南京航空航天大学

硕士学位论文

EFI接口BIOS驱动体系的设计、实现与应用

姓名：石浚菁

申请学位级别：硕士

专业：计算机应用技术

指导教师：李真

20060301

摘要

针对 20 多年来 PC BIOS 一直没有大改进的现状，Intel 推出了 EFI(Extensible Firmware Interface)接口标准，并联合主要 IT 巨头坐在了一起组成了 UEFI 联盟。目前，正值传统 BIOS 向新一代 BIOS 规范的过渡过程，是一个发展自主知识产权的国产 BIOS 技术与产品的极好时机，可以预见新一代的 BIOS 必将成为下一个核心技术的热点，我们研究并掌握它具有重要意义。

本文依据 EFI 接口标准,结合 Intel 对 EFI 实现提供的 Framework 参考模型，针对主流 IA32 架构的 PC 机，给出了符合 EFI 接口规范的新一代 BIOS 驱动架构，将 BIOS 设计成多层次、多阶段的体系架构，赋予了新架构较大的跨平台性。

本文介绍了如何在具体的主板上，实现上述符合 EFI 接口规范的 BIOS 驱动体系，重点是描述 EFI 驱动在具体主板上的设计和实现方法。本文针对 Intel440 BX 芯片主板给出了符合 EFI 接口 BIOS 的驱动体系设计，并给出了体系中具体驱动的实现例子和基于此驱动体系实现的安全应用，在驱动体系大框架下实现了启动 Windows、Linux、Dos 操作系统。Intel440BX 主板是经典的 IA32 架构主板，当今主流 PC 主板基本沿用了 Intel440BX 主板的设计。

本文的驱动实例重点介绍了 USB 驱动的实现方法，分别在 EFI 和传统 BIOS 不同的架构下给出了详细的设计和实现细节。

本文同时介绍了两个安全应用，一个是基于硬盘驱动的硬盘安全方案，可以同时应用在 EFI 架构的 BIOS 和传统 BIOS 上；另一个是 EFI 固件的安全方案，是针对本文在 Intel440BX 主板上的 BIOS 设计给出的多层次安全方案。

关键词： BIOS、EFI、Framework、 Driver、PC 体系结构、IA32、USB、Hardware

Abstract

PC BIOS, which is the interface between platform hardware and operating system, hasn't changed much since two decades ago. The traditional BIOS become a bottleneck in the current PC systems. To get rid of the traditional BIOS, Intel delivered new BIOS interface, called EFI, and associated with other IT giants to form an alliance, called UEFI, to popularize the new BIOS interface. It is a good chance to enter the BIOS area and develop Chinese own BIOS.

This thesis gives a new generation BIOS drivers' architecture which is based on Intel EFI interface and Framework code reference. The new BIOS drivers' architecture has multi-layers and multi-phases, which has strong portability.

On the base of the BIOS drivers' architecture design, this thesis gives some specific hardware BIOS drivers in Intel 440BX platform. Especially, the paper presents a full implement of USB1.1 driver stack which have EFI interface and traditional bios interface, the driver offers various functions controlling hardware directly. This paper analyses the architecture and key flows of this driver by UML.

Finally some security solutions are presented. The solutions are different from the traditional ways, and they are not software-only but are based on hardware chipset. The solutions not only solve the BIOS security problem but also the PC's. The solutions meet the demands of the trusted PC security standards and it is practical. Two associated application are described to show how the solution solves PC security problems.

Keywords: BIOS、EFI、Framework、 Driver、PC Architecture、IA32、USB、Hardware

承诺书

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

本人授权南京航空航天大学可以有权保留送交论文的复印件，允许论文被查阅和借阅，可以将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。

作者签名：_____

日　　期：_____

图表目录

图 3.1 驱动体系层次结构图 8
图 3.2 驱动体系交互关系图 9
图 4.1 初始化层结构图 13
图 4.2 EFI 对象 15
图 4.3 句柄库示意图 17
图 4.4 驱动与协议关系图 18
图 4.5 输入控制台设备连接图 25
图 4.6 驱动执行层的结构图 26
图 4.7 操作系统启动选择层位置图 27
图 4.8 CSM 交互图 29
图 4.9 有 CSM 的驱动层结构图 29
图 5.1 主板的结构框图[6] [7] 30
图 5.2 USB 驱动层次结构 32
图 5.3 EFI 接口 USB 驱动栈 33
图 5.4 CSM 模块 USB 驱动架构逻辑骨架 38
图 5.5 CSM 模块 USB 驱动架构骨架及实例 40
图 5.6 USB 总线初始化请求顺序图 41
图 5.7 打开请求顺序图 42
图 5.8 UHCI 主机控制传输 44
图 6.1 BIOS 安全策略 45
图 6.2 加入本安全策略后各部分交互关系 46
图 6.3 TPM 方案层次图 48
图 6.4 TPM 方案对传统 BIOS 安全启动的设计 49
图 6.5 TPM 一个结合网络的应用 50
图 6.6 硬盘安全方案体系结构及各个部分交互关系 51
图 6.7 硬盘方案控制程序和一般应用程序层次对比图 53
图 6.8 安全控制实施例 54
图 6.9 USB 硬盘锁实施例部署图 56
图 6.10 USB 硬盘锁实施例部署图 57

注释表

EFI ——Extensible Firmware Interface 的简称，可扩展固件接口。Intel 公司提出的下一代固件接口规范。

ATA —— Advanced Technology Attachment 的简称，高级技术附加装置。是近年来通用的一种硬盘接口标准。[35]

BIOS —— 基本输入输出系统，Basic Input and Output System 的简称。计算机上的一块固件代码，用于硬件上电初始化和系统引导。[35]

CACHE ——介于 CPU 和主存之间的高速小容量存储器称为高速缓冲存储器，简称 Cache。[35]

CHS —— Cylinder-Head-Sector，采用柱面号、磁头号、扇区号进行硬盘设备寻址访问的方式。[35]

DMA—— Direct Memory Access，直接内存存取。是 I/O 设备与主存储器之间由硬件组成的直接数据通道，用于高速 I/O 设备与主存储器之间的成组数据传送。[35]

DRAM —— 动态 RAM。采用刷新电容方式存储数据方式。[35]

EXT2 —— Linux 采用的一种主流文件系统。[35]

LBA —— Logical Block Address，采用扇区线性地址进行硬盘设备寻址访问的方式。[35]

MBR —— 主引导扇区 (Master Boot Record)。在硬盘的第一个柱面的最开始的一个 512 字节的扇区，上面有一段引导程序和分区表。[35]

MSR —— Model Specific Registers，是一组主要用于进行调试控制、性能监测、机器检验（如用于微码更新）、以及内存类型范围设定（MTRR）的寄存器。[35]

MTRR —— Memory Type Range Registers，内存类型范围设定寄存器。可以通过它们对不同的存储区域进行不同的 cache 设置。[35]

PCI —— Peripheral Component Interconnect，外设组件互联。基于奔腾等新一代微处理器而发展的总线。打破高速数据传输高性能局部总线。[35]

PIO —— Programmed Input/Output 的简称，通过设备寄存器访问的一种数据传输方式。[35]

SMBUS —— 系统组件管理总线，是 System Management Bus 的简称，该协议用于对外围零组件进行检测、定位、读写参数等设置。[35]

SPD —— Serial Presence Detect，串行存在探测。[35]

USB —— Universal Serial Bus，即通用串行总线。是应用在 PC 领域的新型接口技术，是为了解决日益增加的 PC 外设与有限的主板插槽和端口之间的矛盾而制定的一种串行通信的标准。[35]

保护模式 —— 保护模式是 32 位处理器本身的操作模式，在这种模式下，所有的指令和特定结构能发挥最高的性能和兼容性。保护模式下可实现分段存储和分页存储。[35]

北桥（NorthBridge） —— PC 主板上负责沟通 CPU 和主存，内存与 AGP 接口显卡的集成芯片。[35]

超级 IO 芯片（Superio） —— PC 主板上负责整合较为中低速率的接口的集成芯片。[35]

南桥（Southbridge） —— PC 主板上负责沟通 PCI 外围设备、IDE/USB 装置、ISA 以及其他外设端口的集成芯片。[35]

实模式 —— x86 处理器的一种操作模式。在实模式下 CPU 寻址是由逻辑地址(Logical Address)中的段值(Segment Selector)和偏移(Offset)通过如下形式形成的。[35]

第一章 绪论

1.1 引言

1.1.1 发展国产 **BIOS** 的意义

BIOS 作为 PC 核心技术中的关键一环，其功能也在不断拓展，未来必将在整个 PC 体系中扮演更重要的角色。目前，正值传统 BIOS 向新一代 BIOS 规范的过渡过程中，是一个发展自主知识产权的国产 BIOS 技术与产品的极好时机。

首先，从经济角度来看发展国产 BIOS 的必要性。

随着我国 PC 硬件产业的迅猛发展，世界上更多的 PC 主板和整机将变为中国制造。而作为 PC 必备的 BIOS 主要源于美国 Phoenix 等公司，这样中国公司每生产一块 PC 主板就要支付美国公司相应 BIOS 的使用许可费用。基于 PC 的庞大生产数量，这也是一笔不小的费用支出。

其次，从安全角度来看发展国产 BIOS 的必要性。

PC 流行至今，已经覆盖了社会各个应用领域，其安全问题也日趋严重。作为系统软件核心的操作系统的重要性已广为人知，但对于 PC 而言，开机执行的第一条指令以及第一组程序并非源自操作系统，而是主板 BIOS。如今，主板 BIOS 程序容量逐步增大，Intel 主推的 EFI 更是将 BIOS 的功能推向复杂极致。

BIOS 在操作系统获得系统控制权之前，BIOS 掌管 PC 的一切资源。如果其中留下后门或者被感染病毒，即使格式化硬盘，甚至更换硬盘，也无济于事。

因此，我们国防部门、政府机关等重要电脑应用领域在担忧 Windows 的同时，更多的时候忽略了 BIOS，Windows 可以用国产 Linux 替代，而 BIOS 无可替代，且不可或缺。

最后，从发展角度来看发展国产 BIOS 的必要性。

可以预见新一代的 BIOS 必将成为下一个核心技术的热点，我们研究并掌握它具有重要意义。研发国产 BIOS 不仅可以填补该领域的国内空白，而且也是实现 PC 产品真正国产化的必经之路。

综上所述，不论基于经济因素，安全因素，还是发展因素等，都有必要研制中国自己的 PC BIOS，并在此基础上，发展国产操作系统，即可构筑真正令人放心的国产系统软件平台。

1.1.2 如何发展国产 **BIOS**

BIOS 作为衔接硬、软件的桥梁，技术涉及面广，需要 PC 硬件核心厂商和操作系统厂商的支持，开发难度大，而其推广应用则有赖于主板和整机厂商的支持。因此，发展国产 BIOS，并成功投入商业应用推广不是一件简单的事情。

- **BIOS 系统方案选择**

Intel 的 EFI 1.10 规范获得了众多 IT 厂商的支持，但因 UEFI 联盟的诞生，EFI 规范将止步于 1.10，Intel 将不会再发布更新版本的 EFI 规范。目前 UEFI 尚未制订出新的 BIOS 规范，暂时将以 EFI 1.10 作为一个过渡的起始规范。因此，新的 BIOS 系统方案满足 EFI 1.10 规范是一个不错选择。

但是 EFI 1.10 并非一个完备的 BIOS 系统方案，例如其并未制订出硬件初始化模块的细节（传统 BIOS 的 Power On Self Test）以及并未将 PC 的安全性因素充分纳入规范等。

对于国产 BIOS 的发展而言，这也是一个很好的机遇。可以借鉴各种 BIOS 新老规范，并融入系统安全设计等因素，形成自己有特色和竞争力的 BIOS 系统方案。

- **BIOS 的兼容性**

PC 的发展历史无数次证明了兼容性的重要性，因此新一代安全 BIOS 必须充分考虑到新老硬软件的兼容问题。但因为传统 BIOS 的内核为 16 位的实模式工作方式，而诸如 EFI 这样的新 BIOS 规范的内核是 32 位的保护模式工作方式，因此，看起来似乎要实现兼容是一个矛盾，但实际上解决这些问题的技术手段不止一种。

对于国产 BIOS 而言，为了保证市场成功，至少需要同时提供传统 BIOS 的兼容接口和 EFI 接口。例如微软的新一代长角操作系统就同时提供 ACPI3.0、SMBIOS、UEFI 和传统 BIOS 四种固件接口标准支持。

- **BIOS 核心技术自主知识产权**

无论借鉴采用还是兼容某种 BIOS 规范，要获得长远的发展，保证 BIOS 核心技术的自主知识产权是非常必要的。

EFI 1.10 的知识产权是 Intel 公司独家拥有的，该规范已经提供了很多开源代码，不过还不够完备，例如 USB1.1 OHCI 等部分老标准和 USB2.0 等部分新标准尚未实现。

在 EFI 1.10 已有实现代码的基础上，进一步补充完善，不失为一个快捷的 BIOS 发展方案，但这只是权益之计，不能总为人作嫁。长远看来，实现完全自主知识产权的 BIOS 核心，并在其上提供 EFI 等流行 BIOS 规范的兼容接口是一个更有利的方案。

- **参与 BIOS 新规范的制订并发展部分自主知识产权的 BIOS 规范**

计算机产业已经形成了一个全球性的产业链，任何一家公司都不可能闭门造车，独行其是。甚至作为业界硬软件领袖的 Intel 和 Microsoft 也无法以独霸天下的方式成功推广应用其倡导的 EFI 和 CSS 规范，最终还得大家一起坐下来，达成一个 UEFI 联盟。

因此，争取积极参与新一代 BIOS 新规范的制订是一个获得业界认同的好手段，当然这个需要企业积累和具备一定的实力。

此外，规范有大小之分，BIOS 就有整体性或者部分性的诸多种规范。客观说来，目前国内 IT 企业的实力远不足以制订整体性的 BIOS 规范，但是制订自己的一些局部性规范标准则是完全可行的。比如，结合目前的安全 PC 发展需求，就可以制订一个完全国内自主知识产权的 BIOS 层 PC 安全规范，在国内实行成功后还可进一步推广到海外。

- **发展基于 BIOS 的底层应用**

新一代的 BIOS 规范已经将传统的固件领域向操作系统乃至应用软件的方向拓展。

基于 BIOS 的接口，可以图形界面方式提供安全功能、远程维护、网络和多媒体服务等底层应用。

其中，安全功能可包括 PC 用户身份认证、访问权限管理、数据恢复、防病毒、网络隔离、网络防火墙等。

远程维护可用于 PC 厂商在用户电脑操作系统崩溃后，直接通过 BIOS 底层应用联网用户电脑进行远程故障诊断与维护，降低售后服务成本。

底层网络应用可以无需引导操作系统，即可进行上网浏览网页、收发电子邮件、玩联网游戏等，不仅可避免用户电脑硬盘因上网而感染病毒或者资料泄密，而且适用于低端无盘电脑等应用。

听音乐、看碟片等多媒体服务无需开机等待引导操作系统，通过 BIOS 底层多媒体应用，可使电脑随时可象家电一样方便使用。

底层应用的范围广泛，以上只是罗列其中部分，应该说这些崭新的领域蕴涵着巨大的商机。

- **结合安全芯片、安全操作系统等形成国产安全 PC 系统方案**

随着 PC 的深入广泛应用，PC 的安全也成为一个全社会的问题。为此，各种 PC 安全产品应需而生，但至今没有任何一个所谓的安全解决方案可以彻底解决 PC 安全问题。

其核心原因在于 PC 作为一个包括硬件、固件和软件的复杂系统，不着眼于全局，采用头痛医头、脚痛医脚的局部安全解决方案是不可能很好解决 PC 安全

问题的。

因此，借此 BIOS 更新换代和安全 PC 呼声正响之良机，结合国产安全芯片、安全 BIOS 和安全操作系统，发展一套国产安全 PC 的系统方案会是一件很有意义的事情。

1.2 课题来源及项目背景

本课题来属于自选课题，尝试在具体的主板上实现 EFI 接口的 BIOS。BIOS 作为 PC 核心技术中的关键一环，其功能也在不断拓展，未来必将在整个 PC 体系中扮演更重要的角色。目前，正值传统 BIOS 向新一代 EFI BIOS 规范的过渡过程中，是一个发展自主知识产权的国产 BIOS 技术与产品的极好时机。国家有关部门也已经对 BIOS 产生了相当的热情并已经开始投入资金研究。

1.3 论文研究内容

本文根据 Intel 提出的 EFI1.1 标准，针对 Intel440 BX 芯片主板给出了符合 EFI 接口 BIOS 的驱动体系设计，并给出了体系中具体驱动的实现例子和基于此驱动体系实现的安全应用，在驱动体系大框架下实现了启动 Windows、Linux、Dos 操作系统。内容包括：

(1) 分析 Frame/EFI 的驱动体系，针对 Intel440BX 技术特点，设计具体的 EFI 驱动实施方案。

(2) 针对设计的 EFI 驱动实施方案，给出具体的驱动实现例子。

(3) 在实现的具体驱动之上，设计、实现具体的安全应用。

(4) 在驱动体系大框架下实现通用操作系统引导。

1.4 论文结构

全文共分为六章，根据 Intel 提出的 EFI1.1 标准，针对 Intel440 BX 芯片主板给出了符合 EFI 接口 BIOS 的驱动体系设计，并给出了体系中具体驱动的实现例子和基于此驱动体系实现的安全应用，在驱动体系大框架下实现了启动 Windows、Linux、Dos 操作系统。具体各章节的内容如下：

第一章、 绪论

述课题背景、来源、论文研究的内容和全文的组织结构。

第二章、 传统 BIOS 的缺点

简要阐述了传统 BIOS 的目前存在的问题。

第三章、 EFI 接口 BIOS 驱动体系系统设计

根据 Intel 提出的 Frame/EFI 体系，针对主流 PC 主板给出整个 BIOS 驱动框架的系统设计方案，并概要描述了各个层次、模块。

第四章、 EFI 接口 BIOS 驱动体系详细设计

本章针对第三章的系统设计给出各个部分的详细设计。尤其是详细描述了固件驱动执行层的驱动模型的设计，对于 EFI 接口的固件，为其添加的驱动需要符合 EFI 的驱动模型规范，这章将简要介绍 EFI 驱动的写法和它的一般执行流程。

第五章、 EFI 接口 BIOS 具体驱动的实现

针对第三、第四两章的的设计给出了具体驱动的实现范例，重点介绍了 USB1.1 驱动栈的原理及实现方法。

第六章、 EFI 接口 BIOS 驱动体系的安全应用方案

驱动体系的应用部分，针对安全问题给出两个安全应用的方案。

第二章 传统 BIOS 的缺点

传统 BIOS 从 20 年前的第一台 PC 机诞生到现在没有本质性的变化，它存在不少的缺点：

首先，传统 BIOS 都采用简陋的字符界面，且各选项参数均为英文语言，这让很多初接触的用户都望而生畏，更别提什么参数调整了。

其次，无法达到完全的即插即用，很多时候都要对硬件驱动和 I/O 资源进行手动调整，如果没有一定的专业知识，很难完成这一操作。

第三，BIOS 刷新操作难度高且危险。如果要支持新型号的 CPU，或要修正某个 Bug，抑或是要获得什么新功能，用户就需要通过手动刷新 BIOS 的方法才可顺利支持，但这一操作必须在 DOS 下执行（有些主板厂商为方便用户，也提供 Windows 下的刷新软件）。多数初学者根本都没接触过 DOS，刷新 BIOS 对他们来说是一项高难度的工作，而该过程中若出现断电之类的意外，不可避免地将导致 BIOS 损坏，系统无法启动，用户除了送厂维修外没有别的挽救办法。

第四，现有 BIOS 都是以 16 位模式运行，代码执行缓慢，启动时间长，而代码与堆栈的最大容量仅为 1MB，这也导致现有的 BIOS 任务单一，除了常规的参数调校外不提供任何扩展功能，这显然没有充分发掘 BIOS 应有的潜力。

第五，传统 BIOS 无法为调试硬件提供一个比较强的环境，硬件厂商无法在传统 BIOS 的架构下完成某个硬件驱动的加载、运行、中止等操作，不能方便地更换某个硬件的驱动。

第三章 EFI 接口 BIOS 驱动体系系统设计

EFI 做为一个开放的业界标准接口，它在系统固件（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。EFI 的规范并没有限定这个编程接口的具体实现，标准的 EFI 接口可以有许多种不同架构的实现，它们对外都表现为相同的接口。

在 EFI 刚提出来的早期，Intel 曾经给出了一个基于传统 BIOS 而扩展的 EFI 实现范例：EFI Sample Implement。在这个范例中，Intel 利用传统 BIOS 的中断服务向上实现了一个 EFI 接口。这样的设计有很明显的缺点：1、过大的容量。这种方案需要在已有的 BIOS 基础上增加对 EFI 接口的支持，整体固件的存储容量显然远大过传统 BIOS。2、较低的运行效率。方案 1 需要在 16 位内核和 32 位接口间不停的转换。3、较大的开发难度。不能将 C 语言编程、结构化设计等优秀软件工程方法引入芯片底层固件的开发，也无法从全局整体设计 BIOS 架构，传统 BIOS 复杂、琐碎的接口还会明显提高这种方案的开发难度。

鉴于 EFI Sample Implement 存在的很多问题，Intel 重新设计了 EFI 的实现方案，彻底抛弃了对传统 BIOS 的依赖，新设计取名为 Platform Innovation Framework for EFI，简称 Framework。Framework 其实是一个设计文档族，这一系列的文档是新方案的实现指导方针，详细定义了 EFI 新的实现方案。同时，Intel 也给出了基于 Framework 设计的部分代码，这套代码公布了和硬件无关的部分，在 Intel 内部称为 Tiano，在大多情况下，Framework 和 Tiano 两个概念是一个意思，都指 Intel 对 EFI 实现的设计范例，可以互换使用。这部分将借鉴 Framework 的设计思想，针对主流 PC 主板给出整个 BIOS 驱动框架的系统设计方案。

3.1 驱动体系层次结构

本文驱动体系方案基本上有以下部分组成：

- EFI 初始化准备层（Pre-EFI Initialization），以下简称准备层（PEI）。
- 固件驱动执行层（Driver Execution Environment），以下简称驱动层（DXE）。
- EFI 固件驱动
- 传统 BIOS 接口
- EFI 接口操作系统加载器

