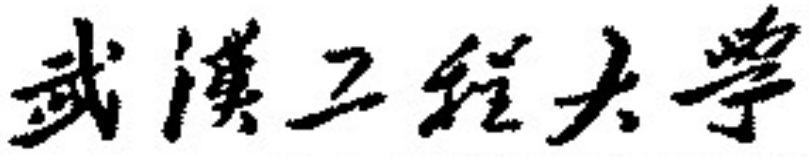

Wuhan Institute of Technology

硕士学位论文

基于 ARM9 全自动化生化分析仪器设备驱动的研究

----EFI 下 EXTx 文件系统驱动设计

学科专业：　模式识别与智能系统

研 究 生：　郭　旭

指导教师：　冉　全 副教授

培养单位：　计算机科学与工程学院

二 〇 一 三 年 五 月

分类号：TP368.1 学校代号：10490

学　号：201005013 秘　　级：公 开

武汉工程大学硕士学位论文

基于 ARM9 全自动化生化分析仪器设备驱动的研究
----EFI 下 EXTx 文件系统驱动设计

作者姓名：郭旭　　指导教师姓名、职称：冉全 副教授

申请学位类别：工学硕士　　学科专业名称：模式识别与智能系统

研究方向：嵌入式开发

论文提交日期：2013 年 5 月 27 日　　论文答辩日期：2013 年 5 月 25 日

学位授予单位：武汉工程大学　　学位授予日期：　年　月　日

答辩委员会主席：陈建勋

Based on ARM9 fully automated biochemical analysis equipment driven research

--EXTx file system driver design under EFI

A Thesis Submitted for the Degree of Master

Major : Pattern Recognition and Intelligent Systems

Candidate : Guo Xu

Supervisor: Associate Prof. Ran Quan

Wuhan Institute of Technology

Wuhan, Hubei 430074, P. R. China

May, 2013

独 创 性 声 明

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

学位论文作者签名：

20 年 月 日

学位论文版权使用授权书

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

保 密□ ， 在____年解密后适用本授权书。

本论文属于

不保密☑。

（请在以上方框内打“√”）

学位论文作者签名： 指导教师签名：

20 年 月 日 20 年 月 日

摘 要

生化分析仪是一种医疗机构进行临床诊断所的设备仪器。它可以快速准确的对人体体液的生化指标。这些常规生化指标可以帮助医生诊断疾病。生化分析仪在临床诊断和化学检验中具有重要作用。

目前的自动生化分析仪大多使用通用 PC 加单片机构成的上下位机控制方式，成本高，使用复杂。本文讨论的是将嵌入式技术应用于生化分析仪的研制当中，选用 32 位的 ARM9 处理器 S3C2410A 和嵌入 Linux 操作系统，搭建 ARM+Linux 的平台。

在自动生化分析仪软件设计方面，本文详细介绍了基于 ARM 引导装载程序 uefi 的 EXTx 文件系统驱动的设计，阐述了 EFI BIOS 的基本概念与总体架构，讨论了 EFI 运行的各个阶段，给出了 EXTx 文件系统的详细论述，并且说明了 EXTx 文件系统的实现原理组织架构。

最后，本文基于 EFI BIOS 的规范标准，设计和实现了 EFI 下 EXTx 文件系统驱动，给出了设计和实现的具体流程与步骤。

关键词：ARM9； EFI；EXTx 文件系统；驱动

ABSTRACT

Biochemical analyzer is a medical institution for clinical diagnosis of the equipment and instruments. It can be fast and accurate biochemical markers of human body fluids. These conventional biochemical indicators can help doctors diagnose disease. Biochemical analyzer has an important role in the clinical diagnosis and chemical test.

Most of the current automatic biochemical analyzer using a general-purpose PC and microcontroller upper and lower computer control mode, the high cost and complicated to use. This article discusses the development of embedded technology used in biochemical analyzer which use 32-bit ARM9 processor S3C2410A and embedded Linux operating system, build ARM + Linux platform.

Automatic biochemical analyzer software design, detailed design based on the the ARM Boot Loader the uefi EXTx file system driver, expounded the basic concepts and overall architecture of the EFI BIOS, EFI running, given EXTxFor a detailed discussion of the file system, and illustrates the organizational structure of the realization of the principle of EXTx file system.

Finally, based on EFI BIOS standards, designed and implemented the EFI EXTx file system drivers, given the design and implementation process step.

Keywords: ARM9;　EFI；　EXTx file system；Driver

目 录

摘 要.....I
ABSTRACT.....III
目 录.....V
第 1 章 绪 论.....1
1.1 研究历史与现状.....1
1.2 研究背景及其主要内容.....2
1.2.1 研究背景.....2
1.2.2 UEFI 的 ARM 引导系统.....3
1.2.3 UEFI 文件系统支持.....4
1.3 本文的主要工作.....5
第 2 章 EFI BIOS 基本架构.....7
2.1 传统 BIOS 方案的局限.....7
2.2 EFI 规范的优点和特性.....7
2.3 EFI 的基本架构图.....9
2.4 EFI 固件文件系统.....9
2.4.1 固件设备.....9
2.4.2 固件卷（FV）.....10
2.4.2 固件文件（FILE）.....11
2.4.3 固件块（SECTION）.....12
2.4.4 文件的操作.....13
2.4.5 分派器如何使用卷、文件和块.....13
2.4.6 基本格式.....13
2.4.7 FV 的管理.....14

2.4.8 固件卷的接口 14
2.5 EFI 运行的各个阶段概述 15
2.5.1 SEC 15
2.5.2 PEI 17
2.6.3 DXE 22
2.7.4 BDS 27
2.7.5 TSL 28
2.7.6 RT Phase 28
2.7.8 AL（After Life） 28
2.8 本章小结 28
第 3 章 EXTx 文件系统概念及重要数据结构实现 29
3.1 引 言 29
3.2 概述 29
3.2.1 总体结构 29
3.2.2 详细描述 31
3.3 Super block 31
3.3.1 SUPER BLOCK 数据结构 32
3.4 块组描述符表和块组描述符 34
3.4.1 块组描述符表和块组描述符数据结构 34
3.4.2 BLOCK BITMAP（区块对照表） 36
3.4.3 INODE BITMAP（INODE 对照表） 37
3.5 i-node 表与 i-node 37
3.5.1 I-NODE 数据结构 38

3.6 目录项 38
3.6.1 目录项数据结构 44
3.7 extent 39
第 4 章 EFI EXTx 文件系统设计与实现 43
4.1 系统结构 43
4.2 EXTx 文件系统核心模块 44
4.2 重要 I/O 协议接口卷 46
4.2.1 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL 47
4.2.2 EFI_DISK_IO_PROTOCOL 47
4.2.3 EFI Block I/O Protocol 49
4.3 协议接口的实现 50
第 5 章 EFI BIOS 文件系统驱动设计 53
5.1 EFI 驱动模块 53
5.1.2 EFI 设备驱动 54
5.1.3 总线驱动 56
5.2 EXTx 文件系统驱动设计实现 57
5.2.1 ExtDriverBinding 57
5.2.2 DriverBindingSupported 58
5.2.3 DriverBindingStart 59
5.2.4 ExtDriverBindingStop 61
第 6 章 总结与展望 63
6.1 总 结 63
6.2 展望 63
参考文献 65

攻读硕士学位期间已撰写和发表的论文.....69
致　谢.....71

第 1 章　绪 论

1.1　研究历史与现状

20 世纪 50 年代美国 Technicon 公司依据 Skeggs 教授的设计方案推出世界上第一台自动生化仪。此后，该类仪器发展迅速，实现了单通道、双通道、多通道仪器。 1965 年研制出分立式自动分析仪，工作原理与人工操作差不多，样品在相互分立的反应杯中进行化学反应。70 年代干化学试剂的反射光学分析仪发展迅速，它不需要液态试剂，在准确性、精密度、以及分析速度方面都得到了提高，方便了急诊和高科技应用。

80 年代初在生化分析仪中普遍存在样品和试剂之间交叉污染，美国 TECHNICON 公司为解决这个问题，设计生产了任选式测定方式的仪器，把自动生化分析仪的开发水平推到更高的水平。此后有很多任选式的仪器推出。90 年代后，自动分析的发展方向主要是完善仪器的各种功能，尤其是计算机技术的广泛应用，使仪器在分析的准确性、精密度、效率、消耗等其他方面都更好地满足用户的需求。

目前，国际上生化分析仪正朝着专业化、通用化、自动化、微量化、组合式和微机化的方向发展。

专业化是指专门用于某一种生化指标测定的仪器，其具有体积小、便于携带和使用方便等特点，尤其适合在家庭中使用。

通用化是指对于大型医疗机构来说，经常需要检测非常多的各种样本，需要测试的项目数目巨大，因此需要大而全的生化分析仪。

自动化是指目前生化分析仪需要人工参与的越来越少，从而简化了检验室的操作流程。

微量化是现代生化分析仪发展的又一特点，包括样品用量微量化和试剂用量微量化，最少的用量可达微升级别，即减轻了病人的痛苦，又降低了医院的成本。

组合式是将整台生化分析仪设计成相互独立的模块，每个模块实现各自不同的功能。该方式避免了由于仪器过于复杂造成的维修保养不便

的缺点，另一方面同一公司生产的不同型号的分析仪之间也可以实现组合，即系列组合。这种设计使各种型号不但可以用于各种特定需要，而且能方便组合，以发挥更大功效。

微机化是分析仪未来的必然发展方向。随着计算机技术的发展，目前生产的各种生化分析都带有内部计算机系统，并可通过标准接口与外部设备互连。微机不但能用于整个操作过程的控制，还可以提供良好的人机界面，方便各种操作和数据的处理。此外，还可以通过软件的不断升级来完善和扩展仪器的功能。

1.2 研究背景及其主要内容

1.2.1 研究背景

目前，国内大中型医院大多使用的是价格昂贵的进口全自动生化分析仪，但对于中小型医院难以负担的起。因此我国的医疗机构，特别是中小医疗机构对国产全自动化生化分析仪的需求越来越迫切。随着社会的发展，中小型医院医疗质量也不断地提高对医疗器械的要求也逐步提高，简单、易用、灵活、价格适合的国产全自动生化分析仪最能适应中小医院的需求。

因此开发出低成本、小体积、高性能的全自动生化分析仪，满足了我国这方面的现实需求，缩小了与先进仪器之间的差距，具有较高的社会效益和经济效益。

目前的自动生化分析仪大多使用通用 PC 加单片机构成的上下位机控制方式，成本高，使用复杂。本文讨论的是将嵌入式技术应用于生化分析仪的研制当中，选用 32 位的 ARM9 处理器 S3C2410A 和嵌入 Linux 操作系统，搭建 ARM+Linux 的平台。

全自动生化分析仪的基本结构和主要部件可以分为硬件部分和软件部分。硬件部分主要分为加样系统、恒温系统、搅拌系统、操作系统等。软件部分包括系统引导程序和上位机操作系统，包含数据的存储、调度、管理、报表打印、通信以及人机交互等几大模块。

要实现生化分析仪的程序控制和自动化，必须要求计算机软件和硬件有很好的配合。软件系统的开发应用和不断完善是提高生化分析仪性能的主要手段之一，其中驱动程序在主板初始化和系统启动后都担负着重要作用。

这些驱动主要是指 linux 系统下的设备驱动，既包括步进电机驱动、LCD 驱动、A/D 驱动等具体工作设备的驱动，也包括引导程序里的文件和总线类设备驱动，如 GPIO、I2C、PCI、Serial、Usb、fs 等，其中 fs（即文件系统）包括了 ext2、fat、yaffs2 等嵌入式文件系统的支持，只有完成了 fs 设备文件驱动才能引导启动 linux 系统。

在 CPU 方面，ARM 公司 32 位 RISC 处理器以其低成本、低功耗、高性能的特点而得到广泛使用，基于 ARM 微处理器构架的嵌入式操作系统也获得了快速发展。其中，嵌入式 Linux 因其源码开放、内核稳定高效及支持多种体系结构等优点，在 ARM 处理器构成的嵌入式系统中占据主流地位。

1.2.2　UEFI 的 ARM 引导系统

UEFI，全称“统一的可扩展固件接口”(Unified Extensible Firmware Interface)， 是一种详细描述全新类型接口的标准。这种接口用于操作系统自动从预启动的操作环境，加载到一种操作系统上，从而使开机程序化繁为简，节省时间。

ARM 公司、苹果、惠普和微软的专家们一起确定了 UEFI 的 ARM 绑定。这样，使用 UEFI 就可以最大限度地实现不同设计间的代码重用，包括那些使用不同的处理器架构的设计。

然而，ARM 系统没有一个预引导固件的标准，使得每个设计都有自己独特的与所引导的操作系统紧密结合的模式。这种传统的方法意味着固件开发者必须保持完全不同的代码库，即使系统可能使用的外围设备(网络，SATA 接口，USB 控制器等)和整个设计功能集是相同的。因此，传统的 ARM 软件设计者仍然依赖诸如 UBoot，Redboot 类启动程序，或者专有的软件启动包。

UEFI 对 ARM 系统的预引导固件是一个创新。UEFI 负责定义操作系统和系统固件之间、固件驱动程序和系统固件之间的接口，而 UEFI 的平台初始化(PI)负责定义固件到芯片之间和固件内部的接口。

UEFI 是与处理器架构无关的，基于 UEFI 构成的 EFI BIOS 拥有很好的可扩展性。EFI BIOS 的模块化设计将其在逻辑上被分成 OS 软件管理与硬件控制两部分，OS 软件管理是一个可编程的开放接口，EFI BIOS 通过这个接口，实现更多的扩展功能。硬件控制部分则是全部 EFI BIOS 都有的共同部分。

此外，EFI BIOS 中还实现了对 TCP/IP 协议的支持，这使得 EFI BIOS 有了使用网络的功能，可以通过网络直接访问 EFI 的方式进行可靠的诊断，并且可以设置各硬件子系统参数的配置或更换驱动程序等使系统恢复到正常状态。

本文的研究内容即是基于 EFI BIOS 的 ARM 引导系统。

1.2.3 UEFI 文件系统支持

ARM 系统在上电或复位时从地址 0x00000000 处开始执行。Nor Flash 将被映射到这个地址，通常在这个地址开始的是系统的引导装载程序 UEFI。UEFI 是系统加电之后在操作系统内核运行之前主板上运行的第一段程序代码。这段代码将系统的软硬件环境初始化到一个合适的状态，主要任务是初始化硬件设备、建立内存空间的映射图、将内核映象和文件系统从 Flash 拷贝到 RAM 中，然后跳转到 Linux 内核的入口地址将控制权交给操作系统启动操作系统。

在 UEFI 中实现 uboot 的两种启动模式：启动加载模式和下载模式。启动模式是 UEFI 从目标机的 Flash 上将操作系统加载到 RAM 中运行，这是正常工作模式。下载模式是 UEFI 实现了与宿主机的 xmodem 或 tftp 等通信协议，从而目标机可通过串口连接或网络连接等通信手段从宿主机下载内核映像、根文件系统映像和应用程序等，工作于这种模式下的 UEFI 向它的终端用户提供一个简单的命令行接口。

UEFI 是严重地依赖于硬件而实现的，UEFI 分为 sec、pei、dxe、runtime

几个阶段。依赖于 CPU 体系结构的代码，通常都放在 sec 和 pei 中。

sec 阶段的开始部分是需要用汇编语言实现的，因为这时候内存还没有初始化，没有分配堆栈，C 语言程序还不能运行。到了 dxe 阶段后因内存可以使用，所以可以使用 C 语言，以实现复杂的功能，且代码会具有更好的可读性和可移植性。

pei 的内容包括初始化时钟和 RAM，为加载 dxe 准备 RAM 空间并将 dxe 拷贝到 RAM 中，并设置好堆栈。

dxe 负责主板的硬件设备的初始化，包括块设备驱动、DMA 驱动、GPIO 驱动、I2C 驱动、PCI 驱动、fs 驱动等都在这个阶段完成，其中 fs 的驱动包括 cramfs、ext2、fat、fdos 等设备文件系统的支持，只有完成了对 ext 文件系统的支持才能最终完成主板的环境配置并启动系统。

