产生一个驱动程序绑定协议— EFI Driver Binding Protocol 。这个协议产生以后设备驱动程序就会等待系统的固件把它连接到一个控制器上。做完这一步以后设备驱动程序负责在控制器的句柄上产生一个控制器所支持的提供统一的 I/O 操作的协议。一个符合规范的总线驱动程序要完成同样的这些工作而且总线驱动程序还要负责发现在总线上的任何一个子控制器并为每一个被找的子控制器创建一个设备句柄。



如图2-4所示[10]，EFI 驱动模型提高了驱动的可移植性。驱动模式还可以为将来的总线类型进行扩展。符合驱动模式的驱动程序可以在不同的平台不同的处理器之间进行移植。EFI 驱动模式应该适用于各种类型的平台这些平台。包括了台式机，便携机，嵌入式系统，工作站和服务器。它能够列举所有的设备或者仅列举那些启动操作系统所需要的设备。最小化的设备列举可以使系统快速的启动。而列举所有的设备则可以完成操作系统的安装，系统的维护或者在系统中存在的启动设备上进行系统诊断。

2.7 本章小结

本章对EFI体系结构作简单了分析。EFI作为一个接口将硬件与软件分隔开来。利用有相对稳定的接口，使BIOS和OS之间藕合度较低，易于开发和管理。且通过对通用步骤的抽象，EFI可以跨平台使用，并且对未来的技术变革，有一定的适应能力。四大组件之中，启动器管理器用于管理，负责和用户交互，同时协调服务、协议和驱动模型共同工作。EFI核心、协议和驱动模型都是对平台的某一部分进行抽象的结果。EFI 核心是对平台提供的底层功能调用进行的抽象，协议是硬件设备的抽象，而驱动模型则是硬件功能的抽象。通过这四大组件，EFI抽象出一个通用的引导环境，可以在不同的硬件环境下正确引导平台。

第 3 章 EFI 测试原理与流程

3.1 引言

软件测试的定义是什么?1983年工EEE定义:使用人工或自动手段来运行或测定某个系统的过程，其目的在于检验它是否满足规定的需求或是弄清楚预期结果与实际结果之间的差别。这就非常明确地提出了软件测试以检验是否满足需求为目标[4]。1993年IEEE给出了一个更加综合的定义:①将系统化的、规范的、可度量的方法应用于软件的开发、运行和维护的过程，即将丁程化应用于软件中;②是对①中所述的方法的研究。它指出软件工程是一种层次化的技术。科学的测试是贯穿整个产品生命周期中的测试[11]。

软件测试的作用是什么？首先用来执行或者模拟一个系统或者程序的操作。第二，确定软件是按照它所要求的方式执行的，而不会执行它不被希望的操作。第三，是带着发现问题和错误的意图来分析程序的。第四，用来度量程序的功能和质量。第四，评价程序和项目工作产品的属性和能力，并且评估其是否获得了期望和可接受的结果。

3.2 EFI 的软件测试原则

权衡投入/产出比的原则：不充分的测试是不负责任的，但是过量的测试是一种浪费，应平衡分配测试资源。全部测试用例必须满足EFI 标准架构。这是因为测试的目的在于被测试软件是否符合EFI标准[12]。

1. 尽早制定测试计划，由于软件的复杂性和抽象性，以及软件开发各个阶段的多样性等，使得开发的每个环节都可能产生错误。所以不应把测试仅仅看作是软件开发的一个独立阶段，应该把软件测试贯穿到软件开发的各个阶段，以期尽早发现错误，提高软件质量。

2. 测试发现的错误可能集中在极少部分的程序模块，这表明应当对错误群集的程序模块进行重点测试。

3. 应从“小规模”开始，逐步转向“大规模”。这也就是为什么测试会

涉及生命周期的两个阶段的原因。

4. 一个大小适度的程序,其路径组合是一个天文数字,因此穷举测试是不可能的。充分覆盖程序逻辑并确保程序设计中使用的所有条件是可能的。

5. 应该由独立的第三方进行测试。由于基于思维定势，人们很难发现自己的错误；而且由于心理因素，人们不愿意否定自己的工作。因此，为达到软件测试的目的，应启用客观、冷静、严格的独立第三方。

3.3 EFI 的软件测试技术与方法

软件测试技术分为两大类，即静态测试和动态测试。动态测试方法中又根据测试用例的设计方法不同，分为黑盒测试和白盒测试两类[13]。

1. 静态测试技术

静态测试是指被测试程序不在机器上运行，而采用人工检测和计算机辅助静态分析的手段对程序进行检测。具体有两种:

人工测试:是指不依靠计算机而靠人工审查程序的办法。

人工审查程序偏重于程序质量的检验，而软件审查除了审查程序质量还要对各阶段的软件产品进行检验。人工检验可以发现计算机不易发现的错误。

计算机辅助静态分析:指利用静态分析工具对被测试程序进行特性分析，从程序中提取一些信息，以便检查程序逻辑的各种缺陷和可疑的程序构造。如用错的局部变量和全程变量、不匹配参数、不适当的循环嵌套和分支嵌套、潜在的死循环及不会执行到的代码等。

2. 动态测试技术

动态测试是指通过运行程序发现错误。一般意义上的测试大多是指动态测试。对软件进行动态测试时，使用黑盒测试法和白盒测试法。

第一种方法：黑盒测试法，该方法把测试对象看成一个黑盒子，测试人员完全不考虑程序的内部结构和处理过程，只在软件的接口处进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息，并且保持外部信息（如数据库或文件）的完整性。因此，黑盒测试又称为功能测试或数据驱动测试。

通过黑盒测试主要发现以下错误:是否有不正确或遗漏的功能；在接口处，能否正确的接受输入数据，能否产生正确的输出信息；访问外部信息是

否有错；性能上是否满足要求等。用黑盒测试时，必须在所有可能的输入条件和输出条件中确定测试数据。

第二种方法： 白盒测试法，该方法把测试对象看作一个打开的盒子，测试人员必须了解程序的内部结构和处理过程，以检查处理过程的细节为基础，对程序中尽可能多的逻辑路径进行测试，检验内部控制结构和数据结构是否有错，实际的运行状态与预期的状态是否一致。

3.4 工业标准测试

对EFI（Tinao）软件进行日常软件测试分为按范围划分为三种：Smoke Test(冒烟测试)、Stable Test（稳定测试）、Extended Test（扩展测试）。在手动测试中最重要的测试是工业标准的测试方法。符合EFI标准 的BIOS 上必须实现一系列工业标准的驱动程序,例如PCI, USB 等。不同的厂商有不同的PCI设备产品，因此测试他们的实现有没有符合这些工业标准成为必然。

1. Somke Test:

在软件里，Smoke test指的是一组小规模的测试，检查一个运行中系统基本功能，以此保证系统状态是正常的。在EFI/Tiano 测试中Smoke test范围包括：map –r（检查是否装载所有的磁盘设备），reconnect –r（卸载设备驱动，再装载设备驱动到设备）。Smoke test与daily build是结合在一起，每次增加Patch（补丁/bug修改后的代码）都要进行Smoke test，检测一天的修改是否影响到系统正常运行，如果有问题则检查的当天的代码改动。

2. Stable Test:

在EFI/Tinao软件中，Stable Test指一组中等范围的测试，能覆盖EFI/Tinao系统的大部分基本功能。包括界面信息是否完整；shell 的基本命令的实现是否正确；存储设备、PCI、USB 设备能否正确识别，能否启动到操作系统等。

3. Extended Test:

测试覆盖范围最大，包括PCI卡：SCSI 卡、显卡、声卡等，AGP显卡，所有USB接口，最大内存容量的测试（测试最大内存容量：Bridgeport 内存容量最大16G）。所有操作系统能否在Tiano软件上正常运行。

3.4.1 EFI/UEFI shell

EFI/UEFI为用户提供了一个交互环境：EFI/UEFI shell，用户可以通过UEFI shell来导入自己编写的特定的Application和Driver。UEFI Application可以是硬件检测，引导管理设置软件，也可以是操作系统引导软件等。EFI/UEFI Driver提供一系列与系统设备通信的接口，它可以从任何支持UEFI环境的设备中导入[14]。

EFI/UEFI App和UEFI库函数提供基本控制台IO，基本磁盘IO，内存管理以及字符串操作功能。EFI/UEFI App以可执行程序．efi的形式存在，执行完后返回控制权，不会驻留内存，可以方便移植到不同的平台。

3.4.2 PCI Option Rom 的测试

测试可选ROM (Option ROMs)的抽象。EFI定义的接口也支持现在的PCI,USB等总线类型，同时为了适应将来的总线类型，EFI提出了“EFI驱动模型”。通过这个模型，驱动程序能够象操作系统引导程序那样，无需知道底层细节，能使用抽象的接口对设备进行驱动。EFI 集合了PCI Specification 和PE/COFF Specification 优点，将EFI 映像装入 PCI Option Rom中。PCI 总线驱动必须扫描 PCI Option Rom 中装载的PCI 设备驱动[14]。在PCI 设备列举过程中发现PCI Option Rom,把PCI Option Rom 的一个副本放到内存缓冲区。PCI 总线驱动通过内存中的副本去查询EFI驱动。PCI 总线驱动通过查询PCI Option Rom中的镜像列表，找到一个在PCIR DATA含有0x03编码类型（EFI image）和在EFI PCI 扩展ROM头中值0xEF1（标示EFI image 头），则PCI 总线驱动去load PCI Expansion Rom Header中的指向EFI 映象头的偏移地址（0x04），这个偏移地址用来获得一个指向PCI 设备驱动的PE/COFF image的指针[15]。

Table PCI Expansion ROM Code Types

Code Type	Description
0x00	Intel IA-32,PC-At compatible
0x01	Open Firmware standard for PCI