- EFI 接口应用模块
- 传统接口操作系统加载器
- 传统接口应用模块

驱动体系层次结构图如图 3.1：

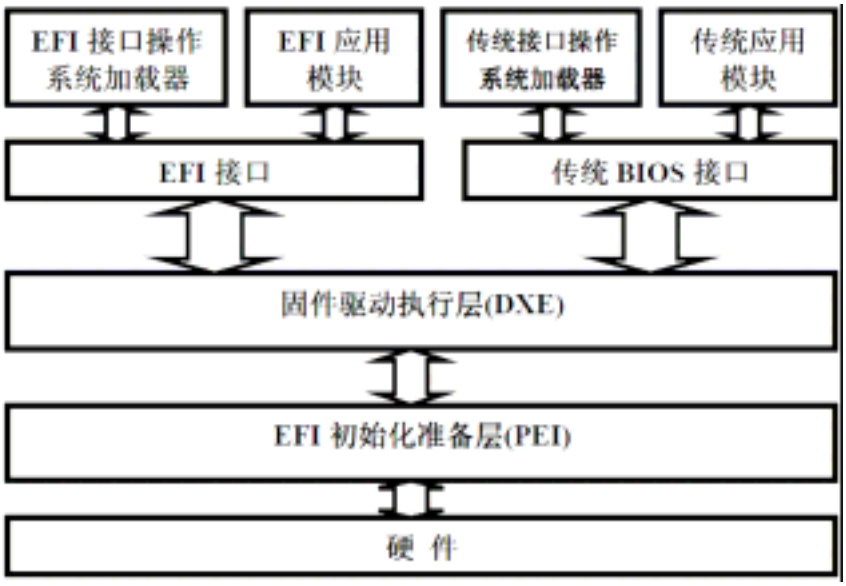

图 3.1 驱动体系层次结构图

整个体系的层次分工是：

EFI 初始化准备层，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。

固件驱动执行层，彻底完成所有硬件初始化，并为上层接口实现所有 EFI1.1 规范中定义的各种服务，固件驱动执行层是本方案的核心。

EFI 接口层，EFI1.1 规范的接口层，是固件对外的接口，提供标准 EFI1.1 所规定的各种协议、接口。

传统 BIOS 接口层，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。

EFI 接口操作系统加载器，支持 EFI 的操作系统中用来通过 EFI 接口加载操作系统的部分。

传统接口操作系统加载器，传统操作系统中通过传统接口加载操作系统的部分。

EFI 接口应用模块，基于 EFI 接口调用，实现操作系统启动前应用的部分。

传统接口应用模块，基于传统接口调用，实现传统操作系统启动前应用的部分。

3.2 驱动体系交互关系

图 3.2 给出本设计方案各部分的交互关系，显示了在计算机开机到操作系统关闭一个完整周期内，本方案各个部分的调度关系。

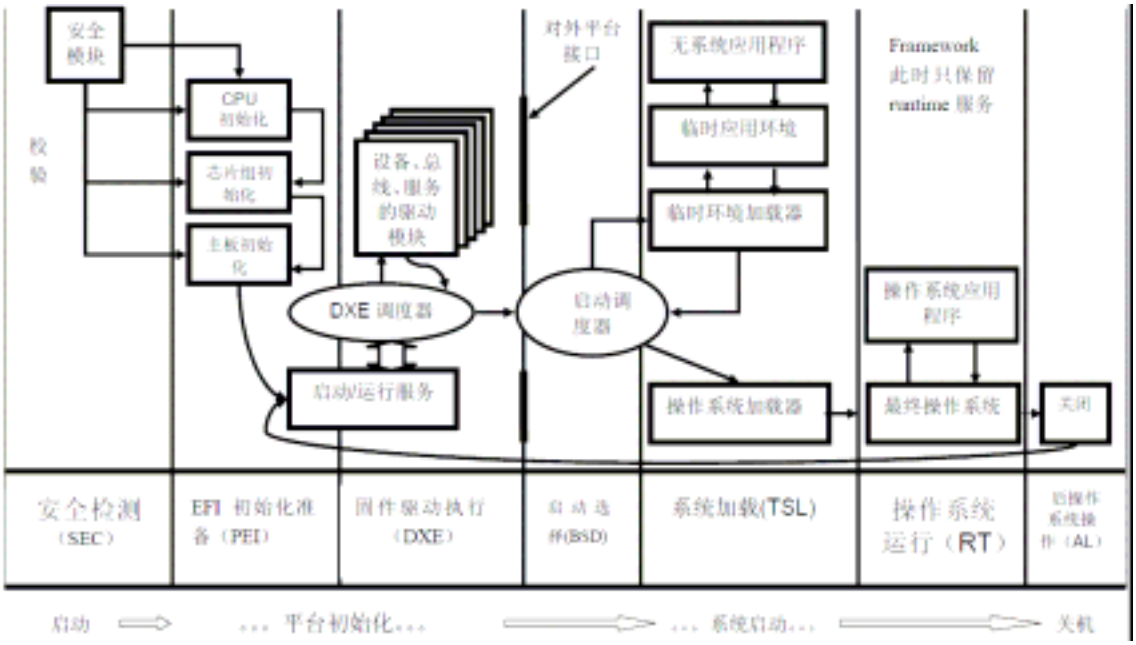

图 3.2 驱动体系交互关系图

图 3.2 各层次交互关系是：

开机后即进入安全检测阶段，一个安全模块将得到控制权，此模块的作用是对后续部分提供安全校验，这个安全模块的校验将贯穿整个固件的生命期。安全模块在自身的初始化后会首先对整个固件进行总体校验，然后启动 EFI 初始化准备的任务，即进入 EFI 初始化准备阶段。

EFI 初始化准备阶段即对应于基本 EFI 初始化准备层，它通过执行一个个的基本初始化模块来完成这个阶段的初始化工作。每个初始化模块在执行前都需经过安全模块的校验，初始化层寻找并初始化最少的硬件资源，这些资源可以满足启动驱动层即可，基本 EFI 初始化准备完成后加载驱动层，即进入固件驱动执行阶段。

固件驱动执行阶段即对应固件驱动执行层，它彻底完成所有硬件初始化，并为上层接口实现所需要的各种服务。驱动层将通过查找一个个驱动模块来完成所有硬件的登记，并在服务被调用时实际执行相应功能的驱动模块。本方案的安全应用在驱动层中加入了内部安全服务，在调度器登记和执行每个驱动模块时，模块都要经过内部安全检测。当驱动层完成所有初始化后，就对外提供了统一的接口：EFI 接口和传统 BIOS 接口（可选），这两个接口为上层的各种应用模块屏蔽

了底层硬件细节，提供了各自统一的运行环境，在统一的运行环境中可以运行启动管理器等上层应用模块。一旦驱动层完成初始化并尝试启动操作系统时，就进入了启动选择阶段。

启动选择阶段实际上是一个启动选择器选择执行应用模块，几乎所有的无操作系统应用均在这一层完成，可以选择执行的应用模块即对应层次图中的 4 个部分：EFI 接口操作系统加载器、EFI 接口应用模块、传统接口操作系统加载器、传统接口应用模块。一旦具体的模块开始执行即进入了无系统的应用阶段。

无系统的应用是外部用户基于本方案固件提供的接口、服务展开的应用。在众多应用当中有一个最重要的应用就是操作系统的加载。当操作系统被成功加载后就进入操作系统运行阶段。

操作系统启动后本方案核心固件仅仅保留操作系统运行时及运行后需要的接口、服务。

当操作系统结束运行后控制权将最终又交换给固件，操作系统结束运行可能是合法的关机也可能是被强迫的非法关闭，上图的内部安全服务在操作系统交还控制权后要进行最后的可信校验。

3.3 驱动体系特点

本设计方案优点：

- 本方案从体系结构上整体设计，可以保证整个体系这个层次的工整、严谨。
- 本方案的整体结构设计，可以从体系结构上来思考一些策略的实现，比如全局安全策略、整体电源管理策略、系统自检策略等等。
- 本方案的分层设计，可以提供方便的固件存储方式，本地 ROM 可以只存储容量很小的初始化层，而将后续执行的部分存储在其他地方，比如硬盘、网络、USB 闪存盘等等。
- 本方案的驱动模块设计，符合面向对象设计的思路，可以灵活兼容各种硬件外设和服务需求，很方便地扩展或缩小对不同硬件的支持。
- 本方案的分层、模块驱动化设计，便于固件开发的分工，基于标准驱动模块的接口可以让不同的专业厂商完成自己硬件的固件开发，大大提供固件的开发速度和固件稳定性。
- 本方案的传统兼容性支持是可以选择的，可以兼容传统操作系统和基于传统 BIOS 接口的各种应用。如果无需支持传统 BIOS，可以移除相应模

块，减少 ROM 使用量。

- 本方案可以将 C 语言编程、结构化设计等优秀软件工程方法引入芯片底层固件的开发。
- 本方案是最大程度的利用现有各种开源项目，符合现有开源架构，可以很方便地移植各种开源成果。利用已有的成熟代码可以极大提高最终代码的质量并缩短开发进度。

本设计方案缺点：

- 借鉴并移植开源项目搭建的平台存在被攻击的风险，所以利用开源平台时必须考虑整体的安全策略，本方案将在安全应用方面重点解决这个问题。

第四章 EFI 接口 BIOS 驱动体系详细设计

本章针对第三章的系统设计给出各个部分的详细设计。

4.1 基本 EFI 初始化准备层（初始化层）

（1）、设计思路

- 提供保障自己安全、可信的方法。
- 只初始化最基本的系统资源。这些系统资源对驱动层是最精简的，其中包括初始化并描述最基本数量的内存、驱动层程序所在 ROM 等等。
- 提供方法完成各个 EFI 初始化准备工作的增添、移除。这些具体任务虽然都是硬件相关的处理器、芯片组、主板基本初始化，但模块依然需要按一定原则来分解，从而可以让不同厂商提供不同的模块。
- 提供修改初始化顺序方法。

（2）、组成体系

初始化层核心层是一个分层的模块化设计，由两个部分组成：

1) 基础模块

2) 初始化模块

这两个部分的相互关系是：初始化层的基础模块是初始化层中相对固定的部分，由它加载一个或多个初始化模块来完成处理器、芯片组、主板等硬件平台相关初始化。根基和初始化模块间有标准接口相互衔接。

4.1.1 基础模块

基础模块是初始化层的公共部分，它完成两个主要任务：加载初始化模块并执行、提供公共服务。基础模块会根据一定的顺序来加载执行初始化模块，这个顺序可以很容易改动。提供的公共服务包括：协助初始化模块间通讯、管理初始化模块所需的临时内存（临时内存可以是处理器 cache、主板 CMOS 等等）、在 ROM 中加载初始化模块、报告初始化模块执行结果、将初始化层最终提供的硬件资源提交给内核层等等。需要特别指出的是基础模块还负责以某种策略来检验自己和所加载模块的安全性，本方案将通过基于硬件芯片的方法实现，具体实现说明见本方案的安全策略。

4.1.2 初始化模块

初始化模块是一个个的可以执行模块，它封装了对处理器、芯片组等等和硬件相关的功能实现。初始化模块会提供标准接口来实现初始化模块之间的通讯、初始化模块和基础模块的通讯。重要的是初始化模块还有本模块的信息描述（包括自己在 ROM 中的定位、自己对其他初始化模块的依赖），可供基础模块判断各个初始化模块的执行顺序。方案将初始化模块以不压缩的形式存储在 ROM 中，这样初始化模块可以直接执行而无需 RAM 来解压缩。因为不压缩所以初始化模块的数量和容量将尽量的少，来节省 ROM 存储空间。

在各种初始化模块中有一个特别的模块，是用来启动驱动层并传递硬件资源信息的。这个模块是初始化层中执行的最后一个模块。

本方案希望由硬件制造商来提供初始化模块的代码，比如处理器、芯片组的初始化模块等等。

图 4.1 给出初始化层的结构图。在下图中可以看到整个初始化层由基础模块和初始化模块组成，其中基础模块提供多了 N 个基础公共服务；初始化模块有 N＋1 个组成，即要完成 N＋1 个初始化任务，其中最后一个要执行的任务是启动驱动层。基础模块通过标准接口来调度初始化模块，并为其提供各种服务。初始化模块可以通过基础模块来间接通讯，也可以通过对方接口的直接通讯。

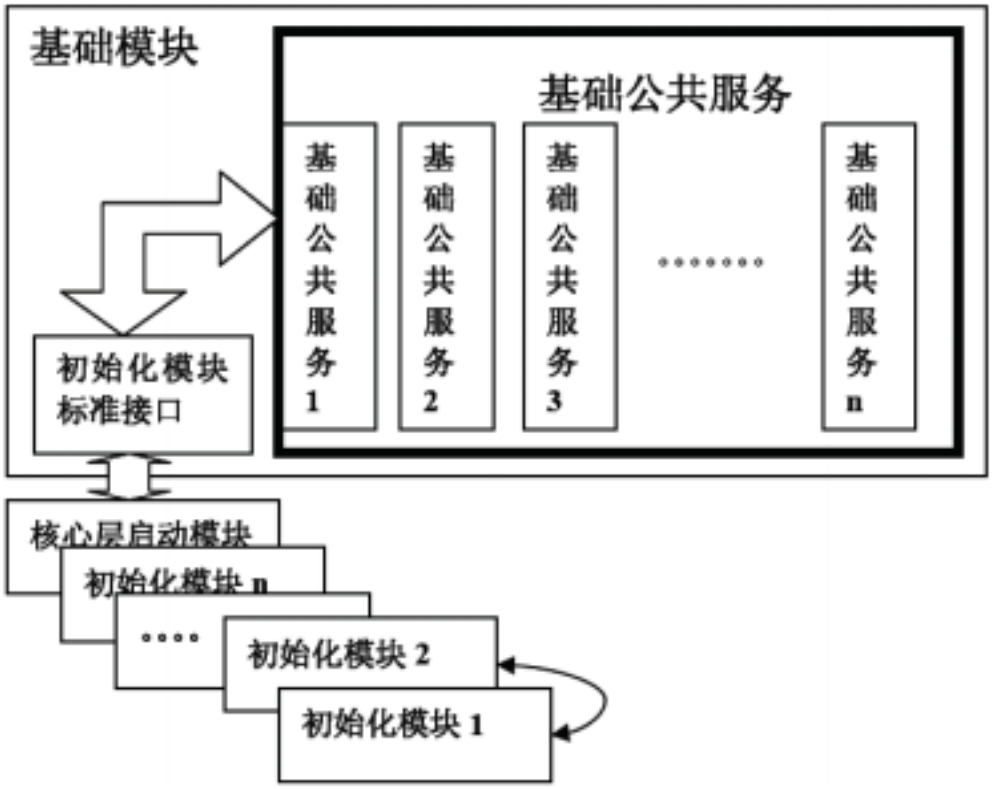

图 4.1 初始化层结构图

4.2 固件驱动执行层（驱动层）

（1）、设计思路

- 提供保障自己安全、可信的方法。
- 完成绝大部分的硬件平台初始化，并对 EFI1.1 接口规范的提供完整实现。
- 区分通用部分，支持跨平台。驱动层是本方案的主体部分，它即有通用的核心业务又涉及大量具体的硬件操作，所以这一层本身必须能够区分哪些是硬件平台相关的，哪些是独立于硬件平台的部分。对于独立于硬件平台的部分将作为进行跨平台的基础，可以支持包括台式机、服务器、便携机及嵌入式平台在内的多种平台。
- 要为增添、移除不同的硬件、服务的提供灵活的方法。驱动层的功能应当是可以很方便的增加和裁减的，需要在设计上提供灵活的部件接口。
- 支持 ACPI 3.0 规范。
- 支持 PCI 3.0 规范。
- 支持 PCI Express 规范。
- 支持 USB1.1 及 USB2.0 规范。
- 支持 x86 IA-32(32 位)微处理器及 IA-64（64 位）微处理器。
- 提供 TCP/IP 网络通讯协议栈。

（2）、组成体系

驱动层是一个分层、模块化的设计，包括三个组成部分：

1) 基础模块
2) 驱动模块
3) 驱动模块调度器（以下简称调度器）。

三个部分的相互关系是，驱动层基础模块提供一系列的通用平台服务：操作系统启动服务、操作系统运行时间服务、驱动层内部服务，这些服务都不涉及具体硬件的细节操作；驱动模块负责处理器、芯片组等硬件平台的初始化，并且提供硬件使用的软件抽象。调度器负责发现驱动模块，并以正确的顺序执行驱动模块；这三部分中基础模块是底层，它通过调度器来控制上层的驱动模块。

4.2.1 基础模块

驱动层的基础模块是一个完全可跨平台的部分，它不依赖任何处理器、芯片组等硬件平台。基础模块只依赖初始化层提供的基本硬件资源，所以基础模块不

需要前面阶段（初始化层）保留任何服务调用，一旦驱动层的基础模块得到所需基本硬件资源后，初始化层的所有部分都可以移除；基础模块将不使用任何硬件的绝对地址，所以基础模块可以被加载到任何物理存储器中，无论给所在存储器分配怎样的逻辑地址，基础模块均可以正常工作；基础模块中没有任何处理器、芯片组等硬件信息，它是通过一套协议接口来抽象各种硬件，这套协议接口是由调度器调用的一套驱动模块产生的。

基础模块可以抽象各种具体的硬件是通过一套协议接口来完成。为抽象各种硬件（设备、总线、网络等）这套协议接口必须足够完备，并且接口足够简单方便。在本方案中将对同一类硬件将提供统一的协议接口。各个硬件的支持将以驱动模块的形式加入到驱动层供基础模块使用。

基础模块会向上提供 EFI1.1 规范中的 EFI 系统表（EFI System Table）和它相关联的 EFI 启动服务（EFI Boot Services）、EFI 操作系统运行时服务（EFI Runtime Services）。

需要特别指出的是基础模块还负责以某种策略来检验自己本身和所加载驱动模块的安全性。本方案将通过基于硬件芯片的方法实现，具体实现说明见本方案的总体安全策略。

4.2.1.1 对象

EFI 可以通过服务来管理很多不同类型的对象。下图给出了对于驱动来说很重要的对象。其中，EFI 系统表对驱动是最重要的一个数据结构，它向驱动提供了各种服务的接口，来使用整个硬、固件平台的数据、服务。

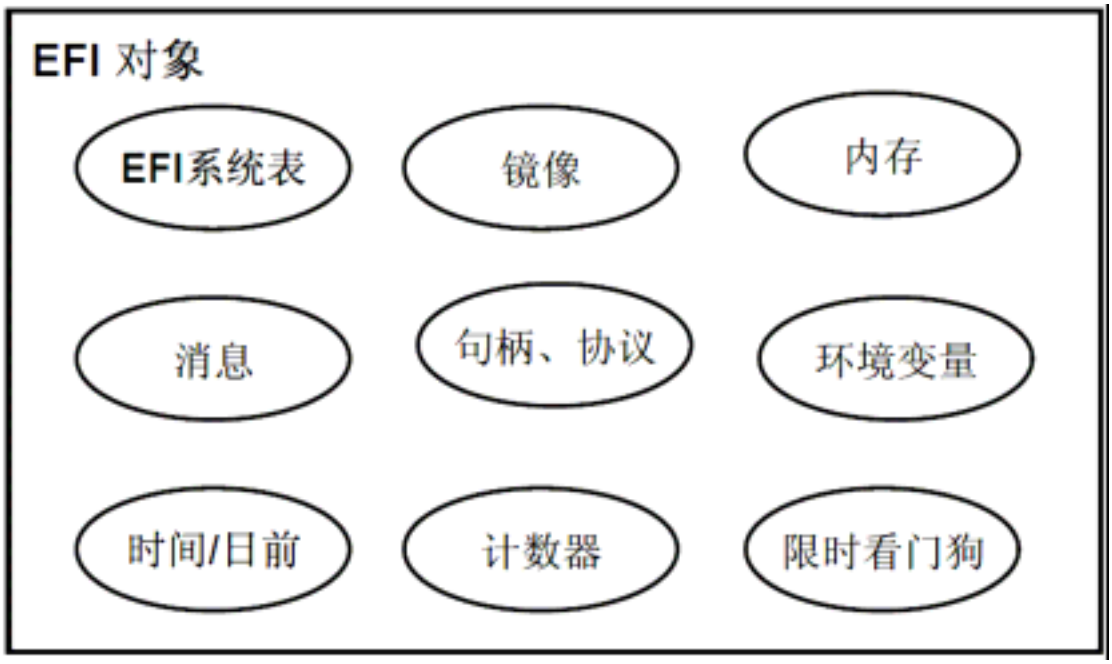

图 4.2　EFI 对象

4.2.1.2 EFI 系统表

EFI 系统表是 EFI 当中最重要的一个数据结构。从 EFI 系统表，一个 EFI 可执行镜像可以访问到系统的各种配置信息和丰富的 EFI 服务。EFI 的服务包括：EFI 启动服务（EFI Boot Services）；EFI OS 运行服务（EFI Runtime Services）；协议服务（Protocol Services）。

EFI 启动服务和 EFI OS 运行服务是通过 EFI 系统表中的 EFI 启动服务表和 EFI OS 运行服务表来对外提供访问接口。这两个服务表的结构是 EFI 规范中定义的。

协议服务是一组由 GUID（Globally Unique Identifier）命名的函数和数据，它提供了诸如控制台、磁盘、网络等硬件设备的软件抽象[1]。它同时也提供了一般固件服务的扩展。一个个的协议是扩展固件的基本单位，EFI 规范已经定义了 30 多种协议，具体的固件还可以定义自己的协议。

4.2.1.3 句柄库

句柄库是由句柄和协议组成的集合[1]。句柄其实是一个指向一个或多个协议的指针，而协议是由 GUID（Globally Unique Identifier）命名的数据结构。协议的数据结构可以是空，也可以包含数据项、服务函数。在 EFI 初始化过程中，系统固件、EFI 驱动和 EFI 应用程序都可能生成句柄，并且在生成的句柄上装载协议。句柄库中的信息是对 EFI 的对象公开的，可以供任何的 EFI 镜像访问。

句柄库可以看作是 EFI 固件维护的一个数据库，结构上就是一列列的句柄串，每个句柄由唯一的句柄号表示，句柄号就就是找到一个句柄的钥匙。每个句柄又是指向一个或多个协议的指针。协议的类型有 GUID 来确定。一个句柄及它指向的协议可以代表：驱动或应用程序；诸如网卡适配器、磁盘分区的硬件设备；EFI 的公共服务。

下图 4.3 举例给出了句柄库的一部分，图中粗线条的部分就是句柄和协议。图中某些协议关联了一些和协议相关的对象，这些对象可以串连在一起，它们的作用是跟踪所关联协议的使用情况，即这个句柄被哪些其它的驱动、镜像或镜像驱动使用，以及它自己使用了其它的哪些句柄。这些信息可以被 EFI 的驱动使用，来保证在驱动加载、启动、暂停、卸载时不会有相互冲突。

