

Beyond BIOS

中文预览版

WWW.BIOSREN.COM

Rev 0.2

声明

本资料仅供学习参考之用，请在学习参考完之后 **24** 小时内删除，未经授权者不得进行任何复制、转载、传播、出版等非法之使用，否则后果自负。

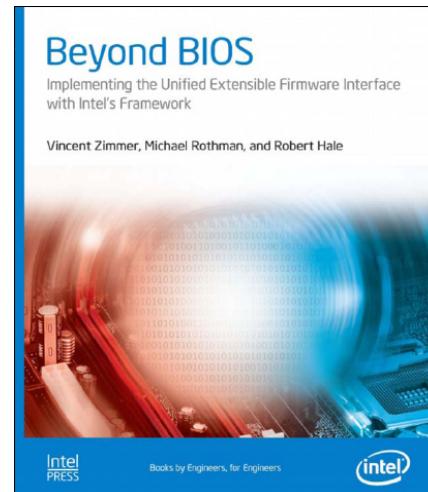
本资料内容来源于网络，由热心网友，英语爱好者协作而成，不保证其中无纰漏、无瑕疵，甚至有些地方完全错误，请阅者体谅。

本资料内容尚不完整，属预览版本，尚有第一章、第五章、第十章部分内容待完成。

若要获得完整版本的学习书籍，请购买英文正版书《**Beyond BIOS –Implementing the Unified Extensible Firmware Interface with Intel's Framework**》作者：**Vincent Zimmer, Michael Rothman, and Robert Hale**

出版：Intel Press www.intel.com/intelpress

感谢网友 Perseverance、xtdumpling、Libeili、Rex、Samli、snowcuso、charlie、Xtjiayou、LightSeed、Strength、habou、yuxi1127、xueyin000、luciferli、wyz198461、mayfang、edmoon、gbyHW、ella、Zonel、Kaeshine、gssxd、dingchao0205、passionapple、wendyw00、freasy、biosye、Delphi、Edwin、Rainbow、gikidy、hitpir、debugcode、manliangpai、wj025、shen 等为此付出的艰辛努力，感谢其他在这里没列出名字的网友的默默支持，希望能在完整版本的感谢名单中看到所有支持和付出过努力的网友的名字。



Giantt.Peng
2010.5.10

第一章

概述

来这个更好的地方找我，装作一只小白兔……

—Harlan Ellison

普通用户并不知道一台现代计算机其幕后的工作原理。设计使然，开发者耗费大量的时间、思考和精力将组成计算机的零散部件整合成一个完全合乎逻辑的整体。较计算机启动程序而言，这种整合，甚至对于那些编写计算机其他程序的人，都更加令人费解。

可启动固件的复杂性与日俱增，现在已经发展到完全可以与底层硬件及其加载的软件的复杂性并驾齐驱了。最新的可启动固件的架构是可扩展固件接口（Extensible Firmware Interface，以下亦简称 EFI），EFI 定义了固件与操作系统以及 option ROMs 之间的接口。实现 EFI 的软件架构可以有无数种。本书首先描述了 EFI 的各组成部分及其特性，然后详述了一个功能丰富的 EFI 的架构，现在被称之为英特尔® EFI 平台创新架构（Intel® Platform Innovation Framework for EFI），或者简单的称作 Framework。

EFI 与 Framework 均是衍生自现代软件开发标准的概念与技术。虽然 EFI 与 Framework 并没有定义一个操作系统，但从任何一个传统的视角来看，它们都借用了许多该领域的概念和想法。EFI 以及 Tiano 的设计方法反映了软件工程的现状和趋势，同样也考虑了嵌入式领域的实际经验。而这两条路线都需要对之前可启动固件架构获取的经验有深入的理解。

历史

最早的电脑没有可启动固件。用户不得不手动操作电脑前面板上的开关来输入一个启动程序。这个过程非常缓慢、费力，并且易于出错。一些电脑使用很复杂的指令集（想想输入这些指令本身的工作就令人发疯 译者），所以他们的设计者们简化了这一流程，比如从一个纸带上来读入程序。这个时代普遍使用一种独特的“自动化”设备——一块在每个开关处都有切割轨迹的胶合板。在使用控制板沿着木板直线滑动时，切割轨迹使之先下滑再上滑，控制开关会在正确的位置动作。对于操作员来说开发更小的程序启动系统来取得竞争优势是非常常见的做法，更普遍的是采用带有几个转换开关的程序来启动系统。

之后，最初的开关位置被板卡上的二极管所代替，这种装置看起来像是一个所谓的外围设备。只有一个从二极管“存储”复制到 RAM 里的小的程序可以被植入。

随着小型电脑的发展，典型的不超过 256 字节长的微型程序被存储在最新的可用的 ROM 里面，并可以从少数设备上启动。如果操作员想从另外一个设备启动，比如从一个纸带读出器而不是从磁鼓或者磁盘上，一般情况下他只能更换 ROM。

一旦加载之后，操作系统基本上不再使用这些 ROM 里的代码。相反地大多数操作系统都带有可启动设备的驱动，因此可以立即初始化这些设备。

固件作为硬件抽象层

这个领域的一个主要转折是在 70 年代早期由 Gary Kildall 以及其他一些人界定的，而 Gary 后来成为了著名的操作系统 CP/M 的发起人，同时也是 Digital Research 的创建者。（也正是这个人，以其傲慢的态度迫使 IBM 选择了微软，浪费了一次成为比尔盖茨的机会—译者）他提出可启动固件是计算机硬件与操作系统之间的一个抽象层。该抽象层用于启动操作系统以及与系统基本外设进行简单通讯。可启动固件试图解决一个操作系统的窘境：如何得到从外围设备所加载的驱动。

PC 机被开发出来的时候，开发者们应用了一个与 Gary 等人提出的可启动固件类似的扩展概念，叫做基本输入输出系统，简写为 BIOS (IBM 公司, 1985)。BIOS 主要包含两个部分：上电自检部分 (POST) 以及运行时库 (runtime)，这一部分主要是为早期的 PC 操作系统提供基本的运行服务。

BIOS 的基本目标是自检以及初始化系统，检测输入输出设备及可启动设备，输入设备一般指键盘，输出设备一般指图形显示器，而可启动设备一般指的是磁盘。接下来 BIOS 会从磁盘读取操作系统最开始的 512 字节的启动程序来启动操作系统并将控制权传递给它。之后 BIOS 将对其所能识别的设备继续发挥其硬件抽象的作用。

随着时间的推移，这一想法中的缺陷日益彰显。BIOS 提供的抽象层是极其原始且不同步的。由于缺乏同步性，BIOS 抽象层不得不采用轮询 (polled) 而不是中断的策略。进一步说，由抽象层所假定的处理器模式假设处理器工作于实模式，这并不是操作系统运行时所使用的模式。上述类似的抽象，事实上一直在使用，但仅仅在操作系统自举的时候使用。