0x02	Hewlett-Packard PA RISC
0x03	EFI image
0x04	Reserved

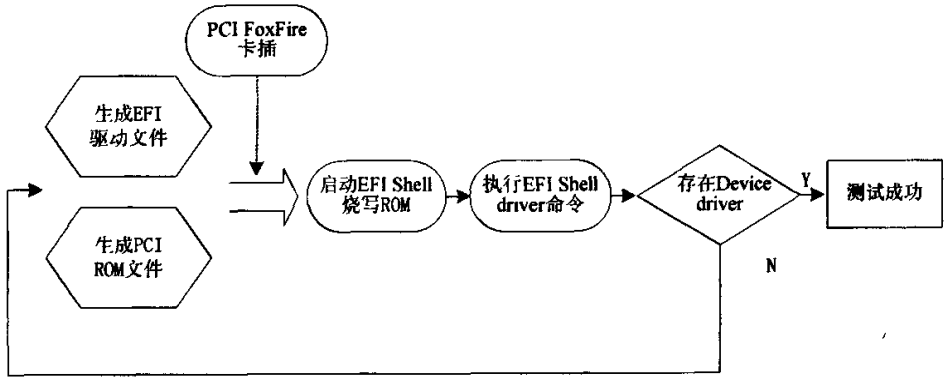
Table: EFI PCI Expansion Rom Header

offset	Byte Length	Value	Description
0x00	1	0x55	Rom Signatiure,byte1
0x01	1	0xAA	Rom Signatiure,byte1
0x02	2	xxxx	Initialization Size-size of this image in units of 512 bytes. The size Include this header.
0x04	4	0xoEF1	Signature from EFI image header
0x08	2	xx	Subsystem value for EFI image header
0x0a	2	xx	Machine type from EFI image header
0x0c	2	xx	Compression type 0x0000 –The image is uncompressed 0x0001 –The image is compressed.
0x0e	8	0x00	Reserved
0x16	2	xx	Offset to EFI image
0x18	2	xx	Offset to PCI Data Structure

3.4.3 UEFI Application（driver.efi）的编译和运行

UEFI App是带有修改过的头标识的PE / COFF文件。头标识用来区分UEFI映像和一般的PE / COFF映像。Intel提供了相应的把PE / COFF文件转换成EFI映像文件的工具。在一个新的UEFI App能够编译之前，每个编译终端(build tip)的makefile文件还需要修改。这些文件包括 \ ... \ build \ ia32 \ makefile， 。其中每个文件中都有一个标识为everything to build的部分，并加入了对makemaker的具体描述。这些步骤完成之后，当编译终端中运行nmake时，新的App就可以编译了。一般常用的.efi 都可以在\Edk\Other\Maintained\Application\Shell\bin\ia32\Apps目录下找到。我们可以在UEFI shell命令行敲入App的名字来运行相关UEFI App。例如：driver.efi

手动测试过程：EFI 的 BIOS 是用 C 语言写成，但 EFI 的 C 语言没有 Main 函数，是应用而是在 Makefile 文件中加入程序入口点（Entry Point）。所以 EFI 程序都需要编译过程。



1 编译 EFI 驱动文件　　图 3-1 PCI Rom 测试流程图

2 编译 PCI Rom 镜象文件

3 PCI FoxFire 卡插在主板的 PCI 插槽上

4 启动EFI 或 UEFI Shell 环境，用.efi 驱动程序把各自的相应.rom 镜像烧入FoxFire卡的Option Rom 中。

5 在EFI Shell 环境下，执行driver命令

```
Shell> drivers
                 T   D
                 Y C I
                 P F A
V  VERSION       E G G #D #C DRIVER NAME                          IMAGE NAME
== ======== = = = == == ================================== ============
5D 00000001 D - -  1  - IDE Cootroller Init Driver          VenMedia(B708
5E 00000001 D - -  1  - SATA Controller Init Driver         VenMedia(B708
5F 00000010 D - -  1  - <UNKNOWN>                           VenMedia(B708
60 00000010 B - -  1 23 PCI Bus Driver                      VenMedia(B708
61 00000010 D - -  2  - PCI IDE/ATAPI Bus Driver            VenMedia(B708
```

图 3-2 输入 dirver 命令后显示现有系统的所有已经安装的驱动

6 检查是否有FoxFire 卡的Device driver ，如果是，则测试成功，否则是测试失败，重新测试。

3.5 软件配置管理

研发人员分工明确：主要的三个角色：程序经理 (Program Manager)、开发人员(Developer)、测试人员（Tester）三者分工明确、接口清晰，程序经

理来定义需求、书写出来每个功能特性(Feature)的设计文档(Spec)，开发人员写代码来实现这个设计文档，测试人员来测试开发人员做出来的东西是否符合程序经理定义的设计文档，三个角色之间并无必然的上下级关系，只是分工合作完成某个功能(Feature)。

程序经理将写好的需求设计文档(Spec)保存到共享文档库中，所有相关的人都可以随时查看；开发人员(Developer) 用WinCVS 的微软内部源代码管理工具）来保存源程序；测试人员把发现的Bug 记录到PVCS中以有效跟踪这个问题的处理流程。

配置管理工具大致可以分为3个级别：

第一个级别--简单的版本控制工具，是入门级的工具，例如:CVS,VSS;

第二个级别—项目级配置管理工具，适合管理中小型的项目，例如：PVCS，Perforce；

第三个级别—企业级配置管理工具，具有强大的过程管理功能，例如：CCC/Harvest，ClearCase。

3.5.1 VSS 文件备份

Visual SourceSafe是Microsoft公司推出的配置管理工具，是Visual Studio的套件之一。SourceSafe因其简单易用性成为国内最流行的配置管理工具，使用Visual SourceSafe Administration可以方便地对指定的路径进行备份。Tinao小组利用VSS客户端备份测试手册、测试工具和测试人员的测试结果，测试工程师的工作进度安排。服务器端已经创建好名为Tinao的VSS数据库。

3.5.2 WinCVS 版本控制

CVS并发版本系统（Concurrent Versions System），用于版本管理。通过客户机/服务器存取模式使得开发者可以从客户端存取代码。用于多人协作开发的大型系统，可以在服务器端存放不同的软件版本，开发和测试人员可以下载到本地进行修改。

WinCVS的主要功能：

1. 版本控制系统跟踪一系列文件所作改变的历史记录。

代码集中的配置：客户机/服务器存取方法使得任意连到因特网上的开发者都能存取在同一台CVS服务器上的文件。

2. 调整代码：

无限制的版本管理检出(checkout)的模式避免了通常的因为排它检出模式而引起的人工冲突。check out命令只需在开始建立本地代码树时使用一次，其后更新本地代码则使用update命令。update命令比较服务器和本地代码库的区别，并把本地代码树中过时的文件自动更新。

例如：在EFI/Tiano项目组，虽然CVS主服务器在美国intel总部，但在中国、美国、印度等分公司可同时修改Tinao源代码，最新项目代码名称为Tinao，开发人员和测试人员通过check out /check in 命令下载和提交代码到服务器。经过一段时间的测试修改代码，在提交代码之前同样需要使用update命令，以获取他人并行修改的的代码。如果出现冲突（即对同一文件同时进行了修改），CVS将在本地代码中把两者都保留并标记出来，要求开发者处理冲突。在冲突不存在或已解决的情况下，使用commit命令将服务器代码更新为本地代码。CVS要求为更改提供注释，并自动为更新的文件处理版本编号。通过CVS客户端选择Tinao 文件在菜单栏选择"modify->creat a tag on selection；creat tag setting"窗体，在"new tag name"中填入版本号（INTEG_20061031）提交到服务器端。

与Visual Source Safe(VSS)相比,CVS主要由两个不同之处：

一是VSS依靠服务器上的一个共享目录提供服务，每一个client必须能够访问这个共享目录。这也就决定了source safe在TCP/IP环境下使用很困难。而CVS依靠TCP/IP连接提供服务，虽然基本的pserver连接安全性不是很高，但是通过使用SSH,可以获得很高的安全性。

二是CVS反对对文件上锁的机制。VSS以及其他很多传统版本控制工具要求一个文件只能有一个使用者，它必须先checkout声明编辑文件的独享权力，直到checkin为止。CVS采取多个用户可以同时对一个文件进行编辑，然后commit的方式解决这个问题。

3.6 Bug 管理流程（PVCS 的应用）

在实际软件测试过程中，每个Bug都要经过测试、确认、修复、验证等的管理过程，这是软件测试的重要环节。

Bug管理的主要流程是：正确设计每个错误的包含信息的字段内容和记录错误的处理信息的全部内容。字段内容可能包括测试软件名称，测试版本号，测试人名称，测试事件，测试软件和硬件配置环境，发现软件错误的类型，错误的严重等级，详细步骤，测试注释。处理信息包括处理者姓名，处理时间，处理步骤，错误记录的当前状态。

为了正确跟踪每个软件错误的处理过程，通常将软件测试发现的每个错误作为一条条记录输入到制定的错误跟踪管理系统。一个缺陷跟踪管理系统，需要正确设计每个错误的包含信息的字段内容和记录错误的处理信息的全部内容。字段内容可能包括测试软件名称，测试版本号，测试人名称，测试事件，测试软件和硬件配置环境，发现软件错误的类型，错误的严重等级，详细步骤，测试注释。处理信息包括处理者姓名，处理时间，处理步骤，错误记录的当前状态。