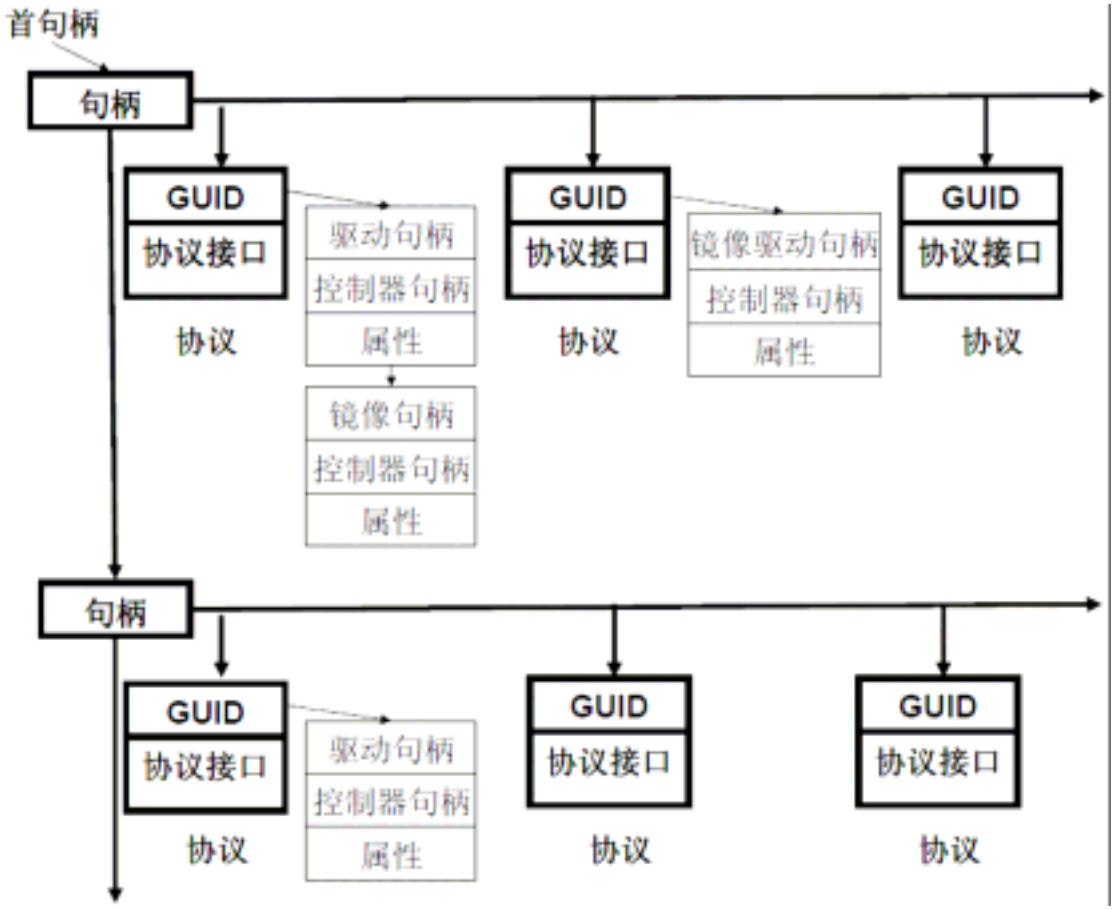

图 4.3 句柄库示意图

4.2.1.4 协议

EFI 的可扩展性很大程度上来自协议。Protocol 有点面向对象中的 class，它定义了某个对象应该有的私有数据和功能接口[1]。最简单的情况下，Protocol 会定义一个 GUID，这个数字是这个 Protocol 的真正的名字，并且用来在定位这个 protocol。通常一个 Protocol 通常会包含一组函数（procedure），以及数据结构（protocol interface structure）。下面给出一个 Protocol 的代码实例。

GUID 的定义实例：

```
#define EFI_COMPONENT_NAME_PROTOCOL_GUID \
{ 0x107a772c,0xd5e1,0x11d4,0x9a,0x46,0x0,0x90,0x27,0x3f,0xc1,0x4d }
```

Protocol Interface Structure 的定义实例：

```
typedef struct    _EFI_COMPONENT_NAME_PROTOCOL {
EFI_COMPONENT_NAME_GET_DRIVER_NAME GetDriverName;
EFI_COMPONENT_NAME_GET_CONTROLLER_NAME GetControllerName;
CHAR8 *SupportedLanguages;
} EFI_COMPONENT_NAME_PROTOCOL;
```

EFI 驱动和 EFI 协议是有不同的，EFI 驱动是一个可执行的镜像，驱动可以根据需要在不同的句柄上安装多个协议；EFI Protocol 是一些函数指针和数据结

构，或者由规范所定义的 APIs。

下面的图 4.4 展示了一个 EFI 驱动程序在句柄库中生成了一个句柄和一个协议。这个协议由一个 GUID 和协议接口组成。大多数情况下，由 EFI 驱动程序产生的协议接口可能会附加一些私有的数据域，而协议接口本身包括协议函数的指针，协议函数实际上包含在 EFI 驱动程序内。一个 EFI 驱动程序生成一个协议还是多个协议只取决于驱动程序的复杂性。

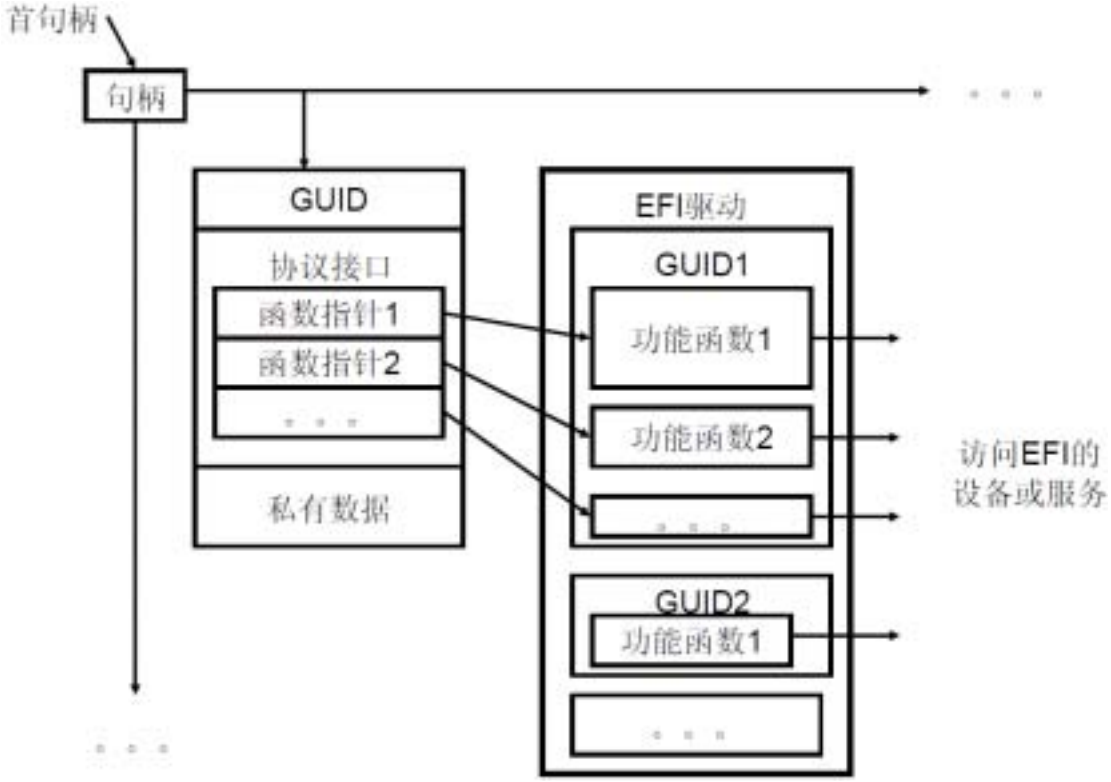

图 4.4 驱动与协议关系图

4.2.1.5 EFI 设备路径

EFI 设备路径是一种 EFI 安装在设备句柄上的协议，它是为了帮助操作系统及其加载器定位设备句柄代表的真实设备[1]。EFI 设备路径协议为系统中存在的硬件给出了唯一的名字。EFI 固件中全部的 EFI 设备路径协议构成了系统的设备名空间。EFI 设备路径协议实际上来自 ACPI 设备名空间的简化。

一条设备路径有一个或多个设备路径节点组成，每个节点又由节点头和具体设备路径数据组成。下面是个具体的例子。

```
//
// Generic device path node header
//
typedef struct{
    UINT8 Type;
    UINT8 SubType;
```

```
    UINT8 Length[2];
} EFI_DEVICE_PATH_PROTOCOL;
//
// PCI Device Path Node that includes a header and PCI-specific fields
//
typedef struct _PCI_DEVICE_PATH {
    EFI_DEVICE_PATH_PROTOCOL Header;
    UINT8 Function;
    UINT8 Device;
  } PCI_DEVICE_PATH;
```

4.2.2 驱动模块

驱动模块提供对硬件平台的抽象，提供硬件服务的接口和封装访问硬件的方法。本方案计划在驱动层实现两种驱动模块：起始驱动模块、EFI1.1 驱动模块。起始驱动模块是不符合 EFI1.1 驱动模型的模块，当 EFI 初始化准备层传递控制权给驱动层后，这些起始驱动来完成驱动层最开始自身初始化的工作，它将根据特定的依赖性来决定自身的执行顺序，起始驱动模块一般包括处理器、芯片组等硬件平台的初始化代码，它还要为基础模块屏蔽硬件建立协议接口；EFI1.1 驱动模块是符合 EFI1.1 驱动模型的模块，它在被调度器执行时并不做任何实际的硬件初始化，而仅仅是在驱动层中注册一个 EFI1.1 的驱动绑定协议（Driver Binding Protocol ），各种注册的驱动绑定协议将在后续启动阶段以 EFI 启动服务（EFI boot service）供使用，所以 EFI1.1 驱动模块只有在明确地需要执行时才完成对硬件的软件抽象。一个个的驱动模块为驱动层详细定义了硬件平台的实际情况。

本方案将支持 ACPI3.0 规范、PCI3.0 规范、PCI Express 规范、USB1.1 及 USB2.0 规范、IA32 架构的 32 位和 64 位处理、IA64 的处理器，支持的方法就是通过添加各种符合规范的驱动模块，让驱动层通过各种规范的驱动模块来使用具体的功能。

4.2.2.1 驱动模型

EFI1.1 规范定义了 EFI 驱动模型，符合 EFI 驱动模型的驱动和 EFI 的应用程序很相似，都是可执行的镜像，但驱动模型通过分别定义了驱动必须支持的一些操作，诸如加载、启动、暂停等，赋予了驱动更强的执行控制能力。下表列出驱动模型定义的，一个符合驱动需要支持的协议。

协议	描述
驱动绑定协议	这个协议提供以下几个基本功能：鉴别驱动是否适用某个硬件；启动驱动运行；停止驱动运行。这个协议是必备的。[1]
组件名称协议	这个协议提供驱动和驱动所控制硬件的名称服务，就是对外提供驱动和硬件的字符名称，增加用户可读性。可选。[1]
驱动诊断协议	这个协议是调试诊断硬件设备的接口，提供了驱动所控制硬件相关的调试功能。可选。[1]
驱动配置协议	这个协议为用户提供了配置驱动所控制硬件的功能。可选。[1]

可以看出符合 EFI 驱动模型的驱动一定要支持驱动绑定协议，可以有选择的支持其它的协议。

驱动在自己的镜像句柄上注册新的协议。在 EFI 驱动模型中，驱动入口的主要工作就是安装这些协议。在后续时间点的系统初始化中，EFI 可以使用这些协议的功能函数操作驱动。复杂的驱动可以生产多个驱动绑定协议的实例，增加的协议将被安装在新的句柄上。这些新的句柄可以有选择地支持组件名称协议、驱动诊断协议和驱动配置协议。

EFI 驱动模型将驱动分成设备驱动和总线驱动，它们之间的区别主要在于驱动绑定协议中启动和暂停功能函数的不同。

4.2.2.1.1 设备驱动

设备驱动的启动功能函数 start()将直接在控制器的句柄上安装协议，并且会把这个控制器的句柄做为一个参数传递给 start()函数。被设备驱动安装的协议会使用这个控制器上提供的 I/O 服务，而这个 I/O 服务是由另一个安装在这个控制器句柄上的 I/O 协议提供的。比如，一个 PCI 控制器的设备驱动将使用 PCI I/O 协议的服务，而一个 USB 控制器的设备驱动将使用 USB I/O 协议的服务。这个使用 I/O 协议的过程称为消费这个总线的 I/O 抽象。下面是设备驱动的主要工作：

- 初始化控制器
- 在被 EFI 固件用来启动操作系统的设备上安装 I/O 协议

设备驱动在 EFI 定义中的一个基本特点是它不会产生任何子句柄。这个特点是区分设备和总线驱动的基本准则。

4.2.2.1.2 总线驱动

总线驱动除了可以产生子句柄外，和设备驱动几乎一样，但因为要产生子句柄，所以总线驱动要额外增加一些责任。

象设备驱动一样，总线驱动的 start()功能函数将消费父总线的 I/O 抽象并以协议的形式生成新的 I/O 抽象。比如，PCI 总线的驱动将消费根 PCI 桥协议（EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL）的服务，并使用它来扫描一个 PCI 总线上的 PCI 控制器。每当一个 PCI 控制器被找到，一个子句柄就会被生成，并且会在这个生成的子句柄上安装一个 PCI 的 I/O 协议（EFI_PCI_IO_PROTOCOL）。此时 EFI_PCI_IO_PROTOCOL 协议的实现是通过使用 EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL 协议的服务来完成的。

下面是总线驱动的主要工作：

- 初始化总线控制器。
- 扫描决定有多少个子句柄要生成。
- 为找到的子控制器分配资源并在句柄库中生成子句柄。
- 在生成的子句柄上安装 I/O 协议，来提供对 I/O 操作的抽象。
- 如果一个子句柄代表的是一个实在的物理设备，还要在这子句柄上安装设备路径协议。
- 如果在 option ROMs 中有这个设备的驱动，从 option ROMs 中加载它的驱动。（目前只有 PCI 设备这样做）

4.2.2.2 驱动模块运行流程及实例

符合 EFI 驱动模型的驱动在有基本相同的运行流程，可以分成驱动的加载和驱动的连接阶段。驱动的加载只是将驱动镜像程序从自己所在的存储设备上加载到系统的内存，让系统固件知道系统中有一个可用的驱动；驱动真正的运行、访问并控制硬件设备是在驱动的连接以后，由系统固件在已经登记的诸多驱动中选择一个合适的驱动，将硬件的控制权交给它。

驱动被加载时会做以下工作：它会在自己的镜像句柄上安装一个驱动绑定协议，有的驱动还会更新卸载镜像的功能函数 Unloader()。当做完这些时候后驱动就从它的调用口离开，并把自己的驻留在系统的内存中。驱动绑定协议提供了驱动的版本号和以下三个功能函数：supported()、start()、stop()。这三个功能函数在驱动被加载后，即完成安装驱动绑定协议并从调用口离开后，都是可用的。

在系统固件需要使用一个硬件设备时，它会尝试在登记的驱动中找到一个最合适的驱动，这个过程即称为驱动与设备的连接。系统固件会使用驱动绑定协议中的 supported()功能函数来检查安装这个协议的驱动是否是某个硬件设备的合适的驱动。每个驱动都会在自己的镜像句柄上提供驱动绑定协议，也即提供了 support()功能函数。如果 support()功能函数返回成功，系统固件会知道这个驱动是一个合适的驱动，系统固件接下来会调用驱动绑定协议中的 start()功能函数并将硬件设备的句柄做为参数传递给它。当 start()开始执行，驱动就和设备连接成

功，驱动就接管了硬件设备的控制权。当系统固件决定不再使用某个设备，它可以通过调用驱动绑定协议中的 stop()功能函数来中止设备运行并断开驱动和设备的连接。

下面以一个具体平台的实例来说明系统固件一个大致的初始化流程。下图中的每个方框都代表一个系统的物理设备。在 EFI 连接操作前，下图中的所有设备都没有在句柄库中登记。下面将一步步介绍各个驱动将如果加载、创建句柄、安装协议。

在启动的一开始，EFI 会不加考虑地创建解压缩、EFI 代码虚拟机协议的相应句柄，并安装协议。这些协议将为后面可能的压缩驱动和 EBC 格式的驱动提供服务。此时句柄库的状态如下：

Handle Database

…

6: Ebc

…

9: Image(Decompress)

A: Decompress

4.2.2.2.1 连接 PCI Root Bridge

EFI 固件接下来会通过 LoadImage()为 PCI Root Bridge 装载驱动。和其它驱动一样，PCI Root Bridge 的加载会在句柄库中为这个驱动创建一句柄，并安装记录这个驱动信息的镜像加载协议：EFI_LOADED_IMAGE_PROTOCOL。因为这个驱动是针对系统 Root bridge 的，所以它不用符合 EFI 驱动模型。当它被加载，它不会安装驱动绑定协议、组件名称协议、驱动诊断协议、驱动配置协议。它会根据自己对具体平台的了解，立刻创建 PCI root bridge 的句柄，然后安装 EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL 和 EFI_DEVICE_PATH_PROTOCOL 到句柄。这样做，是因为 PCI root bridge 是必须要一直工作，不能打断、在连接的。

连接 PCI root bridge 后，句柄库中可能生成如下图内容：

Handle Database

…

B: Image(PcatPciRootBridge)

C: DevIo PciRootBridgeIo DevPath (Acpi(HWP0002,0,PNP0A03))

4.2.2.2.2 连接 PCI 总线

EFI 固件接下来会加载 PCI 总线的驱动。还是和其它驱动一样首先在自己的句柄上安装一个镜像加载协议。PCI 总线的驱动是符合 EFI 驱动模型的，它会在自己驱动的入口安装驱动绑定协议、组件名称协议，但是它在连接 PCI root bridge 以前不会生成任何新的 PCI 控制器句柄。

Handle Database

…

14: Image(PciBus) DriverBinding ComponentName

接下来，EFI 固件将使用 ConnectController()尝试连接 PCI root bridge 的控制器（就是上面的句柄 C）。连接某个设备时，EFI 固件会依据一些优先级策略查询句柄库中的各个驱动，在这个例子中，EFI 固件会找到句柄 14 指向的驱动，并尝试调用 supported()功能函数，并把句柄 C 做为参数传递给它。这个句柄 14 的 PCI 总线驱动判断 PCI root bridge 的控制器标准是它是否支持设备路径协议和 PCI Root Bridge IO 协议，所以 PCI 总线驱动的 support()功能函数会返回成功。然后 ConnectController()会使用 PCI 总线驱动的 start()功能函数来启动句柄 C 的 PCI root bridge 控制器。

因为 PCI 总线驱动是个总线驱动，它的 start()功能函数会调用 PCI Root Bridge IO 协议的功能函数来查找所有的 PCI 设备。对每个在 PCI 总线上发现的 PCI 设备/功能都会创建一个子句柄并且在子句柄上安装一个 PCI IO 协议。这个子句柄在句柄库中会被登记成 PCI root bridge 的控制器的子句柄，它的设备路径也是由 PCI root bridge 控制器的设备路径加上自己的设备路径节点 PCI（Dev|Func）。下图给出一组 PCI 总线驱动连接 PCI root bridge 控制器后，句柄库中可能的情况：

…

16: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(1|0))
17: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(1|1))
18: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(2|0))
19: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(2|1))
1A: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(2|2))
1B: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(3|0))
1C: Image(Acpi(HWP0002,0,PNP0A03)/Pci(3|0)) DriverBinding
1D: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(4|0))
1E: PciIo DevPath (Acpi(HWP0002,100,PNP0A03)/Pci(1|0))
1F: PciIo DevPath (Acpi(HWP0002,100,PNP0A03)/Pci(1|1))

这组句柄显示了，有 9 个 PCI 设备被发现；句柄为 1B 的 PCI 设备有一个 EFI 驱动在自己的 option Rom 中；这个 1B 设备中的驱动被加载并执行，句柄是

1C。

4.2.2.2.3 连接控制台设备

一般扫描整个 PCI 总线后，EFI 固件会初始化控制台设备，连接好控制台设备，EFI 固件就可以输入、输出信息。注意，用做调试输出的串口也是控制台设备，它也会在此时重新被初始化，但它在开机后很早的时候就会被一个非 EFI 驱动模型的驱动完成配置。此时会再次用一个符合 EFI 驱动模型的驱动连接一次，以后的调试信息打印就使用新的串口驱动输出了。

典型的控制台设备有 USB 键盘、USB 鼠标、串口。这些设备的驱动和 PCI 总线驱动一样一般都是系统固件中集成的驱动，如果这些设备是以 PCI 板卡的形式存在的，那么相应的驱动会放在 PCI 板卡的 option ROM 中。

当系统固件将各个控制台设备的驱动加载后，接下来会根据事先设定的设备路径来连接具体的控制台设备。EFI 会把事先设定的设备路径存放在三个全局的 EFI 变量中，即 ConIn、ConOut、ErrOut。

下面以一个具体的输入控制台设备例子，来说明 EFI 固件连接控制台设备的全过程。假设输入控制台的设备路径是 ConIn = Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600 N81)/VenMsg(Vt100+)。这是一个串口设备的路径。所谓连接设备就是系统固件通过一系列的动作，找到并登记符合设备路径要求的设备。

1) EFI 系统固件会首先在句柄库中查找设备路径和 ConIn 变量最吻合的设备句柄。假设它发现句柄 17 最接近，句柄 17 的设备路径是 Acpi(HWP0002,0,PNP0A03)/Pci(1|1)，显然此设备指匹配了 ConIn 的一部分。Uart(9600 N81)/VenMsg(Vt100+)是剩余没有匹配的部分，这段没有匹配的路径就叫剩余路径。其实，ConIn 代表的串口设备是通过一个 PCI 控制器挂接在 PCI 总线上的，句柄 17 正是那个 PCI 控制器设备，而剩余路径就是具体指定这是一个什么样的 PCI 设备。此时，在系统固件看来，这个句柄 17 控制器最有可能就是 ConIn，下面它会从句柄 17 出发，通过一系列的动作来尝试匹配 ConIn 所定义的设备路径。
2) 系统固件会使用启动服务函数 ConnectController()，以句柄 17 和剩余路径 Uart(9600 N81)/VenMsg(Vt100+)做为参数。这个 Connect 函数会论询所有系统固件中已有的驱动，把句柄 17 和剩余路径做为参数调用各个驱动的 Supported()功能函数，检测驱动是否支持这个设备。假设发现句柄 30 的驱动返回 Success。这时系统固件知道已经找到了一个支持句柄 17 控制器的

驱动，它会把句柄 17 和剩余路径做为参数调用句柄 30 驱动中的 Start()函数。这个驱动函数会使用句柄 17 控制器上的 PCI I/O 协议，生成一个新的子句柄 3B，并在新的句柄 3B 上安装 Serial I/O 协议和设备路径协议。这个新的句柄 3B 代表的是一个串口设备，它的设备路径就是句柄 17 的设备路径加上自己的设备节点 Uart（9600 N81）：Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600 N81)。此时第一次 ConnectController()结束。

3) 系统固件经过一次 Connect，并没有找到完全匹配 ConIn 设备路径的设备，但是现在句柄 3B 成了和 ConIn 变量最吻合的设备句柄。系统固件此时会重复上述 2）中的动作，但这次的剩余路径是 VenMsg(Vt100+)，句柄 3B 和剩余路径 VenMsg(Vt100+) 被做为参数，第二次使用启动服务函数 ConnectController()。假设这次发现句柄 31 的驱动支持，那么句柄 31 驱动的 Start()函数将被执行。句柄 31 驱动的 Start()函数将生成新的设备句柄 3C，并生成新句柄的设备路径：Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600 N81)/VenMsg(Vt100+)。