这一模型还存在其他扩展性问题：当有新的技术出现的时候，其升级和扩展能力有限。比如，视频抽象 (INT 10H) 依靠枚举的模式来选择图形分辨率。而大多数早期的分辨率，比如 320X200，很快就过时了。IBM 发明这一接口时，一开始就考虑到该图形模式的扩展性。但由于使用者的分散性，图形模式的扩展由标准制定组织或者公司群体来管理，这些组织只是简单的扩展原有模式而没有考虑到它们的实际意义，比如，对其他的产品的重要性。

这一软件模型允许实现许多底层架构且彼此相互兼容。架构的实现上从单一到多元的软件组件均有。基本思路是：几乎所有的底层架构都是汇编语言实现，原因如下：

- 传统
- 空间限制
- 并非所有的编译器都支持大多数处理器的存储模式，也就是 big real mode
- 非高级语言描述的软件接口必须工作于有限的堆栈空间

需要注意的是，很多近期的 BIOS 实现都使用 C 语言来创建一些特定的组件，主要是 Setup 程序。典型的 BIOS 开发工具同样也是用高级语言实现的，包括 C, C++ 以及 Perl。

Option ROMs

PC BIOS 为固件中的硬件抽象提供了一个很重要扩展概念：Option ROM。Option ROM 是存在于扩展卡中的 BIOS 扩展部分。尽管当时不是如此表述，但 Option ROM 却提供了与操作系统中硬件驱动程序一样的功能：允许基础软件访问其无法直接访问的外设。

PC BIOS Option ROM 的实现存在许多缺陷，包括：

- Option ROM 获取系统内存或者其他资源并没有一个标准。资源利用同样没用很好的规定，尤其是没有定义在调用过程中所允许的最大堆栈空间。
- 并没有一个明确的方法来规定 Option ROM 如何将其特性加入到硬件抽象层。举个例子，它并没有为 SCSI 卡定义相关特性，从而可以将 SCSI 设备驱动加入到系统硬件抽象层。由此导致 Option ROM 无法使用而利用标准的 INT 13 磁盘中断来实现。
- 没有标准方法来控制由 Option ROM 管理的和主板上自带的设备之间的启动顺序。
- 没有一个进入 Option ROM 配置程序的通用方法。虽然许多 Option ROM 都需要进行配置，但远程访问 Option ROM 的配置信息可不是一件容易的事情，因为由 VBIOS 定义的底层的视频接口需要直接的硬件访问才能达到一个可以接受的性能。

主板硬件初始化

POST 的出现，象征着硬件复位时只需更少的初始化步骤的趋势。软件开始初始化系统硬件，这种办法的几点优势随着时间的推移变得更加突出：

- 硬件，尤其是集成电路，可以通过允许平台固件按需求初始化其配置来使之通用。
- 相当一部分硬件的默认值都非常复杂，其初始化易于出错、费时，且在初始化时占用大量的空间。
- 已初始化的硬件配置需要十分昂贵的硬件变更来修复问题。而固件最多只需要更新一个 ROM，这一方法更为经济。这种情况下固件已经成为了修复芯片组硬件缺陷的一个主要手段。

在早期的 PC 以及 AT 系统上，POST 的主要功能并不是初始化，而是自检。随着电路器件越来越稳定，集成度越来越高，相应的对设备进行检测的必要性逐渐降低了，但并没有完全消失。

而另一些需求则随着时代的变更，开始逐渐添加到 BIOS 最大的一部分中了。举个例子，BIOS 负责加载 CPU 的 Microcodes，同时开始要负责描述系统内无法被操作系统所探测到的硬件的属性。

1980 年生产第一台 PC 时，主板以及几乎所有的外设卡都有一些 DIP 开关组，你可以用这些 DIP 开关组来配置这些硬件设备。配置的内容则从基本的操作，诸如 I/O 资源分配，直到描述特殊的设备特性，比如主板的内存大小，网卡的 MAC 地址等，以及其他类似的东西。那个时候的系统配置是十分困难的，因为设置板卡的 I/O 范围及中断很容易引起冲突。

随着板载的二极管成本的显著降低，对板卡以及板载设备的软件配置就变得可能了。总线现在已经可以枚举了；软件系统，包括 BIOS 在内，都可以置于嵌入式设备中，大多数情况下都可以支配并满足他们的资源需求。

从长远的角度看其趋势是不挂载于串行总线的外围设备并没有使用并行设备所使用的那些传统资源。1394 和 USB 是一些这样的例子，一个串行总线可以使用相当少的同一硬件资源来支持很多 USB 设备。

在一些高端服务器上，总线的初始化过程变得极其复杂，这一复杂性是由需要定位所有可能输入，输出以及启动设备而带来的。

(未完，剩余 11 页)

www.BIOSREN.COM

第二章

基本的 EFI 架构

我相信标准，每个人都应该有
---乔治·莫罗

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，中央处理单元（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 OS loaders，诊断程序，和一些系统的应用软件执行及相通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。

理解 EFI 应用程序和驱动的基础是一些 EFI 的概念，这个概念在 EFI1.1 规范中定义。假设你现在是刚接触 EFI，下面的描述将为你解释一小部分关键的 EFI 概念，这个概念应该时刻在你的脑海中以便于你更好的学习 EFI 规范。

- 基于 EFI 固件所管理的对象---用来管理系统的状态，包括 I/O 设备，内存，和事件。 ■ EFI System Table --- 用来与系统交互所用到的主要的数据信息和函数调用所构成数据结构表。
- Handle 数据库和 Protocols---已经注册的可被调用的一些方法。
- EFI images --- 代码中使用的具有可执行特性的二进制代码集合
- Events --- 软件用来触发的事件和响应其他事件活动一种方式
- Device paths --- 用来描述硬件实体的一组数据结构，例如总线，分区，一个格式化过的磁上面 EFI image 的文件名。

基于 EFI 固件所管理的对象

几个不同类型的对像可以通过 EFI 提供的服务程序来管理，一些 EFI 驱动可能需要访问环境变量，但是大多数驱动是不需要这样的。事实上很少的 EFI 驱动需要用到计数器，看门狗或者实时的时钟信号。EFI 的 System Table 才是最重要的数据结构，因为它可以提供访问 EFI 本身提供的所有服务程序和所有其他描述平台配置信息的数据结构。

EFI System Table

EFI System Table 是 EFI 中最重要的数据结构，EFI System Table 的指针被当作入口地址的一部分参数传递给每个驱动和应用程序。从这个数据结构，EFI 的可执行 Image 能得到系统的配置信息和丰富的 EFI 服务程序，包括如下：