除此之外 dxe 阶段还要检测系统内存映射，将内核映象和根文件系统映象从 flash 复制到 RAM 中、为内核设置启动参数、调用内核。将内核映象解压后复制到 SDRAM 0x30008000 开始的地方，将根文件系统映象解压后拷贝到 0x30800000 开始的地方。在将内核映像和根文件系统映像拷贝到 RAM 空间中后，就可以准备启动 Linux 内核了。但是在调用内核之前，应设置 Linux 内核的启动参数。

本文详细讨论和研究 fs 中对 ext2、ext3、ext4 设备文件系统的支持。

1.3　本文的主要工作

本论文具体各章节的内容如下:

第一章：文章绪论，给出了论文的选题背景意义及文章的组织结构。

第二章：详细的介绍了 EFI 的基本架构，EFI BIOS 各个阶段的工作及运行流程,对许多重要概念作了具体的解释。

第三章：介绍 EXTx 文件系统并分析其组织结构。

第四章：论述 EFI EXTx 文件系统驱动设计。

第五章: 给出 EFI BIOS EXTx 文件系统驱动设计。

第 2 章 EFI BIOS 基本架构

EFI 描述了对平台的一个可编程的接口，这个平台包括主板，芯片组，cpu 和其他组成部分。EFI 允许操作系统预处理，这里预处理 agents 可以是 OSloaders、诊断程序、和一些系统的应用软件执行所需要的其它应用程序，包括 EFI 驱动和 EFI 应用程序。EFI 为驱动和应用程序提供了一个非常清晰的接口规范，在这一章中我们重点指出这个接口的一些架构轮廓，这个架构轮廓包括了在 EFI 规范中描述的一组对象和接口。

2.1 传统 BIOS 方案的局限

传统 BIOS 规范从一开始就固定不变，它已经不能适用于发展迅速的现代计算机体系架构。当新的平台功能或者新硬件出现，固件公司都必须设计开发新的解决方案，同时操作系统的开发商需要做出相应修改以配合新的启动代码，这是个相当复杂耗时的过程，并且还需要投入非常大的资源。传统的 BIOS 规范的缺点有:

(1)16 位的实模式下 BIOS 的寻址空间只有 1K 字节。

(2)只能由汇编语言开发，开发效率低下。

(3) BIOS 的界面可操控性和易用性较差。

(5)没有统一的 BIOS 标准规范，开发厂商设计生产的产品相互之间兼容性很差。

2.2 EFI 规范的优点和特性

EFI 标准规范为 BIOS 开发提供了统一的规范，使得固件开发过程中的众多环节得以简化。该标准规范的主要特点如下:

(1) 一致的平台环境：EFI 规范对于所有的 IA 架构系统固件的平台特性及平台在启动过程中给操作系统所提供的功能给出了一致的定义。

(2) 为 OS 提供了一致的固件接口：规范定义了可以供 OS 调用的硬件平台抽象接口。通过对抽象接口的调用，使得操作系统引导程序

(OSloader)的实现与接口之下的硬件平台和固件无关，这组固件抽象接口位于 OS loader 和平台固件之间，这样可保证在符合接口定义的条件下对 OS loader 和平台固件更改而互不影响。

(3) 合理的设备抽象：BIOS 接口需 OS loader 知道特定硬件设备的工作细节。而在 EFI 规范下，OS loader 的开发人员利用 EFI 所提供的设备抽象接口编程就不需要关注设备的工作细节。

(4) Option ROMs 的抽象：EFI 规范定义了硬件平台的功能接口，包括标准总线类型 ,例如 PCI、USB、SCSI 等等。EFI 驱动模型(EFI Driver Model)很好的解决了对未来要使用的总线类型的支持。和传统 BIOS 用中断来管理硬件不同，EFI 驱动模型是用操作系统中的驱动程序的形式管理硬件资源。EFI 驱动模型的解决了现存于“PC-AT” Option ROM 中的兼容性问题。驱动使用抽象接口，这样使得设备驱动和总线驱动的构建减少了与接口下面的硬件平台以及固件的相关性。

(5) 可共享架构的系统分区：扩展硬件的平台功能和添加新的设备经常要求相关软件的支持，通常需要更新固件平台的部分代码以支持底层硬件。常用的解决方案是在硬件平台生产的时候嵌入这些代码。而作为对之前的解决方案的补充，该规范规定了非易失性存储平台相关支持代码的方式。在 EFI 规范中详细的描述了相关的工作机制，使得操作系统厂商、OEM 厂商、固件开发者、第三方可以在安全的共享空间内添加平台功能。与以前 IA 架构所使用的传统 BIOS 相比，它主要有以下一些特点：

(1)不同种类平台使用同样的代码基础。

(2)基于 C 语言的固件开发。

(3)支持未来平台上的新特性新功能的灵活设计。

(4)模块化的组件设计。

2.3 EFI 的基本架构图

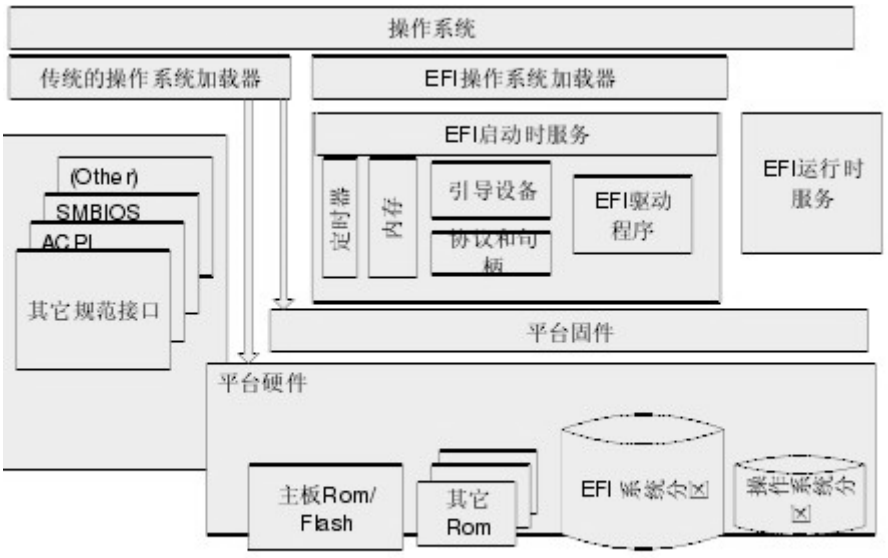

图 2-1 EFI 基本架构

图 2-1 表明了 EFI 相关组件和平台硬件之间的关系。还展示了系统在启动操作系统时 EFI 架构的各个组件之间的交互过程，平台固件从 EFI 系统分区中找到 OS loader 镜像，而 EFI 不仅支持从大容量存储介质启动 OS loader，同时也支持从网络启动。OS loader 启动后， OS loader 运行到操作系统的启动。在此期间，OS loader 会调用 EFI 的启动时服务(Boot Services)和相关接口，初始化与平台固件相关的各个组件，同时初始化 OS 中对平台进行管理的模块。EFI 的运行时服务(Runtime Services)能在 OS 启动后继续使用。

2.4 EFI 固件文件系统

2.4.1 固件设备

固件设备(FD)是指任何可以存储固件的物理设备或设备的集合。通常多个设备的集合，只能被当做为单个设备来使用。闪存芯片是常见的固件设备，其他的一些非易于性的存储设备也常被用作固件设备。

现在常用的是用闪存芯片存储固件卷。另外闪存需要被分成大小不

同的块。EFI 的固件文件系统对固件设备有如下要求，它可以以块或扇区为单位擦除。在擦除后，全部置 0 或者全部置 1。

固件设备的一些特性：

(1)当正在写入或擦除时，禁止其他操作。

(2)完成写入和擦除动时所用的时间比读要多。

(3)可以锁定当前的读写状态到系统下次重新启动。

(4)可以完成读写使能或关闭对整个设备或者是单独的块。

(5)要从非擦除状态变成擦除状态，只有通过对整个块进行擦除。

(6)可以按位写入。

2.4.2 固件卷（FV）

固件卷(FV)是指在单个固件设备上划分出的连续的一个部分并格式后的卷。这里明确规定一个 FV 不能来自于多的 FD (但是一个 FD 确可以来自于多个设备)。和大部分操作系统区别的是，在固件设备中的全部固件卷(FV)可以仅仅占有此固件设备的有效空间中的一部分。FVs 在 FD 中的位置可通过在程序编译的时候给出然后由驱动来描述，或者说，它并不必要固件卷(FV)是可被发现的，这位置也许只是事先知道并通过其它驱动填充到此驱动程序的堆栈中来。就像可以在 FV0 里的驱动去描述其他的 FVs。从定义明确出 FV0 是引导向量的目标。在固件中找出 FVs(通常是由驱动来检测)和建立对基本操作所需要的运行环境都是很重要的工作。

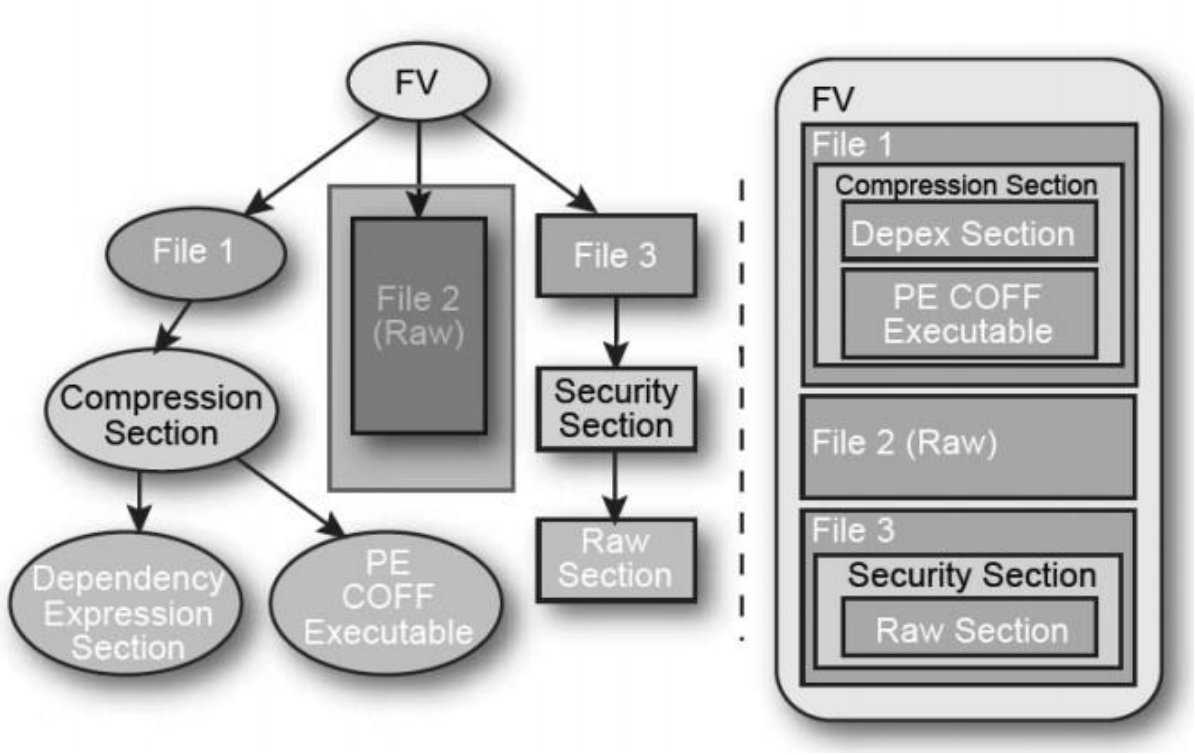

图 2-2 固件卷组织结构

2.4.2 固件文件（FILE）

固件文件(File)是指存储在固件卷里并命名了的信息的集合。这里的文件名在架构中对应的是 GUIDs(GUID 是全球唯一标识符，用来保证文件名的唯一性)。文件名(GUIDs)在一个固件卷(FV)里要确保是唯一的，但是在不同的 FVs 里会有一样的文件名的文件。这样，我们就需要使用文件时要保证 FV 和文件名配对。这与许多的操作系统里的文件系统规则是差不多的。相同后缀名命名的文件代表相同的功能，但是在 framework 中对这个的要求更加严格，例如，DXE 调度器只能派遣相对应类型的文件。FVs 只提供一级目录供文件使用，根本没有子目录存在。但是，文件可能将 FVs 包含在其中，跟子目录的作用是一致的。这种机制也促进了一个更好的方法来构成由大量文件组成单元的封装模块。对于文件有检测(固件卷)FV、打开文件、读取文件、关闭文件等简单的操作。

2.4.3 固件块（SECTION）

块(section):一个文件的子区域。块是有类型的但没有命名的。有两种类型的块：容器区域块和叶形区域块。容器器区域快(也叫做封装区域快)。容器包含另外的区域块而叶节点不包含。

2.4.3.1 封装 section

一般来说压缩的卷(Volumes)和文件是不能在 FFS 里直接存放(压缩)的，但它们是能够在压缩块里面存储的。已经压缩的 vlumes 在一个压缩块（section）。压缩块(的 section 中)描述了怎样去寻找已有(一个被定义)的解压数据接口的协议。将 DXE 的卷放到一起，一种通用做法就是，用压缩块将它封装起来(一种常见的方法就是把用于 DXE 的卷与压缩的 section 放到一起)，储存到一个卷(Volume)里面。另外一种是被称作 GUIDed 块(section)，这种块会通过某种协议与另外的块相联合。这个块的头部包含一个请求明确的 GUID。在使用这个 section 之前就需要定义好带有定义接口的协议(就比如一个 GUIDed section 抽取的协议)和以请求明确的 GUID 作为名字的 section。一旦对应的的 GUID 的 section 被使用，这个接口的协议就会被调用。例如 GUIDed 的类型可能用于通过内部数字签名的方式来对块进行确认。DXE dispatcher 会根据这个 GUID 来查找相关的可执行程序，将其放到调度队列当中。

2.4.3.2 普通的叶子块类型

有超过 10 种 leaf section 类型。它们可分为几类:

(1)为一个可支持的执行格式:PE32+映像、与位置无关的代码(PIC)映像，以及简洁的执行（TE)。PIC 代码是一种可重新部署代码，能在任何地址执行(受对齐的限制)。以及为兼容模块(CSM)的可执行程序保留的专用的 section 类型。

(2)DXE 依赖性表达以及 PEI 依赖性表达的块。

(3)为了用户可以更方便管理文件的一种版本和文件名块。

(4)固件卷映像是一种能够把卷存储在文件中的块类型。

(5)自由形态的子类型 GUIDed 和裸的块和文件中的意义相同。

2.4.4 文件的操作

除文件外和 section 定位功能已经被论述之外，Firmware volume 协议还提供了希望通过文件接口来得到的服务:获得属性(读取一个卷的当前状态)，设置属性，读取文件，以及写文件。每一个协议的实例对应与一个固件卷。标准协议服务可以用于扫描固件卷。

2.4.5 分派器如何使用卷、文件和块

DXE 分派器用于去寻找和分派驱动。DXE 分派器通过固件文件基础架构去完成这些操作。经过这个过程，它提供了关于如何使用 FVs、文件和块的一种更为复杂的范例。系统中的每一个 FV 都是与一个协议来相对应的。这样一来，分派器就通过等待一种新版本的协议(即通过 EFI 的 RequestProtocolNotify 服务)来寻找新卷。在 PEI 中被发现的卷是通过 HOB 传递给分派器的，在分派器在初始化的时候处理。当分派器收到一个新卷出现的通知时，它便利用 Frimwarevolume 的协议来查找 DXE 驱动类型的文件，一旦找到了这些文件，继续搜索依赖性关系块，并根据这些块的信息为这个驱动列出遵循的分派顺序。分派器依据依赖表达式去读取 PE/COFF 映像块，分配这个驱动执行。

2.4.6 基本格式

Framework 的目的是支持任意数量的文件系统，但是开机的时候它还是要依赖于一个文件系统，才可以找到用来告知它怎样去访问其他文件系统的那个模块。就是所谓的 Framework 文件系统格式，也就是我们所说的 FFS。

固件卷是指包含一些文件的一些线性数据块。文件由头、数据体、

和一个尾构成。固件卷从头部在最低地址开始，这个头标明了一个 GUID、卷大小以及其他相关数据的属性。卷的头部从最低的地址开始，这样它的位置可以被直接明确地找到。

卷中的文件连续地存储在从卷头部结的位置。文件头包含名字(GUIU)、大小、类型、属性、完整性校验和状态。这个大小指的是文件的大小。

状态信息将被用作更新操作的一部分。更新操作就是删除和写操作。状态信息就是保证更新时的过程的原子性。在更新中的任何状态下，有且只有一个文件是有效的。此实现假定在不擦除整个区块的情况下，媒体能够轻松地从一个值变成另外的值(如从 1 到 0)，对现在的元件来说，这是对的。FD 擦除的状态通常被认为是 FD 的极化。一个 FD 的极化信息被存储在它的头部。