4) 此时，系统固件找到了完全匹配 ConIn 设备路径的设备：句柄 3C 代表的设备。系统固件会以句柄 3C 为参数，最后一次启动服务函数 ConnectController()，完成 ConIn 的连接。在这次 Connect 过程中，会有多个 ConIn 的驱动被执行，在 3C 句柄上安装相关的协议。

下图演示了上述连接过程。

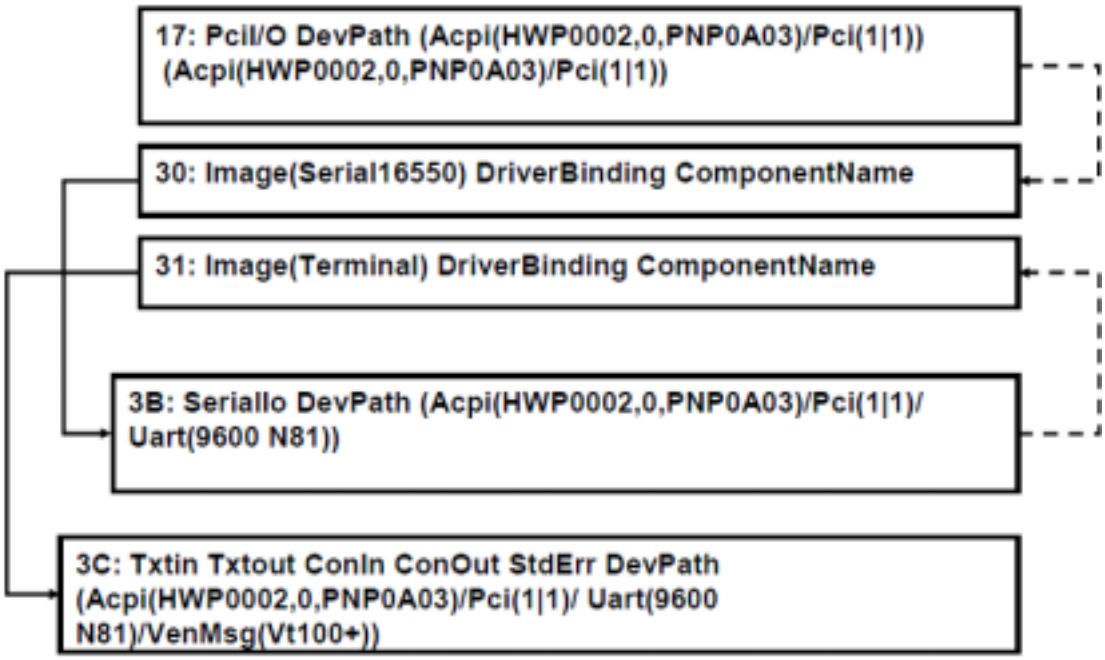

图 4.5 输入控制台设备连接图

4.2.3 调度器

面对众多的驱动模块，它们需要专门的管理。调度器负责在整个固件中寻找驱动模块并调用，一开始调度器将从初始化层传递的硬件资源中寻找驱动模块，随着后续执行，调度器也将从新发现的固件存储设备中寻找驱动模块。

调度器在完成驱动模块的寻找并调用后，驱动层提供的 EFI 接口和各种 EFI 的服务已经准备就绪，此时将控制权交给 EFI 接口以上的操作系统加载部分。

图 4.6 显示了固件驱动执行层的结构图，由图可见固件驱动执行层主要由基础模块和驱动模块构成。基础模块在装载了驱动模块后，即可使用驱动模块提供的服务依据硬件协议接口来控制硬件。

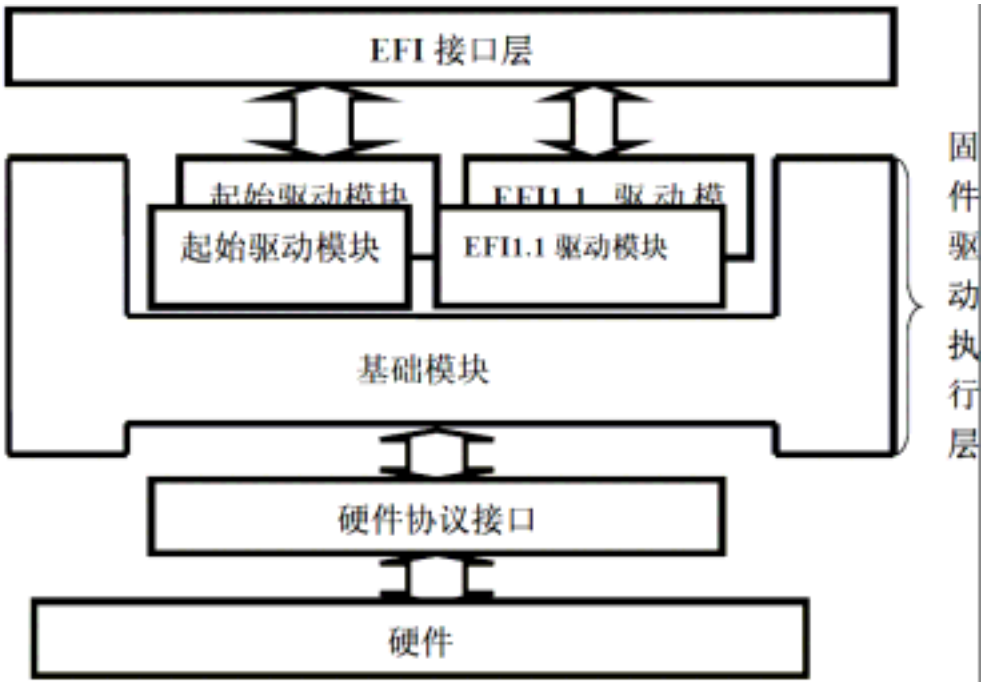

图 4.6 驱动执行层的结构图

4.3 操作系统启动选择层

(1)、设计思路

- 提供保障自己安全、可信的方法。
- 提供传统 BIOS 的接口。
- 为用户提供参数配置界面。
- 支持 EFI Shell。
- 提供包括简体中文及英文在内的多语言显示能力。

(2)、组成体系

操作系统启动选择层将从驱动层的调度器中接过控制权，几乎所有的无操作系统应用均在这一层完成。在这一层，方案将设计一个启动设备调度器，这个调度器将选择执行各种在操作系统启动前的应用程序。这样的应用程序可以是一个传统操作系统的加载器，进而去引导一个传统的操作系统而不是识别 EFI 接口的操作系统；这样的应用也可以是各种扩展的服务程序，比如硬件诊断、BIOS 升级、各种第三方的应用软件等等；最基本的一个应用当然还是识别 EFI 操作系统的加载器。

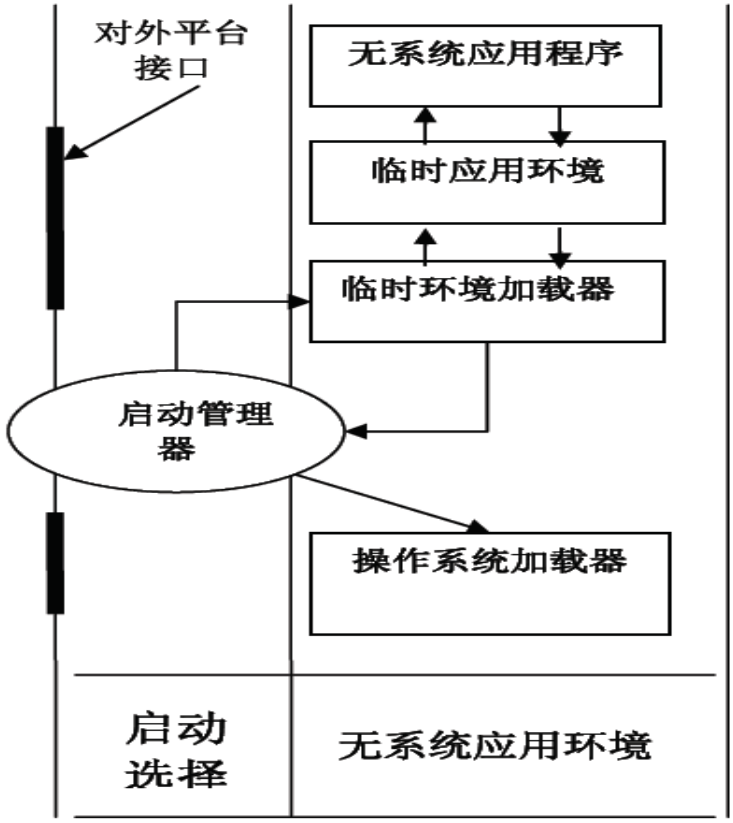

图 4.7 操作系统启动选择层位置图

图 4.7 演示了操作系统启动选择层在整个启动过程中所处的位置：即启动选择阶段和无系统应用阶段。无系统应用如果需要临时的运行环境还需要加载、设置临时的应用环境。如果启动管理器加载的操作系统加载器不能完成对应操作系统的启动，启动管理器将重新检查并尝试执行新的操作系统加载器。传统 BIOS 的兼容、显示、网络方面的应用将在本方案的兼容策略和应用部分介绍。

4.4 传统 BIOS 兼容策略

新一代 EFI 接口的 BIOS 普及需要一个过程，以前各种基于传统 BIOS 的应用不可能被一夜丢弃，可以预见的是在相当长的时间里，传统 BIOS 还是会继续使用，尤其是，第三方外设厂商基于传统 BIOS 构架开发的可选 ROM 的使用，将会延续更长时间。

4.4.1 传统 BIOS 兼容的要求

- 提供对传统操作系统启动、传统 BIOS 应用程序运行环境的完整支持。
- 兼容的部分可以方便添加、移除。

4.4.2 实现方法

对于兼容传统 BIOS，本方案将通过添加两个部分的内容来实现：

1) 传统支持模块（Compatibility Support Module，以下简称 CSM）。
2) 为支持 CSM 而增加的一系列驱动和相关代码，简称 EFI 兼容驱动。

CSM 模块在本方案中表现为多个驱动层的 32 位驱动，它是根基于本方案其他部分而添加的功能，不需要重做传统 BIOS 中的硬件初始化工作，CSM 将利用驱动层提供的服务、信息，对外提供传统 BIOS 的 16 位 INT 调用接口、服务。EFI 兼容驱动提供 CSM 加载、移除及内部信息转换的支持。一般来说 CSM 模块应该由主板生产商来提供，因为传统 BIOS 接口中有大量琐碎的和硬件相关的功能，这些应该由主板厂商设计并实现。

CSM 是在驱动层被加载，加载后在将在操作系统启动选择层形成传统 BIOS 的接口，图 4.8 反映了通过 CSM 来启动传统操作系统和运行基于传统 BIOS 服务的应用。CSM 提供的传统 16 位 BIOS 服务，还包括传统操作系统运行时的 BIOS 支持。CSM 的启动管理器清楚传统操作系统启动方式，并可以加载传统操作系统的加载器。

因为 CSM 实际上是多个驱动层的 32 位驱动，如果用户不需要传统 BIOS 的支持，可轻松卸载这些驱动，CSM 启动管理器将以进行正常的 EFI 启动，而不会加载各种可选 ROM。

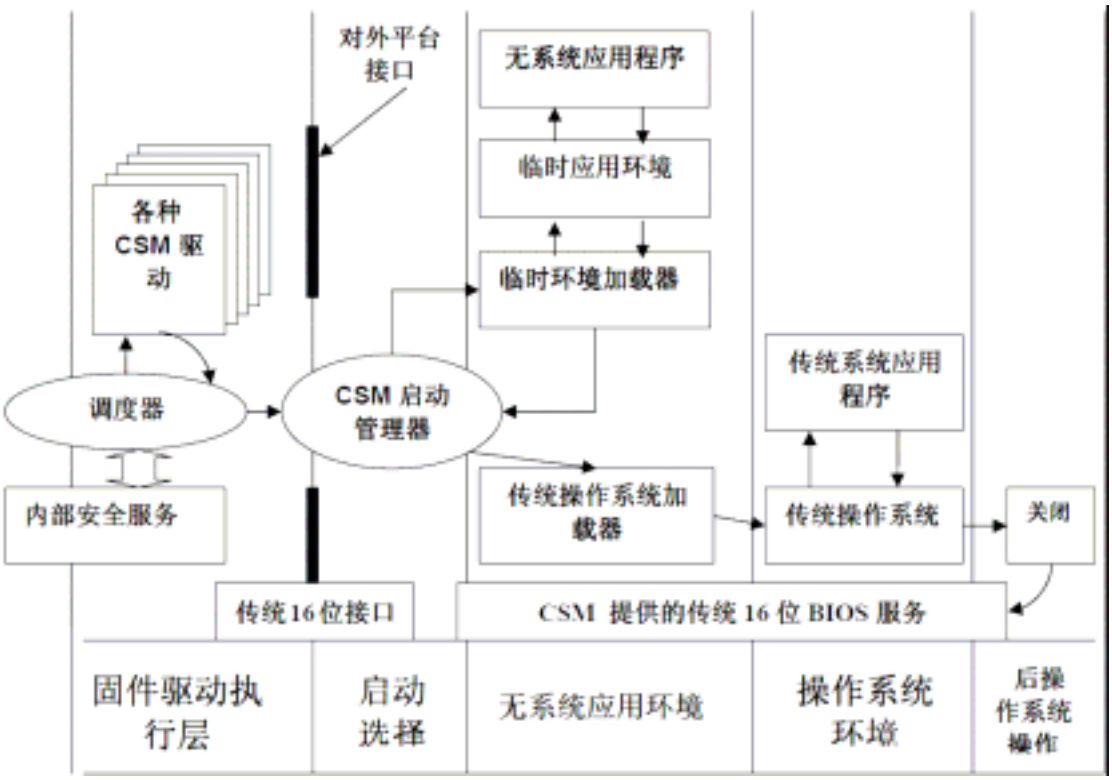

图 4.8　CSM 交互图

在本方案上添加 CSM 实现与传统 BIOS 兼容后，这个驱动层的结构图 4.9 下。图中可以看到通过 CSM 来提供对传统 BIOS 的兼容性支持，不会对其他部分的固件造成影响，而且 EFI 接口和传统 BIOS 接口可以并存，EFI 接口的应用和传统 BIOS 接口的应用也可以并存。

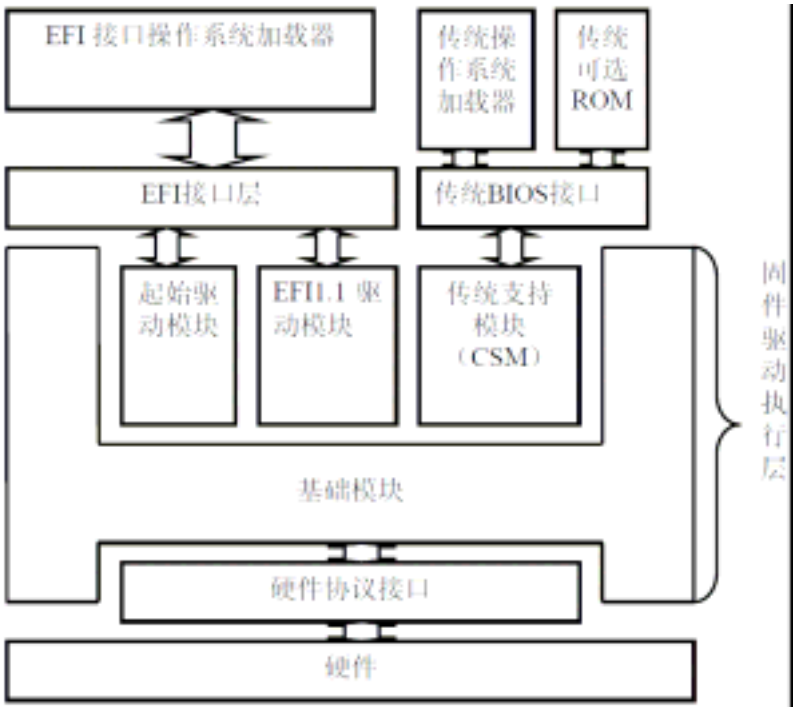

图 4.9 有 CSM 的驱动层结构图

第五章 EFI 接口 BIOS 具体驱动的实现

5.1 目标主板技术特点

我们的目标板是一款以 440BX 芯片组为核心的技嘉主板(6bxc),这款主板主要特性为:

- 一个Slot1插槽，可支持233-633MHz的Pentium II/III/Celeron 处理器。[6] [7]
- 440BX芯片组，支持AGP/SDRAM/Ultra DMA/33IDE/ACPI等功能。[6] [7]
- 提供 3 个 DIMM 插槽。[6] [7]
- 提供一个 AGP 插槽，四个 PCI Bus 插槽，三个 ISA Bus 插槽。[6] [7]
- 提供两组 IDE 接口，最多可接四块硬盘。[6] [7]
- 提供两个串行口、一个并口、以及一个软驱接口。[6] [7]
- 它支持两个 USB 接口，一个 PS/2 规范的鼠标和键盘。[6] [7]
- 使用 AWARD BIOS，闪存的容量为 2M bits。[6] [7]
- 采用四层设计符合 ATX 规格。[6] [7]

这款主板的结构框图如下图所示：

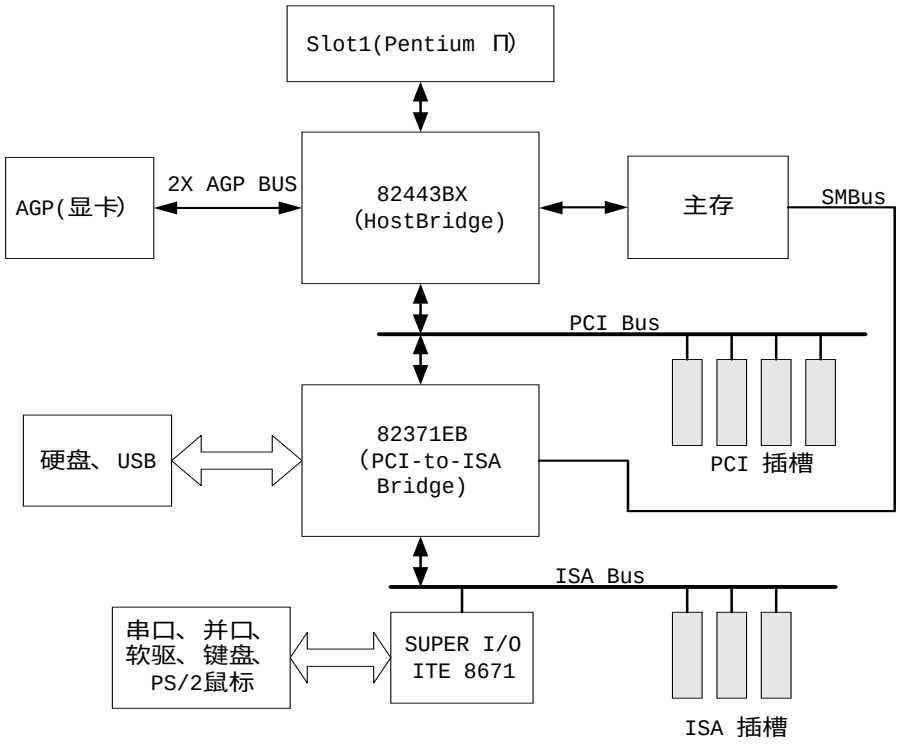

图 5.1 主板的结构框图[6] [7]

440BX 芯片组是这块主板的核心器件，它由北桥芯片（82443BX）和南桥芯片（82371EB）组成。北桥芯片在主板芯片组中起主导作用，因此也成为称为主桥（Host Bridge）。它主要负责与 CPU 的联系并控制内存、AGP、PCI 数据在北桥内部传输，提供对 CPU 的类型和主频、系统的前端总线频率、内存的类型（SDRAM，DDR SDRAM 以及 RDRAM 等等）和最大容量 1G、ISA/PCI/AGP 插槽、ECC 纠错等支持。北桥芯片也是主板上离 CPU 最近的芯片，这主要是考虑到北桥芯片与处理器之间的通信最密切，为了提高通信性能而缩短传输距离。由于北桥芯片的数据处理量非常大，发热量也越来越大，所以现在的北桥芯片往往都覆盖着散热片用来加强北桥芯片的散热，有些主板的北桥芯片还会配合风扇进行散热。值得注意的是:北桥芯片的主要功能之一是控制内存，而内存标准与处理器一样变化比较频繁，因此不同芯片组中北桥芯片是肯定不同的，虽然这并不是说所采用的内存技术就完全不一样，而是不同的芯片组北桥芯片间肯定在一些地方会有差别，这在编程时需要注意。[35]

南桥芯片（South Bridge）是主板芯片组的另一个组成部分，一般位于主板上离 CPU 插槽较远的地方，与 PCI 插槽临近，这种布局是考虑到它所连接的 I/O 总线较多，离处理器远一点有利于布线。这款主板的南桥芯片是 82371EB，提供对 KBC（键盘控制器）、RTC（实时时钟控制器）、USB（通用串行总线）、Ultra DMA/33(66)EIDE 数据传输方式和 ACPI（高级能源管理）等的支持。[35]

从上述的技术特点可以看出，Intel440BX 主板是一款经典硬件架构的主板，它具备目前主流 PC 的各种硬件设备，所以它的 BIOS 架构设计应该是 EFI 实现中最规范的一种。

5.2 USB 驱动实现

5.2.1 USB 控制器

计算机方的 USB 硬件主要是 USB 控制器，它是 PCI 设备，通过 PCI 总线和主机连接。目前 USB 控制器按照传输速度分有两种：中低速的 USB1.1 控制器和高速的 USB2.0 控制器，按照接口分有三种：UHCI、OHCI、EHCI，其中 UHCI、OHCI 是 USB1.1 的中低速控制器接口，EHCI 是高速的 USB2.0 控制器接口。USB2.0 控制器一般会捆绑一个或几个 USB1.1 的控制器，所以在市面上的各种工控机、PC、笔记本等 USB 应用系统中，USB1.1 的控制器一定存在。

USB 控制器管理 USB 总线上的设备，一个控制器对应了一条 USB 总线，驱动模块就是通过 USB 控制器来间接控制插入总线的 USB 设备。USB 控制器的控制方法和一般 PCI 设备类似，首先要得到主机 BIOS 分配给 USB 控制器的资源，

其次使用分配的资源结合接口定义来控制控制器运行。USB1.1 控制器使用的系统资源主要有两个：控制器的中断向量号、控制器接口寄存器映射地址。UHCI 将地址映射到计算机的 IO 空间，OHCI 将地址映射到内存空间。