- EFI Boot Services

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

- EFI Runtime Services
- Protocol Services

EFI Boot Service 和 EFI Runtime Services 是通过各自的表来访问的，这两个表的入口地址存在于 EFI System Table，这两个表里面可以使用的 Services 的个数和类型在每个 EFI 版本的规范是固定的，EFI Boot Service 和 EFI Runtime Services 在 EFI1.10 规范中定义，在 EFI1.10 第四章描述了这些 EFI 驱动组成的服务程序的常规用法。

Protocol services 是一群 GUID 命名的函数和数据的集合，GUID 是一个 16 个字节的唯一的数据，在 EFI1.10 规范的附录 A 中有定义。比较典型的，Protocol services 用来提供设备的软件抽象，例如输入输出控制台，磁盘和网络设备，然而他们能够把平台上很多可用的设备抽象成通用服务程序。Protocols 是 EFI 固件扩展功能的机制，EFI1.10 规范定义了超过 30 种不同的 Protocols，和 EFI 固件的不同用法，EFI 驱动可以产生更多的 Protocols 来扩展平台的功能。

Handle 数据库

Handle 数据库由 handles 和 protocols 组成，handles 由一个或多个 protocols 组成，protocols 则是由 GUID 来命名的数据结构体，这个数据结构体可能是空，可能包含数据，可能包含服务程序，或者同时包括数据和服务程序。在 EFI 的初始化中，系统固件，EFI 驱动和 EFI 应用程序创建 handles，并为每个 handle 挂上一个或多个 protocols。在 handle 数据库中信息是全局的，而且能被任何一个可执行的 EFI Image 访问。

Handle 数据库是 EFI 固件需要维护的最重要的对象库。这个 handle 数据库是所有 EFI handles 的列表，每个 EFI handle 由系统固件分配一个唯一的 handle 编号。这个唯一的编号就是 handle 数据库中入口的“钥匙”，每个 handle 数据库中入口处都是一个或多个 protocol 的集合，挂到 EFI handle 上面的 protocol 类型决定了这个 handle 的类型。一个 EFI handle 可以体现下面这样的类型组件：

- 可执行的 Images，例如 EFI Driver 和 EFI 应用软件。
- 设备，例如网络控制器和硬盘分区
- EFI 服务程序，例如 EFI 解压缩和 EBC 虚拟机

下面的图 2.1 展示了一部分的 handle 数据库。除了 handles 和 protocols 以外，每个 protocol 还有一个对象的列表，这个列表用来跟踪哪些 agents 正在使用哪些 protocols。这个信息对于 EFI 驱动的运行来讲是非常关键的，因为正是通过这样的信息来使 EFI 驱动被安全的装载，开始，停止，和卸载时没有任何的资源冲突。

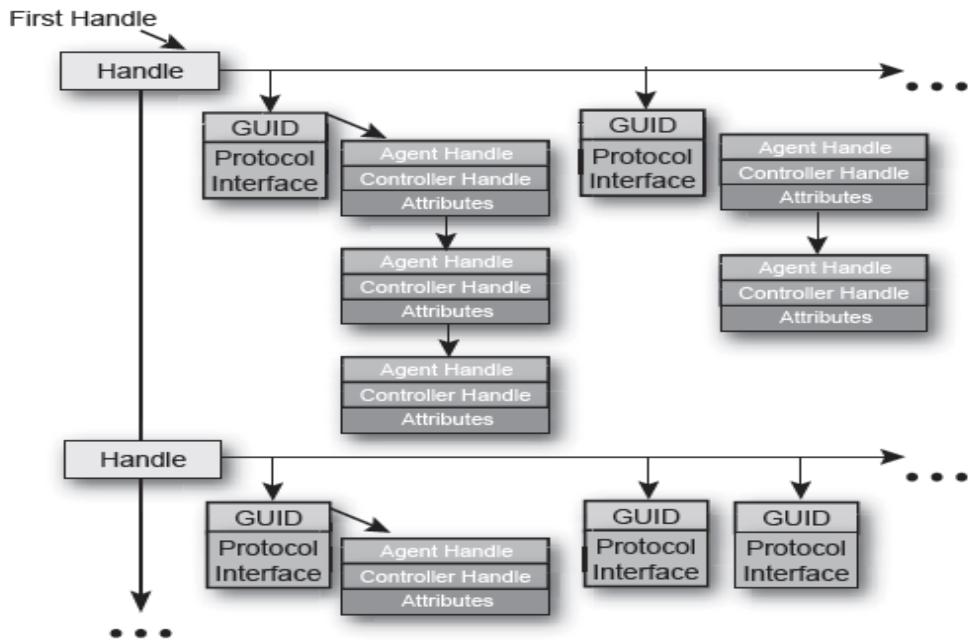
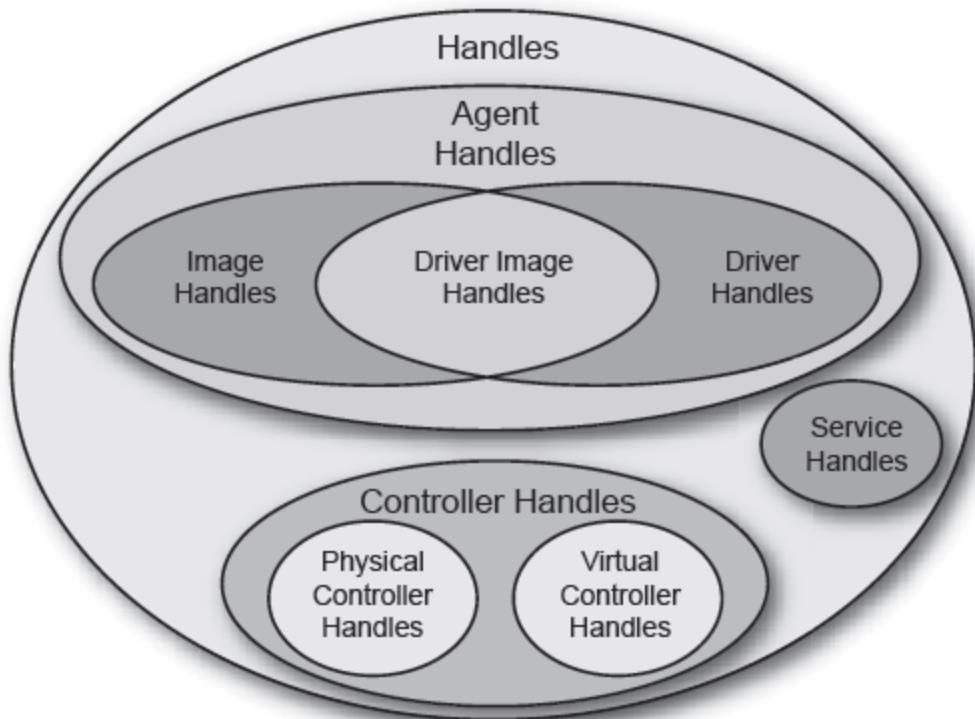
**Figure 2.1** Handle Database