PVCS 是世界领先的软件开发管理工具,市场占有率达 70%以上,是公认的事实上的工业标准，能够支持客户端利用 Web 浏览器访问配置管理库。包括：

PVCSTracker:在整个开发过程中确定和追踪软件的每一变更的要求。

PVCSNotify:将软件状态的变更通过 E-mail 通知组织机构中的其他成员。

PVCSReporter:为 GUI 界面环境提供一个客户报表工具,使用它能很容易地生成和存储多个项目的报表。

Bug 的状态：

1. 新信息(New)：测试中新报告的软件缺陷；
2. 打开 (Open)：被确认并分配给相关开发人员处理；
3. 修正(Fixed)：开发人员已完成修正，等待测试人员验证；
4. 验证（Verification）测试人员验证；
5. 拒绝(Declined):拒绝修改缺陷；
6. 延期(Deferred): 不在当前版本修复的错误，下一版修复
7. 关闭(Closed)：错误已被修复；

测试管理过程：

1. 测试人员提交新的Bug入库，错误状态为New。

高级测试人员验证错误，如果确认是错误，分配给相应的开发人员，设置状态为Open。如果不是错误，则拒绝，设置为Declined状态,如果有争议可是设置为 return to submitter 返回给bug申报的测试人员。

开发人员查询状态为 Open 的 Bug,如果不是错误,把状态设置为 Declined 如果是 Bug 则修复并置状态为 Fixed。不能解决的 Bug，要留下文字说明及保持 Bug 为 Open 状态。对于不能解决和延期解决的 Bug，不能由开发人员自己决定，一般要通过评审会通过才能认可。

测试人员查询状态为 Fixed 的 Bug，然后验证 Bug 是否已解决，如解决置 Bug 的状态为 Closed，如没有解决设置状态为 Reopen。

通过 PVCS 可以实现：

1. 可以完备的记录、跟踪 Bug 的一生：从出生（创建新的 Bug）、不断成长（记录相关人员寻找产生 Bug 的原因的讨论过程）、发育成熟（找到了一个处理办法）到最后死亡（关闭），同时也要允许 Bug 的复活（问题重现），继续其生长过程。

2. 方便的查询功能，快速找到符合所需条件的 Bug。比如：最近 N 个指派给某人的 Bug；最近 N 个由某人创建的 Bug；各种自定义条件的查询。

3. 提供各种 Bug 统计信息。比如每个人有多少个 Bug、到目前为止 Bug 总数的统计、最近一周 Bug 曲线图等等，视具体需要可以有很多种统计。

4. 方便的项目和模块管理，可以有很多项目、每个项目有多个模块，要能够很方便的增加、删除、修改。

5. 简单的用户管理。作为一个可独立使用的系统，需要能够增加、删除用户。当然最好的是直接使用公司已有的管理系统中的用户认证。

3.7 本章小结

本章主要介绍了软件测试的基本概念，介绍了 EFI 的测试原理及其软件测试流程，利用软件配置管理工具管理测试过程，bug 的管理流程和生存周期。在 EFI/Tiano 软件测试中主要的手工测试在于工业标准测试，详细分析了 PCI

Rom 的测试流程，介绍了.efi 文件的编辑和运行环境。介绍软件测试管理的方法和工具，详细介绍了如何通过 PVCS 软件进行 bug 周期的跟踪管理的方法。

第 4 章 SCT 系统的详细设计

EFI 自验证测试系统(SCT)利用 EFI 的驱动-协议模型通过统一的测试协议,管理测试用例集合用于验证一个 EFI 的实现是否符合 EFI1.1，UEFI2.0，Framework 规范[16]。 EFI SCT 是一个独立于平台的测试软件。它会测试每一个 EFI 规范中定义的启动时服务 Boot Service, 运行时服务 Runtime Service 和协议 Protocol 的实现,从而来验证这个实现是否符合规范本身。为了与 EFI BIOS 兼容，SCT 也采用了 EFI 的驱动-协议 Driver-Protocol 模型可以满足 EFI 测试系统独立与底层实现,所有的测试用例都是 EFI 软件的驱动(Driver),测试通过统一的接口和唯一的出错标识（GUID）管理和执行测试用例，并且获得统一格式的测试日志和测试报告[17]。

4.1 黑盒测试方法概述

在已知产品应具有的功能的条件下，测试每个功能是否都能正常使用。在测试时，完全不考虑程序内部结构和内部特性的情况下，测试者在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息，并且保持外部信息（如数据库或文件）的完整性。“黑盒”不考虑内部逻辑结构而是注重程序外部结构、针对软件界面和软件功能进行测试。

测试方法包括等价类划分、因果图、正交实验设计法、边界值分析、判定表驱动法、功能测试等。

1. 等价类划分

定义：所有可能的输入数据,即程序的输入域划分成若干部分（子集）,然后从每一个子集中选取少数具有代表性的数据作为测试用例。

方法：

划分等价类：等价类是指某个输入域的子集合.在该子集合中,各个输入数据对于揭露程序中的错误都是等效的.并合理地假定:测试某等价类的代表值就等于对这一类其它值的测试。因此,可以把全部输入数据合理划分为若干等

价类,在每一个等价类中取一个数据作为测试的输入条件,就可以用少量代表性的测试数据。取得较好的测试结果.等价类可以分为有效等价类和无效等价类。

有效等价类:是指对于程序的规格说明来说是合理的,有意义的输入数据构成的集合.利用有效等价类可检验程序是否实现了规格说明中所规定的功能和性能。

无效等价类:与有效等价类的定义恰巧相反。

2. 边界值分析法

边界值分析(Boundary Value Analysis, BVA)是一种补充等价类分的测试用例设计技术，它不是选择等价类的任意元素，而是选择等价类边界的测试用例。首先应确定边界情况.通常输入和输出等价类的边界,就是应着重测试的边界情况。应当选取正好等于,刚刚大于或刚刚小于边界的值作为测试数据,而不是选取等价类中的典型值或任意值作为测试数据。

4.2 EFI 的驱动-协议模型

协议（Protocol）：EFI 的协议可以被 EFI Boot Services 的 HandleProtocol 和 OpenProtocol 所发现每个协议都应该包括下面几项[18]：

1. 协议的 GUID Globally Unique ID
2. 协议接口的结构
3. 协议的服务

EFI 协议的 GUID 是 128 位的。协议接口结构中含有函数和用来存取设备的实例数据。函数用来允许应用程序把协议装在一个句柄上。一个协议的 GUID 要传个 HandleProtocol 或者 OpenProtocol 来确定确定一个设备句柄是否支持一个给定的协议。如果这个设备支持所需要的协议那么指向这个协议接口结构的指针将会被返回。这个协议的接口结构会链接到为这个设备所调用的协议的服务[19]。

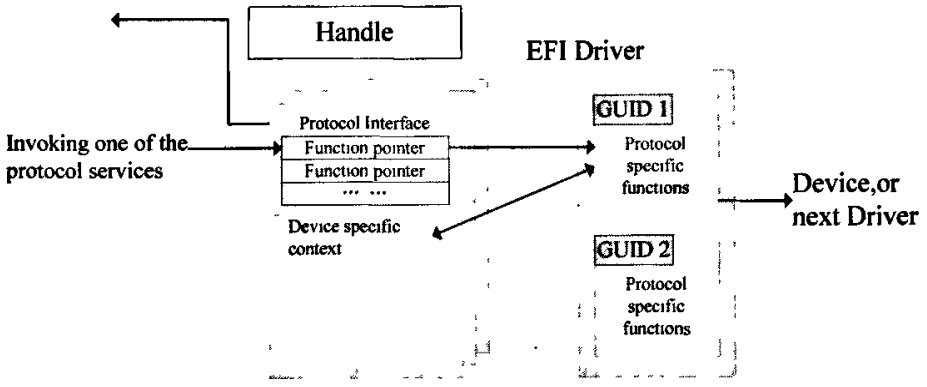


图 4-1 协议结构图

SCT应用的驱动-协议模型是测试系统定义的EFI 测试专用协EFI_TEST_PROTOCOL ，相应的驱动就是测试用例的实现。在初始化时，测试用例的都安装了EFI_TEST_PROTOCOL，在这里不同的测试用例可视为不同的软件设备，它们的实现的测试功能各不相同。 在EFI Shell 下通过命令启动SCT 测试系统，就能通过测试协议获得测试用例的接口，运行测试用例并获得测试结果[20]。

4.3 SCT 组成模块

SCT 测试系统包括测试框架（SCT Framework），辅助驱动模块(Support Driver)和一系列的测试用例(Test Cases) 。如图所示：测试框架(Framework)是控制测试流程的主体模块,包括启动并初始化测试进程配置测试的环境参数,显示可用的测试用例供用户选择,执行测试用例以及生成测试结果报告。辅助驱动模块被测试框架调用与测试用例无关完成一些独立的功能如:记录测试日志和记录系统的重起恢复状态等辅助驱动模块与测试框架间的接口是EFI_TEST_SUPPORT_PROTOCOL 协议[21]。

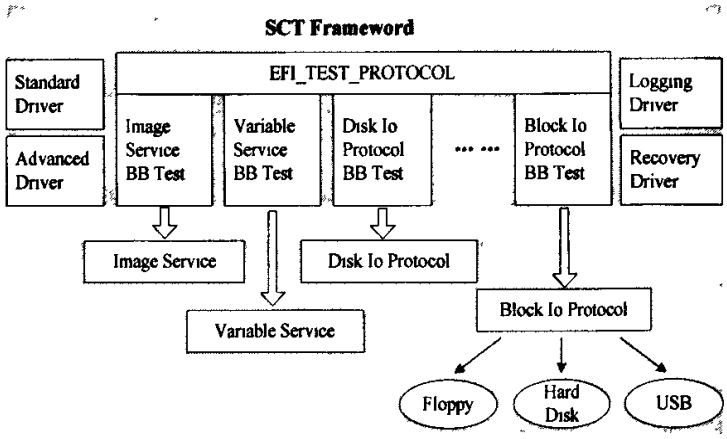


图 4-2 SCT 架构图

如图4-2所示，测试用例可以手动添加或删减，在运行每一个测试用例时就会调用EFI规范定义的服务EFI Boot Service, EFI Runtime Service或是协议EFI Protocol， 从而判断其实现是否符合规范定义。对于一个服务或协议的几个测试用例组织成一个安装了测试协议的驱动，实现了测试服务或协议的函数并把函数入口地址存放在测试协议的接口中供框架调用。每个驱动经过编译生成相应.efi 驱动文件，存放在SCT 的指定目录下。测试框架能检测到该目录下所有的测试驱动文件[22]。在SCT 测试系统的初始化过程中测试框架把检测到的测试驱动的文件映像载入内存备用。驱动中的测试函数入口也被存储于数据结构中。这也就是自动侦测测试用例的过程，执行测试用例时测试函数被调用并记录测试数据，测试结束时内存资源被释放。