2.4.7 FV 的管理

文件开始于在卷中从最低地址处，然后依次连接(允许对齐而产生空隙)，直到卷的的最顶部。后面的的文件会写到之前刚写入的文件位置的上面。文件的删除并不会马上引起上面的文件位置发生从上到下的变动。但是文件会头部的删除标志位从原来的擦除状态改变为非擦除状态。

当空间不能满足能响应写的请求，就要使用合并的操作。实际中并没有一种固定的唯一的方式去合并 FVs。在我们对于机制做选择的时候，往往要考虑可用资源、容错能力要求和其它因素。一个相当简单的方法通过复制没有删除的文件到内存的缓冲器中，建立一个新的 FV 的映像。擦除原有的 FV，然后写入新的 FV 映像。虽然说这不是一种容错解决方案，但对很多应用是可以接受的。另外一种机制是保存两个固件卷，这是一种容错的方式，但是成本太高了。FFS 状态的数据可以用来实现安全的更新或者刷写的机制。

2.4.8 固件卷的接口

FVB 接口定义了 7 种接口：

(1)属性：GetAttributes 和 SetAttributes 函数会对 FV 的副本执行相似的动作。所谈到的属性包括读写保护、锁定和对齐。Getblocksize 会获得卷中区块的大小。如果存在，GetPhysicalAddrass 准许驱动去获取卷的物理地址。

(2)I/O 操作:设备通过读和写操作这样的基本功能来完成他们的工作。

(3)擦除:合并操作的一个部分就是使用 EraseBlocks 擦除区块。

2.5 EFI 运行的各个阶段概述

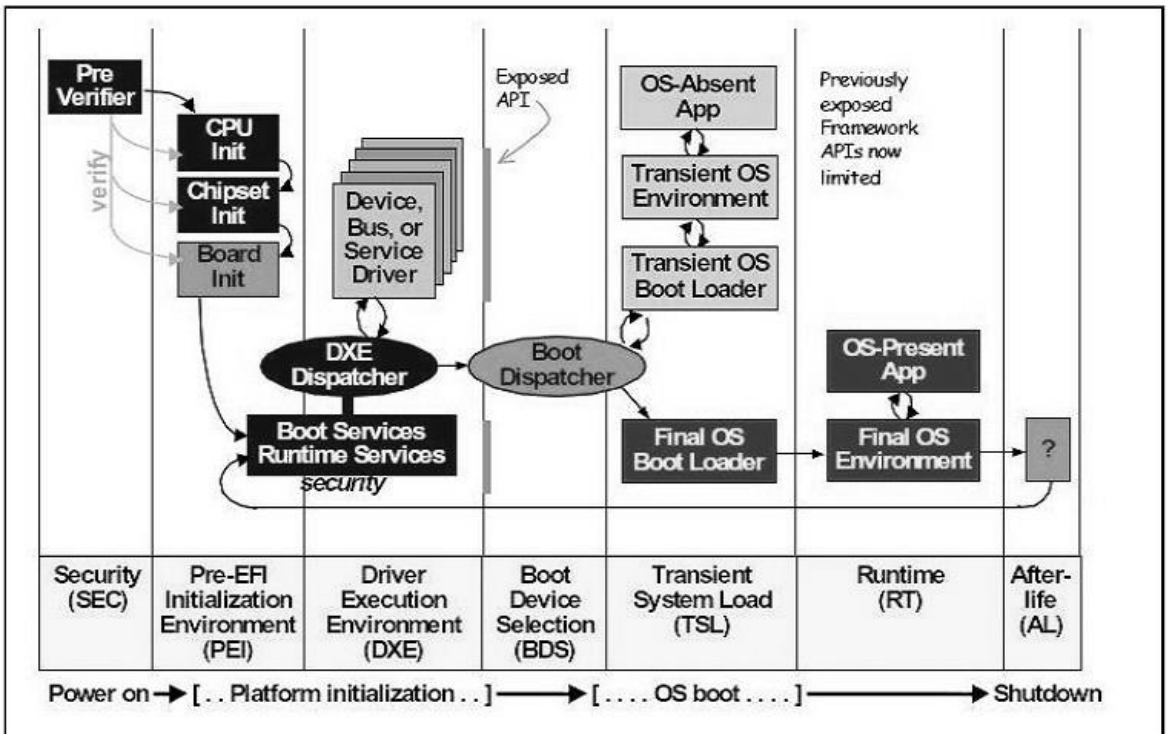

图 2-3 EFI 运行流程

2.5.1 SEC

开机之后，系统开始执行第一条指令，此时就已经进入了 SEC 阶段。这时的 Memory 还没有被初始化还不可用，所以这一阶段最主要的工作就是建立一些临时的 Memory，它可以是处理器的 Cache，或是 system Static RAM(SRAM)，并且使 CPU 进入 Protect Mode。另外 SEC Phase 可

以先天知道（Prior Knowledge）这些早期的内存被映射到的位置以及 BFV（Boot Firmware Volume）的位置。

2.5.1.1 什么要有 SEC Phase

(1)需要用汇编语言来完成 C 无法处理的工作，如 C 语言无法处理 CPU 的特殊寄存器（MSR，MTRR，CRX）。

(2)C 语言需要 Memory 当成 Stack 来处理 Local 变量，刚开机 Memory 还没有被初始化，还不可用，所以需要 CAR（Cache As Ram）的初始化。

(3)让 CPU 进入 Protected Mode(Flat Mode)。

2.5.1.2 SEC 阶段任务

SEC Phase 是整个 UEFI Boot 过程中的第一个阶段，它主要完成的任务有：

(1)系统上电/重启的入口，处理所有的平台 restart events，包括开机，重启，或是各种异常条件下的启动。

(2)创建一块临时的内存区域，在系统内存初始化之前使用，比如用 CAR（Cache As Ram）或 SRAM。

(3)在安全方面，是信任链的根（the root of trust）。之后模块的任何安全相关的设计都必须有个根，而因为系统开机之后最初的代码实现是在 SEC 阶段，所以平台设计者在调用 PEI Foundation 之前可以在 SEC 阶段验证 PEIFoundation 的安全性。所以说 SEC 阶段是系统信任链的根。

(4)传送 Handoff 信息到 PEI Foundation（这才是 SEC Phase 的最终目的），这些信息包括：

a.平台的状态，

b.BFV(Boot Firmware Volume)的位置和大小，

c.临时 RAM 的位置和大小，

d.栈的位置和大小。

数据结构为：EFI_PEI_STARTUP_DESCRIPTOR。

typedef struct {

```
    UINTN                       BootFirmwareVolume;
    UINTN                       SizeOfCacheAsRam;
    EFI_PEI_PPI_DESCRIPTOR    *DispatchTable;
} EFI_PEI_STARTUP_DESCRIPTOR;
```

另外，还有一个 PPI：EFI_SEC_PLATFORM_INFORMATION_PPI 可以用来传送 Handoff 信息。

2.5.2 PEI

PEI 阶段最主要的工作就是 Memory 的初始化以及一些必要的 CPU、Chipset 等等的初始化。由于这些都是没有压缩的 Code，所以要求越精简越好。另外，PEI Phase 还要确定系统的 Boot Path，初始化和描述最小数量的包含 DXEfoundation 和 DXE Architecture Protocols 的 System RAM 及 firmware volume。

2.5.2.1 PEI 阶段作用范围

PEI 阶段负责为后续阶段提供一个稳定的基础去初始化足够的系统。同时负责从失效的固件储存空间检测和恢复并提供启动模式。计算机从原始的状态开始执行。处理器可能需要更新它们的内部微码，芯片组需要大量的初始化。PEI 负责初始化这些基本的子系统，提供一个简单的基础结构，使之转移到更先进的 DXE 阶段的任务能够被简单地完成。PEI 阶段只负责平台启动需要的一小部分任务，也就是只需要履行那些为启动 DXE 所需的那部分。如下：

PEI Phase 的任务：

(1)基本的 Chipset 初始化

(2)Memory Sizing

(3)BIOS Recovery

(4)S3 Resume

(5)切换 Stack 到 Memory （Disable CAR， Enable Cache）

(6)启动 DXEIPL（DXE Initial Program Loader）

2.5.2.2 PEI 基本组成

PEI 阶段包含几个部分：

(1)一个 PEI Foundation，存在于 BFV。

(2)一个或多个 PEIMs（Pre-EFI Initialize Module），存在于 FVs。

PEI 基础的目标是为特殊的处理器体系结构保留相关的常量，为支持从不同供应商提供的特殊处理器、芯片组、平台和其它的器件加入模块。若没有模块间的相互关联，这些加入的模块通常不能被编码，就算可以，它们将也起不到作用。

2.5.2.3 执行条件

2.5.2.3.1 临时 RAM

在系统内存还没有完全初始化时 PEI 阶段将使用 SEC 阶段提供的的暂存式 RAM 用于数据存储。这个 RAM 可以被当做常规 RAM 那样去被访问。一旦系统被完全初始化好后，临时 RAM 可能会被重新初始化给供其他用。

2.5.2.3.2 启动 FV

(BFV)包含了 PEI Foundation 和 PEIMs 模块。它们必须装载在系统的内存地址空间中。并且他们还会包含有该处理器架构的复位矢量。

BFV 的格式是按照 EFI flash 的文件系统格式。PEI Foundation 按照 EFI flash 格式来找到 BFV 中的 PEIMs。平台的一个 PEIM 会告知 PEI Foundation 系统中其他固件卷的位置供 PEI Foundation 在其他的固件卷中寻找到 PEIMs。 PEI Foundation 和 PEIMs 在 EFI flash 文件系统中有唯一 ID 标明。

2.5.3 PEI 用到的概念

2.5.3.1 PEI Foundation

PEI Foundation 存在于 FV0(即 BFV)，它是在 SEC 阶段被发现并通过验证的，这也就允许 PEI 阶段能够确定 FV 文件有没有被破坏掉。

PEI Foundation 负责：

(1)Dispatching PEIM

(2)Maintaining the boot mode

(3) Initialize permanent memory

(4)Invoking DXEIPL

2.5.3.2 PEI 初始化

PEIM 可包含处理器和其它特殊平台功能的可执行二进制文件。PEIM 提供 PEIM 或 PEI Foundation 与 PEIM 和硬件间通信的接口。PEIM 是不被压缩的驻留在 ROM 中的单独编译的二进制文件。一小部分 PEIM 也可以在 RAM 中运行，这些 PEIM 在 ROM 中压缩格式存放的。ROM 中的 PEIM 由位置相关代码与重定位信息位置或者独立代码一起构成的内置可执行的模块。

2.5.3.3 PEI 服务

在 PEI Foundation 是在在系统中建立一个系统表。这个表被命名为 PEIServices Table，它对所有 PEIM 可见的。一个 PEI 服务程序可以被定义成一个命令或者其它一些功能。因为 PEI 前阶段没有可用的固定存储器，所以 PEI 后阶段能被创建的服务程序的范围要广泛一些。编译的时候 PEI Foundation 及其临时内存的位置是未知的，指向 PEI Services Table 的指针被放进每一个 PEIM 和 PPI 中。PEI Foundation 有下面几种类型的服务:

(1)PPI 服务:管理 PPI 和管理 PEIM 模块间的调用。通过临时 RAM 中

管理接口被跟踪、安装的信息数据库。

(2)启动模式服务:管理系统 S3、 S5、诊断和正常启动等启动模式

(3)HOB 服务:建立叫 Hand-Off Blocks 结构传递信息给下一个阶段

(4)Firmware Volume 服务:在 Flash 的 FV 中扫描各 FFS 寻找 PEIM 和其余固件

(5)PEI Memory 服务:在固定存储器被发现的前后提供存储器

(6)状态码 Services: 报告统一的进展和错误代码服务

(7)ResetServices:重启动系统

2.5.3.4 PEIM 到 PEIM 间的接口(PPI)

在 PEIM 可以通过 PPI 接口调用其它 PEIM。每个接口使用 GUID 命名保证独立性。PPI 被定义成结构，它可以包含数据、函数或者这两个的联合。PEI Foundation 管注册 PPI 的数据库，PEIM 必须要使用 PPI。PEIM 想要使用一个具体的 PPI 可以查询 PEI Foundation 来找到他所需要的接口。类型如下：

(1)服务程序

(2)通知

2.5.3.5 简单堆

在固定系统存储器还不能使用的时候，pei Foundation 使用临时 RAM 当做简单的堆存储。PEIM 可以申请从堆中分配需要的存储单元，从堆中释放内存的机制不存在。当固定存储器可以使用，堆就被重新定位到固定系统内存中，PEI Foundation 不会处理堆中存在的数据。PEIM 不能将堆中其它数据指针存入此堆中。

2.5.3.6 传递下去的信息块(HOBS)

HOBS 用来从 PEI 阶段到 DXE 阶段的传递系统状态信息的结构机制。

HOB 包含一个头和数据部分。 所有 HOB 有统一的头的定义且允许任何代码使用这个定义去了解数据部分的结构和 HOB 总大小。

固定内存可以使用后，HOB 在内存中被顺序地分配提供给 PEIM 使用。依据一些核心服务构成内存中 HOB 的顺序列表称做 HOB 列表。PHIT HOB 是 HOB 列表中的第一个 HOB。它给出了 PEI 阶段所使用的内存和这期间发现的启动模式。

只有 PEI 阶段的元件才可以增加或更改 HOB。DXE 原件只能有效的读取 HOB 列表和一些握手信息。HOB 列表包含 PEI 阶段传递到 DXE 阶段时所包含的系统状态数据，但不代表 DXE 阶段期间的系统当前状态而只能通过 DXE 定义的那些服务代替通过分析 HOB 列表获得。

2.5.3.7 PEI 阶段的操作

PEI 包括调用 PEI Foundation、发现和调用下一个阶段全部的 PEIM。初始化时，PEI Foundation 初始化需要给 PEIM 提供通用服务的函数和内部数据区域。PEIM 调度时，PEI Dispatcher 遍历全部的 FV,根据 flash 文件系统的定义发现 PEIM。如果下面的条件满足的话 PEI Dispatcher 调度所有的 PEIM:

(1)PEIM 还没有被调用过

(2)PEIM 文件已经被正确的格式化好

(3)PEIM 可信赖

(4)PEIM 的依赖性条件满足

2.5.3.7.1 PEI Dependency 表达式

相关联的条件判断表达式来决定 PEIM 的执行顺序。这个二进制表达式来描述每个 PEIM 运行的必要条件，它强调了 PEIM 的弱序化。在这个弱序化中，PEIM 可被以随意顺序初始化。在 Dependency 参考(引用)PPI 和文件名中的 GUID。 Dependency 表达式是操作中一个有语法，可以在大部分 Dependency 中运行。PEIFoundation 用内部可运行的的 PEIM 反通

过逻辑与逻辑或、逻辑非和之前、之后的顺序操作 Dependency 表达式并注册 PPI。

2.5.3.7.2 PEIM 的执行过程

PEIM 被 PEI Foundation 调用执行。且每个 PEIM 只能被调用一次，还要完成它自己的工作。必要的时候 PEIM 也可以注册一个 callback 用于其它的 PEIM 运行完后，那么这 PEIM 需要重新获得控制权。

2.5.3.7.3 内存的发现

发现内存是 PEI 阶段中一个重大的结构事件。当 PEIM 成功地发现个连续范围的系统内存时，它会把这个 RAM 告知给 PEI Foundation。当那个 PEIM 退出时，PEIMFoundation 把 PEI 使用的临时内存转移到真实系统 RAM 中，包括 PEI Foundation 必须把临时 RAM 中使用的堆栈转换到固定的系统内存和 PEI Foundation 必须把 PEIM 分配好的简单堆转移到真是的 RAM 中两个任务。这个过程完成后。 PEI Foundation 就会安装一个 PPI 通知 PIEM 真正的系统内存已经建立好可以使用了。