图 2.2 展示了在 handle 数据库中可能出现的不同类型的 handle 以及各种不同 handle 类型之间的关系，所有的 handle 都存在于同一个 handle 数据库，每个 handle 类型的区别由 handle 下面的 protocol 的类型决定。在一个操作系统环境下的文件系统 handles，在一个对话期间是固定的，但是他们的值是不定的。与此类似的，在 C 语言库中，从一个 fopen 函数中返回的 handle，在不同的进程或在同一进程的不同调用中，不需要提供一个特定的值。Handle 只是一个用来管理状态的临时值。

**Figure 2.2** Handle Types

Protocols

EFI 的可扩展特性可以为 Protocol 提供很大限度的组合, EFI 驱动和 EFI Protocol 有的时候比较容易混淆。虽然他们的关系很密切, 但是他们显然是不同的东西, EFI 驱动是一个可执行的 EFI Image, 通过同时安装多种 protocol 和多种 handle 来实现它的功能。

EFI protocol 是一些函数指针和数据结构体或者规范定义的 APIs 的一个集合, 最少来讲, GUID 必须要被定义, 我们可以认为这个 GUID 是 Protocol 的真实名字, 启动服务程序, 例如 LocateProtocol 就是用这个 GUID 来在 handle 数据库中查找它所需要的 Protocol。Protocol 通常会包含一些函数和数据结构体, 我们称之为 Protocol 接口结构体, 下面的代码依次展示了在 EFI1.10 规范 9.6 章节中定义的 Protocol 例子, 注意看里面两个函数和一个数据区是怎么定义的。

Sample GUID

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

Protocol Interface Structure

```
typedef struct _EFI_COMPONENT_NAME_PROTOCOL {
    EFI_COMPONENT_NAME_GET_DRIVER_NAME
```

GetDriverName:

声明: 本资料仅供学习参考用, 未经授权, 不得进行任何复制、转载、传播、出版等非法之用途, 请在学习后 24 小时内删除, 否则后果自负。

```

EFI_COMPONENT_NAME_GET_CONTROLLER_NAME
GetControllerName:
CHAR8
*SupportedLanguages:
} EFI_COMPONENT_NAME_PROTOCOL;

```

图片 2.3 展示了在 handle 数据库中的一个 handle 和 protocol, 这个 handle 和 protocol 是由一个 EFI 驱动产生的, Protocol 由 GUID 和 Protocol 的接口结构体组成。很多时候, Protocol 还有一些私有数据需要产生这个 Protocol 的 EFI driver 去维护。Protocol 的接口结构体本身只是简单包含了指向 Protocol 具体函数的指针, Protocol 函数本身是被包含在 EFI 驱动里面的, 单个 EFI 驱动能够产生一个或多个 Protocol, 这取决于这个驱动本身的复杂性。

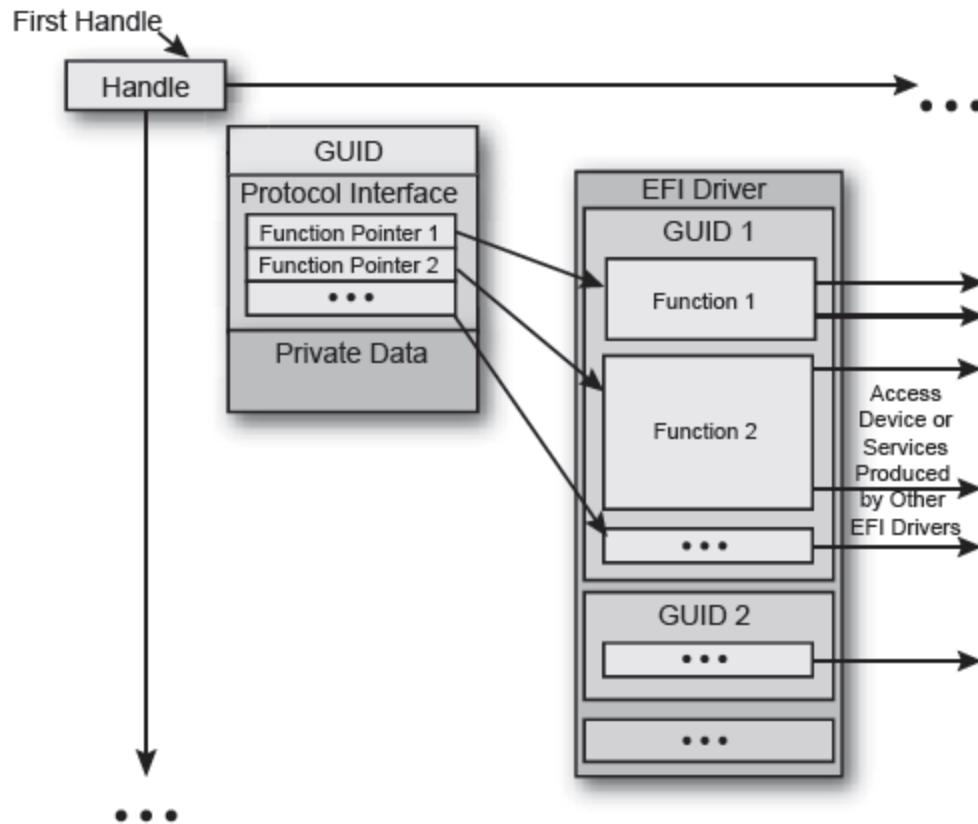


Figure 2.3 Construction of a Protocol

不是所有的 Protocols 在 EFI1.10 规范中都有定义，在开发包（EDK）中就有很多这样的 Protocols，它们提供了非常广阔的功能扩展，在特定应用开发中会用得到，但是这些 Protocols 没有在 EFI1.10 的规范中定义，应为它们不会提供启动 OS 和写一个驱动所必需外部接口。正是新的 Protocol 的产生才能使基于 EFI 的系统能够随着时间的推移而得到扩展，因为新的设备，新的总线，新的技术的会随之不断的出现。

- Varstore -- 对 EFI 统一的二进制存储对象抽象的接口

- ConIn--提供字符输入的服务程序
- ConOut--提供字符输出的服务程序
- StdErr-- 提供错误信息输出的服务程序
- PrimaryConIn--一级视图输入控制台
- VgaMiniPort--提供视频图形矩阵输出的服务程序。
- UsbAtapi-- 在 USB 总线上抽象区块访问的服务程序

EFI 应用程序工具包也包含了能够在一些平台找到的 EFI Protocols

- PPP Deamon--点对点协议驱动
- Ramdisk -- 在随机存取存储器 (RAM) 缓冲里面的文件系统
- TCP/IP--传输控制/网络通讯协定