4.4 SCT 系统架构

4.4.1 黑盒测试协议 EFI_BB_TEST_PROTOCOL 的定义如下

```
typedef struct {
  UINT64              TestRevision;
  EFI_GUID            ClientGuid;
  CHAR16              *Name;
  CHAR16              *Description;
  EFI_BB_TEST_ENTRY   *EntryList;
} EFI_BB_TEST_PROTOCOL;
```

测试协议的接口用C 语言的结构类型来定义并引用了EFI 实现中所定义的EFI_GUID 类型[23]。说明:

TestRevision: 用来标识测试的版本号。

ClientGuid: 被测试的启动服务（Boot Service），运行时服务(RuntimeService)和协议(Protocol)的GUID。

EFI_BOOT_SERVICES_CLIENT_GUID: 一个指针指向boot services table。

EFI_RUNTIME_SERVICES_CLIENT_GUID: 一个指针指向 the runtime Services table。

EFI_GENERIC_CLIENT_GUID: 一个指针指向NULL。

Boot service:用 EFI_BOOT_SERVICES_CLIENT_GUID。

Run time service: 用EFI_RUNTIME_SERVICES_CLIENT_GUID。

不确定类型：用EFI_GENERIC_CLIENT_GUID。

Name 是测试的名称。

Description 是本测试用例的描述。

EntryList: 记录了存放所有函数入口的链表的头地址。

4.4.2 黑盒测试入口定义如下

```
EFI_BB_TEST_ENTRY
typedef struct {
  EFI_BB_TEST_ENTRY     *Next;
  EFI_GUID              EntryId;
  CHAR16                *Name;
  CHAR16                *Description;
  UINT32                TestLevelSupportMap;
  EFI_GUID              *SupportProtocols;
  UINT32                CaseAttribute;
  EFI_BB_ENTRY_POINT    EntryPoint;
} EFI_BB_TEST_ENTRY;
```

EntryList 所指向的链表中的每个节点元素是执行测试的最小单位其类型为EFI_BB_TEST_ENTRY。对应一个测试函数EFI_BB_TEST_ENTRY 类型

包含了测试入口的ID，测试函数的名称和描述测试的级别（功能测试或完整性测试），所需的辅助测试协议，测试的属性（自动或手动），指针supportProtocol指向GUID数组，其中每个GUID都代表一个测试用到的协议，以及测试函数入口地址EntryPoint 所有的测试函数的入口都是统一EFI_ENTRY_POINT类型。

4.4.3 黑盒测试入口点定义如下

```
EFI_STATUS
  BBTestDriverEntryPoint1 (
    IN EFI_BB_TEST_PROTOCOL        *This;
    IN void                        *ClientInterface;
    IN TEST_LEVEL                  TestLevel;
    IN EFI_HANDLE                  SupportHandle;
  )
```

每一个测试用例都是从EFI_ENTRY_POINT开始的，测试函数的第二个参数ClientInterface 用于传递被测试协议的接口（EFI_BB_TEST_PROTOCOL的 GilentGuid ）[23]。如果被测试的是一个EFI 服务测试协议的成员ClientGuid和测试函数的参数ClientInterface 都为空，执行测试函数时，通过系统全局的数据结构获得被测试服务的接口。从参数SupportHandle中得到库协议接口（例如：Standard Test Support Library）。通过库协议接口可以完成架构的测试过程。

4.4.4 标准库协议接口（Standard Test Support Library）

```
typedef struct {
  UINT64          LibraryRevision;
  CHAR16          *Name;
  CHAR16          *Description;
  EFI_RECORD_ASSERTION RecordAssertion;
  EFI_RECORD_MESSAGE   RecordMessage;
  EFI_GET_RESULT_COUNT GetResultCount;
} EFI_STANDARD_TEST_LIBRARY_PROTOCOL;
```

DriverEntryPoint 调用库接口函数可以获得pass/warn/fail信息，还有导致

出错的错误信息，返回出错结果数目。

4.4.5 卸载测试用例

```
EFI_STATUS
    BBTestDriverUnload (
        IN EFI_HANDLE          ImageHandle;
    )
```

函数可以释放除了测试协议以外的所有资源，如果从句柄上测试协议接口，同时释放测试入口结点。

4.4.6 黑盒测试协议架构

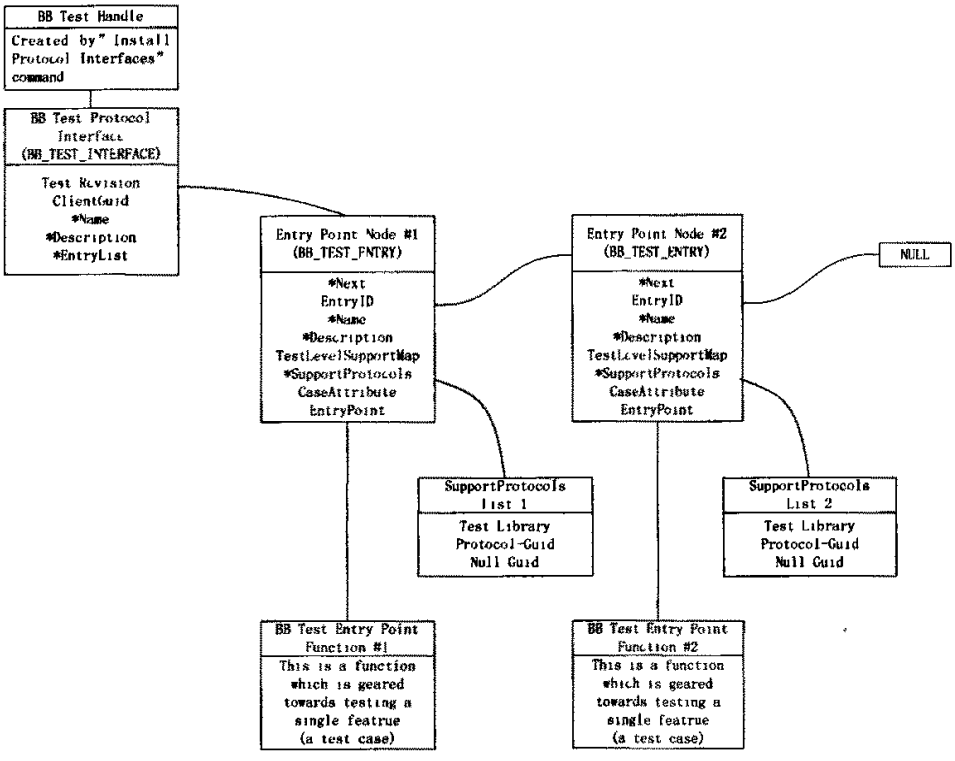


图 4-2 SCT 测试协议接口图

测试协议和具体执行测试的测试函数之间的关系如图4-2所示。一个测试驱动Test Driver包括完成对某个EFI 服务或协议的测试的所有测试函数。一个测试驱动Test Driver包括完成对某个EFI 服务或协议的测试的所有测试函数。

这个测试驱动上安装了测试协议，测试函数用合法或不合法的参数调用来检验被测试服务或协议的接口是否符合协议规范，并返回测试结果。如果被测试的是一个EFI 协议，EFI 系统中的每个协议都有独一无二的编号GUID ，被测试协议的编号存放在测试协议EFI_TEST_PROTOCOL 接口的第二个成员ClientGuid 中。测试框架Framework 根据这个编号查询到系统中支持该协议的设备并把要测试的接口通过测试协议传给测试函数[24]。

测试入口EFI_TEST_ENTRY 结构包括测试函数和成员变量，这些成员变量用来标识测试函数的属性和级别等配置信息，执行测试函数前后可根据这些配置信息完成必要的初始化和结束工作，并且每个测试函数各自独立不会相互干扰。测试入口结构组成的链表存放在测试协议的EntryList 成员中，这样测试框架就能通过测试协议访问每个测试入口继而调用到测试函数。

4.4.7 SCT 黑盒测试用例设计

测试活动分4个步骤进行：（1）确定测试标准；（2）基于测试标准构造测试用例；（3）执行测试用例；（4）分析判断测试结果/。生成测试用例在整个测试活动中占有重要地位，测试用例质量直接影响测试的有效性。如何从大量的测试用例中选择出尽可能少的有效测试用例去发现程序中的错误是软件测试人员研究的目标。构造测试用例取决于第一步的测试用例标准。由于SCT中测试用例众多，所以不列举所有测试用例，这里只选取串口通信的测试用例作为例子，对SCT的黑盒测试用例设计方法加以说明[25]。

串口通信的测试用例分为功能性测试，一致性测试，压力测试。这里重点介绍功能性测试。功能测试指测试软件各个功能模块是否正确，逻辑是否正确。对测试对象的功能测试应侧重于所有可直接追踪到用例或业务功能和业务规则的测试需求。这种测试的目标是核实数据的接受、处理和检索是否正确，以及业务规则的实施是否恰当。