2.6.3 DXE

DXE 阶段是实现 EFI 的最重要的阶段，大部分的工作都是在这个阶段实现的。DXE 阶段完成了大多数的系统初始化。DXE 通过 HOBs 获得 PEI 传递的系统状态信息，HOBs 在内存中的位置不是固定的。DXE 阶段包括 DXE 基础框架、DXE 调度器、DXE 设各驱动。

DXE 基础框架包含了启动服务、DXE 服务、运行时服务。DXE 调度器发现并按照规定顺序执行 DXE 驱动。DXE 驱动包含有初始化处理器、以及平台组件、芯片组等。全部工作共同协作完成平台的初始化。DXE 阶段一直持续到操作系统成功开始启动。

2.6.3.1 DXE 基础框架

DXE 有以下的特点，其始化仅依赖于 HOB 链表，这种依赖关系意味着它不会调用前阶段创建的所有服务 HOB 链表被传递给 DXE 后，会把所有阶段中的服务和数据释放。其次不使用硬编码的地址空间。它可以在物理内存的任何位置被加载无论在内存的哪个地方，无论固件卷(firmware volume)位于处理器的哪个地址，DXE 都是可以执行的。最后，DXE 的基础框架不包括任何特定处理器、特定平台的信息、特定芯片组。因为。DXE 的基础框架是来自对系统硬件的抽象。由相应 protocol 接口架构实现的。

DXE 创建了 EFI 系统表和一系列的 EFI 运行时服务和 EFI 启动服务。DXE 基础框架也包括了 DXE 调度器其主要要作用是发现并执行存储在固件卷(firmware volume)中的 DXE 设备驱动。执行顺序取决于一组可配置的依赖关系描述文件是 PE/COFF 格式，DXE 通过 PE/COFF 格式解释器加载和执行。

2.6.3.2 交接块（Hand-Off Block，HOB）链表

HOB 链表中包括了各种信息被 DXE 框架结构用于去创建基于存储器的服务。HOB 链表中包括 PEI 阶段发现的系统内存、启动模式信息、处理器指令集、初始化的系统信息描述及 PEI 阶段发现的固件设备信息。固件设备信息包含有系统内存在固件卷和固件设备中的位置。固件卷包括 DXE 调度器和 DXE 设备驱动。调度器负责加载并执行 DXE 设备驱动，之后被存储在固件卷中。HOB 链表还包含有 PEI 阶段找到的内存映射 I0 资源和 I0 资源。

2.6.3.3 DXE 架构协议（Architecture Protocols）

DXE 框架是从平台硬件抽象出来，并且由一些 DXE 架构 protocolt 成的分为 EFI 启动服务和 EFI 运行时服务。从固件卷中加载的 DXE 设备驱动创建了 DXE 架构协议。从这个设计中我们可以看出，DXE 框架从一

开始必须包含足够的服务和启动 DXE 设备。

DXE 框架的 HOB 链表须包含至少一个固件卷和系统内存的描述信息。系统内存描述信息用于初始化 EFI 服务。整个系统还必须运行在单处理器环境，以及没有中断服务的 flat physical mode。由 HOB 被传递给 DXE 调度器的固件卷必须包括有一个只读的 FFS 驱动用来解析和搜索指定固件卷中的 DXE 设备驱动和依赖描述文件。当一个依赖关系满足的驱动被找到，DXE 调度器就会使用 PE/COFF 解析器加载并运行这个 DXE 驱动。DXE 架构 protocol 会被前期的 DXE 驱动创建，以便在后面的 DXE 基础结构能创建出全部的 EFI 启动服务和 EFI 运行时服务。

一些 DXE 架构协议：

Security Architectural Protocol:用作 DXE 验证固件卷中的文件能否是可以用的。

CPU Architectural Protocol:提供用于管理缓存、中断、处理器频率恢复和基于处理器的时钟查询服务。

Metronome Architectural Protocol:提供让处理器暂停并确认改动时间的服务。

Timer Architectural Protocol:提供在 DXE 使用的 heartbeat timer 中断服务。

BDS Architectural Protocol:提供在全部的 DXE 设备驱动完成之后被调用的入口地址。当系统运行从 DXE 阶段运行到 BDS 阶段时，建立控制台并使能启动服务运行。

Watchdog Timer Architectural Protocol:提供用于控制平台监视时钟开启和关闭的服务。

Runtime Architectural Protocol:提供运行时服务和从物理地址到虚拟地址转换运行时驱动。

Variable Architectural Protocol:提供获取环境变量和存储临时变量服务。

VariableWrite Architectural Protocol:提供持久保存变量的存储服务。

Monotonic Counter Architectural Protocol:提供一个 64 位计数器。

Reset Architectural Protocol:提供用于系统重置和系统关闭的服务。

Status Code Architectural Protocol:提供处理 DXE 基础结构和 DXE 设备驱动返回码的服务。

Real Time Clock Architectural Protocol:提供对当前时间和日期查询设置的服务，和可选择的用于特定时间唤醒功能的计时器。

2.6.3.4 EFI 系统表

DXE 基础结构创建了 EFI 系统表供所有 DXE 设备驱动和 BDS 阶段调用的可执行映像文件同时调用，它包含了被 DXE 调用的全部组件和服务。

DXE 基础结构创建了 EFI Boot Services、EFI Runtime Services、和符合 DXE 架构 protocol 的所有服务。 EFI 系统表包含了全部可用设备访问方法和一些 EFI 配置表。EFI 配置表是可以进行扩展的表集，它描述了平台上配置信息，包含有 DXE 服务，ACPI, SMBIOS 和 SAL 系统表等可以在需要时扩展各种类型的表。并且，任何可执行映像可以通过 Protocol Handle Services 访问 Handle Date 和所有已在 DXE 驱动表中注册的 protocol 接口。

句柄数据库、EFI 启动服务和 DXE 驱动提供的所有服务在进入运行时状态以后就会中止。被释放的内存空间会留给 OS 来使用，EFI 系统表、系统设置表以及 EFI 运行时服务表不会存被终止它们仍然可以被用于操作系统的环境中。当然你也有一次机会选择将所有的 EFI 运行时服务从物理地址转换成 OS 下的虚拟地址。

EFI 运行时服务表（Runtime Services Table）

运行时服务表的各种服务的简单介绍:

Variable Services:为非易失性存储的环境变量提供查找添加、删除服务。这些服务依赖于变量架构 protocol 和写变量架构 protocol。

Real Time Clock Services:提供设置当前时间和日期的服务以及读取和设置依赖于时间日期架构 protocol 的可选的唤醒计时器的服务。

Reset Services:提供用于重启和关闭系统的服务。

Status Code Services:提供服务去处理状态码并发送到口志或者一个输出设备。

2.6.3.6 DXE 服务表（Services Table）

GCD 管理服务(Global Coherency Domain Services):提供平台上 I/O 资源、系统内存资源、以及 MMIO 资源管理服务用于向动态增加和删除除处理器的 GCD 中的资源。

DXE 调度器服务:提供管理被 Dispatcher 加载和运行的 DXE 设备驱动的服务。

DXE 调度器（Dispatcher）

我们根据一个优先级描述文件来查找新的固件

.3 卷里的驱动。每个优先级描述文件使用固定的文件名同时还包含应被优先加载并执行 DXE 驱动的加载顺序。每个固件卷只能有一个优先级描述文件，当然优先级描述文件不是必须的可以没有，相关的依赖表达式会被更新并依次找出下一个被加载的 DXE 驱动。优先级描述文件给出一个固定的执行顺序的 DXE 驱动列表，供驱动的加载使用这些驱动不使用依赖式，而优先级较低的 DXE 驱动则使用依赖表达式，并且在每个 DXE 驱动都必须过安全架构 protocol 的验证才能执行。禁止加载其他不明来源的 DXE 驱动。

2.6.3.5 DXE 驱动（Drivers）

第一类是 DXE 阶段早期执行的 DXE 驱动，这类在 DXE 早期加载的驱动的执行顺序取决于优先级文件和依赖表达式。典型的早期执行的 DXE 驱动包括处理器、芯片组和平台初始化代码，这些早期加载的 DXE 驱动还会创建 DXE 架构 Protocol，这个 protocol 被 DXE 基础框架用来创建 EFI 启动服务和 EFI 运行时服务。为了减少启动所用时间，可以将尽可能多设备初始化放入第二类按照驱动模型创建的 DXE 驱动。

还有一类是按照驱动模型创建的 DXE 驱动，这一类类 DXE 驱动被 DXE 调度器运行时，驱动不会做任何硬件的初始化。DXE 调度器会将 Driver Binding Protocol 接口注册到 handle 数据库中。这类驱动还提供对控制台设备和启动设备的软件抽象。

DXE 驱动执行时需要使用 EFI Boot Services 和 EFI Runtime Services。这时 DXE 架构 Protocol 可能还没有完成注册，DXE 驱动要使用依赖表达式来保证当前驱动所依赖的其他服务都是可用的。

在 DXE 阶段，第二类 DXE 驱动会被注册到 handle 数据库中，直到现在所有的 DXE 基础架构 protocol 才被加载完成。当 DXE 调度器已经将所有的 DXE 驱动执行完，而这时 DXE Architectural Protocols 还没有完全加载，那么 BIOS 会发出一个严重错误并 halt 住系统。

2.7.4 BDS

BDS Architectural Protocol 是在 DXE 阶段被找到，并且在执行前必须满足两个条件，首先所有 DXE Architectural Protocols 都已经注册到 handle 数据库中同时 EFI Boot Services 和 EFI Runtime Services 要建立好。其次 DXE Dispatcher 加载完所有的符合条件的驱动。BDS 查找和加载许多可以在启动前环境中执行的应用。这些应用程序可以用来创建一个传统的 OS boot loader 或者一个扩展服务来加载最后需要需要启动的 OS。这些扩展启动前服务包括扩展诊断服务、flash 升级支持、OEM 服务等。

BDS 阶段会执行一系列任务。用户界面和响应方式依据不同的平台可能完全不同:但是 BDS 阶段的启动规则却基本相同，基于 ConIn、ConOut 和 StdErr 的初始化控制台设备，尝试加载环境变量中 DriverOrder 中的所有设备，从和 Std Err 这些环境变量初始化控制台设备，从 BootOrder 列出的设备启动系统。

如果 BDS 连接控制台设备失败，加载驱动或者启动失败，那么意味着新固件卷在上次完成调用后并没有被找到，DXE 调度器会被重新调用。在新找到的固件卷中可能包含有管理控制台设备和启动设备的 DXE 驱动。控制权会在新发现的固件卷中的驱动全部都被加载后重新交给 BDS。

这时 BDS 还是连接不到控制设备或者是启动设备，BDS 就启动失败了，失败后会自动尝试从下一个控制台设备启动。

BDS 阶段的主要工作是：

(1)初始化基于环境变量 ConIn、ConOut、StrErr 的控制台设备。

(2)尝试去加载列在环境变量 Driver####和 DriverOrder 上的 Driver。

(3)尝试从列在环境变量 Boot####和 BootOrder 上的启动设备列表中启动。

2.7.5 TSL

指 shell

2.7.6 RT Phase

当 OS 呼叫了 Boot Service ExitBootService（）之后，系统就进入了 RT 阶段。此时，DXE Foundation 和 Boot Service 都已经终止了，只有 EFI Runtime Service 和 EFI System Table 还可以继续被使用。

2.7.8 AL（After Life）

当 OS 呼叫了 EFI Runtime Service ResetSystem()或者是呼叫了 ACPI Sleep State，系统就进入了 AL 阶段。 异步 Event（比如 SMI、NMI）的触发也可使系统进入 AL 阶段，这在 Server 和 Workstation 上比较常见。

2.8 本章小结

本章对 EFI 架构做了详细阐述。详细介绍了 EFI 启动管理器、协议、EFI 内核、句柄数据库和 EFI 驱动模型这些 EFI 系统中各个重要组成部分。对 EFI 运行流程做了具体描述。

第 3 章　EXTx 文件系统概念及重要数据结构实现

3.1　引 言

文件系统是操作系统组织、存取和保存信息的重要手段，每种操作系统都有自己的文件系统， Linux 所用的文件系统主要有 EXT2、EXT3、EXT4 和 ReiserFS 等。

3.2　概述

EXT2 和 EXT3 和 EXT4 是现如今 Linux 操作系统默认的文件系统，我们将其统称为 EXTx。EXTx 基于 UFS(Unix File System)是一种快速、稳定的文件系统。EXTx 在整个文件系统中的多处存放重要数据结构的备份，这样就使得其可恢复性很好方便数据恢复。

3.2.1　总体结构

虽然从 EXT2 到 EXT4，找数据的方式发生了变化，但是，磁盘的布局还是非常相似的。磁盘布局如下：

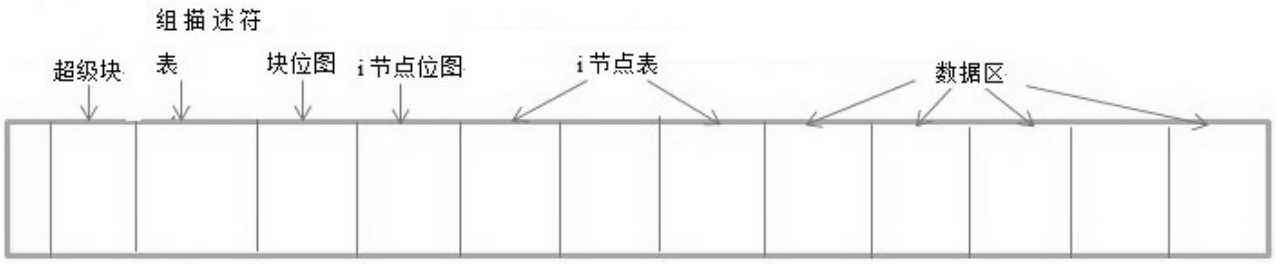

图 3-1 EXTx 文件系统磁盘布局

(1)0-1 号扇区为引导代码保留扇区，只提供给引导代码使用，如果引导代码不存在，则把这两个扇区全部用 0 填充。

(2)2-3 号扇区为 Super block，描述了块的大小、总块数、每块组块数、第一个块前保留块数、i-node 数、每块组 i-node 数、卷名、最后写入时间、

址后挂载时间及挂载路径、一致性检查的标志等，同时还有空闲 i-node 和空闲块的记录信息。

(3)Super block 后面的开始的块为“组描述符表”其具体的起始位置在 Super block 中给出。

特别指出的是，组描述符表是紧接在 Super block 后而的“块”，依据每个块包含的扇区的不同起始位置的扇区号也就不同，当块大小为 1024 个字节(两个扇区〕或2048个字节(4个扇区)时，组描述符表就起始于Super block 后面的 4 号扇区，同样的当块大小为 4096 个字节（8 个扇区），则组描述符起始于 8 号扇区。

“组描述符表”包含有文件系统中全部块组的信息，每组描述符占用 32 个字节。使用默认的参数格式化时，一般来说整个组描述符表的大小不大于一个块组。每个块组中都包含有一个组描述符表备份，激活了稀疏 Super block 特征的除外。

(4)紧接在组描述符表的 Block 块后面的是“块位图块”。在创建文件系统时，每块中的 bit 数和每组的块数设置为相等，这样一个块的大小和块位图的大小是等同的。块组中块的分配情况的管理是块位图的任务，其起始位置在组描述符中给出。块位图的字节大小为组中的块总数除以 8。

(5) 紧接在块位图块后面的是 Block 块是" i-node 位图块”，也只占用一个 Block 块。i-node 位图管理组中 i-node 的分配,起始位置同样是在组描述符中。每组 i-node 数除以 8 可以得到它的大小字节数。通常 i-node 数值的大小是可以选择的。i-node 表的起始位置也在组描述符中给出，每个 i-node 大小为 128 字节。

(6) 接在 i-node 位图块后面的是 i-node 表。每个 i-node 的大小固定为128 个字节，用来存储文件及目录的元数据，全部的 i-node 都存放在 i-node 表中，每个块组中都有一个属于本块组的 i-node 表。

(7)接在 i-node 表后面的是数据区。文件名使用“目录项”进行存储，目录项存储在其父目录分配的块中。目录项由文件的名字和指向这个文件的 i-node 项的指针组成。