5.2.2 USB 驱动需求

对于一个插入 USB 总线的设备，其正常工作需要软件至少提供三个部分的支持：设备驱动、USB 总线驱动、主机控制器驱动。

- 设备驱动，是整个驱动软件体系的最上层，是 USB 底层驱动的使用者，只有它知道设备中数据的含义，它使用 USB 总线驱动提供的功能接口（比如标准 WDM 中提供的各种标准接口函数）来读写数据，并处理读出或写入的数据，比如加密、解密，提供给更上层的程序使用。各种基本的功能函数的实现对设备驱动程序是透明的。
- USB 总线驱动，是 USB 总线的抽象层，它对设备驱动屏蔽了 USB 总线的操作细节，它负责自动检查插入新设备，分配、调度 USB 总线的各种资源，调用下层硬件的各种标准 USB 请求来合成上述设备驱动使用的功能函数。
- 主机控制器驱动，提供了主机控制器硬件的抽象。主机控制器驱动通过硬件的访问完成各种标准 USB 请求（比如 Get Descriptor、Set Address 等），各种标准 USB 请求会被 USB 总线驱动层调用。因为主机控制器都会集成根集线器，所以主机控制器驱动要提供对根集线器的支持。

USB 驱动基本架构如图 1 左侧，右侧是有操作系统时的架构：

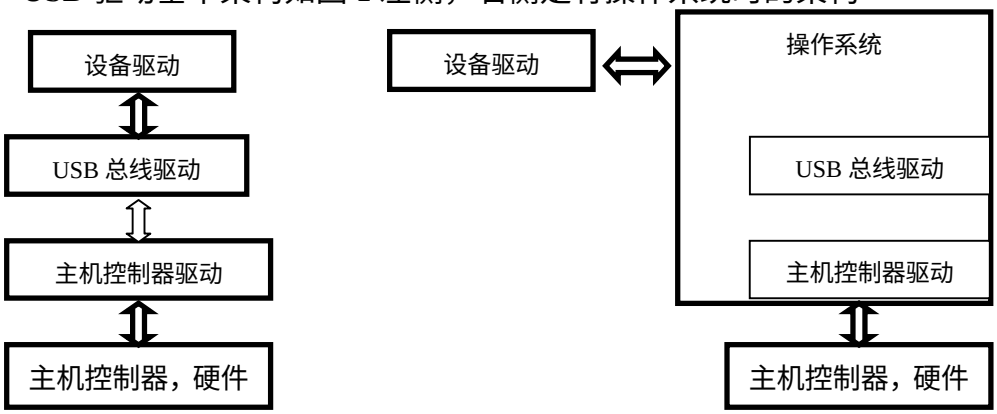

图 5.2　USB 驱动层次结构

在有操作系统的环境下，操作系统一般会提供主机控制器驱动、USB 总线驱动以及 USB HUB 设备驱动。操作系统对自己实现的基本功能进行了封装，形成了各种 API 函数。设备驱动程序是使用主机软件提供的 API 函数来读写数据，

这种方式也是一般 USB 驱动的实现方法。

USB 总线驱动和主机控制器驱动是最主要的两个部分，是 USB 驱动的核心，也是一般操作系统提供的部分，所以实现 USB 总线驱动和主机控制器驱动是实现整个无操作系统支持驱动的关键和难点。

USB 其它细节请参见相关资料，这里直接介绍在 440BX 体系中 USB 驱动的实现。USB 体系本身是一个很有层次的结构，完整的 USB 驱动是一系列驱动组成的多层次协议栈。因为 440BX 体系是兼容传统 BIOS 的，目前对传统 BIOS 接口的支持仍然不可缺少。所以下面给出 USB 驱动栈在 440BX 驱动模型中的实现和在传统 BIOS 部分的实现。

5.2.3 EFI 标准驱动模型的 USB 驱动栈实现

下图给出 EFI 需要的 USB 驱动栈的例子和驱动之间各个协议的相互关系。

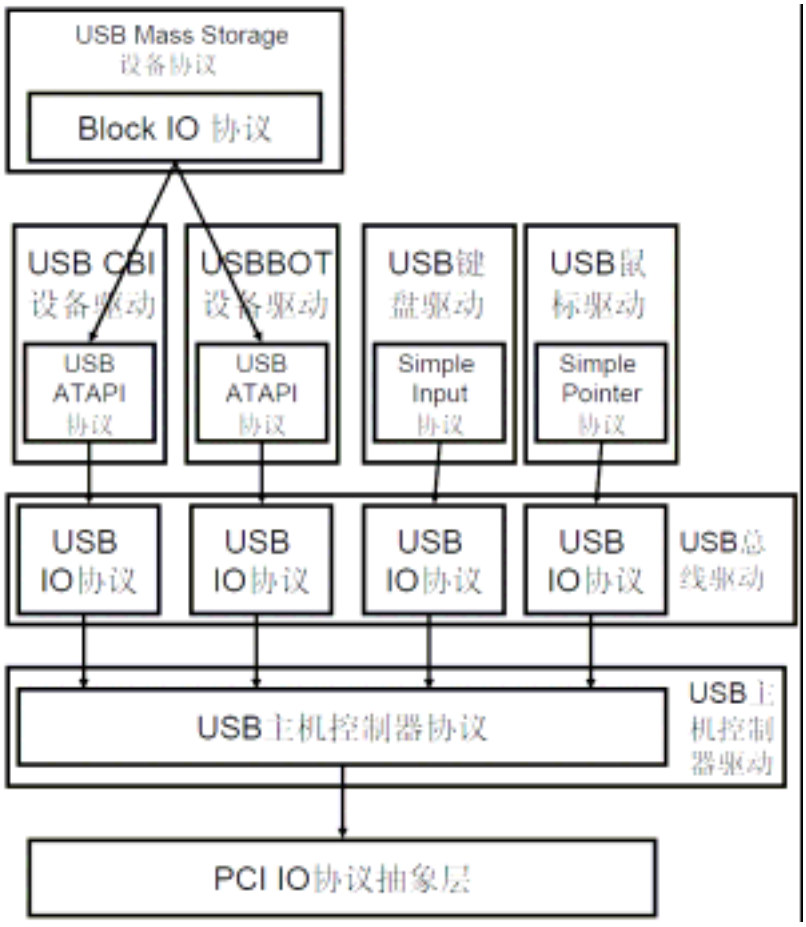

图 5.3　EFI 接口 USB 驱动栈

这个例子中，PCI 总线上有单一的 USB 主机控制器。PCI 的总线驱动会扫描发现这个控制器，生成代表这个控制器的句柄并在句柄上安装 EFI_DEVICE_PATH_PROTOCOL 和 EFI_PCI_IO_PROTOCOL，这就是最下面的 PCI IO 协议抽象层。

USB 主机控制器驱动会使用 EFI_PCI_IO_PROTOCOL 的服务，并在主机控制器的句柄上安装 EFI_USB_HC_PROTOCOL 协议。这就是第 2 层的 USB 主机控制器驱动层。

USB 总线驱动会使用 EFI_USB_HC_PROTOCOL 协议。在这个例子中，USB 总线上会有 USB 键盘、USB 鼠标和两个 USB 的 mass storage 设备，这样 USB 总线驱动会生成 4 个子句柄，来分别对应这 4 个 USB 设备的接口控制器，并在每个子句柄上安装 EFI_DEVICE_PATH_PROTOCOL 和 EFI_USB_IO_PROTOCOL。图中是第 3 层 USB 总线驱动层。

USB 鼠标驱动使用下层的 EFI_USB_IO_PROTOCOL 并且安装 EFI_SIMPLE_POINTER_PROTOCOL 在鼠标对应的句柄上。USb 键盘使用下层的 EFI_USB_IO_PROTOCOL 并且安装 EFI_SIMPLE_INPUT_PROTOCOL 在键盘对应的句柄上。因为 USB 的 mass storage 设备可以有两套传输协议，所以可以有两套命令接口的实现。这里将传输协议的实现也独立出来做为一层，这样 USB mass storage 设备的驱动又分成两层。第一层将使用下层的 EFI_USB_IO_PROTOCOL 并且安装 EFI_USB_ATAPI_PROTOCOL 做为统一接口向上提供服务。第二层将使用下层的 EFI_USB_ATAPI_PROTOCOL 并且安装 EFI_BLOCK_IO_PROTOCOL。这就是最高两层。

5.2.3.1 USB 主机控制器驱动

USB 主机控制器驱动是一个设备驱动，它符合 EFI 驱动模型。一般来说它会使用 EFI_PCI_IO_PROTOCOL 协议的服务并提供 EFI_USB_HC_PROTOCOL 协议.下面具体说明 USB 主机控制器 EFI 驱动模型中驱动绑定协议的实现方法。

Supported()

USB 主机控制器驱动必须要实现驱动绑定协议，也即要实现三个基本的功能函数：Supported()、Start()、Stop()。Support 函数会评估传入的句柄参数 ControllerHandle，检查这个句柄代表的 Usb 主机控制器是否可以被驱动控制。检查的原理是，驱动读取主机控制器的 PCI 配置空间，检查这个控制器的设备类型和设备 ID 即制造商 ID 字段，如果这些字段都符合要求，Supported()会返回成功，否则返回不支持。可以看到，这种检查驱动是否支持硬件的方法没有对硬件进行任何实际控制操作，仅仅读取了硬件控制器的类型信息，没有对硬件做任何的影响。

下面给出了 Supported()具体的实现代码：首先，函数尝试打开控制器句柄上的 PCI IO 协议，如果不能打开，说明这个控制器不是一个 PCI 设备，直接返回

错误。如果可以，再用 PCI IO 协议中的功能服务函数读出 PCI 配置空间中的 PCI 设备类型等字段，如果都匹配则返回成功。

```
EFI_STATUS
EFIAPI
UHCIDriverBindingSupported (。。。。。。)
{
    。。。。。。
  //
  // 检查控制器句柄上是否有PCI IO 协议
  //
  OpenStatus = gBS->OpenProtocol (。。。。。。);
  if (EFI_ERROR (OpenStatus)) {
    return OpenStatus;
  }

  Status = PciIo->Pci.Read (。。。。。。);
  if (EFI_ERROR (Status)) {
    gBS->CloseProtocol (。。。。。。);
    return EFI_UNSUPPORTED;
  }
  //
  // 检查是否是UHCI控制器
  //
  if ((UsbClassCReg.BaseCode != PCI_CLASS_SERIAL)          ||
      (UsbClassCReg.SubClassCode != PCI_CLASS_SERIAL_USB) ||
      (UsbClassCReg.PI != PCI_CLASSC_PI_UHCI)) {

    gBS->CloseProtocol (。。。。。。);
    return EFI_UNSUPPORTED;
  }
  gBS->CloseProtocol (。。。。。。);
  return EFI_SUCCESS;

}
```

Start()和 Stop()

Start()的功能函数完成控制器的初始化和驱动的工作，因为篇幅过大，将在后面的传统 USB 驱动实现中介绍。下面给出 Stop()函数的具体实现。

USB 主机控制器的 Stop()函数会首先检查传入的控制器句柄是否自己支持的设备，然后释放分配给主机控制器的所有资源，主要是内存资源，最后通知驱动执行层卸载这个协议。

```
EFI_STATUS
EFIAPI
UHCIDriverBindingStop (。。。。。。  )

{
    。。。。。。

  OpenStatus = gBS->OpenProtocol (。。。。。。 );

  //
  // 检查传入的控制器句柄是否自己支持的设备
  //
  if (EFI_ERROR (OpenStatus)) {
    return OpenStatus;
  }
  //
  // 释放分配给主机控制器的所有资源并卸载这个协议
  //
  UnInstallUHCInterface (Controller, UsbHc);

  gBS->CloseProtocol (。。。。。。  );
  return EFI_SUCCESS;
}
```

5.2.3.2 USB 总线驱动

USB 总线驱动会使用 USB 主机控制器提供的 EFI_USB_HC_PROTOCOL 协议来枚举 USB 设备，生成各个 USB 设备控制器的句柄，并在每个子句柄上安装 EFI_DEVICE_PATH_PROTOCOL 和 EFI_USB_IO_PROTOCOL。

5.2.3.3 USB 设备驱动

USB 设备驱动的作用是提供 USB 设备的 IO 抽象，它通过 EFI_USB_IO_PROTOCOL 协议来提供抽象服务，即向上提供 USB 设备的功能函数。

Supported()

USB 的设备驱动都是符合 EFI 驱动模型的，下面给出 USB 设备驱动绑定协议中 Supported()的一般实现方法。

首先，检查目标设备句柄是否安装有 EFI_USB_IO_PROTOCOL 协议，如果

有，说明 USB 总线驱动已经辨认出这是一个 USB 设备。

其次，使用 EFI_USB_IO_PROTOCOL 协议提供的功能函数读出目标 USB 设备的设备描述符，描述符中 BaseClass、SubClass、Protocol 字段指定了的 USB 设备的类型，通过检查这些字段就可以知道驱动是否是目标 USB 设备的驱动。

下面假设有个 USB 设备叫 XYZ，它的 Supported()函数实现如下。

```
EFI_STATUS
  USBXYZDriverBindingSupported (。。。。。。)
{
  。。。。。。
//
//检查目标设备句柄是否安装有USB_IO协议
//
OpenStatus = gBS->OpenProtocol (。。。。。。);
if (EFI_ERROR (OpenStatus)) {
    return OpenStatus;
}
//
// 读出目标USB设备的设备描述符
//
    Status = UsbIo->UsbGetInterfaceDescriptor(。。。。。。);
if(EFI_ERROR(Status)) {
    Status = EFI_UNSUPPORTED;
} else {
    //
    // 检查是否是目标USB设备的驱动
    //
    if (InterfaceDescriptor.InterfaceClass == CLASS_XYZCLASS &&
        InterfaceDescriptor.InterfaceSubClass == SUBCLASS_XYZSUBCLASS &&
        InterfaceDescriptor.InterfaceProtocol == PROTOCOL_XYZPROTOCOL) {
        Status = EFI_SUCCESS;
    } else {
        Status = EFI_UNSUPPORTED;
    }
}
        gBS->CloseProtocol (。。。。。。);
return Status;
}
```

5.2.4 CSM 传统兼容模块中 USB 驱动栈的实现

新一代 EFI 接口的 BIOS 普及需要一个过程，以前各种基于传统 BIOS 的应用不可能被一夜丢弃，传统 BIOS 还是会继续使用。所以做为一个完整的驱动，应该给出 CSM 传统兼容模块中的实现。下面将给出在 CSM 中实现的 USB 驱动栈。这部分使用了 UML 的图形语言来描述 USB 的驱动实现。

5.2.4.1 框架设计

下面图 5.4、图 5.5 展示了传统 USB 驱动程序的架构，是整个驱动的逻辑视图，也就是驱动的组成。图 5.4 描述的是逻辑骨架，图 5.5 是添加实例后的视图。

在下面两幅图中，小圆圈代表接口类（比如应用程序接口 API），人形图标代表角色类（比如应用程序 APP），大圆圈下加一横线代表实体类（比如事务描述符），大圆圈上加一个逆时针的肩头代表控制类（USB 总线管理器 UsbMan），小头的箭头代表单向关联关系（即箭头尾的类知道箭头所指类的公共属性和操作），三角头的箭头代表泛化关系（即具体的实现，比如通用主机控制器 UHC 是主机控制器 HC 的泛化），菱形的箭头表示积累关系（即整体与部分的关系，比如 USB 设备协议管理器 USBDevProtMan 是 USB 总线管理器的子部分）。

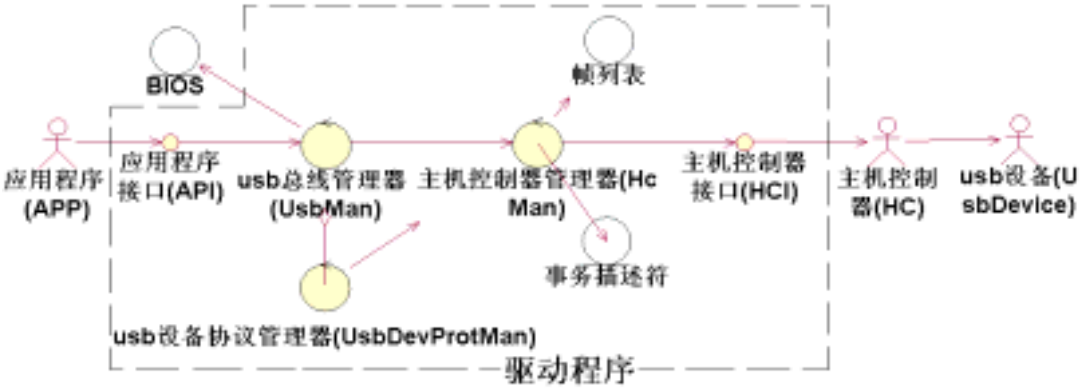

图 5.4　CSM 模块 USB 驱动架构逻辑骨架

图 5.4 中从左而右各个部分解释如下：

1) 应用程序（APP），角色类，是驱动的使用者角色。
2) 应用程序接口（API），接口类，是驱动向 APP 提供的接口，APP 通过 API 向驱动发请求。接口中一共定义了 6 个函数，其中最关键的有 4 个：INIT_USB()、READ()、WRITE()、SHUT_DOWN()，分别是初始化、读、写、关闭。
3) USB 总线管理器（UsbMan），控制类，是对上文 USB 总线驱动层的实

现，它负责管理、调度在 USB 总线上的一般的总线操作，包括初始化/关闭 USB 总线、配置新插入的 USB 设备并分配资源、发送标准的设备请求等等。关键函数包括：UsbInit()、UsbShutDown()、UsbNewDeviceOpen()、UsbDeviceClose()，分别是初始化 USB 总线、关闭 USB 总线、配置新插入的 USB 设备以及关闭设备。USB 设备协议管理器（USBDevProtMan）是 USB 总线管理器的子部分，它负责根据 USB 设备各自的通讯协议访问 USB 设备。当新插入的设备配置完毕后，USB 总线管理器将后续应用程序对 USB 设备的读写请求直接传递给 USB 设备协议管理器，由它来完成读写。USB 设备协议管理器（USBDevProtMan）通过组合使用下层提供的各种标准及非标准请求，来完成具体设备的的读写操作。比如，不同的加密狗根据自己硬件制定的通讯协议，往往需要几次控制传输、中断传输的组合才能完成的一次读写。关键函数包括: DogRead()/DogWrite()、DiskRead()/DiskWrite()，分别是加密狗的读写和 U 盘的读写函数。

4) 主机控制器管理器（HcMan），控制类，也就是主机控制器驱动，是对上文主机控制器驱动层的实现。主机控制器管理器（HcMan）根据 USB 总线管理器（UsbMan）的需要提供主机控制器的相应支持。它要提供 USB 协议中规定的标准请求（比如 Set Address），它根据 USB 总线管理器的要求调整帧列表来完成分配带宽，读写实际的控制器地址来控制 USB 事务（transaction）的运行等等。比如在 UHCI 的主机上，当 UsbMan 调用“Set Address”标准设备请求时，主机控制器管理器则把 USB 总线管理器提供的“Set Address”请求数据装配到 UHCI 的一次控制传输中去，完成本次控制传输后把执行的状态返回给 USB 总线管理器。主机控制器管理器的两种具体泛化是通用主机控制器管理器（UhcMan）和开方主机控制器管理器（OhcMan）。
5) 事务描述符，实体类，包括 TD、QH、ED 等等和事务传输有关的数据实体，它和帧列表构成 USB 驱动的核心数据实体。不同的主机控制器有不一样的事务描述符。
6) 帧列表，实体类，是帧列表的数据结构实体。USB 总线管理器和主机控制器管理器通过设置各种 TDs、QHs、EDs 来组合成四种不同方式的传输，并通过调度挂载到帧列表的位置来为不同的传输分配不同的带宽。不同的主机控制器有不一样的帧列表结构。
7) 主机控制器接口（HCI），接口类，它是主机上的硬件接口的映射，驱动通过 HCI 来控制主机控制器。HCI 的两种具体泛化是通用主机控制器接

口（UHCI）和开放主机控制器接口（OHCI）。

8) 主机控制器（HC），角色类，它完成 USB 线路上实际的信息传输。主机控制器根据帧列表、事务描述符两类规定的任务、路线在主机控制器管理器下完成事务的执行。HC 的两种具体泛化是通用主机控制器（Uhc）和开放主机控制器（Ohc）。

9) USB 设备（USBDevice），角色类，包括 USB disk、USB dog 等。

10) BIOS，实体类，主机 BIOS 为 UsbMan 提供 hc 的系统信息。

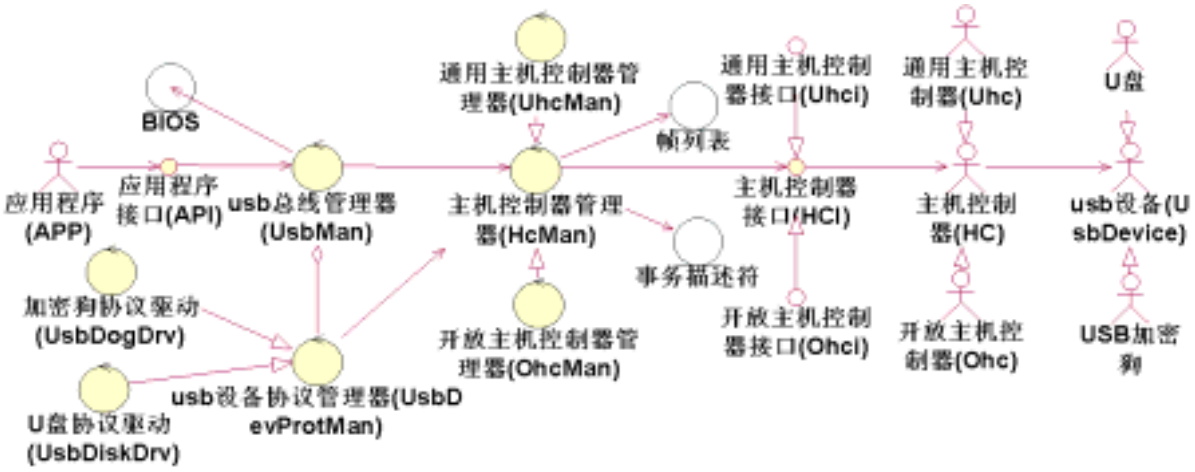

图 5.5　CSM 模块 USB 驱动架构骨架及实例

5.2.4.2 驱动重点流程介绍

下面选择了三个重要的 USB 操作，分析了它们的流程。

5.2.4.2.1 USB 总线初始化

USB 总线初始化是软件工作的第一步，实现的功能是建立驱动的运行环境并启动 USB 控制器，初始化后软件接管 USB 硬件的控制权。USB 总线初始化在设计中通过发送 USB 初始化请求完成。

图 5.6 是驱动中 USB 初始化请求实现的顺序图，图中序列号的前后次序表示了实际步骤的先后关系。关键步骤如下：

图 5.6 步骤 1、应用程序发出 USB 初始化请求。

图 5.6 步骤 2、应用程序接口检查请求是否是合法的。

图 5.6 步骤 4、USB 总线管理器接受到消息后首先通过 BIOS 得到 USB 控制器的 PCI 地址、中断向量，获取的方法是调用 BIOS 的 INT1ah 中的 0B103h 子