测试流程从SerialIoBbTestMain.c开始，SerialIoBbTestMain.c主要是

初始化黑盒测试接口和入口表。下面分为SerialIoBbTestFunction.c(功能测试)，SerialIoBbTestConformance.c（功能性测试），SerialIoBbTestStress.c。作为支持的文件包括guid.c，guid.h(作用在于自动产生出错信息)；SerialIo.c，SerialIo.h（抽象串口设备使之成为一个符合EFI1.0 的协议）。作为一个串口

设备必须具有几项功能，测试用例中以函数来实现。SERIAL_IO.SetAttributes()，为串口设备设定参数（波特率， 接受数据先进先出的深度，传出与接受数据的超时，奇偶校验，停止位。其中奇偶校验，停止位的默认值是由硬件设备规定的，其余都是0。SERIAL_IO.SetControl()，为串口设备设定控制位。例如：

```
#define EFI_SERIAL_CLEAR_TO_SEND 0x0010
#define EFI_SERIAL_DATA_SET_READY 0x0020
#define EFI_SERIAL_RING_INDICATE 0x0040
#define EFI_SERIAL_CARRIER_DETECT 0x0080
#define EFI_SERIAL_REQUEST_TO_SEND 0x0002
#define EFI_SERIAL_DATA_TERMINAL_READY 0x0001
#define EFI_SERIAL_INPUT_BUFFER_EMPTY 0x0100
#define EFI_SERIAL_OUTPUT_BUFFER_EMPTY 0x0200
#define EFI_SERIAL_HARDWARE_LOOPBACK_ENABLE 0x1000
#define EFI_SERIAL_SOFTWARE_LOOPBACK_ENABLE 0x2000
#define EFI_SERIAL_HARDWARE_FLOW_CONTROL_ENABLE 0x4000
```

SERIAL_IO.Write(把数据写入串口设备的缓冲区)，当 write()函数正在把数据字节写入串口设备中也就是数据被送入串口，如果发生超时错误则传输该缓冲区将结束，返回 EFI_TIMEOUT（）。SERIAL_IO.Read(串口设备中读取数据)。

SERIAL_IO.SetAttributes() 函数对其测试每个参数有三种测试：默认参数的测试，标准参数的测试，非标准参数的测试。以 Timeout 参数举例，SERIAL_IO.SetAttributes()首先做所有参数的默认值测试，每个参数都设置为0。然后测试标准输入参数设定超时数组为 TimeoutArray[] = {1, 1000000, 100000000, 0}，测试数组中的数据是否都能正确执行。

4.5 SCT 工作流

SCT 测试执行是在 EFI Shell 中进行的,由测试框架(Framework)的逻辑控制。主要流程分为初始化测试数据（Attach Test Data）， 执行测试函数（Do Execution） 和释放测试数据（Detach Test Data） 三部分。如图所示：

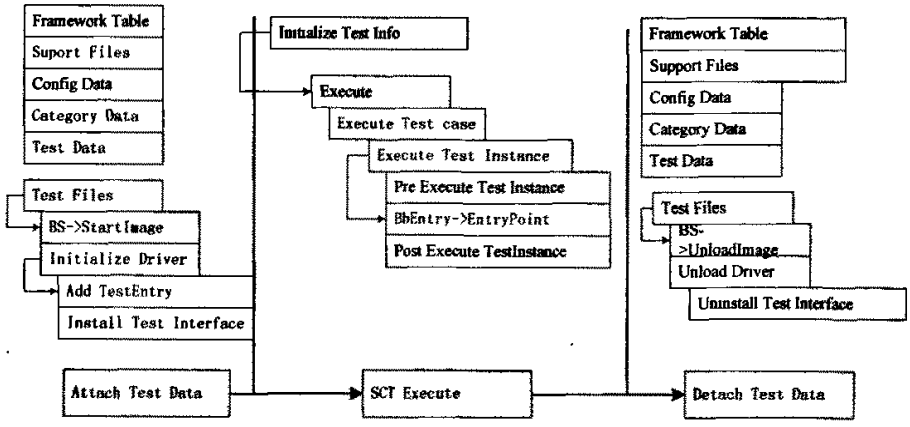


图 4-2 SCT 主流程图

4.5.1 初始化测试数据

初始化数据结构举例：

```
EFI_STATUS
Initialize_Example_BB_Test (
  IN EFI_HANDLE          ImageHandle;
  IN EFI_SYSTEM_TABLE    *SystemTable;
)
```

初始化的功能：创建，初始化测试协议接口（EFI_BB_TEST_PROTOCOl）。启动卸载 image 功能，创建，初始化一个入口结点的链表，并且把这个链表连接到协议的接口上。把测试协议的接口安装到句柄（handle）上。

测试的初始化过程在这一阶段完,成为执行测试函数作准备包括为测试框架分配系统资源,初始化测试框架的数据结构,及建立配置信息(ConfigurationData),等等。最重要的是把每一个测试用例装载（Load ）进来，测试框架得到全局的 EFI_Handle 和 EFI System table,然后用到子目录下所有包含测试用例的.efi 测试驱动文件，调用系统服务 LoadImage()装载文件中的驱动映像（测试所需的库和测试所需的文件）， 用系统服务 StartImage()启动映像并调用驱动（即测试用例）的初始化函数。每个测试用例的初始化函数都建立了测试协议接口并为每个测试函数建立测试入口结构,测试函数入口地址都被记录在测试入口中，然后测试入口记录在测试协议的成员 EntryList 指向的链表中，最后测试协议被安装在测试用例的驱动上。所有检索测试用例所需的信息如测试名称和被测试协议编码等都存储于测试协议的接口中，每

个测试用例对应的测试协议接口初始化之后其指针存储在测试框架的全局变量中备用[26]。

4.5.2 执行测试函数

测试框架管理用户交互的信息。执行用户选择的测试用例时,测试框架调用该测试用例上安装的测试协议接口,获得测试入口链表 EntryList ,从而访问测试入口结构的成员 EntryPoint 即测试函数的入口地址[27]。

如果被测试的是 EFI 服务（Service）， 测试函数通过系统的全局变量获得被测试服务的接口；如果被测试的是 EFI 协议（Protocol）， 测试框架从测试协议接口的成员 ClientGuid 获得被测试协议编号（GUID）,查询系统中安装了被测试协议的设备并把这些设备上的被测试协议接口通过 EFI_ENTRY_POINT 的第二个参数 ClientInterface 传递给测试函数。测试函数将测试每个设备上实现被测试协议的正确性，因此一个测试函数可能被执行多次，每一次执行称为测试函数的一个实例（Instance）[28]。

4.5.3 释放报告

SCT 执行完所有的测试用例，在退出前，测试框架将做一些清理工作.把原来建立起的全局的数据结构配置信息等删除,释放测试协议和测试入口占用的资源再找到从.efi 测试文件载入的文件映像调用 Boot Service 中的 UnloadImage() 来卸载此映像,最后退出。

4.6 本章小结

本章主要介绍 SCT 内部数据结构，架构及 SCT 测试时各个数据结构的调用关系。正因为有着样的数据架构使得 SCT 能够统一的管理测试用例。测试用例通过编译成.efi 文件，作为测试的驱动加载到映象当中。在 SCT 初始化阶段可以被自动检测到并载入。

第 5 章 SCT 实现 TIANO 测试

Tinao 是一项因特尔正在进行的项目，它是 EFI 的一个实现。内置图形驱动功能，可以提供一个高分辨率图形环境，用户进入后可以直接用键盘进行设置。Tiano 能够修改配置或安装新的硬件驱动，将系统成功修复，防止操作系统因为设置问题无法进入。并且支持强大的磁盘管理和启动管理功能，具有脱离操作系统的管理工具，直接进行整机维护工作。Tiano 以硬盘的某个区域作为自己的存储空间，可以执行一些程序，例如硬盘分区、多重操作系统引导等。具有独立的文件系统，能够控制底层硬件[29]。