OS loader 和驱动不应该依赖上面这些 Protocols, 因为这些 Protocols 不能被保证在每个 EFI 兼容的系统中存在。OS loader 和驱动应该只依赖于能在 EFI1.10 规范中找到的以及平台设计指定需要的 Protocols, 例如 64 位服务器设计应用指南。

EFI 的可扩展的特性允许每个不同平台开发者可以增加专门的 Protocols。使用这些 Protocols, 能够扩展 EFI 的能力并且访问独有的设备和接口, 同时和其余的 EFI 体系协调工作。

因为 Protocol 是以 GUID 来“命名”的, 所以不应该有 Protocols 同名。在为一个新的 Protocol 以 GUID 来命名的必须非常小心, EFI 本身假设一个 GUID 只会用到一个 Protocol 上。剪切和粘贴或者手动修改一个现有的 GUID 会使重复 GUID 的出现成为可能, 我们要避免这样做。一个意外包含了重复的 GUID 的系统在找 Protocol 的时候可能会找到另外一个 Protocol, 造成系统的崩溃。这样的 bug, 要找出问题所在是很困难的。GUID 也允许来命名 API, 这样可以不用担心 API 本身名称的冲突。在 PC/AT BIOS 架构的系统中, 服务程序是以枚举的方式来顺序增加的, 例如, 著名的 Int15 接口通过 AX 来传递服务类型的, 因为没有文档来管理和维护 Int15 服务程序的扩展, 一些 BIOS 厂商各自定义了相同的服务类型 (但是功能不同), 这样就使操作系统和系统预处理应用软件的协同工作非常困难。EFI 通过 GUID 来命名 APIs 并写入规范的方式, 使得 API 平衡了扩展的需要和协同工作的能力。

Protocols 的使用

在 Boot time 时, 任何 EFI 的代码可以使用 Protocol, 但是当 ExitBootServices() 执行完, handle 数据库就不在存在。一些 EFI 启动时间的服务程序使用 EFI Protocols。

多个 Protocols 的实例

一个 handle 可以有许多个 Protocols 安装在上面, 但是可能每一种类型的 Protocol 只有一个, 也就是说, 一个 handle 可能不会有超过一个的相同类型的 Protocol。否则, 在使用一个 handle 的特定 Protocol 的时候会有不确定性。

但是, 驱动可以产生多个特定的 Protocol 并且把他们挂在不同的 handle 上面。PCI I/O protocol 就是这样的, PCI 的总线驱动会为每一个的 PCI device 安装一个 PCI I/O Protocol。每个 PCI I/O Protocol 里面的数据是根据其所在的 PCI 设备来配置的, 包括 EFI Option Rom(OpROM)的位置和大小。

同时，每个驱动可以安装同一 Protocol 的不同定制版本，只要他们不被用在同一个 Handle 上面。例如，每个 EFI 驱动会为自己的 Driver Image Handle 安装一个叫做“组件名字”的 protocol，当 EFI_COMPONENT_NAME_PROTOCOL.GetDriverName 这个函数被调用的时候，每个 handle 会返回驱动所对应 Image Handle 的名字。在英语语言环境下，EFI_COMPONENT_NAME_PROTOCOL.GetDriverName() 函数在 USB 总线驱动 handle 上面会返回“USB bus driver”，但是在 PXE 驱动 handle 上面则会返回“PXE base code driver.”

Tag GUID

一个 Protocol 可能只由一个 GUID 组成，这样的 GUID 被叫做标签 GUID，这样的 Protocol 是很有用的，比如用来标记一个特殊的 device handle，或者允许其他的 EFI Images 很容易的找到安装有标签 GUID 的 device handle。在 EDK 中就使用 HOT_PLUG_DEVICE_GUID 这样的方式来标记具有热插拔功能设备的 device handles。

EFI Images

所有的 EFI Images 都包含一个 PE/COFF 格式的头来定义这段可执行代码，PE/COFF 是微软在 1997 可移植的执行体和通用对象文件格式规范中定义的，这种代码可以用在 IA-32 处理器，安腾处理器，或者未知的处理器，以及 EFI 二进制代码。这个头文件定义了处理器和 Image 的类型，目前有 3 种处理器类型和以下 3 种 Image 类型被定义：

- EFI 应用程序——这种 Image 的内存资源和状态会在他们退出的时候释放。
- EFI Boot Service 程序驱动——这种 Image 的内存资源和状态在进入操作系统之前都被保持。当 OS loader 调用 ExitBootServices() 的时候被释放。
- EFI Runtime 驱动——这种 Image 的内存资源和状态会一直存在，这些 Image 和 EFI 操作系统并存，并且能够被支持 EFI 的操作系统调用。

EFI Image 格式的价值在于不同方产生的二进制文件能够互用，例如，微软 Windows 和 Linux 操作系统为支持 EFI 的操作系统的。另外，第三方开发的抽象特定硬件的 EFI 驱动，像网络接口主机总线适配器(HBA)或者其他设备.EFI image 通过 Boot Service gBS->LoadImage() 被装载.EFI images 的几个可用的支持存储的地方有：

- PCI 卡上面的 Expansion ROMs
- 系统的只读存储器或者系统的 flash 芯片
- 媒介设备，像硬盘，软盘，光盘等
- 网络启动服务器

总的来说，EFI images 不是被编译和链接在一个固定位置。相反的，EFI images 能够被重新定位，所以 EFI images 能够被放在系统内存的任何地方.Boot Service gBS->LoadImage() 会做以下工作：

- 为正在被转载的 image 分配内存
- 自动对 image 进行重新定位
- 在 handle 数据库中创建一个新的 image handle，安装 EFI_LOADED_IMAGE_PROTOCOL 的一个实例。

这个 EFI_LOADED_IMAGE_PROTOCOL 的实例包含 EFI image 已经被装载的信息。因为

这个信息在 handle 数据库中是共用的，所以对于所有的 EFI 组件是可用的。

在 EFI Image 被 gBS->LoadImage() 装载以后，它能够通过 gBS->StartImage 的调用执行。EFI Image 的头包含了供 gBS->StartImage 调用的入口地址。这个入口地址有 2 个参数：

- 正要被装载的EFI image的Image handle
- 指向EFI System Table的指针

有了这两项，能够使EFI image做到：

- 访问平台上面可用的所有EFI服务程序。
- 知道EFI Image从哪里被装载以及被放在内存的什么位置。

EFI Image 在他的入口地址里面的操作很大程度上取决于 EFI image 的类型。图 2.4 展示了不同的 EFI Image 的类型和不同层次 image 之间的关系。