3.2.2　详细描述

我们根据块大小为 1024 字节详细描述 EXTx 组织结构。其详细结构如图 4-2 所示。

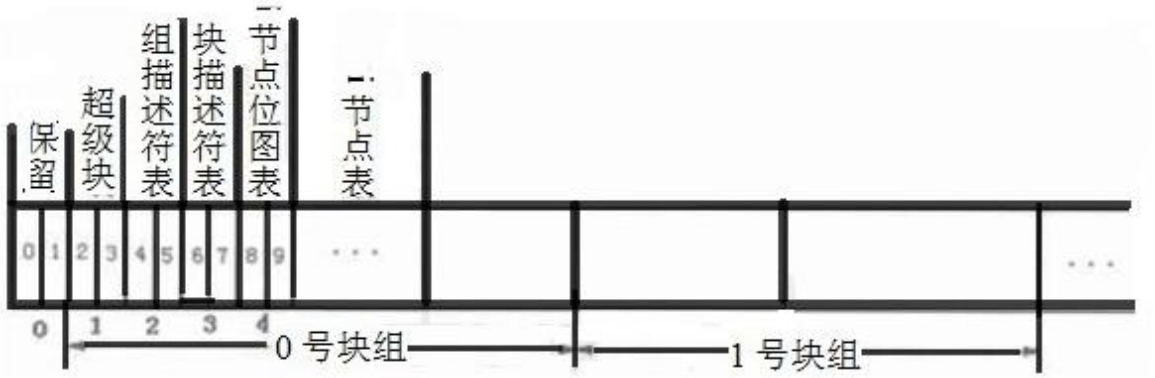

图 4-2 布局图

总结如下：

(1)0-1 号扇区 0 号块，为引导扇区。

(2)2-3 号扇区 1 号块，为 Super block。

(3)4-5 号扇区 2 号块，为组描述符表。

(4)6--7 号扇区 3 号块，为块位图块。

(5)8-9 号扇区 4 号块，为 i-node 位图。

(6) 10 号扇区开始的 5 号块开始为 i-node 表，占用若干个块。然后是数据区。

3.3　Super block

Super block 在文件系统中的位置是固定的在 1024 字节处，其分配到的空间为 1024 字节，但空间并没有完全被使用。

Super block 包含一些如块大小、总块数、每块组块数及 i-node 数和每块组 i-node 数等的配置信息。以及卷名、最后写入时间、最后挂载时间及挂载路径等。 Super block 中还记录空闲块和空闲 i-node 和空闲块信息，当需要分配新的 i-node 和新块时使用。

3.3.1 Super block 数据结构

struct ext2_sblock {
UINT32 total_inodes; 文件系统总的 i-node 数
UINT32 total_blocks; 文件系统总块数
UINT32 reserved_blocks; 为文件系统预保留的块数
UINT32 free_blocks; 空闲块数
UINT32 free_inodes; 空闲 i-node 数
UINT32 first_data_block; O 号块组起始块号
UINT32 log2_block_size; 块大小(此值为需要将 1024 左移的位数)
UINT32 log2_fragment_size; 片段大小(与块大小字段完全相同)
UINT32 blocks_per_group; 每个块组所含块数
UINT32 fragments_per_group; 每个块组所含片段数
UINT32 inodes_per_group; 每块组 i-node 数
UINT32 mtime; 最后挂载时间
UINT32 utime; 最后写入时间
••••••
};

(1)Magic 签名

对于 EXT2 和 EXT3 文件系统来说，这个字段的值应该正好等于 0xEF53。如果不等的话，那么这个硬盘分区上肯定不是一个正常的 EXT2 或 EXT3 文件系统。

(2)s_log_block_size

从这个字段，我们可以得出真正的 block 的大小。我们把真正 block 的大小记作 B，B=1 << s_log_block_size + 10)，单位是 bytes。举例来说，如果这个字段是 0，那么 block 的大小就是 1024bytes，这正好就是最小的 block 大小;如果这个字段是 2,那么 block 大小就是 4096 bytes。从这里我们就得到了 block 的大小这一非常重要的数据。

(3)s_blocks_count 和 s_blocks_per_group

通过这两个成员，我们可以得到硬盘分区上一共有多少个 block group，或者说一共有多少个 group descriptors

s_blocks_count 记录了硬盘分区上的 block 的总数，而 s_blocks_per_group 记录了每个 group 中有多少个 block。显然，文件系统上的 block groups 数量，我们把它记作 G，G=(s_blocks_count-s_first_data_block- 1)/s_blocks_per_group+1。为什么要减去 s_first_data_block，因为 s_blocks_count 是硬盘分区上全 部的 block 的数量，而在 s_first_data_block 之前的 block 是不归 block group 管的，所以当然要减去。最后为什么又要加一，这是因为尾巴上可能多出来一些 block，这些 block 我们要把它划在一个相对较小的 group 里面。

(4)s_inodes_per_group

s_inodes_per_group 记载了每个 block group 中有多少个 inode。在从已知的 inode 号，读取这个 inode 数据的过程中，s_inodes_per_group 起到了至关重要的作用。

用我们得到的 inode 号数除以 s_inodes_per_group，我们就知道了我们要的 这个 inode 是在哪一个 block group 里面，这个除法的余数也告诉我们，我们要的这个 inode 是这个 block group 里面的第几个 inode；然后，我们可以先找到这个 block group 的 group descriptor，从这个 descriptor，我们找到这个 group 的 inode table，再从 inode table 找到我们要的第几个 inode，再以后，我们就可以开始读取 inode 中的用户数据了。这个公式是这样的：

block_group = (ino - 1) / s_inodes_per_group。这里 ino 就是我们的 inode 号数

offset = (ino - 1) % s_inodes_per_group，这个 offset 就指出了我们要的 inode 是这个 block group 里面的第几个 inode。

(5)在 ext4_super_block 结构中，增加了 3 个与此相关的字段：

s_blocks_count_hi、s_r_blocks_count_hi、s_free_blocks_count_hi，它们分别表示 s_blocks_count、s_r_blocks_count、s_free_blocks_count 高 32 位的值，将它们扩充到 64 位。

3.4 块组描述符表和块组描述符

块组描述符表也可以称为组描述符表，它起始于 Super block 所在块的下一个块。它是一个列表，由文件系统中描述每个块组的组描述符组成，文件系统中的协个块组都在这个表中拥有一个包含该组相关信息的描述项。

如果没有激活稀疏 Super block 特征，则在每个块组中都会有一个组描述符表的备份。但通常 Linux 会默认激活稀疏 Super block 特征，因此只会在某些块组中存在组描述符表的备份。其他的块组中则没有。在没有 Super block 备份及组描述符备份的块组中，也不会为 Super block 和组描述符保留空间。

通常每个块组中除文件内容以外，还会有管理数据，如 Super block、组描述符表、i-node 表、i-node 位图块和块位图块，组描述符中说明了这些数据的存储位置。

3.4.1 块组描述符表和块组描述符数据结构

```
struct ext2_block_group {
    UINT32 block_id;  块位图起始地址(块号)
    UINT32 inode_id; i-node 位图起始地址(块号)
    UINT32 inode_table_id; i-node 表起始地址(块号)
    UINT16 free_blocks;  该块组中的空闲块数
    UINT16 free_inodes;  该块组中的空闲块数
    UINT16 used_dir;  该块组中的目录数
    UINT16 pad;
```

UINT32 reserved[3]; 未使用

};

类似的，在 ext4_group_desc 中引入了另外 3 个字段：bg_block_bitmap_hi、bg_inode_bitmap_hi、bg_inode_table_hi，分别表示 bg_block_bitmap、bg_inode_bitmap、bg_inode_table 的高 32 位。采用 48 位块号取代原有的 32 位块号之后，文件系统的最大值还受文件系统中最多块数的制约，这是由于 EXT3 原来采用的结构决定的。

对于 EXT3 类型的分区来说，在每个分区的开头，都有一个引导块，用来保存引导信息。文件系统的数据一般从第 2 个数据块开始（更确切的说，文件系统数据都是从 1KB 之后开始的，对于 1024 字节大小的数据块来说，就是从第 2 个数据块开始，对于超过 1KB 大小的数据块，引导块与后面的 Super block 等信息共同保存在第 1 个数据块中，Super block 从 1KB 之后的位置开始）。

文件系统将剩余磁盘划分为一个个块组。块组前面存储了 Super block、块组描述符、数据块位图、索引节点位图、索引节点表、然后才是数据块。通过有效的管理，EXT2/EXT3 可以尽量将文件的数据放入同一个块组中，从而实现文件数据在磁盘上的最大连续性。

在 EXT3 中，所有的块描述符信息全部被保存到第一个块组中，因此以缺省的 128MB（227 B）大小的块组为例，最多能够支持 227 / 32 = 222 个块组，最大支持的文件系统大小为 222 * 227 = 249 B= 512 TB。而 ext4_group_desc 目前的大小为 44 字节，以后会扩充到 64 字节，所能够支持的文件系统最大只有 256 TB。

为了解决这个问题，EXT4 中采用了元块组（metablock group）的概念。元块组就是指块组描述符可以存储在一个数据块中的一些连续块组。原来在 EXT3 中，要想扩大文件系统的大小，只能在第一个块组中增加更多块描述符，通常这都需要重新格式化文件系统，无法实现在线扩容；另外一种可能的解决方案是为块组描述符预留一部分空间，在增加数据块时，使用这部分空间来存储对应的块组描述符；但是这样也会受到前面介绍的最大容量的限制。而采用元块组概念之后，如果需要扩充文件

系统的大小，可以在现有数据块之后新添加数据块，并将这些数据块也按照元块组的方式进行管理即可，这样就可以突破文件系统大小原有的限制了。在 Super block 结构中需要增加一些字段来记录相关信息。下图给出了 EXT3 为块组描述符预留空间和在 EXT4 中采用元块组后的布局。

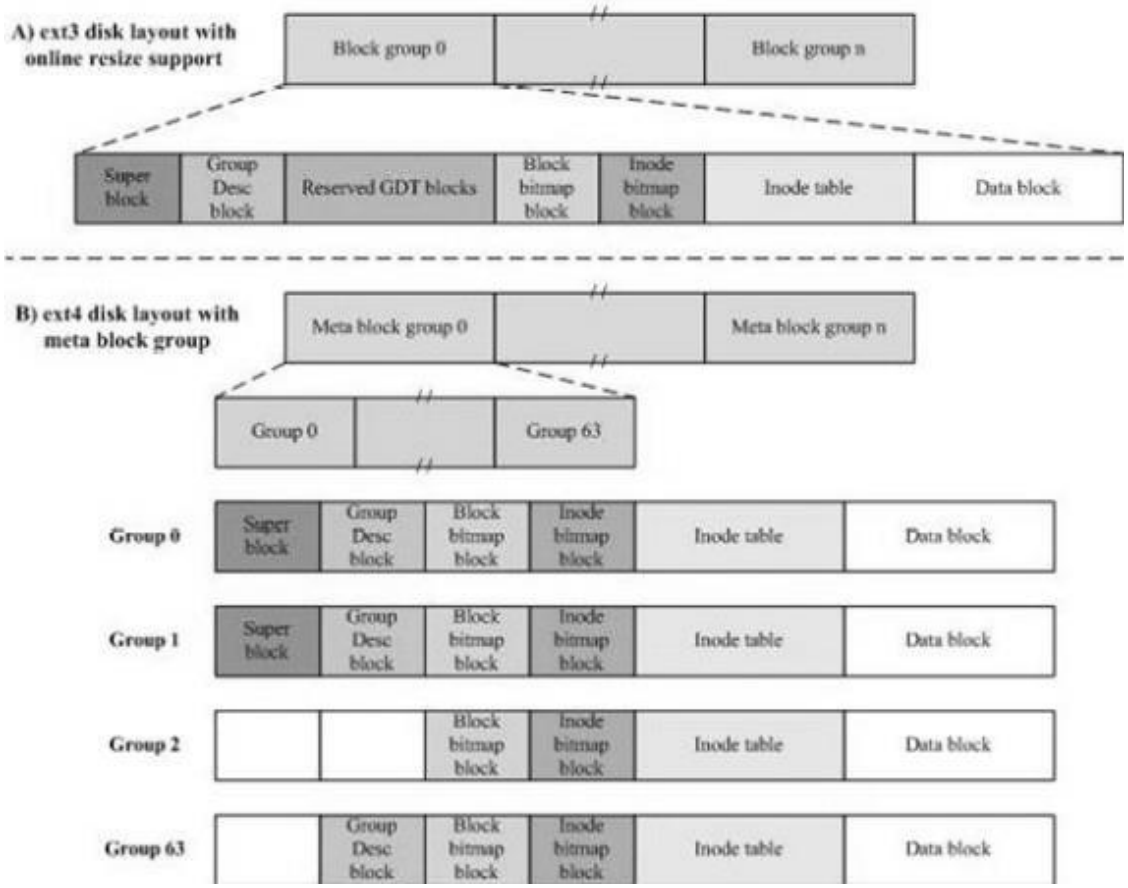

图 3-3 EXT3 与 EXT4 布局对比

3.4.2 block bitmap（区块对照表）

通过 block bitmap 的辅助我们的系统就能够很快速的找到可使用的空间来供给新增文件时的数据使用。 同样的，如果你删除某些文件时，那么那些档案原本占用的 block 号码就得要释放出来， 此时在 block bitmap 当中相对应到该 block 号码的标志就得要修改成为未使用的，这就是 bitmap 的功能。

3.4.3　inode bitmap（inode 对照表）

其实与 block bitmap 是类似的功能，只是 block bitmap 记录的是使用与未使用的 block 号码， 至于 inode bitmap 则是记录使用与未使用的 inode 号码。

3.5　i-node 表与 i-node

i-node 存储文件或目录的元数据，存放在 i-node 表中:

(1) 每个块组中都有一个本组的 i-node 表用于存放本块组内的 i-node。且每个 i-node 的大小都是相同的都是 128 个字节，大小记录在 Super block 中。

(2) 每一个文件或者文件夹对应一个 i-node，所有 i-node 从“1”开始对自己的地址编号。

(3) 全部 i-node 被通过若干个组的形势管理，每个块组中分配一组 i-node，大小由 Super block 中描述。

(4) 每个块组中的 i-node 在该组的 i-node 表中存储，i-node 表的位置组描述符表的组描述符项中给出。

(5) i-node 的 1-10 号被保留使用在 i-node 位图中设置为已分配状态，在 Super block 中给出第一个非保留 i-node 的编号，在保留的 i-node 中，2 号 i-node 是专门留给根目录使用的，1 号 i-node 用于表明坏块， 8 号 i-node 给日志使用，但可以在 Super block 中对其进行重新定义。

(6) 11 号 i-node 一般是留给用户的第一个文件，被"lost+found”目录使用。当某个 i-node 已被分配但却没有文件名指向它时，文件系统程序会把它添加到"lost+found "目录中去同时给它一个文件名。

(7) i-node 的分配情况由 i-node 位图进行描述，i-node 位图的位置在组描述符中给出。

3.5.1 i-node 数据结构

```
struct ext2_inode {
UINT16 mode; 文件模式(类型及权限)
UINT16 uid; UID 的低 16 位
UINT32 size; 大小字节数低 32 位
UINT32 atime; 最后访问时间
UINT32 ctime; i-node 改变时间
UINT32 mtime; 最后修改时间
UINT32 dtime; 删除时间
UINT16 gid; GID 的低 16 位
UINT16 nlinks; 链接数
UINT32 blockcnt; 512 bytes 为单位的块数
      ••••••
};
```

3.6 目录项

目录项用来存放文件或目录的名字。它们被保存在为目录分配的块里,包含有文件或目录的 i-node 地址信息。EXTX 的目录与一个正常文件除了在 i-node 中专用类型值不同外其余相同。

目录项、i-node 和块之间的关系如图 3-4 所示。利用文件的目录项中记录着的它的 i-node 号找到它的 i-node，从 i-node 中可获知其数据内容存储的块位置。