5.1 应用菜单模式的测试流程

5.1.1 EFI Shell

Shell 环境输入命令 SCT –u 开始调用 SCT 测试的主界面

如果测试使得系统重启，使用 Startup.nsh 脚本文件继续未完成的测试。

fs0: //进入文件系统 FS0（fat 或者是 efi 格式）

cd Framework

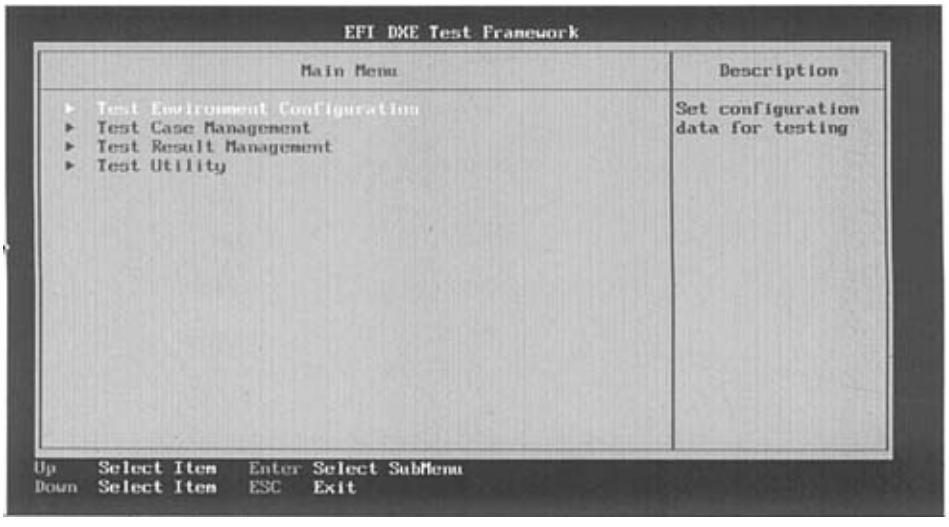
if not exist Framework.Done then

Sct –c //继续测试用例

```
    echo > Framework.Done
  endif
```

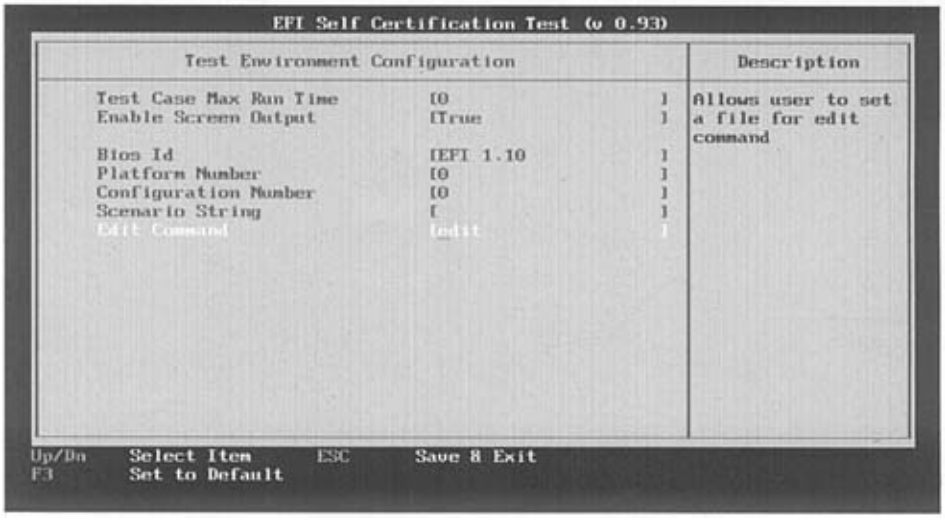
5.1.2 主界面

分为四个部分：测试用例管理——菜单中选择测试用例；测试环境配置——测试配置信息；测试报告生成——报告使用户生成.CSV 格式的报告存放在 Report 子目录下[30]。



重置结果(F4)	重置所有的测试结果在执行完毕后，测试框架会保存所有的测试结果。用户可分开执行测试用例最后一次生成报告这个选项用于删除所有的过时的测试结果。
导入序列(F5)	导出一个测试序列在 Sequence 子目录下找到序列文件根据其信息在 Test Case Management 中选择测试用例。
导出序列(F6)	导出个测试序列保存在当前的Test CaseManagement 中的测试序列保存到序列文件中存放在Sequence 子目录下。
继续执行(F8)	继续执行测试用例测试框架支持系统重起后继续执行测试用例用户可选择继续执行或是重新执行。
执行(F9)	执行所有选中的测试用例

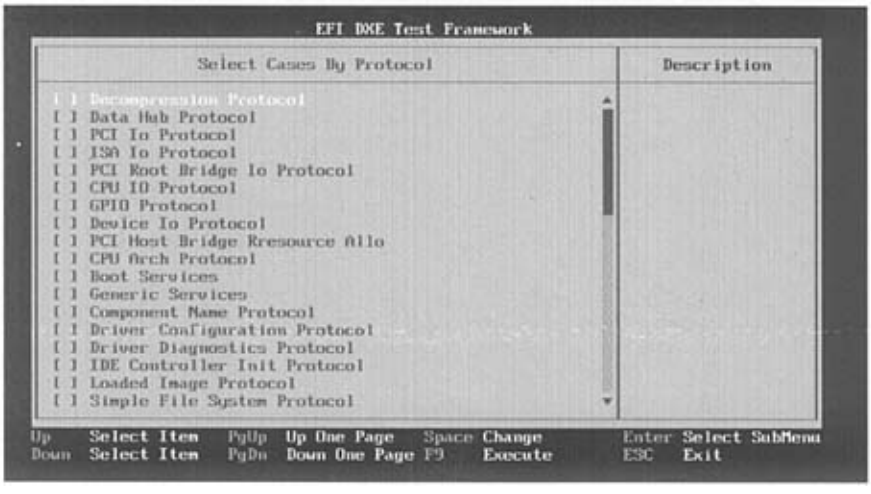
5.1.3 测试环境管理



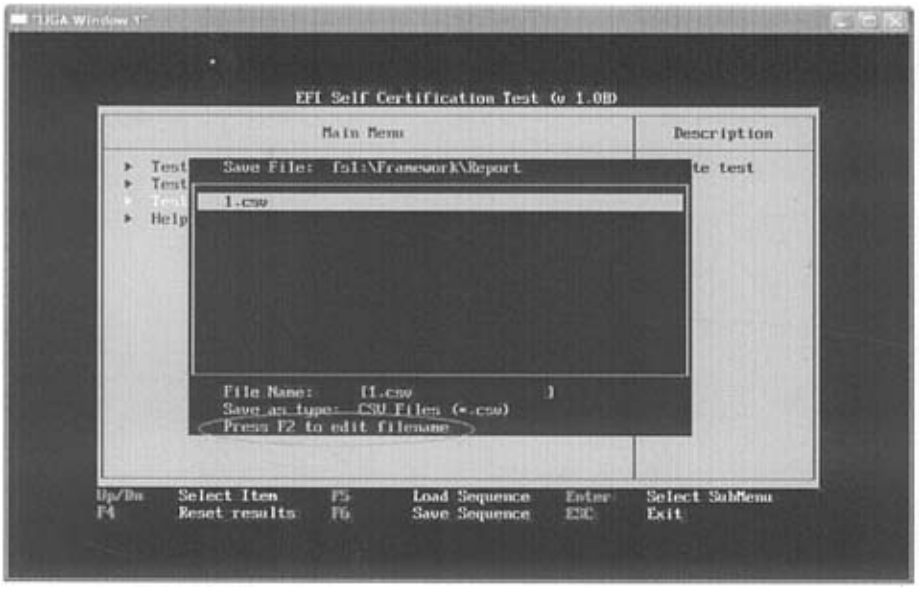
Test Case Max Run Time	每个测试用例的最大运行时间如果用户输入一个非零值测试用例将设置一个 watchdog timer 如果运行时又超时发生系统将发生重起。
Enable Screen Output	控制运行时至信息是否向屏幕输出。
Bios Id	用户输入测试 Bios 的信息
Platform Number	用户输入测试平台的信息
Configuration Number	用户输入测试配置的信息
Scenario String	用户输入的描述测试环境的信息

5.1.4 测试用例管理

测试用例管理用来管理选中要测试的用例。



5.1.5 测试报告生成器



5.1.6 报告生成

测试函数执行过程中,会把测试数据记录在日志文件中,存储在磁盘上。在被测试服务或协议接口的每个检测点上都会生成一条测试信息，即这个检测点的测试是成功（Pass） 还是失败（Failure）。 日志文件按照 EFI 规范章节分类对于每一个协议（Protocol ）的接口（Interface）都有一个.log 日志文件和.ekl 日志文件.log 日志文件是可读的文件.ekl Efi Ket Log file 是供报告生成器分析的日志文件两者仅在格式上有所区别。在所有的 Test Case 运行完毕后，SCT 自带的一个报告生成器会检索所有的.ekl 日志文件根据 EFI 规范章节分类获取每个检测点的详细信息根据 Pass 和 Failure 进行归类统计生成最后 CSV 格式的报表。CSV 格式的文件在 Windows 平台与 Linux 平台都有应用程序可解析用户只需察看 CSV 报表便可知道当前的 EFI 实现的服务和协议的状况。

5.2 测试结果分析

BroadwaterX64Uefi64UefiSCT.csv 文件是在 Desktop Broadwater 平台上对 UEFI x64 位的 Tinao BIOS 的 SCT 报告日志文件。

Self Certification Test Report

Service\Protocol Name	Total	Failed	Passed
Generic Test\EFI Compliant Test	15	0	15
Boot Services Test\Event, Timer, and Priority Services Test	30	0	30
Boot Services Test\Memory Allocation Services Test	134	0	134
Boot Services Test\Protocol Handler Services Test	1203	0	1203
Boot Services Test\Image Services Test	118	0	118
Boot Services Test\Misc Boot Services Test	132	0	132
Runtime Services Test\Variable Services Test	52	0	52
Runtime Services Test\Time Services Test	84	0	84
Runtime Services Test\Misc Runtime Services Test	6	0	6
Loaded Image Protocol Test	2622	0	2622
Device Path Procotols\Device Path Procotol Test	307	0	307
Device Path Procotols\Device Path Utilities Procotol Test	17	0	17
Device Path Procotols\Device Path To Text Procotol Test	41	0	41
Device Path Procotols\Device Path From Text Procotol Test	41	0	41
Driver Model Test\Component Name Protocol Test	255	0	255
Console Support Test\Simple Input Protocol Test	16	0	16
Console Support Test\Simple Output Protocol Test	153	0	153
Console Support Test\Simple Pointer Protocol Test	18	0	18
Console Support Test\Serial IO Protocol Test	28	0	28
Console Support Test\Graphics Output Protocol Test	110	0	110
Bootable Image Support Test\Simple File System Protocol Test	615	6	609
Bootable Image Support Test\Disk IO Protocol Test	16	1	15
Bootable Image Support Test\Block IO Protocol Test	33	1	32
Bootable Image Support Test\Unicode Collation Protocol Test	12	0	12
PCI Bus Support Test\PCI Root Bridge IO Protocol Test	58	0	58
PCI Bus Support Test\PCI IO Protocol Test	1730	0	1730
USB Support Test\USB Host Controller Protocol Test	185	0	185
USB Support Test\USB2 Host Controller Protocol Test	287	0	287
USB Support Test\USB IO Protocol Test	74	0	74
Network Support Test\Simple Network Protocol Test	42	0	42
Network Support Test\PXE Base Code Protocol Test	8	0	8
Debugger Support Test\Debug Support Protocol Test	2	0	2
Compression Test\Decompress Protocol Test	13	0	13
EFI Byte Code Test\EBC Interpreter Protocol Test	3	0	3
Total service\Protocol	8460	8	8452