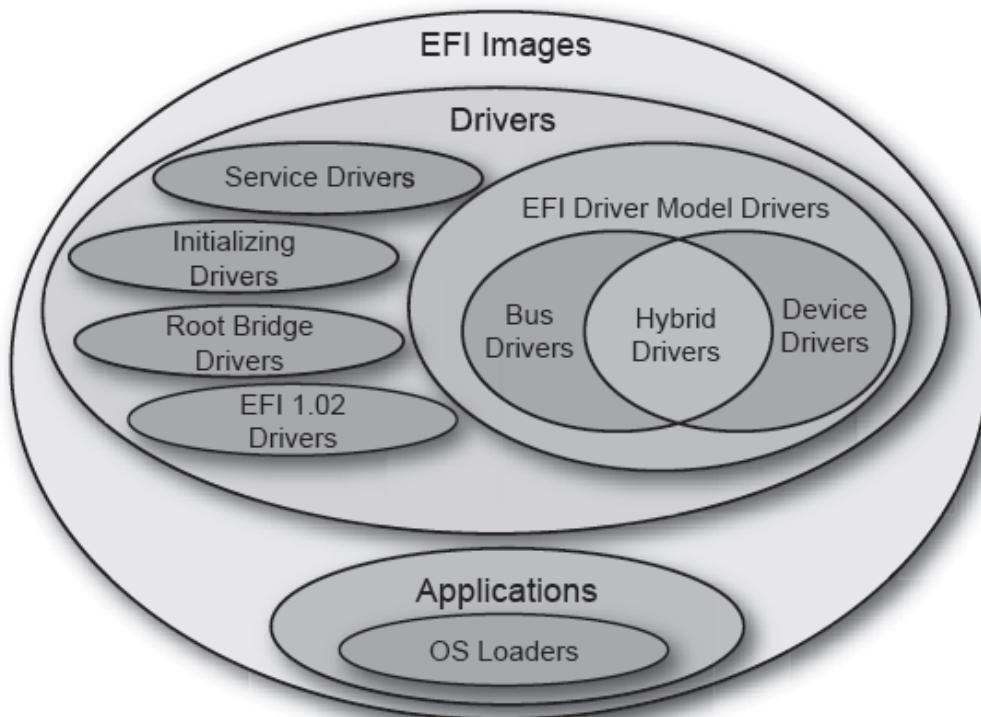
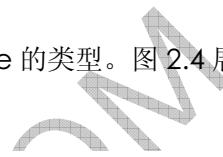


Figure 2.4 Image Types and Their Relationship to One Another

表2.1 Image类型的描述

Image 的类型	描述
应用程序	一种 EFI image 的类型 EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION. 执行时，当 image 退出或从入口点返回时这种 image 会被自动卸载。
OS loader	一种特殊类型的的应用程序，通常不会返回或退出，相反的，它会调用 EFI Boot Service gBS->ExitBootServices() 来把平台的控制权从固件

	交给操作系统。
Driver	<p>一种 EFI image 的类型</p> <p>EFI_IMAGE_SUBSYSTEM_BOOT_SERVICE_DRIVER 或者 EFI_IMAGE_SUBSYSTEM_RUNTIME_DRIVER。如果这种 iamge 返回 EFI_SUCCESS，那么它不会被卸载。如果他返回错误代码，那么它会被自动从系统内存中卸载。在系统内存中驻留的能力是驱动和应用程序不同的地方。因为驱动能在内存中驻留，所以他们能提供服务程序给其他驱动和应用程序，或者操作系统。只有运行时驱动(runtime drivers)在调用 gBS->ExitBootServices()以后还能驻留内存。</p>
Service driver	这种驱动会在一个或多个服务程序 handle 上面加载一个或多个 protocol，并且它的入口点会返回 EFI_SUCESS。
Initializing driver	这种驱动不会产生任何的 handle,也不会在 handle 的数据库中增加任何的 protocol, 它只会进行一些初始化操作，并且返回错误代码。所以这种驱动执行完或被从系统内存中卸载。
Root bridge driver	这种 driver 会产生一个或多个 controller handle，而这种 handle 会包含一个 Device Path Protocol 和一个对芯片根总线提供的 I/O 资源软件方式抽象出来的 protocol，最常见的 Root bridge driver 是为平台上面的 PCI root bridges 产生 handles 的一个驱动，并且产生的 handle 上面支持 Device Path Protocol 和 PCI Root Bridge I/O Protocol.
EFI1.02 driver	一种遵循 EFI1.02 规范的驱动。这种类型的驱动不会用到 EFI Driver Model。这种类型的驱动不会在这份文档中详细讨论。反而我们建议把 EFI1.02 驱动转换为遵循 EFI Driver Model 的驱动。
EFI Driver Model driver	一种遵循 EFI1.10 规范描述的 EFI 驱动模型的驱动。这种类型的驱动和 Service driver, Initializing driver, Root bridge driver, EFI1.02 driver 从根本上是不同的，因为遵循 EFI Driver Model 的驱动是不允许操作硬件或者在入口点产生设备相关的服务程序。取而代之的是，这种驱动的入口点只允许注册一组在后面系统初始化进程中被开始和结束的服务程序。
Device driver	一种遵循 EFI Driver Model 的驱动。这种类型的驱动会在 handle 数据库中产生一个或多个的 driver handle 或者 driver image handle，并在 handle 上安装一个或多个 Driver Binding Protocol 的实例。这种类型的驱动在 Driver Binding Protocol 的 start()函数被调用时不会产生 Child handles，它只会在现有的 Controller handles 上面增加另外的 I/O protocols。
Bus driver	一种遵循 EFI Driver Model 的驱动。这种类型的驱动会在 handle 数据库中产生一个或多个的 driver handle 或者 driver image handle，并在 handle 上安装一个或多个 Driver Binding Protocol 的实例。这种类型的驱动在 Driver Binding Protocol 的 start()函数被调用时会产生新的 Child handles，而且也会在新的 Child handles 上面增加另外的 I/O protocols。
Hybrid driver	一种遵循 EFI Driver Model 的驱动并且兼容了 device drivers 和 bus drivers 的特点，这个特性指，在 Driver Binding Protocol 的 start()函数

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

被调用时会在现有的 handles 上面增加另外的 I/O protocols，而且也会产生新的 Child handles。

应用程序

一个 EFI 的应用程序从它的入口点开始执行，并且一直执行直到在入口点中碰到返回或者 boot service 的 EXIT()函数被调用。当结束时，它的 Image 会被从内存中卸载。一般的 EFI 应用程序的例子有 EFI shell，EFI shell 的指令，刷 flash 芯片的工具，诊断工具。从应用程序中调用其他应用程序是完全可以的。