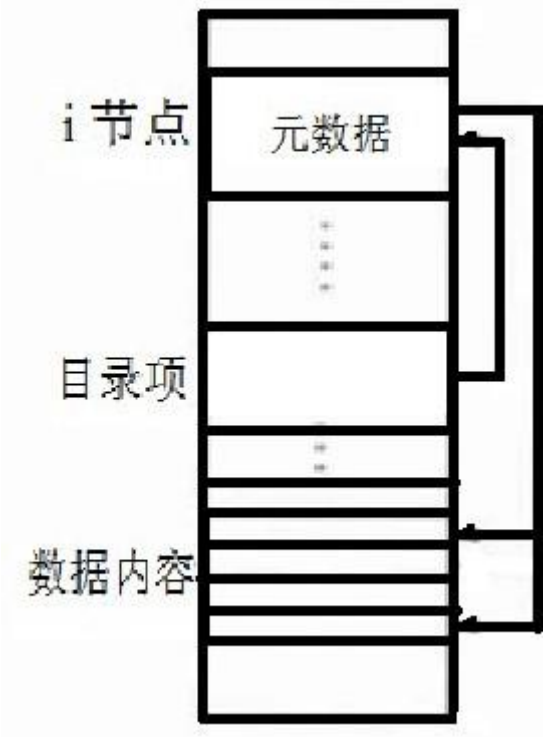

图 3-4 目录项、i 节点与块的关系

3.7　extent

EXT2/EXT3 文件系统都使用了直接、间接、二级间接和三级间接块的形式来定位磁盘中的数据块。

在 EXT4 中引入了 extent 的概念来表示文件数据所在的位置。所谓 extent 就是描述保存文件数据使用的连续物理块的一段范围。每个 extent 都是一个 ext4_extent 类型的结构，大小为 12 字节。定义如下所示：

```
struct ext4_extent {
__le32     ee_block;
__le16     ee_len;
__le16     ee_start_hi;
__le32     ee_start_lo;
};
struct ext4_extent_idx {
```

```
__le32    ei_block;
__le32    ei_leaf_lo;
__le16    ei_leaf_hi;
__u16     ei_unused;
};

struct ext4_extent_header {
__le16    eh_magic;
__le16    eh_entries;
__le16    eh_max;
__le16    eh_depth;
__le32    eh_generation;
};
```

每个 ext4_extent 结构可以表示该文件从 ee_block 开始的 ee_len 个数据块，它们的位置是从 ee_start_hi<<32+ee_start 开始，到 ee_start_hi<<32 + ee_start + ee_len －1 结束，全部都是连续的。尽管 ee_len 是一个 16 位的无符号整数，但是其最高位被在预分配特性中用来标识这个 extent 是否被初始化过了，因此可以一个 extent 可以表示 215 个连续的数据块，如果采用 4KB 大小的数据块，就相当于 128MB。

如果文件大小超过了一个 ext4_extent 结构能够表示的范围，或者其中有不连续的数据块，就需要使用多个 ext4_extent 结构来表示了。为了解决这个问题，EXT4 文件系统的设计者们采用了一棵 extent 树结构，它是一棵高度固定的树，其布局如下图 3-5 所示

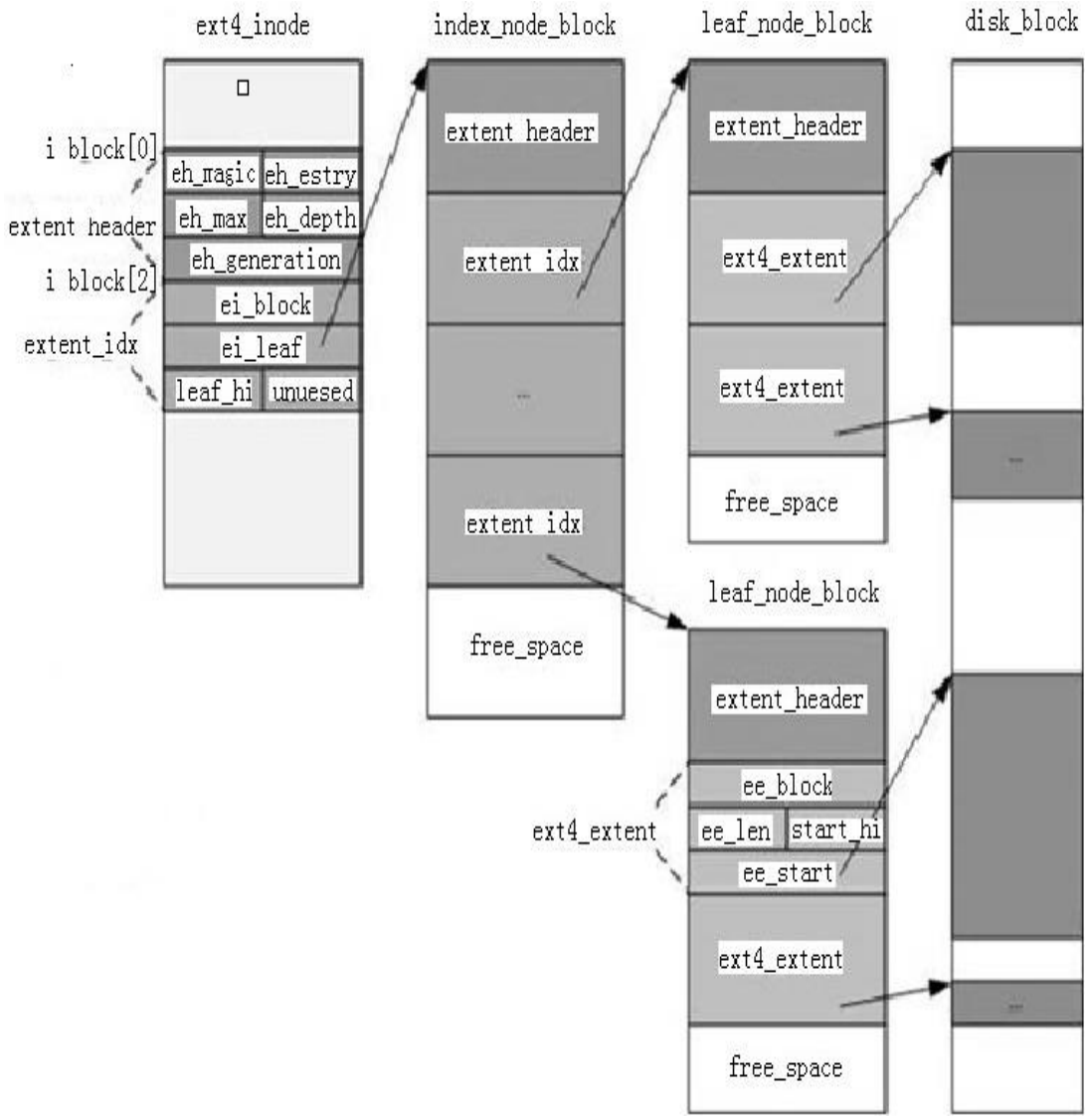

图 3-5　extent 布局图

在 extent 树中，节点一共有两类：叶子节点和索引节点。保存文件数据的块信息全部记录在叶子节点中；而索引节点中则存储了叶子节点的位置和相对顺序。不管是叶子节点还是索引节点，最开始的 12 个字节总是一个 ext4_extent_header 结构，用来标识该数据块中有效项（ext4_extent 或 ext4_extent_idx 结构）的个数（eh_entries 域的值），其中 eh_depth 域用来表示它在 extent 树中的位置：对于叶子节点来说，该值为 0，之上每层索引节点依次加 1。extent 树的根节点保存在索引节点结构中的 i_block 域中，我们知道它是一个大小为 60 字节的数组，

最多可以保存一个 ext4_extent_header 结构以及 4 个 ext4_extent 结构。对于小文件来说，只需要一次寻址就可以获得保存文件数据块的位置，而超出此限制的文件只能通过遍历 extent 树来获得数据块的位置。

第 4 章 EFI EXTx 文件系统设计与实现

4.1 系统结构

本篇 EFI EXTx 文件系统驱动的设计要达到的目标是支持 EXTx 基本功能。因为 EFI 是一个单用户、单任务的运行环境，对于 EXTx 对文件操作的一些高级特性并不能支持，所以实际目标为实现 EFI 文件协议所定义的接口。

按照 EFI 文件系统的规范，对整个系统设计为 3 各部分：协议层、核心层和 I/ O 层。系统结构见图 4-1

图 4-1 EFI 文件系统结构

下面给出三个层次的功能的解释：

(1) 协议层负责提供给外界使用调用的接口，它对外隐藏具体的文件系统内部的复杂的实现细节。该层的主要任务是实现协议所定义的接口通过调用 EXTx 文件系统提供的各个模块的接口。

(2) 核心层是本 EXTx 文件系统重点实现部分。从逻辑上看这是一个比较简单的模块，它就是只要去处理 EXTx 文件系统定义的所需的各种磁盘数据结构。鉴于 EXTx 文件系统的结构复杂性，该层的实现是本章的难点也是必须要解决的重点。

(3) I/O 层完成磁盘数据的读写。EFI 提供了相应的 I/O 接口供该层所使用，实现对 I/O 的操作。

3.6.1 目录项数据结构

```
struct ext2_dirent {
UINT32 inode; i-node 号
UINT16 direntlen;  本目录项的长度字节数
UINT8 namelen;  名字长度
UINT8 filetype;
CHAR8 name[255];  名字的 ASCll 码
};
```

4.2 EXTx 文件系统核心模块

EXTx 文件系统核心模块主要目标是抽象出文件系统和完成对文件系统所有的数据结构的操作。在本 EFI 文件系统驱动里完成了对文件系统对象进行统一的抽象，如 Super Block、Inode 。使用到了 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL（简单文件系统协议）和 EFI 文件协议。在 EFI EXTx 文件系统驱动中，主要对文件 (EXT_OFILE)和卷 (Volume)这两个数据结构的抽象实现。另一方面，根据 EXTx 文件系统结构，可把文件系统分为几个模块：卷模块，目录索引模块，文件操作模块，文件模块，文件属性模块。这些模块完成 EXTx 各种数据结构的操作。EXTx 文件系统核心模块如图 4-2 所示。

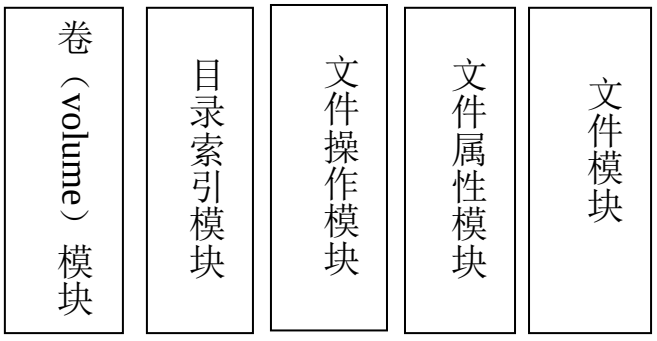

图 4-2 EXTx 文件系统核心模块

卷和文件的数据结构是 EXTx 文件系统的两个重要结构，它们抽象出卷和文件(目录) 。卷的数据结构描述了 EXTx 分区的全局信息，如文

件句柄（Handle）、BlockIo 以及 DiskIo 读写函数等信息。卷在 EXTx 文件系统驱动加载首先被创建，提供给其他函数使用。卷的结构如下所示：

```
typedef    struct _EXT_VOLUME {
    ••••••
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL   VolumeInterface;   EFI 简单文件系统协议结构
    EFI_BLOCK_IO_PROTOCOL                *BlockIo;
    EFI_DISK_IO_PROTOCOL                *DiskIo;
    UINT64            VolumeSize;
    struct    ext2_data      *extFileSystem;
    ••••••
    EXT_DIRENT          RootDirEnt;
    RootFileString[1];
    struct _EXT_OFILE       *Root;
    }EXT_VOLUME;
```

文件对象数据结构 EXT_OFILE 描述了一个文件或目录的全部信息。它记录了文件(目录) 的原数据信息。EXT_OFILE 包含很多最重要的字段其中 volume 指向文件所在的卷的数据结构，来获得全局信息。Parent 指向父文件对象数据结构。ODir 指向打开的目录文件。DirEnt 指向打开的目录的入口地址。CheckLink 是链接文件卷链表。extFileSystem 包含了文件 i-node 重要数据它记录了文件的所有属性。EXT_OFILE 的结构如下：

```
Typedef      struct _EXT_OFILE {
    UINTN                    Signature;
    struct _EXT_VOLUME    *Volume;
    •••••••
    struct    ext2_data       *extFileSystem;
    struct _EXT_OFILE      *Parent;
```

```
EXT_ODIR                *ODir;
EXT_DIRENT              *DirEnt;
EFI_LIST_ENTRY          CheckLink;
}EXT_OFILE;
```

其他所有的模块都是为这两个数据的做的具体操作。EXTx 文件系统驱动加载时调用 AllocateVolume，创建并初始化 Volume 结构。ext2_data 记录文件信息更新 i-node 节点，在新建和删除所有对文件的操作都涉及到对 i-node 节点的读写。文件操作模块给出磁盘读写工具接口。文件属性控制块模块完成对文件中各属性添加、删除、修改的操作，对任何文件的所有操作，都体现在对文件属性控制块的操作上。文件块模块是各完成文件的创建、删除和数据读写等操作。目录索引模块用于操作文件时目录索引的管理。EXT4 文件系统中，目录中的索引是以 B+ 树结构的的。目录索引模块抽象出 B+树，以完成在对文件查询、添加和修改时对 B+ 树索引的操作。各个模块的关系如图 4-3 所示。

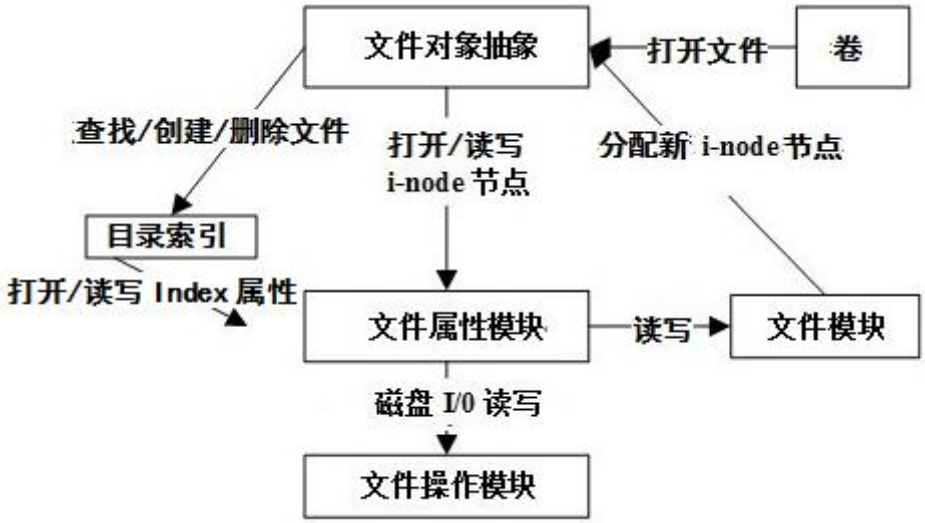

图 4-3 各模块关系图

4.2 重要 I/0 协议接口卷

EXTx 文件系统核心模块负责处理 EXTx 文件系统的具体逻辑，

Ext2AllocateVolume（）分配 volume 资源并查询 EXTx 文件系统，初始化 EFI_DISK_IO_PROTOCOL 和 EFI Block I/O Protocol 以及 cache。

4.2.1 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL

EFI_SIMPLE_FILE_SYSTEM_PROTOCOL 提供了一个最小的文件类型的访问的设备的接口。该协议只支持某些设备。

支持简单的文件系统协议的设备返回一个 EFI_FILE_PROTOCOL。该接口的唯一功能是打开文件系统卷上的根目录的句柄。一旦打开，所有的进行卷的访问都是通过使用 EFI_FILE_PROTOCOL 协议文件卷句柄文件处理。关闭所有打开的文件句柄关闭卷。如下：

```
Typedef    struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL
{
UINT64   Revision;
EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME
OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

OpenVolume 指向 OpenVolume（）。OpenVolume（）函数打开一个卷，并返回一个文件卷的根目录句柄。这个句柄是用来执行所有其他的文件 I/ O 操作。卷保持打开状态，直到所有的文件处理它的文件关闭。

如果介质的改变并且有打开的文件句柄的卷，所有的文件卷的句柄将返回 EFI_MEDIA_CHANGED。要访问新的介质上的文件，该卷必须利用 OpenVolume（）重新打开。如果新媒体是一个与 EFI_HANDLE 的设备路径为 EFI_SIMPLE_SYSTEM_PROTOCOL 提供的不同的文件系统，OpenVolume（）将返回 EFI_UNSUPPORTED。

4.2.2 EFI_DISK_IO_PROTOCOL

EFI_DISK_IO_PROTOCOL 是用于控制块 I/ O 接口。

磁盘 I/ O 功能允许 I / O 操作不需要在底层设备的数据块范围内也不

需要有对齐的要求。这是通过将数据复制去或者从内部缓冲区中，根据需要，以提供适当的情况下的块 I / O 设备的请求。优秀的写缓冲区的数据使用设备句柄 EFI_BLOCK_IO_PROTOCOL 的 FlushBlocks（）功能来刷新。

固件会自动添加一个 EFI_DISK_IO_PROTOCOL 界面到任何 EFI_BLOCK_IO_PROTOCOL 的接口。它也增加了文件系统，或逻辑块 I / O，任何 EFI_DISK_IO_PROTOCOL 接口包含任何可识别的文件系统或逻辑块 I/ O 设备的接口。实现如下：