5.2.1 Log 分析与问题的定位

Service\Protocol Name	Index	Instance	Iteration	Guid	Result	Title	Runtime Information	Case Revision	Case GUID
Bootable Image Support Test\Simple File System	5.7.3.	1	0	C5DA4980-0	FAIL	EFI_FILE.Cl	C:\ReleaseTest\UefiSc	0x00010000	3EAEA622-
Bootable Image Support Test\Simple File System	5.7.3.	1	0	B9478756-4	FAIL	EFI_FILE.Cl	C:\ReleaseTest\UefiSc	0x00010000	3EAEA622-
Bootable Image Support Test\Simple File System	5.7.3.	1	0	B656663F-5	FAIL	EFI_FILE.De	C:\ReleaseTest\UefiSc	0x00010000	B709D88F-
Bootable Image Support Test\Simple File System	5.7.3.	1	0	B0678DAE-5	FAIL	EFI_FILE.De	C:\ReleaseTest\UefiSc	0x00010000	B709D88F-
Bootable Image Support Test\Simple File System	5.7.3.	1	0	0F51D637-4	FAIL	EFI_FILE.De	C:\ReleaseTest\UefiSc	0x00010000	B709D88F-
Bootable Image Support Test\Simple File System	5.7.3.	1	0	B9C79E4E-1	FAIL	EFI_FILE.De	C:\ReleaseTest\UefiSc	0x00010000	B709D88F-
Bootable Image Support Test\Disk IO Protocol Tes	5.7.4.	0	0	B0D7A6E7-4	FAIL	EFI_DISK_IO_C	C:\ReleaseTest\UefiSc	0x00010000	75DFBA09-
Bootable Image Support Test\Block IO Protocol Te	5.7.5.	0	0	91CFDE2C-6	FAIL	EFI_BLOCK_IO_C	C:\ReleaseTest\UefiSc	0x00010000	82615903-

.env 文件包括出错信息的详情

Serviec\Protocol 名称：说明测试的是哪项服务的指定协议(Bootable Image Support Test\Block IO Protocol Test)。Instance:共执行了几个实例，GUID ：测试用例的 GUID（91CFDE2C-619E-4C88-800D-99CE53AD3B25），可以从 EFI Self-Certification Test Case Specification 查询得到本次测试详细原因。Result：表示这项测试是通过（ Pass）还是失败(Fail).Title 表示这项测试失败的大概原因（EFI_BLOCK_IO_PROTOCOL.Readblocks - ReadBlocks() returns EFI_NO_MEDIA with no media present in the device. ）。RuntimeInformation: 说明测试出错的详细代码位置（C:\ReleaseTest\UefiSct\Platform\IntelTest\TestCase\EFI\Protocol\BlockIo\BlackBoxTest\BlockIoBBTestConformance.c:415:BufferSize=512, Status=Device Error, Expected=No Media）等等。

5.2.2 Bug 的修正

修正 Bootable Image Support Test\Block IO Protocol Test 失败的 bug。根据 GUID 91CFDE2C-619E-4C88-800D-99CE53AD3B25 在 EFI Self-Certification Test Case Specification 中查询，得到测试描述为：For device with MediaPresent being FALSE:1. Call ReadBlocks() with validparameter.Expected Behavior:The return code must be EFI_NO_MEDIA.，得出传入测试用例的参数错误。接着到 Tinao 代码中找到这个参数。测试失败的最终原因为：当一个空的盘插入到设备中，也就是说 Tiano 把它作为最后一个块设备并且为 0，AtapiDetectMedia 函数就认为这个设备是错误的。解决方案：假定磁盘容量是空的时候,设备的依旧是正确的。代码见附录。

5.3 本章小结

本章主要介绍了基于 Tiano 的 SCT 的测试流程，以及测试报告的分析。然后举一个例子说明怎样根据 SCT 给出的测试结果来修正错误。SCT 同样就有平台无关性，能够在所有符合 EFI/UEFI 规范的平台上运行.测试报告的 CSV 格式也都是跨平台的,使 SCT 易于移植. 用户界面简洁清晰方便用户管理测试。

第 6 章 结束语

由于传统 BIOS 的种种弊病，Intel 推出了一种用来替代现有的传统 BIOS 的全新的技术—EFI Extensible Firmware Interface EFI 全称是可扩展固件接口 EFI。EFI 是在操作系统与平台固件（platform firmware）之间的一套完整的接口规范。EFI 定义了许多重要的数据结构以及系统服务，如果完全实现了这些数据结构与系统服务，也就相当于实现了一个真正的 BIOS 核心。

本论文的研究是利用软件手动或自动化测试技术测试基于 EFI（Extensible Firmware Interface）和 UEFI（Unified Extensible Firmware Interfac）规范（efi spec and uefi spec）的新型 BIOS--Tinao 软件。以基于 EFI /UEFI 计算机的预启动环境为背景，将 EFI 的驱动模式技术以及 EFI 的事件机制相结合。着重研究了可扩展（EFI）的平台架构包括：EFI 标准扩展接口的结构模式的分析，EFI 平台的启动过程，平台的管理过程，其中 EFI 特有的关键概念（协议）的介绍。

根据正在蓬勃兴起的软件管理技术，结合传统的软件测试原理，开发适合 EFI/UEFI 的自我认证测试系统(SCT)。它会测试每一个 EFI 规范中定义的启动时服务 Boot Service, 运行时服务 Runtime Service 和协议 Protocol 的实现,从而来验证这个实现是否符合规范本身。所有的测试用例都是 EFI 软件的驱动（Driver）,测试通过统一的接口和唯一的出错标识（GUID）管理和执行测试用例，并且获得统一格式的测试日志和测试报告。分析测试日志和报告，可以定位到软件出错位置并且修订 bug。

参考文献

[1] EFI 1. 1 Specification, ver. 1. 1 [DB]. Intel Corporation, http: //developer. intel. com/technoledge /efi, 2002。

[2] R.Avresky,Formal Verification and Testing of Protocol,Computer communications 22,1999,p681-690。

[3] Mark Doran, EFI 1. 10 and Beyond An Overview, Intel Developer Forum 2003 Spring, 2003。

[4] Mark Doran, Extensible Firmware Interface Changing the face of Bios, IntelDeveloper Forum 2001 Spring, 2001。

[5] Brian Richardson, Robert P. Hale, BIOS Compatibility within the Intel Platform Innovation Framework for EFI, Intel Developer Forum 2003 Fall, 2003。

[6] Brian Richardson, Robert P. Hale, BIOS Compatibility within the Intel Platform Innovation Framework for EFI, Intel Developer Forum 2003 Fall, 2003。

[7] Michael Kinney, EFI 1.1 Driver Writer's Guide Presentation, Intel Developer Forum 2001 Spring, 2001。

[8] Michael Kinney, EFI 1.1 Driver Model and Protocol Services Overview, Intel Developer Forum 2001 Fall, 2001。

[9] Dong Wei, Extending EFI the Right Way : Case Studies, Intel Developer Forum 2001 Fall, 2001。

[10] Harry Hsiung, EFI 1.1 Driver Model Demonstration, Intel Developer Forum 2001 Fall, 2001。

[11] R.Avresky,Formal Verification and Testing of Protocol,Computer communications 22,1999,p681-690。

[12] 许胜，支定镛等. 航天软件工程实施技术指南及范例. 北京:航天 5 院，2003。

[13] 万年红，李翔，软件黑盒测试的方法与实践，计算机工程，2000. 12. 26。

[14] Micah Elliott, Building Extensions for the EFI Shell, Intel Developer Forum 2001 Fall, 2001。

[15] Vincent Zimmer, EFI Specification Evolution, Intel Developer Forum 2003Spring, 2003。

[16] Intel Platform Innovation Framework for EFI Self Certification Test Specification Intel Corporation，http：//developer. intel.com/ technoledge / efi，2003。

[17] Michael S. Richmond, Intel's BIOS Replacement Roadmap, Intel Developer Forum 2004 Fall, 2004。

[18] Daniel J.Mosely,Bruce A.posey.Just Enough Software Test Automation..NewYork:Pearson Education,2002.87-91。

[19] Paul C.Jorgensen. Software Testing -A Craftsman ' s Approach. 2,Florida:CRC Press,2002.260-296。

[20] Roy Wade, Case Study : Porting to the EFI 1.1 Driver Model, Intel Developer Forum 2001 Fall, 2001。

[21] Yosi Govezensky, EFI Update & EFI Application Tookit : Providing a jump start to EFI application development and a uniform pre-boot environment, Intel Developer Forum 2000 Spring, 2000。

[22] Harry Hsiung, Implementing EFI 1.0, Intel Developer Forum 2000 Fall, 2000。

[23] Michael Kinney, Curtis E. Stevens, EFI 1.1 Presentation, Intel Developer Forum 2000 Fall, 2000。

[24] Vincent Zimmer, Michael Kinney, Non-IA Silicon Support within the Intel Platform Innovation Framework for the Extensible Firmware Interface, Intel Developer Forum 2003 Fall, 2003。

[25] Andrew Fish, Evolving Firmware for Remote Manageability, Intel Developer Forum 2003 Spring, 2003。

[26] Intel Corporation, EFI 1.10 Driver Writer's Guide, Version 0.7, 2003, 177-201,211-215, 323-337。

[27] Geoffrey Cox, Debugging C-Source Code for EFI, American Arium, Tustin, CA,2001。

[28] Michael Kinney, Dong Wei, Writing and Debugging EFI Drivers, IntelDeveloper Forum 2003 Spring, 2003。