功能。

图 5.6 步骤 6、在建立新工作环境前，备份控制器原来的环境。

图 5.6 步骤 9、建立新的工作环境，主要是建立新的帧列表和它所指向的事务描述符。

图 5.6 步骤 12、在加载新的工作环境时，应该注意应先停止 USB 控制器。

图 5.6 步骤 13、在新工作环境建立完成后，启动控制器。

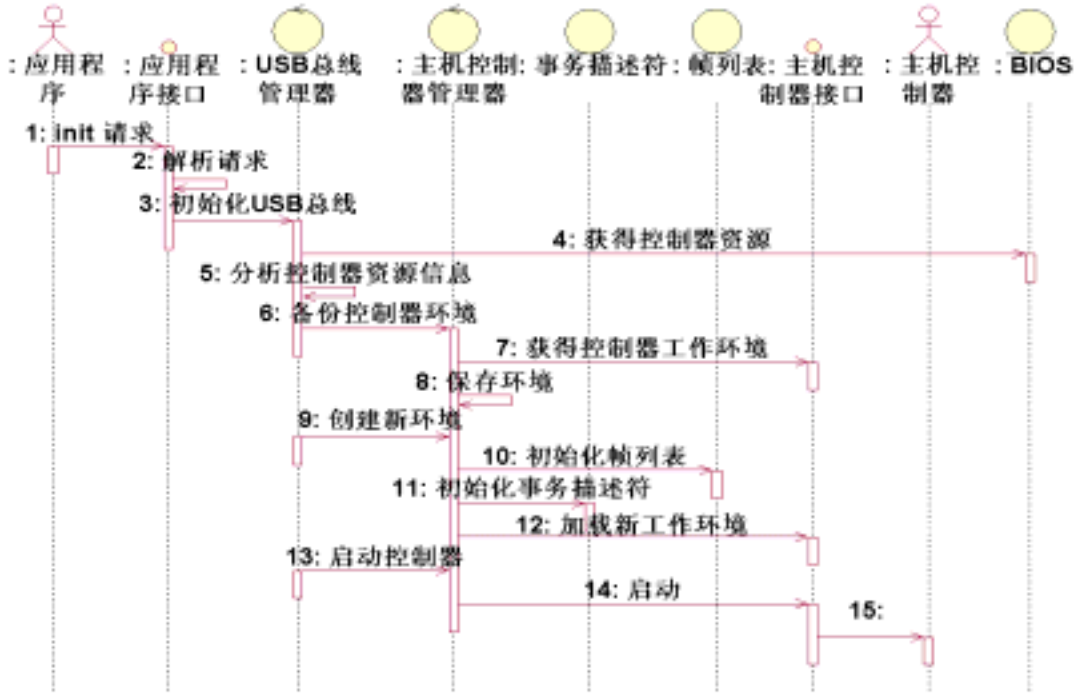

图 5.6　USB 总线初始化请求顺序图

5.2.4.2.2 打开 USB 设备与中断传输的实现

打开设备是主机与 USB 设备建立连接的过程，过程中将会给 USB 设备分配地址、带宽等各种资源，在这一过程中使用了大量的控制传输。打开 USB 设备在设计中通过发送打开请求完成。

图 5.7 是驱动中打开请求的顺序图，打开请求是在 USB 初始化请求后执行的操作，所以 USB 总线管理器需要检查 USB 总线目前所处的状态，即图 5.7 步骤 4。从图 5.7 步骤 5 开始，USB 总线管理器将设法找到新设备并与之建立连接。因为新设备是通过 USB 控制器中的根集线器连接到 USB 总线的，所以根集线器在发现有新设备插入时，需要通过中断通知 USB 总线管理器。这里就要介绍一下 USB 中断传输的实现原理。

USB 数据传输的管理方式都是主从式的，就是说传输的双方一定有一个

“主”，即 master，和一个“从”，即 slave，而主机的 USB 控制器永远是 master，USB 设备永远是 slave，在 USB 总线上的每次数据传输永远是由 master，即 USB 控制器发起的。而对于中断传输一般要求中断源，即 USB 设备，可以引起中断而发起中断传输，这就和 USB 数据传输管理方式有矛盾。为了解决这个矛盾，USB 总线使用伪中断来实现中断传输。这里所谓的“伪中断”是 USB 设备在 USB 控制器预先知道的情况下发起中断，而不是一般的在主控方不知情的状况下打断主控方，也就是说 USB 控制器预先就设计好了“中断时间点”，在固定的中断时间点会通知 USB 设备让它“中断”。所以 USB 的中断传输实际上是 USB 控制器以“定时查询”的方式来控制 USB 设备传输数据。图 5.7 步骤 5～9，就完成了中断传输的设置，方法如下：

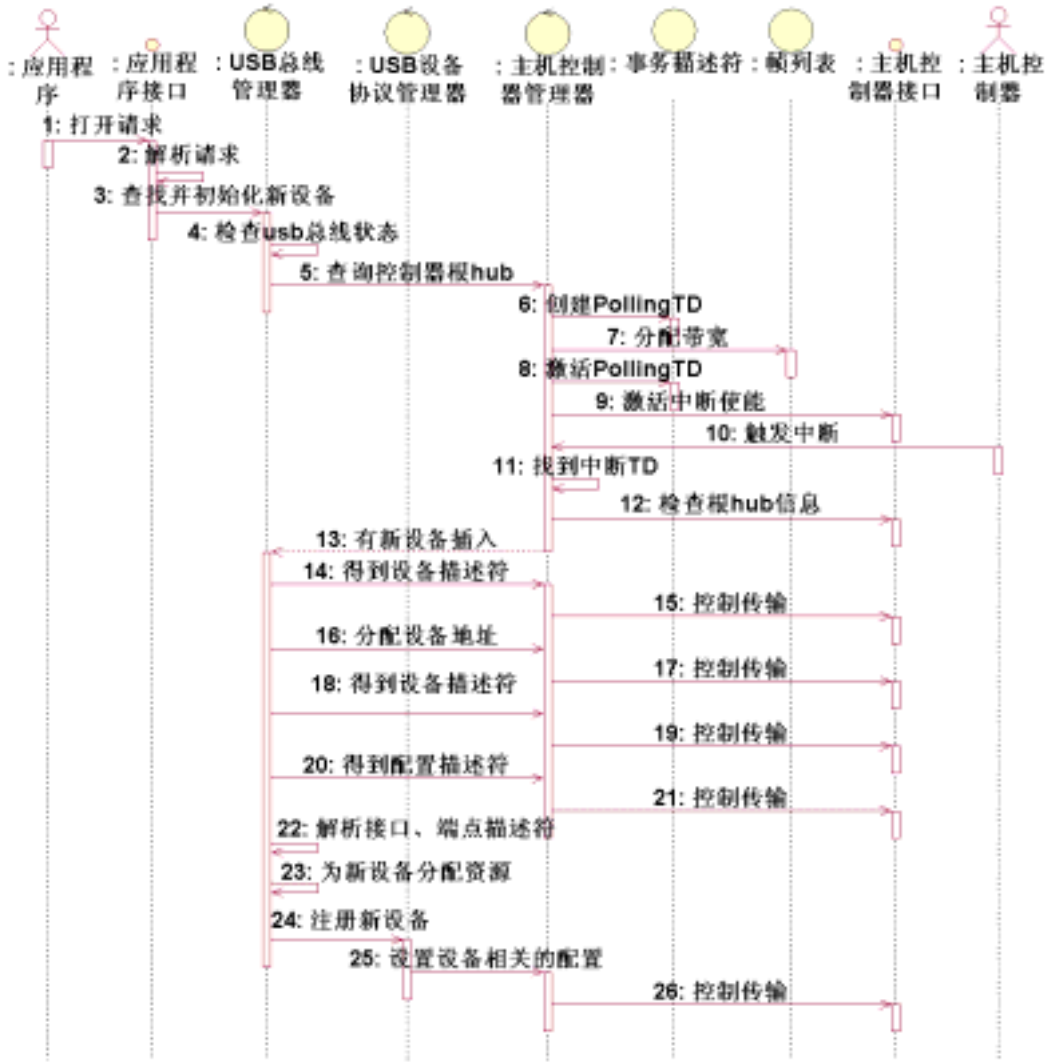

图 5.7 打开请求顺序图

图 5.7 步骤 5、USB 总线管理器通知主机控制器管理器需要根集线器（根 hub）对新设备插入做一次“ 中断 ”反应。

图 5.7 步骤 6、主机控制器管理器为触发中断准备了一个特别的事务描述符，即 Polling TD。这个 TD 的设备地址是一个 Dummy 地址，即在 USB 总线上不会出现的地址，同时这个 TD 的允许出错次数设置为 1。当这个 TD 在 USB 总线上执行时，因为目标设备地址不存在一定会报错，同时又因为此 TD 的允许出错次数为 1，所以当此 TD 执行会立刻触发错误中断。

图 5.7 步骤 7、主机控制器管理器将此 TD 挂载到帧列表中适当位置，使得 HC 可以周期性地执行此 TD。

图 5.7 步骤 8 与 9 中当此 TD 激活并打开主机控制器中断使能后，USB 设备就在每次 Polling TD 执行时触发“ 中断 ”。

图 5.7 步骤 10、是设备通过主机控制器触发中断后将程序控制权移交给主机控制器管理器。主机控制器管理器发现是 Polling TD 触发中断后会在步骤 12 检查根集线器的状态，并在有设备插入时通知 USB 总线管理器。

图 5.7 步骤 14～步骤 26 是标准的 USB 设备配置流程：

图 5.7 步骤 14、首先，USB 总线管理器用新设备的默认 0 地址 0 端点号发送 get Descriptor 请求来获得设备的描述符，请求中 bMaxPacketSize0 默认长度定为 8，在设备描述符中偏移 7 的位置得到 bMaxPacketSize0 的实际大小。

图 5.7 步骤 16、其次，USB 总线管理器使用新得到的 bMaxPacketSize0 发送 set Address 请求为新设备分配一个总线上唯一的地址。

图 5.7 步骤 18、在必要的延时后，USB 总线管理器用新的设备地址和实际得到的 bMaxPacketSize0 重新发送 get Descriptor 请求来获得设备的描述符。在新得到的设备的描述符中有 USB 设备的厂商 ID、设备 ID 和配置数量等重要信息。

图 5.7 步骤 20、根据等到的配置数量发送 Get Configuration 请求依次读出每一个配置描述符。在这个步骤中可能因为设备支持多个配置而需要多次发送 Get Configuration 请求。

图 5.7 步骤 22、在每次得到的配置描述符中有当前配置包含得所有接口、端点描述符。分析每个接口描述符即可知道设备的功能类别，即设备有什么功能。分析每个端点描述符即可知道此接口的通讯方式。步骤 20、22 可以说是识别 USB 设备的关键，也是工作量最大的地方。

图 5.7 步骤 24、在识别了新设备并了解所有功能后，USB 总线管理器会为之分配必要的资源，即步骤 23，并通知 USB 设备协议管理器注册新设备。

图 5.7 步骤 25、USB 设备协议管理器会根据具体设备的配置设备协议对新设备做专有的配置。

5.2.4.2.3 控制传输实现

在上述打开 USB 设备过程中用到了大量控制传输，控制传输是 USB 使用的最频繁的传输方式，在设备配置过程中的各种标准请求就是一次次的控制传输。

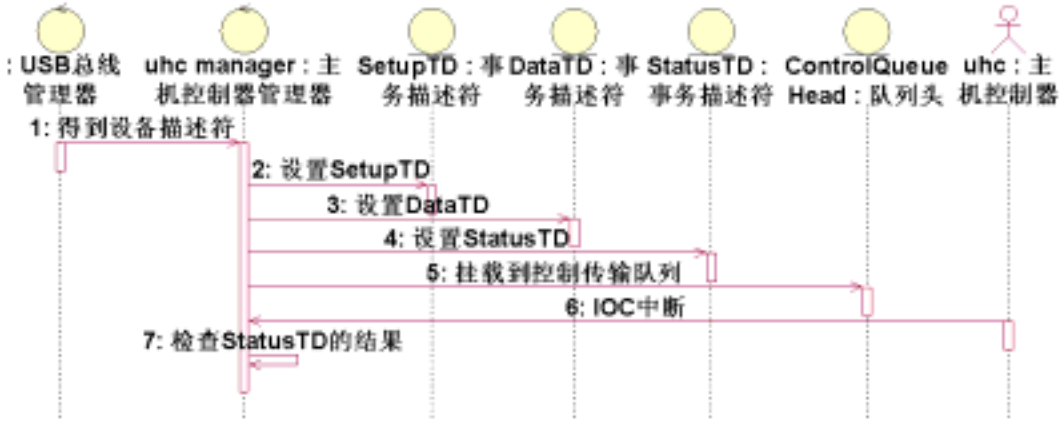

图 5.8 UHCI 主机控制传输

图 5.8 以 UHCI 主机上 get Descriptor 请求的执行展示了一次控制传输的过程。控制传输由三个阶段组成，分别是准备阶段、数据阶段、状态阶段。准备阶段由一个名为 SetupTD 的事务完成；数据阶段由若干名为 DataTD 的事务完成，具体个数由传输的数据量来决定，状态阶段由一个名为 StatusTD 的事务完成。USB 总线管理器在设置好三个阶段的事务后（即步骤 2、3、4），将事务按执行先后顺序链接并挂载到FRAMEs帧列表的控制传输入口ControlQueueHead处(即步骤 5)，ControlQueueHead 是 UHCI 的一个队列头结构（QH），在驱动中这个队列头是控制传输的加载点。当指向入口的帧活动时，USB 控制器会按照链接的顺序依次执行三个阶段的事务。完成后检查 StatusTD 来验证传输是否成功。当三个阶段的任何一个事务失败会将出错信息写入 StatusTD，并通过软件终止这个事务队列的后续执行。

第六章 EFI 接口 BIOS 驱动体系的安全应用方案

本驱动体系借鉴并移植了部分开源项目的架构设计，存在被攻击的风险，所以利用开源平台时必须考虑整体的安全策略。在驱动体系的应用这部分，将针对安全问题给出两个安全应用的方案。

6.1 基于 TPM 的固件的安全方案

6.1.1 安全策略体系结构

对于 BIOS 及 PC 的安全性，各种基于软件、网络和服务器的信息安全解决方案层出不穷，然而 PC 对于各种病毒和信息窃取事件的应付招数依然捉襟见肘，尴尬频频。这是因为传统的 PC 安全保护是通过局部的、应用的和软件的手段来进行的，而真正意义上的安全 PC，是应该可以实现基于硬件的、全面的，从体系结构上实现的最全面和彻底的安全保护，因为信息安全落到最实处还是要依靠硬件的保障。只有从芯片、主板等硬件结构和 BIOS、操作系统等底层软件做起，综合采取措施，才能从源头上得到控制，实现真正的 PC 信息安全。

作为下一代安全 BIOS，它的安全策略应该从一个体系结构上来设计，应该符合未来安全 PC 对 BIOS 的要求，不但要满足自身安全的需要，还要能为上层操作系统提供服务。本方案的安全策略就是从硬、软件结合的全局角度来构建的。

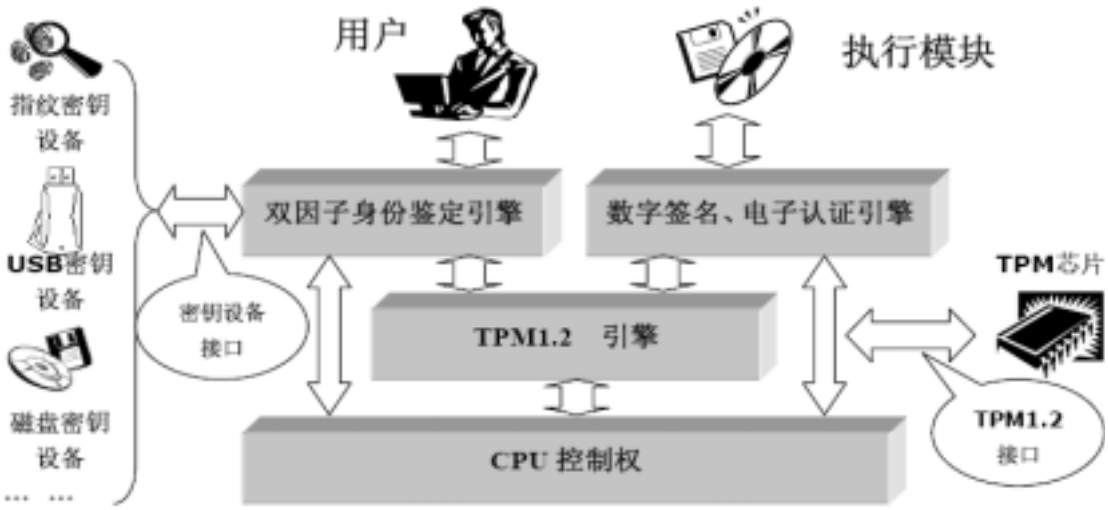

图 6.1 BIOS 安全策略

图 6.1 展示了本文提出的 BIOS 安全策略。如图所示，本文的安全策略分为软硬两部分：硬件的 TPM 芯片、各种安全密钥设备；软件的身份鉴定、电子认证及 TPM 的驱动引擎。

TPM 技术标准来源于“TCG”的组织，它原理是在主板中植入一块不可移动的 TPM 芯片，由它来提供和安全相关的基本功能。TPM 芯片包含加密与解密引擎、受保护的存储区。集成了 TPM 芯片的系统可以有能力将各种密钥本身加密并存储于 TPM 芯片，这样只有通过 TPM 才可以取出密钥。TPM 还提供加封、解封的应用将一个密钥的获取同某个平台的度量值相关联，这样可以来衡量其他平台的安全可信性。最新的 TPM 标准是 TPM1.2。

图 6.1 中的双因子身份鉴定是指，在鉴定用户身份时，不但需要用户提供口令还需要用户提供硬件的密钥设备。这样的设计是参考了银行信用卡的安全原理，通过硬件实体＋秘码的方法来确定用户的合法性。

图 6.1 中的数字签名、电子认证是指，在 BIOS 层中任何要执行的模块在执行前，必须提供自身的认证信息，在核实这些认证信息合法后，BIOS 才会允许模块加载运行。

图 6.1 的双因子身份鉴定、电子认证及 TPM 构成了针对用户、执行模块两层的安全保护。

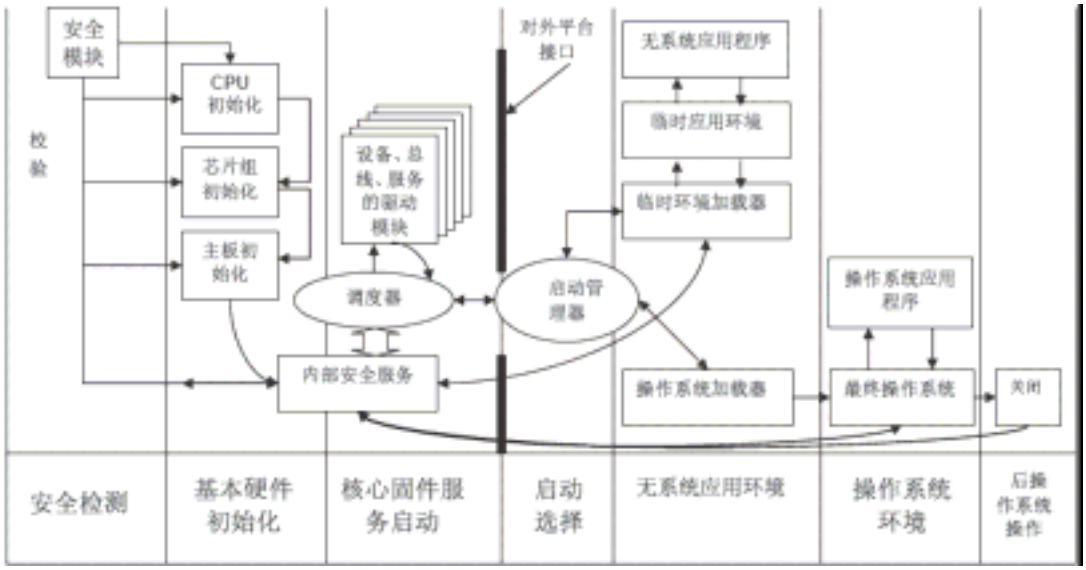

图 6.2 加入本安全策略后各部分交互关系

图 6.2 给出加入本安全策略后，各部分的交互关系，显示了在计算机开机到操作系统关闭一个完整周期内，本方案各个部分的调度关系。

图 6.2 各层次交互关系是：

开机后即进入安全检测阶段，一个安全模块（TPM 芯片）将得到控制权，此模块的作用是对后续部分提供安全校验，这个安全模块的校验将贯穿整个固件的生命期。安全模块在自身的初始化后会首先对整个固件进行总体校验，然后启动基本 EFI 初始化准备的任务，即进入 EFI 初始化准备阶段。

EFI 初始化准备阶段即对应于基本 EFI 初始化准备层，它通过执行一个个的基本初始化模块来完成这个阶段的初始化工作。每个初始化模块在执行前都需经

过安全模块的校验，初始化层寻找并初始化最少的硬件资源，这些资源可以满足启动驱动层即可，基本 EFI 初始化准备完成后加载驱动层，即进入核心固件服务启动阶段。

核心固件服务启动阶段即对应核心固件层，它彻底完成所有 EFI 初始化准备，并为上层接口实现所需要的各种服务。核心固件层将通过查找一个个驱动模块来完成所有硬件的登记，并在服务被调用时实际执行相应功能的驱动模块。本方案的安全应用在驱动层中加入了内部安全服务，在调度器登记和执行每个驱动模块时，模块都要经过内部安全检测。当驱动层完成所有初始化后，就对外提供了统一的接口：EFI 接口和传统 BIOS 接口（可选），这两个接口为上层的各种应用模块屏蔽了底层硬件细节，提供了各自统一的运行环境，在统一的运行环境中可以运行启动管理器等上层应用模块。一旦驱动层完成初始化并尝试启动操作系统时，就进入了启动选择阶段。

启动选择阶段实际上是一个启动选择器选择执行应用模块，几乎所有的无操作系统应用均在这一层完成，可以选择执行的应用模块即对应层次图中的 4 个部分：EFI 接口操作系统加载器、EFI 接口应用模块、传统接口操作系统加载器、传统接口应用模块。一旦具体的模块开始执行即进入了无系统的应用阶段。

无系统的应用是外部用户基于本方案固件提供的接口、服务展开的应用。在众多应用当中有一个最重要的应用就是操作系统的加载。当操作系统被成功加载后就进入操作系统运行阶段。

操作系统启动后本方案核心固件仅仅保留操作系统运行时及运行后需要的接口、服务。

当操作系统结束运行后控制权将最终又交换给固件，操作系统结束运行可能是合法的关机也可能是被强迫的非法关闭，上图的内部安全服务在操作系统交还控制权后要进行最后的可信校验。