```
typedef    struct _EFI_DISK_IO_PROTOCOL
{
UINT64   Revision;
EFI_DISK_READ    ReadDisk;
EFI_DISK_WRITE    WriteDisk;
} EFI_DISK_IO_PROTOCOL;
```

ReadDisk（）函数读取的从 BufferSize 指定的设备的字节数。所有的字节被读取，或者返回一个错误。如果设备中没有媒介，该函数返回 EFI_NO_MEDIA。如果 MediaId 不是目前的设备中的介质 ID，该函数返回 EFI_MEDIA_CHANGED。代码如下：

```
Typedef    EFI_STATUS
(EFIAPI *EFI_DISK_READ) (
IN EFI_DISK_IO_PROTOCOL     *This,
IN UINT32      MediaId,
IN UINT64      Offset,
IN UINTN       BufferSize,
OUT VOID       *Buffer
);
```

WriteDisk（）函数写入 BufferSize 设备所指定的数量的字节。所有字节写入，或者返回一个错误。如果设备中没有媒介，该函数返回 EFI_NO_MEDIA。如果 MediaId 不是目前的设备中的介质 ID，该函数返

回 EFI_MEDIA_CHANGED。实现如下：

```
typedef    EFI_STATUS
(EFIAPI *EFI_DISK_WRITE) (
IN EFI_DISK_IO_PROTOCOL     *This,
IN UINT32      MediaId,
IN UINT64      Offset,
IN UINTN       BufferSize,
IN VOID *Buffer
);
```

4.2.3 EFI Block I/O Protocol

该协议用于抽象的大容量存储设备，以允许在 EFI 引导服务环境中运行的代码访问在没有该类型设备的特别知识或控制器的情况下去管理它。函数被定义为从大容量存储设备以块级别读取和写入数据，以及在 EFI 引导服务环境来管理这样的设备。如下：

```
typedef struct_EFI_BLOCK_IO_PROTOCOL{
UINT64   Revision; EFI_BLOCK_IO_MEDIA     *Media;
EFI_BLOCK_RESET     Reset;
EFI_BLOCK_READ      ReadBlocks;
EFI_BLOCK_WRITE     WriteBlocks;
EFI_BLOCK_FLUSH     FlushBlocks;
} EFI_BLOCK_IO_PROTOCOL;
```

ReadBlocks（）从设备的块读取请求数。如果没有媒体的设备，函数返回 EFI_NO_MEDIA。如果 MediaId 是不是目前的媒体设备的 ID，该函数返回 EFI_MEDIA_CHANGED。该函数必须返回 EFI_NO_MEDIA 或 EFI_MEDIA_CHANGED，即使 LBA, BufferSize，或 Buffer 是无效的，这样调用者可以探测介质状态的变化。如下：

```
typedef EFI_STATUS
(EFIAPI *EFI_BLOCK_READ) (
IN EFI_BLOCK_IO_PROTOCOL *This,
IN UINT32     MediaId, IN EFI_LBA    LBA,
IN UINTN      BufferSize,
OUT VOID      *Buffer
);
```

WriteBlocks（）指定数量的块写入到设备中。WriteBlocks（）函数将所请求的块数写入到该设备。所有的块被写入，或者返回一个错误。

如果没有媒体的设备，在函数返回 EFI_NO_MEDIA 的。如果 MediaId 不是目前的媒体在设备的 ID，该函数返回 EFI_MEDIA_CHANGED。该函数必须返回 EFI_NO_MEDIA 或 EFI_MEDIA_CHANGED，即使 LBA, BufferSize，或 Buffer 是无效的，以致于调用者可以探测介质状态的变化。

4.3 协议接口的实现

要实现 EFI 文件协议在这里是通过使用 Volume 提供的文件系统卷信息，然后实现对 EXTx 文件系统的各种操作。协议中最主要接口的实现如下：

(1) mount 接口：文件系统初始化检测 EXTx 文件系统并 mount，将初始化 inode 节点数据数据。把磁盘上文件系统的信息读取出来。

(2)Open 接口：根据文件名，通过目录进行查询，打开文件。首先是目录文件还是普通文件，如果是目录文件还将轮询查询目录中所有文件。每一个文件对应一个相应的 direntry 其标明了文件的属性，以对文件进行读写等操作。

(3)Rea 和 Write 接口：读写接口主要根据 EFI_DISK_IO_PROTOCOL 和 EFI Block I/O Protocol 读取文件系统的内容。对文件读/写时，首先初始化 EFI_DISK_IO_PROTOCOL 和 EFI Block I/O Protocol，根据这两个协议提供的接口读取文件系统信息。

(4) GetFileSize 接口：主要用来获得文件的大小。

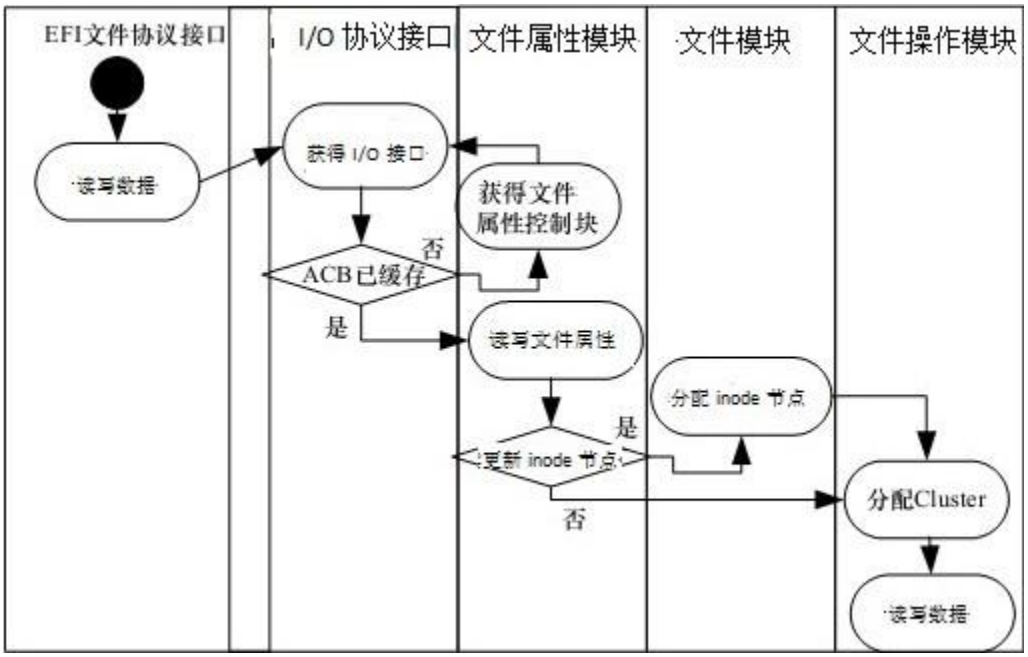

图 4-4 协议工作流程图

第 5 章 EFI BIOS 文件系统驱动设计

5.1 EFI 驱动模块

EFI 提供的驱动模型，可支持符合工业标准总线的设备和将来的架构，它能够减小可执行映像的大小和简化设备驱动的设计。这样很多复杂工作将由总线驱动完成，还有些放到了公共固件服务之中去。设备驱动需要在本驱动 image handle 上生成 Driver Binding Protocol，之后等待系统固件将驱动连接到相对应的控制器上。当连接之后，该设备驱动就会生成在控制器的 devicehandle 上的协议以抽象出控制器所支持的 I/0 操作。总线驱动会执行同样的任务，同时也会负责查找总线上的任何子控制器，并且为发现到的每个子控制器创造 device handle。

5.1.1 EFEI 驱动初始化

driver image 必须从某种媒介中加载，例如 ROM、闪存、光盘硬盘、网络线路。当 driver image 被系统找到，该 image 就会由启动服务的 LoadImage（）加载到系统的内存中去。LoadImage（）的作用就是加载 PE/COFF 格式的镜像进入到系统内存。加载成功后生成驱动的 handle，同时一个 Loaded Image Protocol 例程被挂载到该 handle 上。此刻该驱动在内存之中等待开始。

驱动被 LoadImage（）加载之后，还需启动服务的 StartImage（）进行启动，EFI 系统中所有的 EFI 应用程序和驱动都是同样的过程。遵循 EFI 驱动模型的驱动的入口程序必须遵循一些严格的规则:

1．不能对任何硬件操作，只能在自己的 image handle 上选择性安装一些协议例程。

2. 更新 Loaded Image Protocol 提供 Unload （）函数用来卸载驱动

3. 如果驱动在 ExitBootServices（）调用的时执行其他的操作，它可以通过创建一个带有通知功能的事件，当 ExitBootServices（）被调用的

时候，该通知就会被触发。包含有 Driver BindingProtocol 例程的 image handle 被称作 driver image handle。图 5-1 展示 StartImage( )被调用后 image handle 的可能配置。

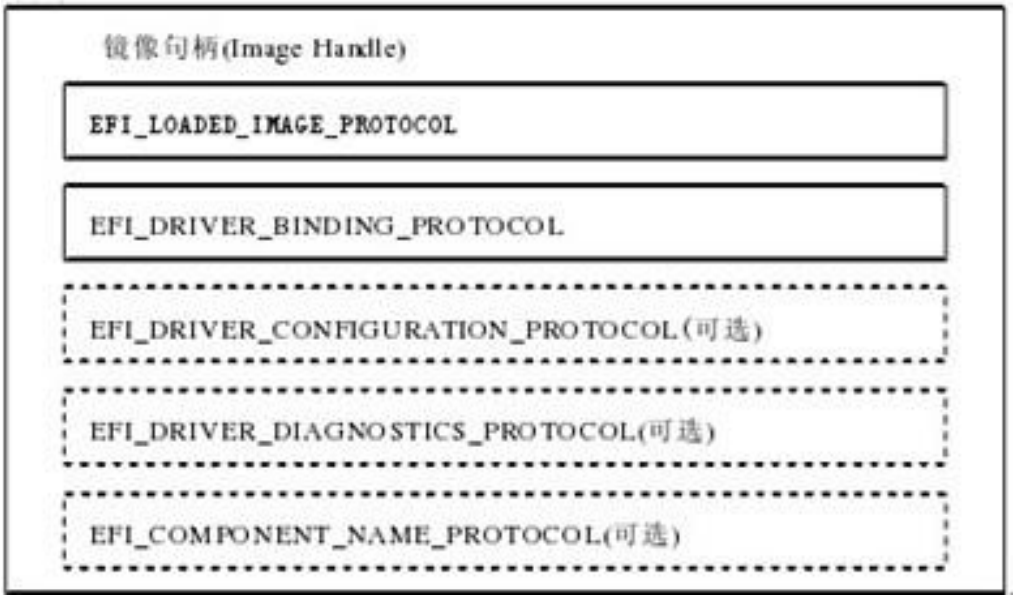

图 5-1 EFI 驱动镜像句柄

5.1.2 EFI 设备驱动

设备驱动不可以创造新的 device handle，它只能在现有的设备 handle 上安装别的协议接口。普遍的设备驱动需要把 I/O 抽象挂载到由总线驱动所生成的设备 handle 上，该 I/O 抽象用于启动 EFI 可兼容的操作系统，这些 I/O 包括 Simple Network Protocol、Simple Input、SimpleText Output、Block I/O。图 5-2 展示了一个设备驱动的 device handle 在连接前后情形。该 device handle 是 HHH 总线的子 handle，它包含了 HHH 总线支持的用于 I/O 服务的 I/O 协议。并且也包含了 HHH 总线驱动建立的 Device Path Protocol。Device Path Protocol 对于系统中物理设备的 device handle 来说必须的。对于虚拟设备的 handle 就不需要。

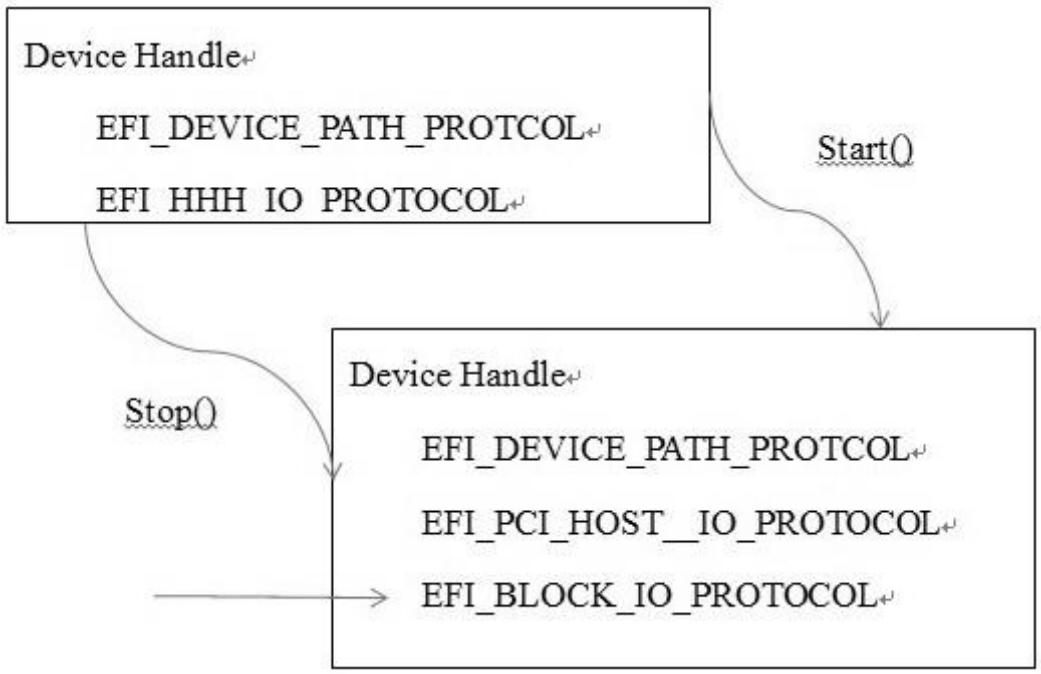

图 5-2 设备驱动连接图

图 5-2 所示的设备驱动要连接到 device handle 上的就必须在自己的 image handle 上安装 Driver Binding Protocol。 Driver Binding Protocol 包含有三个功能函数接口，分别为: Start()、Supported()、Stop()。Supported（）用于检测该驱动是不是被给定的控制器所支持的。如果驱动的 Supported（）检测通过，接下来才可以继续调用驱动的 Start（）函数将驱动连接到控制器上。Start（）函数就是将其他的 I/O 协议安装到 device handle 上。对应的 Driver Binding Protocol 中还有一个 Stop（）函数，使驱动停止管理该 device handle。这样设备驱动才能卸载 StartO 函数安装的所有协议。

在 EFI Driver Binding Protocol 的 Support（）、Start（）和 stop（）函数中， OpenProtocol（）函数用于获取得到一个协议接口。CloseProtocol（）函数用于释放一个协议接口。OpenProtocol（）和 CloseProtocol（）会致使系统固件维护的 handle 数据库更新，以此去追踪驱动和协议接口的使用。Handle 库之中存在有驱动和控制器的信息。

5.1.3 总线驱动

从 EFI 驱动模型的角度设备驱动和总线驱动实质上相同，仅有的差别是总线驱动会为子控制器创造新的 device handle。主要有两种类型的总线驱动：第一个，第一次调用 Start（）函数时总线驱动为所有的子控制器建立 handle，第二个，允许通过多次对 Start（）函数的调用来为所有子控制器建立 handle。图 5-3 展示了总线控制器在 Start（）被调用前后的的树结构

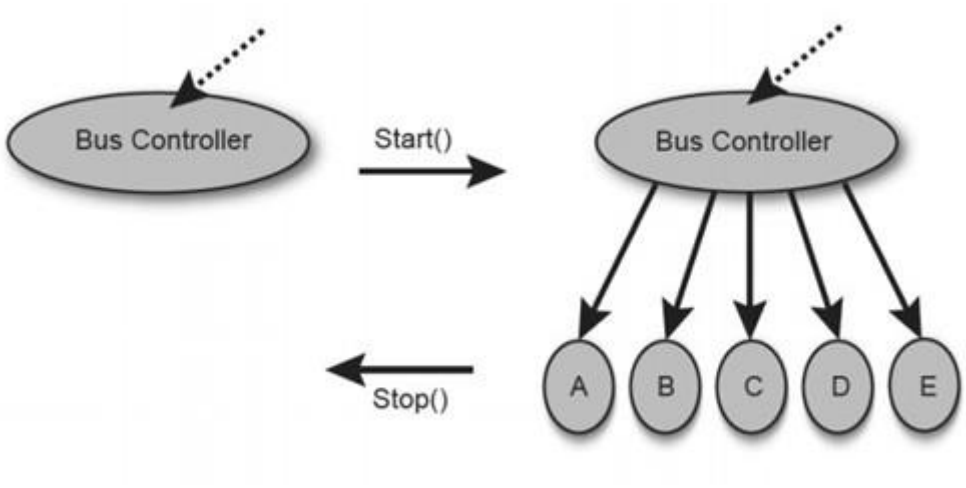

图 5-3 总线控制器驱动连接图

总线驱动要为它的每个子设备 handle 安装协议接口，同时必须为子控制器安装一个协议接口。该接口为总线服务提供了 I/O 抽象。总线驱动必须为一个代表物理设备的子 handle 安装 Device Path Protocol 例程。每个子 handle 可以安装一个 Bus Specific Driver Override Protocol 也可以不安装，Bus Specific Driver Override Protocol 在有多个驱动被连接到子控制器时才使用。ConnectController（）根据架构上定义的优先级别为控制器选择最合适的驱动。图 5-4 展示了一个例子，由 XYZ 总线驱动创建的子设备 handle 支持一个 bus specific driver override 的机制。

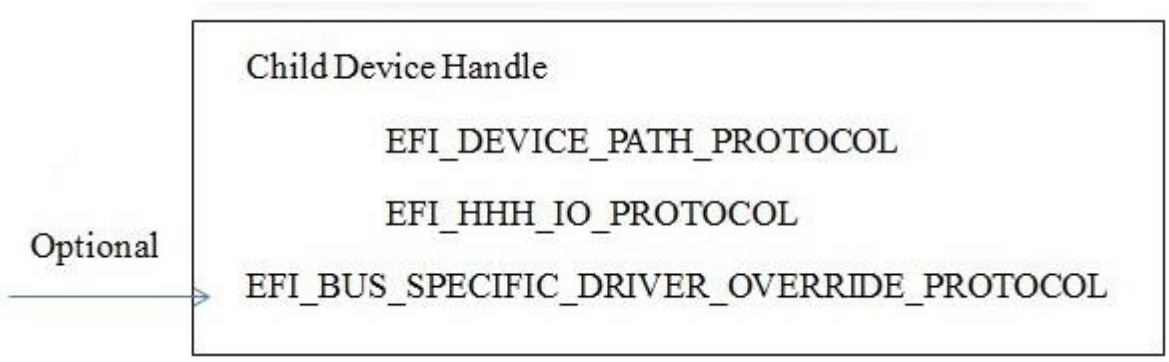

图 5-4

5.2 EXTx 文件系统驱动设计实现

EFI 文件系统驱动遵循 Driver Binding Protocol 协议规定。下面具体说明实现方法。

5.2.1 ExtDriverBinding