OS loader

一种特殊类型的 EFI 应用程序，叫做“系统启动引导程序”，当 OS loader 已经建立了足够的系统基础架构来掌握系统资源时，会调用 ExitBootServices()。这时，EFI 会释放所有的启动时服务程序和驱动，只留下 run-time 服务程序和驱动。

驱动

EFI 驱动和 EFI 应用程序的区别在于，驱动会一直驻留内存，除非一个错误从驱动的入口点返回。EFI 核心固件，启动管理系统，或者其他 EFI 应用程序可以装载这些驱动。

EFI1.02 驱动

一些类型的 EFI 驱动在随后的规范中已经被升级了。在 EFI1.02 的时候，驱动是在驱动模型没有被定义的情况下创建的。EFI1.10 规范提供了一个驱动模型来取代在 EFI1.02 时代创建驱动的方式，但是同时兼容 EFI1.02 的驱动。EFI1.02 会在入口点马上开始它的驱动。这样方式意味着驱动必须马上找到支持的设备，并安装上所需要的 I/O protocols，这种情况下就需要通过定时器轮询来得到设备。但是这种方式不会给系统任何策略来控制驱动的装载和连接，所以 EFI Driver Model 被用来解决这些问题，EFI Driver Model 在 EFI1.10 规范的 1.6 章节中定义。

浮点软件辅助(FPSWA)驱动是一种常见的 EFI1.02 驱动。其他的 EFI1.02 驱动可以在 EFI 应用程序工具包 1.02.12.38 中找到。为了兼容性的需要，EFI1.02 驱动能被转换成遵循 EFI 驱动模型的 EFI1.10 驱动。

Boot Service and Runtime Drivers

Boot-time 驱动会被装载在一个以 EfiBootServicesCode 标记的内存区域内，数据以 EfiBootServicesData 来标记。这些内存在 gBS->ExitBootServices() 调用完以后就被转换为可用的内存。

Runtime 驱动被装载在一个以 EfiRuntimeServicesCode 标记的内存区域内，数据以 EfiRuntimeServicesData 来标记。这些内存在 gBS->ExitBootServices() 调用完以后还是被保留，因此能使 Runtime 驱动能为正在运行的操作系统提供服务程序，所以 Runtime 驱动必须要提供另外一个另外的调用机制，因为 EFI handle 数据库在 OS 运行时已经不存在了。最常见

的 EFI Runtime 驱动是浮点软件辅助驱动(FPSWA.Efi)和通用网络驱动接口(UNDI)驱动。其他的 Runtime 驱动都不怎么常见。另外，Runtime 驱动的实现和验证都比 boot service 驱动困难，因为 EFI 支持 Runtime 服务程序和 Runtime 驱动从物理寻址模式到虚拟寻址地址的变换。在这个转换的帮助下，操作系统能够用虚拟地址来调用 Runtime 驱动。通常，OS 运行在虚拟模式下，它必须转换到物理寻址模式下来调用一般程序。为现代的，多处理器操作系统转换到物理寻址模式是非常浪费的，因为需要冲掉转换寻找区(TLB)，并集合协调所有的 CPU 和其他任务。所以，EFI 的 Runtime 提供了一个非常有效的调用机制，因为不需要模式转换。

事件和任务的优先级

事件是另外一种通过 EFI 服务程序来管理的对象。一个事件能被创建和消除，并且能处在等待状态或者触发状态。一个 EFI image 能做以下的事情。

- 产生一个事件
- 消除一个事件
- 查询一个事件是否被触发
- 等待一个事件被触发
- 请求一个事件从等待状态转换到触发状态

因为 EFI 不支持中断，所以它给了习惯于中断驱动模型的驱动开发者一个挑战。取而代之的，EFI 提供轮询的驱动。EFI 驱动对于事件最常用的用法，是用 timer 事件来允许驱动来周期性的轮询设备。图 2.5 展示了 EFI 所支持的不同类型的事件及这些事件之间的关系。

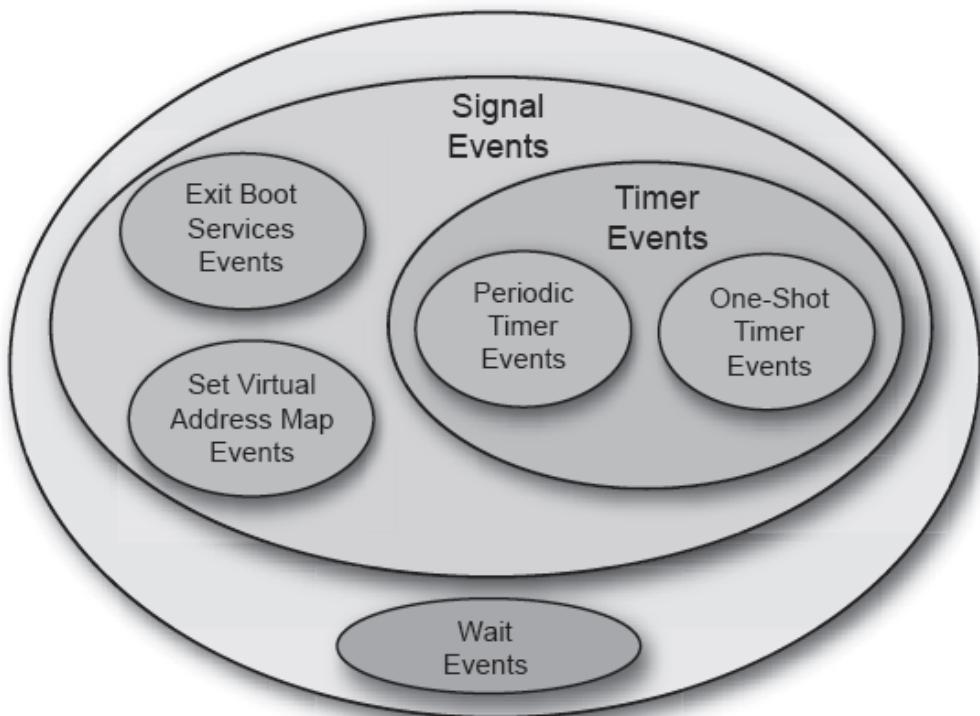


Figure 2.5 Event Types and Relationships

表 2.2 事件类型的描述

事件类型	描述
等待事件	这种事件的通知函数每当被确认或使处于等待的时候被执行。
触发事件	这种事件的通知函数每当事件从等待状态过渡到触发状态的时候被计划执行。
Boot Services 退出事件	一个特殊类型的触发事件，这个事件在 EFI Boot Services ExitBootServices() 被调用的时候从等待状态过渡到触发状态，这个调用正好是平台的控制权从固件交给操作系统的时间点。当 ExitBootServices() 被调用的时候，这个事件的通知函数被计划执行
设置虚拟地址影射事件	一个特殊类型的触发事件，这个事件在 EFI Runtime Service SetVirtualAddressMap() 被调用的时候从等待状态过渡到触发状态。这个调用正好是操作系统请求 EFI runtime 组件从物理寻址模式转换到虚拟寻址模式的事件点。操作系统提供虚拟地址的映射。当 SetVirtualAddressMap() 被调用的时候，这个事件的通知函数被计划执行。
Timer 事件	一个类型触发事件，这个事件会在一个指定时间过后从等待状态过渡到触发状态。支持周期 timer 事件和单次执行 Timer 事件，当一个特定时间过后，这个事件的通知函数被计划执行。
周期 timer 事件	一个类型触发事件，这个事件会以一定的频率从等待状态过渡到触发状态。当一个特定时间过后，这个事件的通知函数被计划执行。
单次执行 Timer 事件	一个类型触发事件，这个事件会在一定时间周期过后从等待状态过渡到触发状态。当一个特定时间过后，这个事件的通知函数被计划执行。