6.1.2 如何在下一代 BIOS 中集成安全策略

本文安全策略中 TPM 引擎是实现的难点，下面介绍如何在下一代 BIOS 中集成安全策略。

TPM 需要固件提供的支持主要是 TPM 芯片的协议接口，为基于固件的应用模块和操作系统加载器提供服务，也就是在本方案的操作系统启动选择层中必须要有 TPM 芯片的协议接口供上层使用。在本方案中 TPM 还要对本方案的固件本身进行校验，所以 TPM 提供的安全服务应当优先被启动。

TPM 在本方案中作为一个基本硬件来处理，它应当在初始化前内存前就可以工作。在本方案设计中会在初始化层加入两个初始化模块： TPM 的设备协议

模块和TPM服务协议模块，来为初始化层本身和后续操作提供TPM芯片的抽象，并提供应用服务。 设计的结构如图 6.3：

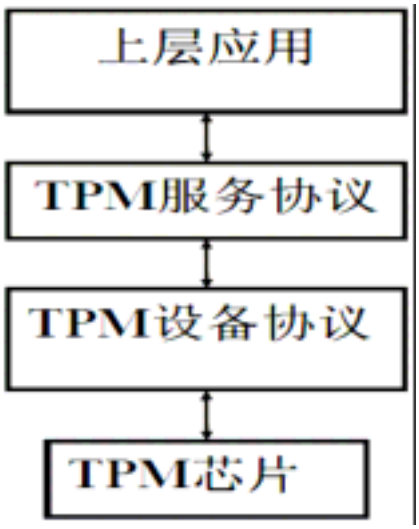

图 6.3 TPM 方案层次图

TPM 设备协议用来抽象各种不同的 TPM 芯片，这个模块提供 TPM 芯片内部各个寄存器、加密存储区域的访问方法。

TPM 服务协议用来为实现各种基于 TPM 的安全服务，比如支持国家认可的加、解密及数字签名算法等等。

TPM 设备协议和 TPM 服务协议提供的 TPM 支持将作为操作系统运行时的固件服务予以保留。

需要注意的是，本方案需要在整个固件生命周期内使用基于 TPM 的校验，包括一开机后对初始化层自身的校验，所以在初始化层运行前也要有基于 TPM 的对固件的校验。这部分功能应当由主板硬件来实现。

6.1.3 安全设计在操作系统安全启动的应用

图 6.4 演示了本方案对传统 BIOS 安全启动的设计。从图 6.4 中可以看出，从开机到操作系统引导的整个过程中，计算机系统的控制权先后经过了 TPM 初始化模块、传统 BIOS、MBR（主引导扇区）、BootSector（分区引导扇区）、BootBlock（分区引导块）、解锁模块、OS Loader（操作系统加载器）。其中每个模块中都有自己的校验部分，在图中是黑颜色部分。

这些模块中 TPM 初始化模块、MBR、BootSector、BootBlock 的校验是无需 TPM 的自检。

传统BIOS部分的校验是基于TPM芯片的校验，它一定要以某种协议和TPM进行交互后才能执行完成，并把控制权传递给 MBR。

解锁模块解锁是本方案为传统操作系统启动设计的安全步骤。本方案将 OS Loader 和操作系统都进行了加密，并且将密钥保存于 TPM 芯片。正常启动时本

方案解锁模块会在 BootBlock 后截获控制权，在和 TPM 交互后，完成后续部分的解锁操作，解锁后再加载 OS Loader 进行后续启动。解锁后的 OS Loader 中仍然需要无需 TPM 的自检，图中 OS Loader 黑颜色反映了这点。

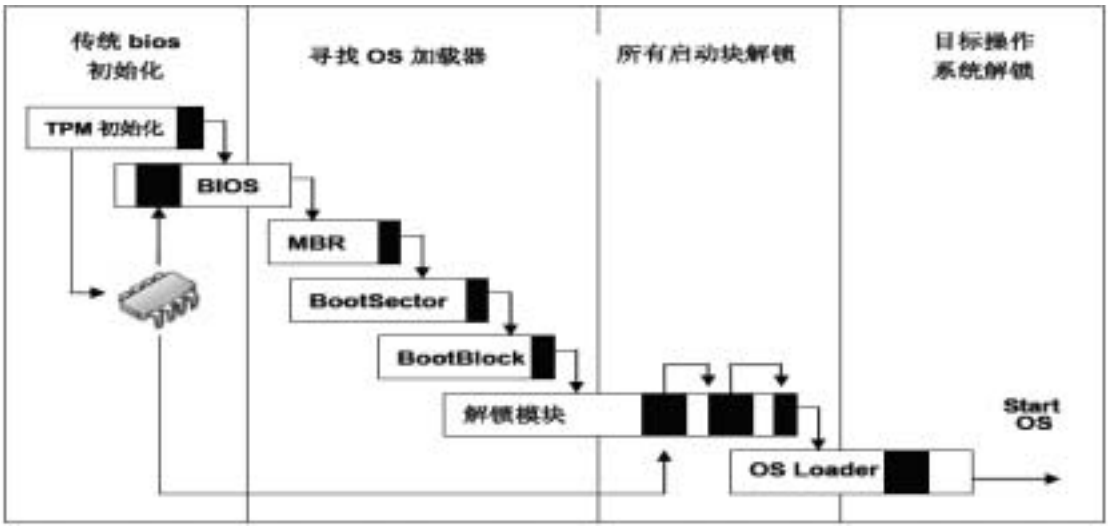

图 6.4 TPM 方案对传统 bios 安全启动的设计

6.1.4 一个基于安全策略并结合网络的应用

图 6.5 展示了实现本方案安全策略后的一个结合网络的应用。由于本方案是分层、模块化的设计，所以本方案的各个部分可以方便的拆分、组装。本方案的各个部分中，占据主体存储量的部分应该是驱动层驱动模块和 EFI 应用模块，这两个部分的存储方式对方案的应用方式影响很大，在图 6.5 所示的应用中展示了如何异地存放这两个部分。

首先，选择所有可以分离存放的模块，然后提取这些模块的定位描述，定位描述会包含这些模块的大小、存储地址、拼装方法，根据这些定位描述可以重新找到并拼装原有的模块，其中模块的拼装有标准的协议接口。

其次，将驱动层驱动模块的定位描述、EFI 应用模块的定位描述、固件其他没有分离的执行部分按一定的顺序存储于 TPM 的保护存储区。因为定位描述和固件其他执行部分都非常的小，所以可以共同存放于 TPM 的保护存储区。

然后，将驱动层驱动模块、EFI 应用模块按照定位描述中指定的路径存放于异地。在图中 EFI 应用模块存放于本地硬盘，驱动层的驱动模块存放于远端网络的服务器中。

在计算机正常启动过程中，固件会从 TPM 芯片中取出定位描述，然后通过两个部分的定位描述从硬盘和网络中找回异地存放的 EFI 应用模块和驱动层驱动模块，并在内存中拼装形成分离前的完整固件。

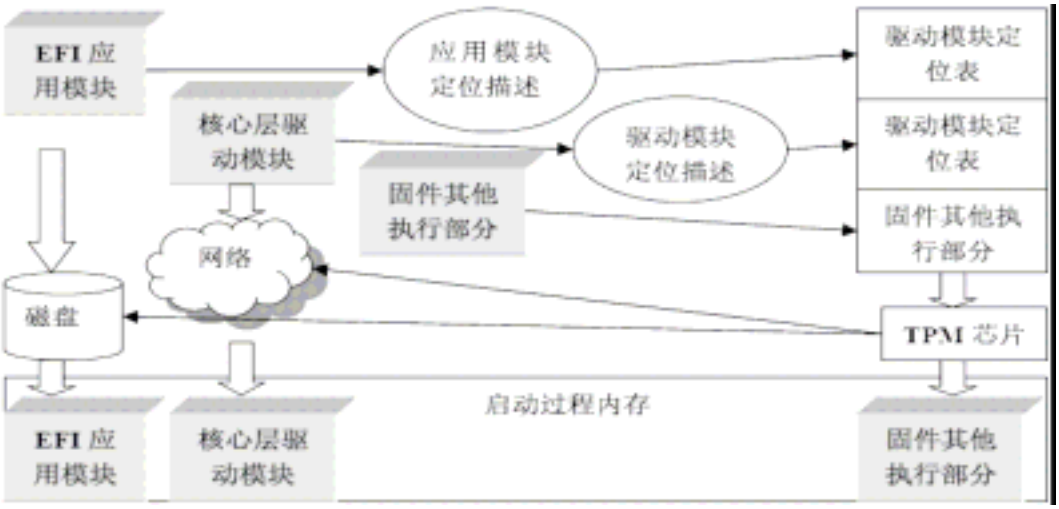

图 6.5 TPM 一个结合网络的应用

图 6.5 的应用方案有不少好处：

- 安全性很高而且实现又很简单。
- 极大解放固件存储空间。
- 固件升级很方便，可以在不同时间、不同地点针对某个异地存放的模块进行升级，用户在开机时不会觉察。

6.2 基于 USB 密钥设备的硬盘安全方案

硬盘作为当今数据存储最重要的媒介是各种计算机广泛采用的设备。在大部分的 PC、笔记本系统中，硬盘是唯一的数据存储设备。确保硬盘数据的安全对整个计算机系统的安全意义重大。据统计各个公司的机密数据外泄绝大部分是由于对自己硬盘的管理不善造成的。内部不法员工或者外人可以偷偷拷贝公司的硬盘数据或者干脆偷盗硬盘从而带走大量机密数据。在计算普及的今天，硬盘的使用环境也日益复杂，除了传统概念中的办公室、机房外，工业计算机上的硬盘还需要工作在厂房、建筑工地等安全环境较差的户外，在这种环境下因盗窃硬盘而造成的机密资料泄漏的机会更大。

防范此类的安全事故需要硬盘具有特别的安全机制：要能保证即使硬盘遗失，外人若没有权限也无法使用它，即不能读出或写入硬盘中的任何一个扇区；并且这种安全机制要能让硬盘的合法用户方便使用。

在现有的硬盘保护方案中，对硬盘的安全保护主要是针对用户误删除，往往通过在硬盘上划分保护区备份来实现。这种保护不能防止数据盗窃的问题。少部分针对数据盗窃而设计的保护方案大都是基于操作系统的对文件、文件夹或分区的加密、解密及隐藏等方法。这种方法操作起来很麻烦，一般用户很难坚持使用，

而且对于一个熟悉加密和文件系统的人来说破解这种方法的可能性比较大。一般的保护方案都是通过用户名和用户密码来鉴定用户身份，这种鉴定方法是基于字符串的，它泄漏的可能性也很大，不如一个实物载体更容易保存，比如银行自动取款机的磁卡，必须有磁卡实物和用户名及密码才能通过身份鉴定。

针对上述问题和需求，本文给出一种新的控制方案。

6.2.1 方案体系结构

本方案的基本思路是：用户在每次开机使用硬盘前，必须提供一个身份的证明来获取硬盘的使用权，即将锁定的硬盘解锁；用户在关机后，硬盘会自动回收使用权，即硬盘自动锁定。硬盘锁定以后，将拒绝对任何扇区的读写操作。用户身份的证明通过 USB 密钥设备和用户密码来实现。

方案分软件、硬件两个部分：一个硬件的 USB 密钥设备、一个软件的硬盘加锁解锁控制程序。其中硬盘加锁解锁控制程序又分成三个子部分：底层 USB 驱动模块、硬盘加锁解锁执行模块、权限信息鉴定模块。如图 6.6。

图 6.6 是展示了本方案的体系结构及各个部分的交互关系。用户提出加锁解锁请求后，“ 权限信息鉴定模块 ” 要求用户插入 “ USB 密钥设备 ” 并且输入用户名及密码，然后 “ 权限信息鉴定模块 ” 会调用 “ 底层 USB 驱动模块 ” 读出 “ USB 密钥设备 ” 中的权限信息，经过核对鉴定用户身份后，“ 权限信息鉴定模块 ” 会调用 “ 硬盘加锁解锁执行模块 ” 来完成硬盘的加锁、解锁。

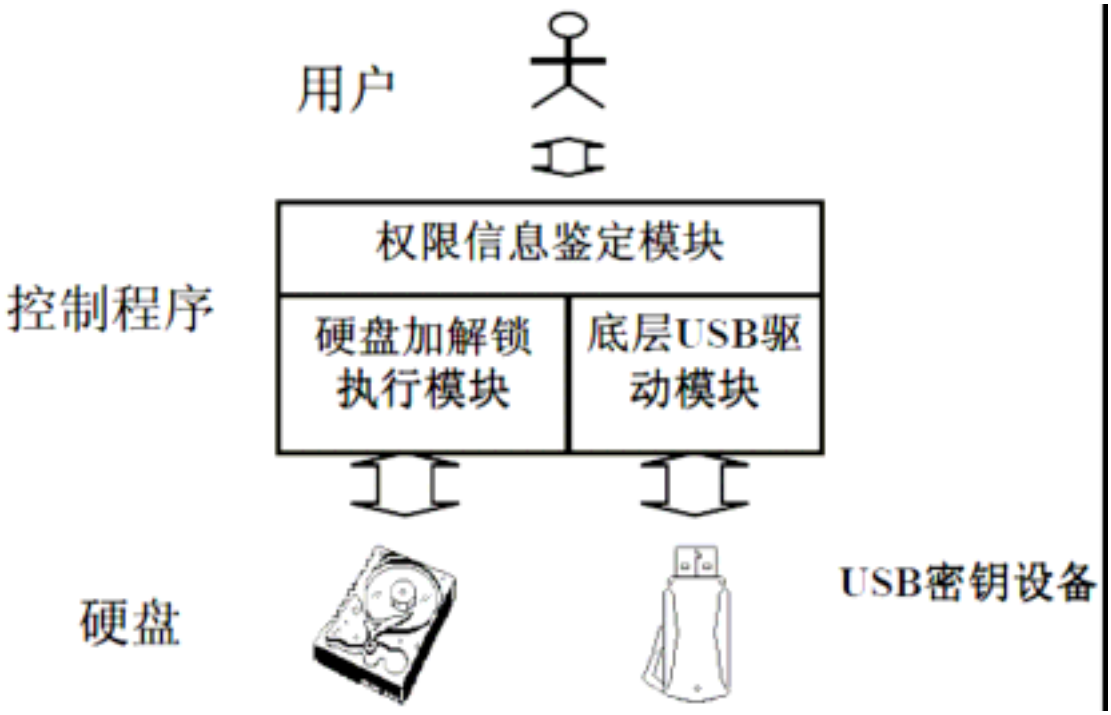

图 6.6 硬盘安全方案体系结构及各个部分交互关系

整个方案中控制程序是核心，由控制程序调度方案中的所有事务。权限信息

鉴定模块提供给用户的是一个图形化的输入界面，对于各种事务执行的状态、结果均以图形化形式显示。底层 USB 驱动模块要实现符合 USB1.1 规范所有主机控制器的驱动，这包括通用主机控制器接口（UHCI）和开放主机控制器接口（OHCI）两种不同的类型。在 USB 设备驱动方面，除了最基本的 USB 密钥设备外，底层 USB 驱动模块还可以增加对 USB 闪存盘的驱动支持。硬盘加锁解锁执行模块在具体实现时还可以加入硬盘低级格式化、全部扇区内容清零等功能，以满足更广泛的需求。

6.2.2 方案各部分描述

6.2.2.1 USB 密钥设备

本方案中的“ USB 密钥设备 ”是对外表现为人机接口类的 USB 设备，包括人机接口设备类的 USB 微控制器、可读写的非易失存储器。可读写的非易失存储器用来存储相关的权限信息，包括：硬盘合法用户名、每个用户对应的权限级别和密码、受保护硬盘的序列号、硬盘加锁解锁控制程序的校验和。USB 密钥设备中的所有数据均经过特定的算法加密，并且它的通讯接口是经过特殊设计的，不同于一般的人机接口类 USB 设备，可以保证即使 USB 密钥设备丢失，存储于其中的权限数据也不会外泄。将 USB 密钥设计成人机接口类 USB 设备可以保证其体积小巧，便于携带保管。USB 密钥设备本身和合法用户的用户名及密码构成了本方案用户身份鉴定的双因子。用户必须同时提供 USB 密钥及正确的用户名、密码才能通过“ 权限信息鉴定模块 ”的检查。

6.2.2.2 底层 USB 驱动模块

本方案中的“ 底层 USB 驱动模块 ”是控制程序中直接操控计算机 USB 硬件控制器来驱动 USB 密钥设备的部分。直接操控硬件控制器是指控制程序不依赖任何操作系统或第三方程序的支持，仅仅利用计算机主板 BIOS 的中断服务，直接访问 USB 控制器的寄存器来完成驱动 USB 密钥设备。之所以要直接控制器访问，首先是因为硬盘一旦加锁并重启后，其中的任何扇区均无法访问，硬盘中的操作系统或应用程序都无法加载，所以本方案的控制程序一定要脱离操作系统工作；其次如果是基于通用操作系统或第三方的程序，整个方案的安全性、可靠性都受制于第三方的系统，从原理上存在安全隐患。“ 底层 USB 驱动模块 ”还保证了整个控制程序可以脱离操作系统存放于任何启动设备上，包括硬盘、软盘、光盘、USB 闪存盘、主板 bios 的 ROM 芯片、计算机板卡 bios 的 ROM 芯片等。

图 6.7 是本方案控制程序和一般操作系统环境中应用程序的层次对比图。图的右半部分是在操作系统环境中应用程序访问 USB 密钥设备的层次结构；图的

左半部分是本方案的权限信息鉴定模块访问 USB 密钥设备的层次结构。

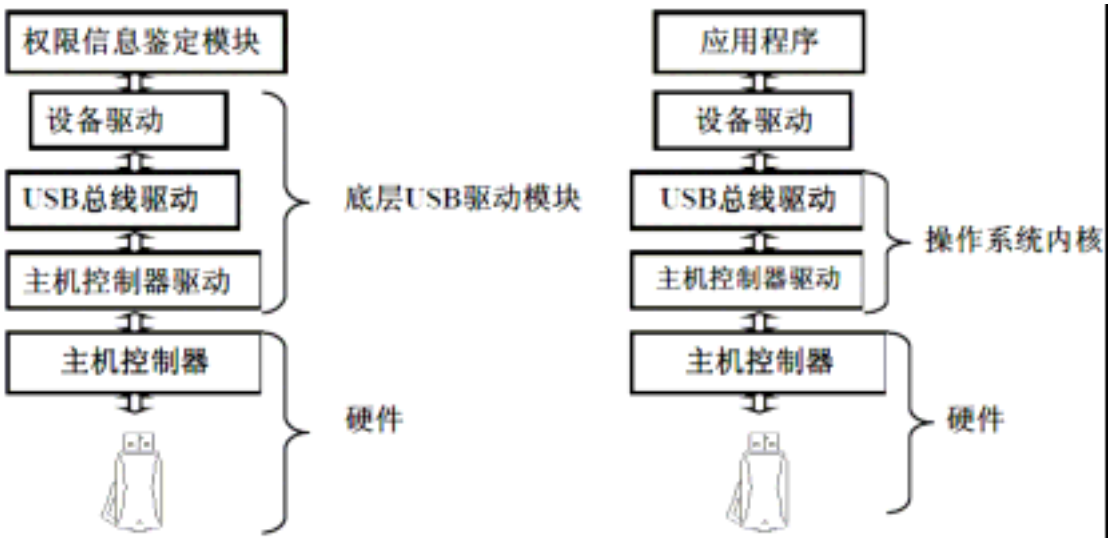

图 6.7 硬盘方案控制程序和一般应用程序层次对比图

一个 USB 设备若想让应用程序正常使用，需要驱动软件至少提供三个部分的支持：设备驱动、USB 总线驱动、主机控制器驱动。

- 设备驱动，是整个驱动软件最上层，是 USB 底层驱动的使用者，只有它知道设备中数据的含义，它使用 USB 总线驱动提供的功能接口（比如标准 WDM 中提供的各种标准接口函数）来读写数据，并处理读出或写入的数据，比如加密、解密，提供给更上层的程序使用。各种基本的功能函数的实现对设备驱动程序是透明的。
- USB 总线驱动，是 USB 总线的抽象层，它对设备驱动屏蔽了 USB 总线的操作细节，它负责自动检查插入新设备，分配、调度 USB 总线的各种资源，调用下层硬件的各种标准 USB 请求来合成上述设备驱动使用的功能函数。
- 主机控制器驱动，提供了主机控制器硬件的抽象。主机控制器驱动通过硬件的访问完成各种标准 USB 请求（比如 Get Descriptor、Set Address 等），各种标准 USB 请求会被 USB 总线驱动层调用。因为主机控制器都会集成根集线器，所以主机控制器驱动要提供对根集线器的支持。

在有操作系统的环境下，操作系统一般会提供主机控制器驱动、USB 总线驱动以及 USB HUB 设备驱动。操作系统对自己实现的基本功能进行了封装，形成了各种 API 函数。设备驱动程序是使用主机软件提供的 API 函数来读写数据，这种方式也是一般 USB 驱动的实现方法。

本方案的底层 USB 驱动模块是无需操作系统的，所以要使用 440BX 体系 BIOS 中的设备驱动、USB 总线驱动和主机控制器驱动这三个部分。从图中可以清楚看到虽然本方案控制程序是脱离操作系统工作的，但它实现的功能一个也不

少。

6.2.2.3 权限信息鉴定模块

本方案的权限级别采用树状结构，越靠近树根的用户权限越高。高级别权限的用户不但可以完成低级别用户的所有操作，还可以读取、修改低级别用户的用户名及密码。在这个树状的权限结构中，每个用户名都是唯一的，它唯一对应了一个 USB 密钥设备。本方案提供专门的工具根据用户名、权限级别来初始化 USB 密钥设备。一个低级别用户的 USB 密钥设备中会记录由他到树根路径上的所有高级别用户的用户名、密码。这样可以允许高级别的用户重新初始化这个 USB 密钥设备。本方案对硬盘加锁解锁的控制原则是：谁加锁谁才能解锁，高级用户可以解锁低级用户加锁的硬盘。在硬盘加锁时会在用户的 USB 密钥设备中同时记录硬盘的序列号，下次在解锁硬盘时，只有含序列号的 USB 密钥设备才能用来解锁那块硬盘。因为高级别的用户可以读出低级别用户名和密码，所以他能以低级别用户的身份来解锁硬盘。