[29] Rob Haydt, Windows, EFI, and Testing : Supporting 32-bit and 64-bit platforms,Intel Developer Forum 2004 Spring, 2004。

[30] Harry Hsiung, Manufacturing and Test solutions with EFI, Intel Developer Forum 2003 Spring, 2003。

附录一 实现代码

EFI SCT 初始化数据结构实现代码

```
EFI_DRIVER_ENTRY_POINT(InstallSct)
EFI_STATUS
InstallSct (
  IN EFI_HANDLE            ImageHandle,
  IN EFI_SYSTEM_TABLE      *SystemTable
  )
{
  EFI_STATUS   Status;
  CHAR16       *DirName;
  //
  // Initialize library
  //
  EFI_SHELL_APP_INIT (ImageHandle, SystemTable);
  //
  // Check parameters
  //
  if (SI->Argc != 1) {
    PrintUsage ();
    return EFI_SUCCESS;
  }
  gImageHandle = ImageHandle;
  //
  // Get the destination directory
  //
  Print (L"\nGather system information ...\n");
  Status = GetDestination (&DirName);
  if (EFI_ERROR (Status)) {
    return Status;
  }
```

Bug fix 的源代码

```
diff -urN --binary -a -x CVS Oldv1.3/Edk/Sample/Bus/Pci/IdeBus/Dxe/atapi.c
Newv1.3/Edk/Sample/Bus/Pci/IdeBus/Dxe/atapi.c
--- Oldv1.3/Edk/Sample/Bus/Pci/IdeBus/Dxe/atapi.c     2007-03-09
14:34:32.615371600 +0800
+++ Newv1.3/Edk/Sample/Bus/Pci/IdeBus/Dxe/atapi.c     2007-03-09
14:34:32.615371600 +0800
@@ -897,7 +897,7 @@
 EFI_STATUS
 AtapiTestUnitReady (
   IN IDE_BLK_IO_DEV          *IdeDev,
-  OUT UINTN                  *SenseCount
+  OUT SENSE_RESULT           *SResult
   )
 /*++
   Name:
@@ -905,7 +905,7 @@
```

```
    Purpose:
            Sends out ATAPI Test Unit Ready Packet Command to the specified device
-           to find out whether device is accessible. Sense count is requested
+           to find out whether device is accessible. Sense result is requested
            after this packet command.

    Parameters:
@@ -913,14 +913,14 @@
              pointer pointing to IDE_BLK_IO_DEV data structure, used
              to record all the information of the IDE device.

-           SenseCount          OUT     *SenseCount
-             sense count for this packet command.
+           SENSE_RESULT        OUT     *SResult
+             sense result for the command
    Returns:
            EFI_SUCCESS
              command is issued and sense data is requested successfully.

            EFI_DEVICE_ERROR
-             exceptional error without sense data (*SenseCount == 0)
+             exceptional error without sense data (SResult is undefined)

    Notes:

@@ -928,8 +928,7 @@
  {
    ATAPI_PACKET_COMMAND    Packet;
    EFI_STATUS              Status;
-
-   *SenseCount = 0;
+   UINTN                   SenseCount;

    //
    // fill command packet
@@ -945,12 +944,13 @@
      return Status;
    }

-   Status = AtapiRequestSense (IdeDev, SenseCount);
+   Status = AtapiRequestSense (IdeDev, &SenseCount);
    if (EFI_ERROR (Status)) {
-     *SenseCount = 0;
      return Status;
    }

+   ParseSenseData (IdeDev, SenseCount, SResult);
+
    return EFI_SUCCESS;
  }

@@ -1075,7 +1075,7 @@
  EFI_STATUS
  AtapiReadCapacity (
    IN IDE_BLK_IO_DEV       *IdeDev,
-   OUT UINTN               *SenseCount
```

```
+   OUT SENSE_RESULT            *SResult
    )
 /*++
    Name:
@@ -1097,8 +1097,8 @@
              pointer pointing to IDE_BLK_IO_DEV data structure, used
              to record all the information of the IDE device.

-           UINTN                 OUT     *SenseCount
-             sense count for this packet command.
+           SENSE_RESULT      OUT     *SResult
+             sense result for this packet command.
    Returns:
           EFI_SUCCESS
             Packet command is issued and sense data is requested
@@ -1106,7 +1106,7 @@
             decide whether this command is successful.

           EFI_DEVICE_ERROR
-            Exceptional error without sense data (*SenseCount == 0)
+            Exceptional error without sense data (SResult is undefined)

    Notes:
           parameter "IdeDev" will be updated in this function.
@@ -1117,6 +1117,7 @@
    EFI_STATUS                  Status;
    EFI_STATUS                  SenseStatus;
+   UINTN                       SenseCount;
    ATAPI_PACKET_COMMAND        Packet;

    //
@@ -1125,8 +1126,6 @@
    READ_CAPACITY_DATA          Data;
    READ_FORMAT_CAPACITY_DATA FormatData;

-   *SenseCount = 0;
-
    EfiZeroMem (&Data, sizeof (Data));
    EfiZeroMem (&FormatData, sizeof (FormatData));

@@ -1159,15 +1158,16 @@
    }

    if (Status == EFI_TIMEOUT) {
-     *SenseCount = 0;
      return Status;
    }

-   SenseStatus = AtapiRequestSense (IdeDev, SenseCount);
+   SenseStatus = AtapiRequestSense (IdeDev, &SenseCount);

    if (!EFI_ERROR (SenseStatus)) {

-     if (!EFI_ERROR (Status)) {
+     ParseSenseData (IdeDev, SenseCount, SResult);
+
```

```
+     if (!EFI_ERROR (Status) && *SResult == SenseNoSenseKey) {

        if (IdeDev->Type == IdeCdRom) {

@@ -1176,19 +1176,8 @@
              (Data.LastLba1 << 8) |
              Data.LastLba0;

-           if (IdeDev->BlkIo.Media->LastBlock != 0) {
-
-             IdeDev->BlkIo.Media->BlockSize = (Data.BlockSize3 << 24) |
-               (Data.BlockSize2 << 16) |
-               (Data.BlockSize1 << 8) |
-               Data.BlockSize0;
-
-             IdeDev->BlkIo.Media->MediaPresent = TRUE;
-           } else {
-             IdeDev->BlkIo.Media->MediaPresent = FALSE;
-             return EFI_DEVICE_ERROR;
-           }
-
+           IdeDev->BlkIo.Media->MediaPresent = TRUE;
+
            IdeDev->BlkIo.Media->ReadOnly = TRUE;

            //
@@ -1234,7 +1223,6 @@
       return EFI_SUCCESS;

     } else {
-      *SenseCount = 0;
       return EFI_DEVICE_ERROR;
     }
   }
@@ -1283,7 +1271,6 @@
     EFI_BLOCK_IO_MEDIA                OldMediaInfo;
     UINTN                             RetryTimes;
     UINTN                             RetryNotReady;
-    UINTN                             SenseCount;
     SENSE_RESULT                      SResult;
     BOOLEAN                           WriteProtected;

@@ -1306,7 +1293,7 @@
     RetryTimes = 5;
     while (RetryTimes != 0) {

-      Status = AtapiTestUnitReady (IdeDev, &SenseCount);
+      Status = AtapiTestUnitReady (IdeDev, &SResult);

       if (EFI_ERROR (Status)) {
         //
@@ -1328,8 +1315,6 @@
         continue;
       } else {

-        ParseSenseData (IdeDev, SenseCount, &SResult);
-
```

```
            switch (SResult) {
            case SenseNoSenseKey:
              if (IdeDev->BlkIo.Media->MediaPresent) {
    @@ -1383,15 +1368,13 @@

          while (RetryTimes != 0) {

    -       Status = AtapiReadCapacity (IdeDev, &SenseCount);
    +       Status = AtapiReadCapacity (IdeDev, &SResult);

            if (EFI_ERROR (Status)) {
              RetryTimes--;
              continue;
            } else {

    -         ParseSenseData (IdeDev, SenseCount, &SResult);
    -
              switch (SResult) {
              case SenseNoSenseKey:
                goto Done;
    diff -urN --binary -a -x CVS Oldv1.3/Edk/Sample/Bus/Pci/IdeBus/Dxe/ide.h
Newv1.3/Edk/Sample/Bus/Pci/IdeBus/Dxe/ide.h
    --- Oldv1.3/Edk/Sample/Bus/Pci/IdeBus/Dxe/ide.h 2007-03-09 14:34:32.615371600
+0800
    +++ Newv1.3/Edk/Sample/Bus/Pci/IdeBus/Dxe/ide.h 2007-03-09
14:34:32.615371600 +0800
    @@ -937,7 +937,7 @@
     EFI_STATUS
     AtapiTestUnitReady (
       IN   IDE_BLK_IO_DEV   *IdeDev,
    - OUT UINTN             *SenseCount
    + OUT SENSE_RESULT      *SResult
       )
     /*++

    @@ -982,7 +982,7 @@
     EFI_STATUS
     AtapiReadCapacity (
       IN   IDE_BLK_IO_DEV   *IdeDev,
    - OUT UINTN             *SenseCount
    + OUT SENSE_RESULT      *SResult
                                        )
```

致 谢

两年求学时光转瞬即逝，回首往昔，一种难以割舍的情结油然而生。

感谢导师贾智平教授的谆谆教诲，无论我将来身在何方，导师严谨求实的治学精神、任劳任怨的工作态度及正直、诚信的人格魅力将激励我终身！

感谢所有关心和帮助我的老师和同学！

感谢父母的辛勤培育，你们的鼓励和关爱将是我人生的最大财富！

此文仅仅是学习阶段的一个总结，更重要的是新的起点，在今后的人生中，我将会记住一句话：气有浩然，学无止境！