```
EFI_DRIVER_BINDING_PROTOCOL gExtDriverBinding = {
  ExtDriverBindingSupported,
  ExtDriverBindingStart,
  ExtDriverBindingStop,
  ……
};
```

EFI 文件系统驱动要实现的驱动绑定协议即要实现三个基本的功能函数 Supported（）、Start（）、Stop（）。 Supported（） 函数会评估传入的句柄参数 ControlerHandle。检查这个驱动是否可以被添加到 ControllerHandle。ControllerHandle 必须支持 Disk IO and Block IO 协议，如果支持返回成功，否则返回不支持。

5.2.2 DriverBindingSupported

下面给出 Supported（）具体的实现代码

```
EFI_STATUS
EFIAPI
ExtDriverBindingSupported (……)
{
  ……
  EFI_DISK_IO_PROTOCOL   *DiskIo;
  EFI_BLOCK_IO_PROTOCOL *BlockIo;
  Status = gBS->OpenProtocol (   ……   );
  gBS->CloseProtocol (   ……   )
  ……
}
```

(1)驱动中，用 OpenProtocol（）打开所有必需的协议。如果该驱动允许与其它驱动共享被打开的协议，那么它应该使用 EFI_OPEN_PROTOCOL_BY_RIVER 属性来 OpenProtocol()(OpenProtocol（）打开协议接口的一种方式)。如果该驱动不允许与其它驱动共享被打开的协议,那么其应该使用 EFI_OPEN_PROTOCOL_BY_DRIVER 与 EFI_OPEN_PROTOCOL_EXCLUSIVE 的属性来 OpenProtocol（）。设备驱动中使用 OpenProtocol（）的属性必须和 Supported（）中使用的 OpenProtocol（）属性相同。

(2)如果在步骤 1 中对 OpenProtocol（）的任何调用返回了一个错误，那么用 CloseProtocol（）关掉步骤 1 中打开的所有协议，并且返回一个状态码，该状态码来自于返回错误的对 OpenProtocol（）的调用

(3)忽略参数 RemainingDevicePath

(4)初始化由 ControllerHandle 指定的设备。如果一个错误出现，用 CloseProtocol（）关闭掉在步骤 1 中打开的所有协议，并且返回 EFI_DEVICE_ERROR。

(5)分配和初始化所有的数据结构，这些数据结构是该驱动去管理 ControllerHandle 指定的设备所需要的，这将会包括公共协议的空间和与 ControllerHandle 相关的任何的另外的私有数据结构。如果分配资源时有错误出现，那么用 CloseProtocol（）关闭掉在步骤 1 中打开的所有协议，并且返回 EFI_OUT_OF_ RESOURS。

(6)使用 InstallProtocollnterface（）在 ControllerHandle 上安装所有的协议接口。如果一个错误出现，用 CloseProtocol（）关闭掉在步骤 1 中打开的所有协议，并且从 InstallProtocolInterface（）返回错误值。

5.2.3 DriverBindingStart

ExtDriverBindingStart（）加载文件需要的协议。

```
EFI_STATUS
EFIAPI
Ext2DriverBindingStart ( …… )
{ ……
Status = ExtAllocateVolume (ControllerHandle, DiskIo, BlockIo);
Status = gBS->OpenProtocol (
                    ……
                    &gEfiBlockIoProtocolGuid,
                    //加载块协议
                    ……    );
Status = gBS->OpenProtocol (
                      ……
                      &gEfiSimpleFileSystemProtocolGuid,
```

```
//加载简单文件系统
…… );
…… }
```

(1)设备驱动中，用 OpenProtocol（）打开所有必需的协议。如果该驱动允许与其它驱动共享被打开的协议，那么它应该使用 EFI_OPEN_PROTOCOL_BY_RIVER 属性来 OpenProtocol（）(OpenProtocol（）打开协议接口的一种方式)。如果该驱动不允许与其它驱动共享被打开的协议，那么其应该使用 EFI_OPEN_PROTOCOL_BY_DRIVER 与 EFI_OPEN_PROTOCOL_EXCLUSIVE 的属性来 OpenProtocol（）)。设备驱动中使用 OpenProtocol（）的属性必须和 Supported（）中使用的 OpenProtocol（）属性相同。

(2)如果在步骤 1 中对 OpenProtocol（）的任何调用返回了一个错误，那么用 CloseProtocol（）关掉步骤 1 中打开的所有协议，并且返回一个状态码，该状态码来自于返回错误的对 OpenProtocol（）的调用。

(3)忽略参数 RemainingDevicePath。

(4)初始化由 ControllerHandle 指定的设备。如果一个错误出现，用 CloseProtocol（）关闭掉在步骤 1 中打开的所有协议，并且返回 EFI_DEVICE_ERROR。

(5)查找由 ControllerHandle 指定的总线控制器的所有子设备 EfiBlockIo 和 EfiSimpleFileSystem。

(6)为由 ControllerHandle 指定的总线控制器的所有子设备分配资源。

(7)循环 ControllerHandle 的每一个子设备。

(8)分配和初始化该驱动管理子设备所需要的所有数据结构，这将包括公共协议空间和与子设备相关的任何另外的私有数据结构。在分配资源时如果出现错误，那么用 CloseProtocol（）关掉在步骤 1 中打开的所有的协议，并且返回 EFI_OUT_OF_RESOURCES。

(9)如果总线驱动为子设备创造设备路径，那么基于 ControllerHandle 的设备路径为子设备创造设备路径。

(10)初始化子设备。如果出现错误，那么用 CloseProtocol（）关闭在步骤 1 中打开的所有的协议，并且返回 EFI_DEVICE_ERROR。

(11)为子设备创造新的 handle，并且为子设备安装协议接口。这可能包括 EFI_DEVICE_PATH_PROTOCOL。

(12)用 EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER 的属性代表子设备调用 OpenProtocol（）。

(13)返回 EFI SUCCESS。

5.2.4 ExtDriverBindingStop

ExtDriverBindingStop（）与 ExtDriverBindingStart（）正好相反，断开驱动器控制器和释放任何资源分配中的服务。实现如下：

```
EFI_STATUS
EFIAPI
Ext2DriverBindingStop ( …… )
{ ……
```

```
extVolume = EXTVOLUME_FROM_VOL_INTERFACE (FileSystem);
Status = ExtAbandonVolume1 (extVolume);
//释放分配的卷

Status = gBS->CloseProtocol ( …… );
//释放加载的协议
…… }
```

第 6 章 总结与展望

6.1 总 结

本文论述了全自动化生化分析仪引导系统 EFI BIOS 平台的 EXTx 文件系统的设计。具体的工作如下：

(1)详细论述了全自动生化分仪发展历史、国内外研究的现状，分析了其工作原理。

(2)重点介绍了 EXTx 文件系统数据结构。

(3)深入研究了 EFI 标准规范，并详细的分析说明了 EFI 的各个阶段。其中包括了 EFI 的组织结构、运行机制。

(4)根据 EFI BIOS 平台的特点和 EXTx 文件系统的特点提出了本设计方案，最终实现了文件的拷贝、删除、移动和信息查看。

6.2 展望

由于时间的紧迫和开发设备等条件限制，本设计方案还是存在不足和一些可以改进的地方。在我之后的学习工作中，我还将要去做进一步的研究和探索。

参考文献

[1] UEFI Forum.Unificd Extensible Firmwarc Interface Spccification, Version 2.0, 2006

[2] Intel Corporation. Intel Platform Innovation Framework for EFI Architecture Spccification, Version 0.9, September 16, 2003

[3] A. Parlraj, R. Roy, and T.Kailath. Estimation of signal Parameters via Rotational Invariance Techniques- ESPRIT. Proc.19th Asilomar Conf, Pacific Grove, CA, 1985,pp.83-89.

[4] Fcng-Li Lian, James R.Moync, Dawn M Tilbury.Pcrfbrmance Evaluation of ControlNctworks, IEEE Control System Magazine. 2001-2.

[5] Intel CorporationEFl Drivcr Model, May 20, 2001

[6] 潘登，刘光明EFI结构分析及Driver开发.计算机工程与科学，2006,(02).

[7] Intel Corporation.IA-32 Intel Software Developer's Volume 1: Base Architecture, 2003.

[8] Intel Corporation. Intel Platform Innovation Frameworkfor EFI Firmware Volume Specification, version 0.9..

[9] Intel Corporation.Intcl Platform Innovaiton Framework for EFI Boot Script Spcc, version 0.9.

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

[11] Intel Corporation, Intel Platform Innovation Framework for EFI Architecture Specification. Version 0.9, 2003.

[12] Intel Corporation, Intel Platform Innovation Framework for EFI Firmware Volume Specification. Version 0.9, September 16, 2003.

[13] Intel Corporation, Intel Platform Innovation Framework for EFI Firmware File System Specification. Version 0.9, September 16, 2003.

[14] UEFI, Unified Extensible Firmware Interface Specification . Version 2.3,2009-02-18.

[15] UEFI, Platform Initialization Specification. Version 1.2, 2009-05-13

[16] Vincent Zimmer, Michael Rothman, and Robert Hale, Beyond BIOS.

[17] Implementing the Unified Extensible Firmware Interface with Intel'sFramework, 2007-06

[18] Intel Corporation, Driver Writer's for UEFI 2.0 Revision 0.95. April 10, 2007.

[19] Intel Corporation and IBM, Trusted Platforms UEFI,PI andTCG-based firmware. September, 2009.

[20] Bird T.Right-sizing Linux Selective Reduction of Linux for Embedded Devices. Embedded System Conference, Chicago. 1999.

[21] Lars Wirzenius. Linux System Administrator's Guide Version 0.6. LDP Document,Nov. 1997.

[22] C.M.Krishna, Kang G.Shin 《Real-Time Systems》清华大学出版社，2001，pp185-221.

[23] V. S. Kedia and B. Chandna,A new algorithm for 2-D DOA algorithm, Signal Process. , 1997,vol.60, no.3,pp.325-332.

[24] Operating Systems-Internals and Design Principles, Third Edition, William Stalling Prentice Hall International, Inc. 1998.

[25] Operating Systems-Design and Implementation, Second Edition, Andrew S.Tanenbaum, Albert S. Woodhull, PrenticeHall International, Inc. 1997..

[26] Fred Butzen, Christopher Hiltonk. The Linux Networking. IDG Books Worldwide, Inc. 1998.

[27] Ba Barkakati. The Linux Secrets. IDG Books Worldwide, Inc. 1996.

[28] Maurice J. Bach. The Design of the UNIX Operating System. Prentice Hall. 1986.

[29] 谭良,周明天.可信操作系统研究.计算机应用研究， 2007,

24(12):10-114.
[30] 陈文钦.BIOS研发技术剖析〔M〕.北京:清华大学出版社，2001.
[31] 黄丰隆.计算机系统与接口原理详解[M].北京:学苑出版社，1993.
[32] 刘乐善，叶济中，叶永坚.微型计算机接口技术原理及应用[M].武汉:华中理工大学出版社，1996.
[33] 周明德.微型计算机硬件软件及其应用(修订版)[M].北京:清华大学出版社，1989..
[34] Intel.IntelItaniumArchitectureSoftware Developer' s Manual [M].2002..
[35] Intel.Extensible Firmware Interface Specification [S].2002..
[36] Intel.IntelArchitectureSoftwareDeveloper'sManual:SystemProgramming [DB].2006..
[37] Intel.Intel Platform Innovation Framework for EFI Pre-EFIInitialization Core Interface Specification [S].2004..
[38] Intel.Intel Platform Innovation Framework for EFI Driver Execution Environment Core Interface Specification [S].2004..
[39] Intel.Intel Platform Innovation Framework for EFI System Management Mode Core Interface Specification [S] .2003..
[40] Intel.Intel Platform Innovation Framework for EFI CPU I/O Protocol Specification [S].2003.

攻读硕士学位期间已撰写和发表的论文

[1]郭旭，冉全，嵌入式 linux 下 USB 摄像头驱动程序开发，软件导刊，Vol. 12, No. 1, 2013.

致 谢

不知不觉中，三年的光阴已悄然逝去。此间发生的一幕幕像是昨天发生的一样，那些曾经在生活和学习上给过我帮助的人一一浮现在我的眼前。

我的导师冉全，读研期间最重要的老师，陪我走过这一生中最难忘的三年。冉老师才思敏捷、治学严谨，在学习和生活上给了我极大的帮助和鼓励，使我可以坦然面对学习上、生活上以及未来的逆境。他为人和蔼，待人谦逊，给我留下了一生中最难忘的记忆。我想用 “知常容”来形容冉老师从研究工作到生活的状态。在此向冉老师致以最衷心的感谢。

我要感谢学院所有老师给我在学习和生活上带来的巨大帮助。

感谢同班同学在生活和学习上给了我很大的支持，陪我度过愉快的三年。

我还要感谢我的父母，他们给予我的爱和支持，是我终生难以回报。

最后感谢百忙之中抽出时间评阅和评议该论文的各位专家和学者，感谢你们为审阅本文所付出的辛勤劳动。