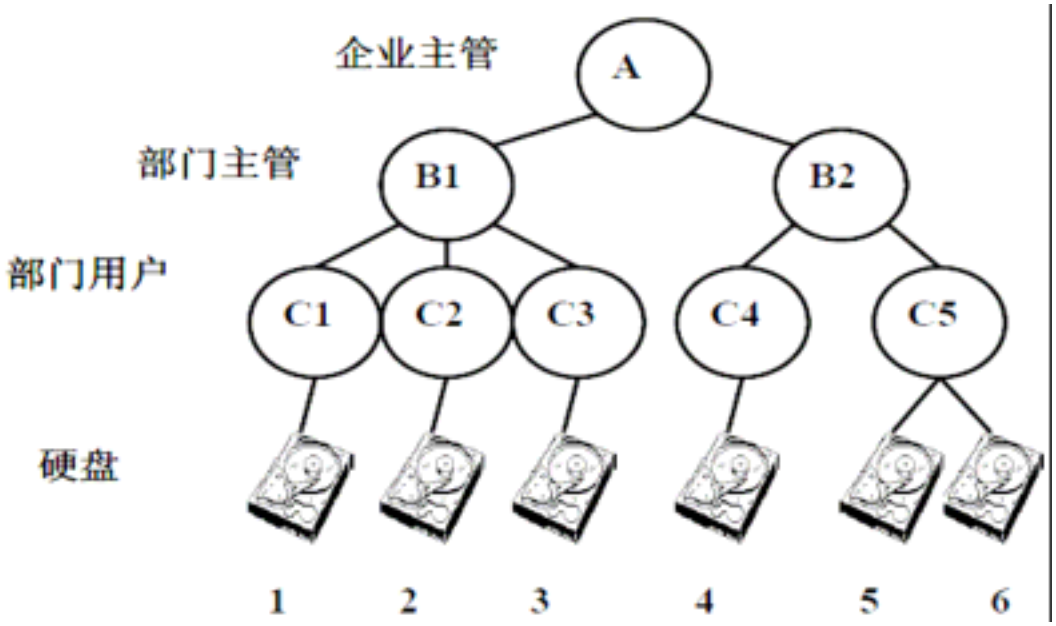

图 6.8 安全控制实施例

在图 6.8 所示的实施例中，A 是企业主管，B1 和 B2 是部门主管，C1～C5 是部门用户。这 8 个用户管理着 6 个硬盘，分别是硬盘 1～硬盘 6。在本实施例中 5 个硬盘均有同一个企业主管密码，企业主管密码作为权限较高的密码会对应于 Master password，但实际写入硬盘控制器的 Master password 不是企业主管设置的密码而是一个独立的密码。这样即使企业主管密码遗失软件也有办法挽救。部门主管密码和部门用户密码会对应于 User password，同样实际写入硬盘控制器的 User password 不是用户设置的密码而是独立密码。在图 5 所示的实施例中，部门用户 C1、C2、C3 的 USB 密钥设备中会分别记录他们的上级用户 B1、A，

部门用户 C4、C5 的 USB 密钥设备中会分别记录他们的上级用户 B2、A。所以 B1 和 A 可以解锁硬盘 1、2、3，B2 和 A 可以解锁硬盘 4、5、6。硬盘 1 一旦被 C1 加锁，部门用户 C2～C7 和部门主管 B2 都无法解锁。部门用户 C5 可以加锁两块硬盘：硬盘 5、6，硬盘 5、6 一旦被用户 C5 加锁，C1～C4 和 B1 都无法解锁。如果某块硬盘被 B1 加锁了，只有 B1 和 A 可以解锁。如果模块硬盘被 A 加锁，只有 A 可以解锁。

6.2.2.4 硬盘加锁解锁执行模块

本方案中的硬盘加锁分为硬加锁和软加锁两种。

方案中的硬加锁是基于硬盘安全指令集来实现的加锁。硬盘 ATA 接口规范中定义了安全模式特色指令集（Security Mode Feature Set），安全模式特色指令集中定义了若干指令专门用来实现硬盘访问权限控制。根据 ATA 接口规范的定义，硬盘用户可以使用密码对硬盘锁定。锁定后的硬盘在每次上电后都需要解锁，否则硬盘拒绝任何访问请求，无法读写硬盘中任何一个扇区。对硬盘解锁的指令是需要提供加锁密码的。这里解锁又分成两种模式：一种是永久解锁，一种是暂时解锁。永久解锁是指一旦硬盘解锁后，硬盘自动永久地恢复成正常状态，以后不再需要使用开锁指令即可正常访问硬盘。暂时解锁是指解锁后硬盘虽然可以在本次启动的后续时间正常使用，但一旦计算机关机或重启硬盘又会自动恢复锁定状态，在下一次开机时用户又必须使用解锁后才能使用硬盘。永久解锁和暂时解锁是指令本身提供的两种解锁模式，本方案的解锁操作正是利用了暂时解锁模式。

方案中的软加锁是对硬加锁的补充。软加锁是通过加密硬盘的 MBR（主引导记录）扇区来使得硬盘中的数据无法读取。软加锁在硬加锁的基础上更进一步地保护了数据。

ATA 安全模式特色指令集中的 SECURITY SET PASSWORD 等指令只能接受两个密码来加锁、解锁，分别是 Master Password 和 User Password。仅仅两个密码不能满足本方案中多级别权限管理的要求。本方案通过引进“独立密码”概念对密码的级别个数、用户个数做了扩展。“独立密码”实际是本方案自定义的两个长密码，与之对应的是用户自己设定的“用户密码”。两个“独立密码”是实际 SECURITY SET PASSWORD 指令设定的两密码，分别作为 Master Password 和 User Password。之所以称作“独立密码”是指这两个密码与用户密码以及这两个密码之间均不同，没有任何关联，是由本方案的实现者决定的。本方案在需要加锁解锁硬盘时会根据用户权限做一个转换，将最高级的“用户密码”替换成 Master Password 对应的“独立密码”，将其它级别的“用户密码”替换成 User Password 对应的“独立密码”，然后再使用 ATA 的安全模式特色指令进行设定。

这样的转换可以将本方案的权限级别做任意的扩展。同时这种方案也可以实现在所有用户都忘记密码的情况下，由方案制造商来解锁被锁定的硬盘。

6.2.3 本方案的具体实施例

图 6.9 是本控制方案的一个具体实施例的部署图。这个实施例采用 USB 闪存盘作为引导设备，方案的控制程序就放在 USB 闪存盘的只读区中。如图 6.9 所示，整张图有两个 USB 设备（USB 闪存盘和 USB 密钥设备）、一条 USB 总线，其中 USB 闪存盘和 USB 密钥设备通过插在计算机主板的 USB 插口上和 USB 总线相连。图 6.9 所示的 USB 闪存盘分两个部分，一个 USB 接口及闪存控制器、一个闪存存储区。闪存存储器又可分成普通的可读写区和用来存放控制程序的只读区。USB 闪存盘作为引导设备，当它启动计算机时它会加载只读区中的控制程序并将执行的控制权交给控制程序，由控制程序完成后续的引导工作。图 6.9 所示的 USB 密钥设备显示了一条实现 USB 密钥的例子。这个例子的密钥由 USB 收发器、USB 微控制器、串行 EEPROM 组成。USB 收发器是负责与 USB 总线通讯的 USB 接口芯片，USB 微控制器 EEPROM 的数据加密解密，EEPROM 是最终权限数据的载体。

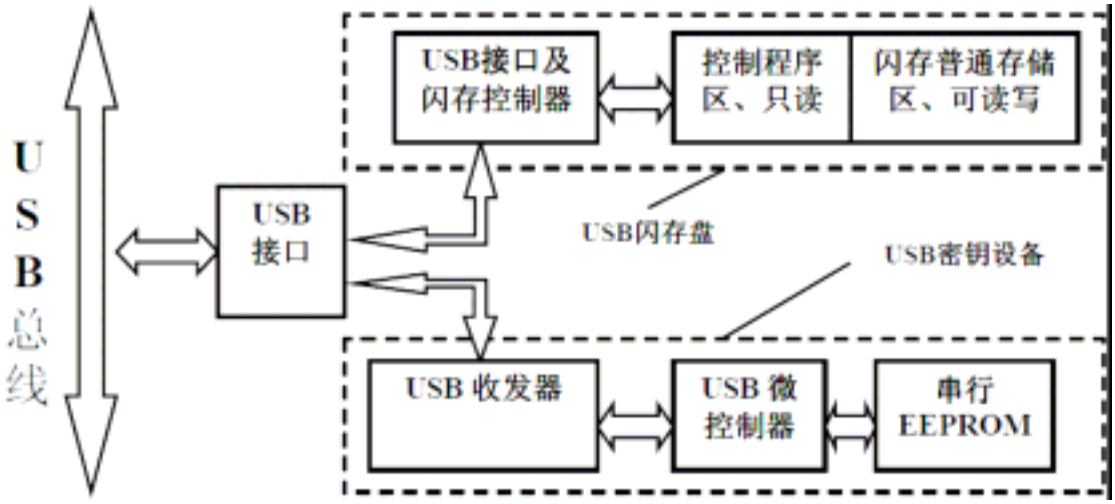

图 6.9 USB 硬盘锁实施例部署图

图 6.10 是本控制方案的另一个具体实施例的部署图，这张图展示了方案的一种合成设备实施例。在图 6.9 的实施例中用到了 USB 闪存盘和 USB 密钥两个 USB 设备，其实两个设备也可以合而为一，通过一个内置的 USB 集线器设备来连接闪存盘和密钥，再通过 USB 集线器和外部的 USB 总线连接。这种合而为一的实施例可以将方案所需的 USB 硬件集成在一个小巧的设备上，使用方便。

控制程序因为可以脱离操作系统，所以可以存放于任何启动设备上，包括硬盘、软盘、光盘、USB 闪存盘、主板 BIOS 的 ROM 芯片、计算机板卡 BIOS 的

ROM 芯片等。

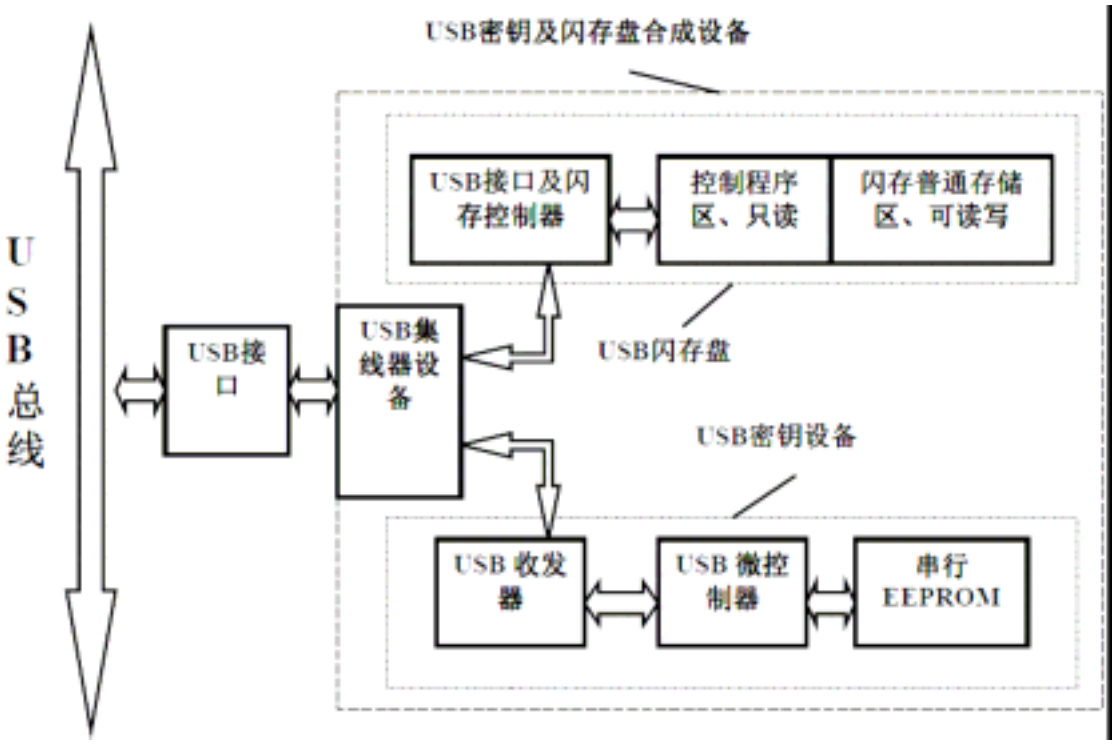

图 6.10 USB 硬盘锁实施例部署图

如上所述，本方案对硬盘进行加锁、解锁控制其效果如下：

本方案的加锁解锁操作根据 ATA 规范可以保证锁定的硬盘在解锁前不可能泄漏任何数据，即使硬盘被偷盗也不用担心数据被访问。本方案加锁解锁操作是快速、安全的原子操作，在瞬间即可完成对硬盘的保护，而且不涉及对硬盘有效数据的加密改写，不额外占用硬盘的空间，也不用担心加密时断电造成的数据严重破坏。本方案采用 USB 密钥设备和用户名、密码双因子来鉴定用户身份，相比单一用户名、密码的权限鉴定具备了更高级别的安全保证。本方案采用不依赖任何操作系统和第三方软件的设计基本上杜绝了破解的可能，这种设计可以使得整个方案不受其它软件质量好坏的影响。

在实际使用中，基于本方案的产品受到了广泛的欢迎。

总结

BIOS 作为 PC 体系中的一个关键组成，不论基于经济因素，安全因素，还是发展因素等，都有必要研制中国自己的 PC BIOS，并在此基础上，发展国产操作系统，即可构筑真正令人放心的国产系统软件平台。近年来，Intel 推出了 EFI 接口标准，目前正值传统 BIOS 向新一代 BIOS 规范的过渡阶段，是一个发展自主知识产权的国产 BIOS 技术与产品的极好时机。

借助 EFI 接口的推广时机，我尝试根据已有的各种资源，设计并实现一个符合 EFI 接口规范的 BIOS 驱动体系，并且在具体的 440BX 主板上检验自己的设计和实现。我具体完成的工作有：

(1) 分析 EFI 的接口要求，针对 Intel440BX 技术特点，设计具体的 EFI 驱动体系方案。

(2) 针对设计的 EFI 驱动实施方案，给出具体的 USB 驱动实现例子。

(3) 在实现的具体驱动之上，设计、实现具体的安全应用。

(4) 在驱动体系大框架下实现通用操作系统引导。

在近一年探索和研发的实践过程中，我感觉开发 BIOS 除了要掌握必要的软硬件知识外，还要克服 3 个难关：

1. 资料不完备：BIOS 主要是针对硬件的一系列操作，因此相关硬件技术文档是开发的必备资料。但一些硬件厂家出于其商业目的，往往不向公众提供资料或仅提供部分资料。只有其商业伙伴能获得完备的技术资料。[34]
2. 代码调试困难：BIOS 代码直接与硬件交互，为了消除操作系统对它的影响，它往往只能在裸机上进行调试，能用到的调试手段也近乎原始，所以最好引进可以控制 CPU 单步运行的 ITP 调试器。[34]
3. 国内支持很少：（1）在技术方面，BIOS 不被国内计算机的研究者所重视，因此在国内几乎没有在这一领域研究的群体。在遇到的问题和困惑时，想在国内获得帮助的可能性很小。在我们研究过程中给我们提供有价值帮助的往往都是一些国外的 BIOS 研发工程师。（2）在资金方面，BIOS 的研发是计算机核心组成的开发，在前期需要大量的资金投入，而这一点在我的科研当中感触很深。现在国内的计算机生产商在 BIOS 上基本没有投入，都靠购买国外 BIOS 开发商的产品，国内的 BIOS 研发没有形成商业的良性循环。[34]

本文给出了一个符合 EFI 接口规范的 BIOS 驱动体系，但以下问题还有待于进一步完善：

1. 我们所实现的 EFI 接口 BIOS 是建立在一款以 440BX 为核心的主板上的，对于这款主板上没有涉及的硬件在我们的实现中也没有体现。因此还有大量的硬件需要进行支持。
2. 从理论上讲，具有 CSM 模块的 EFI 接口 BIOS 应该可以满足现在所有的商业操作系统，但我们引导的 Windows 系列操作系统不能实现即插即用功能。究其原因是由于我没有提供 ACPI 的 BIOS 支持。ACPI 即电源管理是 BIOS 中重要的组成部分，也是 BIOS 和操作系统交互的重要途径，缺少 ACPI 的 BIOS 无法支持操作系统完成资源动态分配等即插即用功能。所以 ACPI 的部分是需要进一步研究并增添的部分。
3. Intel 为了推广 EFI 接口规范，实现了一个 EFI 接口的 Linux 加载器：elilo。这个 elilo 加载器不使用传统 BIOS 的中断服务，而仅仅使用 EFI 接口的协议来加载 Linux 操作系统。通过 elilo 可以来部分地检验一个 EFI 接口 BIOS 的提供的接口协议。因为我一个人精力有限，使用 elilo 来检验 EFI 接口 BIOS 的工作一直没有启动，但可以肯定研究并使用 elilo 一定会对 EFI 接口 BIOS 的研发带来帮助。

参考文献

[1] Extensible Firmware Interface Specification Version 1.10 December 1, 2002 Intel Corporation.

[2] IA-32 Intel® Architecture Software Developer's Manual Volume 1:Basic Architecture

[3] Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference

[4] Intel Architecture Software Developer's Manual Volume 3: System Programming Guide

[5] Phoenix FirstBIOS: The Right Choice for Client PCs Phoenix Technologies Ltd.

[6] Intel® 440BX AGPset:82443BX Host Bridge/Controller Datasheet

[7] Intel® 82371AB PCI-TO-ISA / IDE XCELERATOR (PIIX4) Datasheet

[8] Daniel P.Bovet & Macro Cesati,Understanding the Linux Kerne，中国电力出版社，2001 年 10 月

[9] Intel 82371AB PCI-TO-ISA / IDE XCELERATOR (PIIX4) Datasheet

[10]陈文钦，BIOS 研发技术剖析，清华大学出版社，2001 年 9 月

[11]李善平，刘文峰，Linux 内核 2.4 版源代码分析大全，机械工业出版社

[12]Jean J. Labrosse，嵌入式实时操作系统 uC/OS-2 (第 2 版)，北京航空航天大学出版社

[13]Adam Agnew, Adam Sulmicki, Ronald Minnich, ect，Flexibility in ROM: A Stackable Open Source BIOS，2003 USENIX Annual Technical Conference

[14]PCI Local Bus Specification Revision 2.1

[15]Frank van Gilluwe，PC 技术内幕（精英科技 译)，中国电力出版社

[16]毛德操，胡希明，Linux 内核源代码情景分析，浙江大学出版社

[17]Kris Kaspersky，黑客反汇编揭密，电子工业出版社

[18]H. Peter Anvin ,THE LINUX/I386 BOOT PROTOCOL ,Jan 01,2002

[19]David Mosberger, Stephane Eranian, IA-64 Linux kernel design and implementation, 清华大学出版社

[20]龚建伟，熊光明， Visual C++/Turbo C 串口通信编程实践，电子工业出版社

[21]曾繁泰， 冯保初，PCI 总线与多媒体计算机编著，电子工业出版社

[22]王涛，张伟良，冯重熙，嵌入式系统硬件抽象层的原理与实现，电子技术应用，2001 年第 10 期

[23]The Linux BootPrompt-HowTo Paul Gortmaker. v1.4, Mar 21, 2003 http://www.tldp.org/HOWTO/BootPrompt-HOWTO.html

[24]张益农, 黄文玲，嵌入式操作系统中硬件抽象层的描述，内蒙古工业大学学报，2002 年，第 4 期，第 21 卷

[25]蒋句平，Windows NT HAL 的结构与移植，计算机工程与设计，1997 年 2 月，第 18 卷 第 1 期

[26]郭静寰, 孟祥迪，Windows NT 硬件抽象层 HAL 功能分析，计算机应用，2002 年 7 月，第 22 卷 第 7 期

[27]姜波，陈英，可移植的 USB 协议栈实现原理与技术研究，计算机工程与应用，2003 年 2 月

[28]Microsoft Hardware Abstraction Layer for Microsoft ‘s Windows NT

[29]Operating System . Microsoft Corporation. U. S. A ,1996, Peter G. Viscarola. 实用技术:Windows NT 与 Windows 2000 设备驱动及开发. 北京:电子工业出版社,2000.

[30]David A. Solomon. Windows NT 技术内幕第二版. 北京:清华大学出版社,1999.

[31]William Stallings Operating Systems Internals and Design Principles, Prentice Hall 1998

[32]Bennet t S, Real-Time Computer Control，Prentice Hall，1988

[33]Felice Balarin, Hardw are2Softw are Co2Design of Embedded system s，Kluwer Academic publishers，1997

[34]胡藉, 面向下一代 PC 体系结构的主板 BIOS 研究与开发, 南京航空航天大学，硕士学位论文，2005

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

[36]石浚菁，《无需操作系统支持的 USB1.1 驱动实现及应用》，扬州大学学报，2005，8（8）：109~112

[37]石浚菁，《下一代 BIOS 的安全策略及应用》，南京航空航天大学第七届研究生学术会议，2005.10

[38]石浚菁，《一种基于 USB 密钥设备的硬盘加锁解锁控制方案》，计算机科学与实践，2005.8

致谢

首先要感谢我的导师李真老师和张有成老师给我们提供了优越的实验条件和学习条件，为我们能够更好的完成课题研究打下了坚实的基础。在两年多学习研究过程中，两位老师渊博的学识、踏实的工作作风、正直的人品使我受益匪浅。同时，他们为我创造了良好的学习、实践环境和相互学习、交流的机会，使得我能在一个良好的学术氛围中不断充实和提高自己。

尤其是张有成老师，在长达三年的科研时间里，张老师竭尽所能为我的项目提供资金和技术上的帮助，给我广阔的发挥空间。张老师敏锐的眼光、过人的胆识给我极大启迪，让我受用终身，衷心感谢张老师。

其次要感谢我的两个师兄：邱忠乐、胡藉和两个师弟：陈灯川、肖阳春。在早期，我协助两位师兄研究了 BIOS 的一些相关课题，为我后来的科研工作打下良好基础，他们锐意开拓的创新精神给我很大鼓舞。两个师弟在我后期的科研中给予了力所能及的帮助，感谢他们的支持。

特别要感谢我的女朋友彭星，因为她，我有了一段精彩的人生。在我科研遇到困难时，在我心情低落时，是她鼓励我坚持到底，走出了困境。她用自己的善良、顽皮、体贴驱赶了我原本生活中的寂寞、沉闷，是她让我开朗、乐观，让我沉着、坚强。

最后感谢我的父母，感谢他们在我读书期间对我无微不至的关心。在今后的工作和学习中，我将尽自己最大的努力，以优秀的表现来回报他们对我的爱护。

在学期间发表的学术论文

发表及录用的论文

1. 石浚菁，《无需操作系统支持的 USB1.1 驱动实现及应用》，扬州大学学报，2005，8（8）：109~112
2. 石浚菁，《下一代 BIOS 的安全策略及应用》，南京航空航天大学第七届研究生学术会议，2005.10
3. 石浚菁，《一种基于 USB 密钥设备的硬盘加锁解锁控制方案》，计算机科学与实践，2005.8

在学期间获奖情况

(1) 南京航空航天大学 2003-2004 学年优秀团员。

(2) 南京航空航天大学 2004-2005 学年优秀研究生。