每个事件有以下的 3 个元素：

- 事件的任务优先级 (TPL)
- 通知函数
- 通知环境

等待事件的通知函数在事件被确认或被置于等待状态时被执行，触发事件的通知函数每当在事件从等待状态转换到触发状态时被执行，通知环境在每次通知函数被执行的时候被传递给通知函数。TPL 是通知函数被执行的优先级定义。表 2.3 例出了当前定义的 4 个 TPL 优先级。另外的 TPL 可以在以后被定义出来。比如一个兼容的 TPL list 可以在 TPL_NOTIFY 和 TPL_HIGH_LEVEL 之间增加一系列的中断 TPL，来为 EFI 提供中断方式的 I/O 支持。

表 2.3 在 EFI 中定义的任务优先级

任务优先级	描述
TPL_APPLICATION	EFI Image 执行的优先级
TPL_CALLBACK	大多数通知函数的优先级
TPL_NOTIFY	大多数 I/O 操作执行的优先级
TPL_HIGH_LEVEL	Timer 中断的优先级

TPL 提供了下面 2 个用途：

- 定义通知函数执行的优先级
- 产生锁

对于优先级定义，只有在同时事件有多个事件处于触发状态的时候，你才会用到这个机制。在这些情况下，应用程序先执行被注册成高优先级的通知函数。同时，高优先级的通知函数能中断低优先级通知函数的执行。

对于产生锁，因为 EFI 支持 single-timer 中断，在正常环境中运行的代码和在中断环境下运行的代码能访问到相同的数据结构。如果更新到一个共享的数据区域而这个数据区域又不是原子性的（Atomic（原子性）：事务中包含的操作被看做一个逻辑单元，这个逻辑单元中的操作要么全部成功，要么全部失败），这样的访问能够产生问题和不可预料的结果。EFI 应用软件或 EFI 驱动可以临时提高任务优先级来防止共享数据结构被正常环境和中断环境同时访问。应用程序可以通过临时提高任务优先级到 TPL_HIGH_LEVEL 来创建一个“锁”。在这个级别单次执行 Timer 事件都可以被阻止，但是你必须尽可能的减少系统处于 TPL_HIGH_LEVEL 的时间，因为在一段时间内，所有基于 Timer 的事件都会被阻止，任何需要周期性访问设备的驱动也会被阻止。TPL 与微软 Windows 里面的 IRQL 及一些 Unix 环境下的 SPL 相似，TPL 描述了对资源访问控制的一个优先排序机制。

第三章

EFI 驱动模型

万事万物应该尽量简单，而不是更简单。

—Albert Einstein

为了支持符合现有工业标准总线（譬如 PCI 和 USB 总线）的设备和未来的架构，EFI 提供了一个驱动模型，该模型被期望于能够简化设备驱动的设计和执行，并且能够减小可执行映像的大小。基于此，某些复杂的内容被移入到总线驱动和更大范围的公共固件服务之中去。设备驱动需要在加载这个驱动的 image handle 上产生一个 Driver Binding Protocol，然后该驱动就会等待系统固件将其连接到一个控制器上。当连接之后，该设备驱动就会负责在控制器的 device handle 上产生一个协议，该协议抽象了控制器所支持的 IO 操作。总线驱动会执行这些相同的任务，除此之外，总线驱动也会负责查找总线上的任何子控制器，并且为检测到的每个子控制器创造一个 device handle。

对于一个特定的平台，由固件服务程序，总线驱动程序和设备驱动程序构成的组合很可能由不同的厂商来生产，这些厂商包括 OEM、IBV 和 IHV。而来自不同厂商的这些组件必须相互

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

协作来提供一个统一的协议，该协议用来引导 EFI 兼容的操作系统。因此，为了增加这些组件的协同工作的能力，我们建立了 EFI 驱动模型概念并详细描述。

本章对 EFI 驱动模型做了简要概述，同时描述了驱动的入口点，主机总线控制器，设备驱动和总线驱动的特性，以及 EFI 驱动模型如何调度热插拔事件。

为何要在 OS 启动之前建立驱动模型呢？

在 EFI 驱动模型下，只需要最小数目的 IO 设备处于活动之中。例如，基于当前 BIOS 的系统(非 EFI)，一个拥有多个 SCSI 设备的服务器需要使所有的这些 SCSI 设备都处于活动之中(来引导操作系统)这是因为 BIOS 的 IN19 代码并不能预先知道哪个设备包含 OS 引导程序(其只会循规蹈矩的对每个设备进行尝试)。而 EFI 驱动模型只会使对于启动必需的部分设备处于活动之中，这就使得系统的快速重启成为可能，并且加速了其它设备激活进入到操作系统当中。随着消费电子设备被推向所有的开放平台，对于快速启动时间的需求是十分迫切的。

驱动初始化

driver image 文件必须从某种类型的媒介中加载，这些媒介包括 ROM，闪存，硬盘，软盘，光盘，或者甚至于网络线路。一旦一个 driver image 被系统检测到之后，该 image 就会被启动服务的 LoadImage()加载到系统的内存 (LoadImage()用于加载 PE/COFF 格式的 image 进入到系统内存)。同时该驱动的 handle 被生成，并且一个 Loaded Image Protocol 例程被挂载到该 handle 上，而包含有 Loaded Image Protocol 例程的 handle 叫做 image handle。此刻该驱动并没有开始，其只是驻留在内存之中等待开始。图 3.1 展示了驱动的 image handle 被 LoadImage() 调用之后的状态。

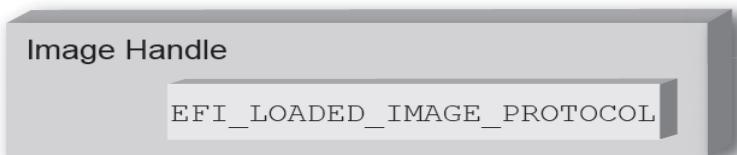
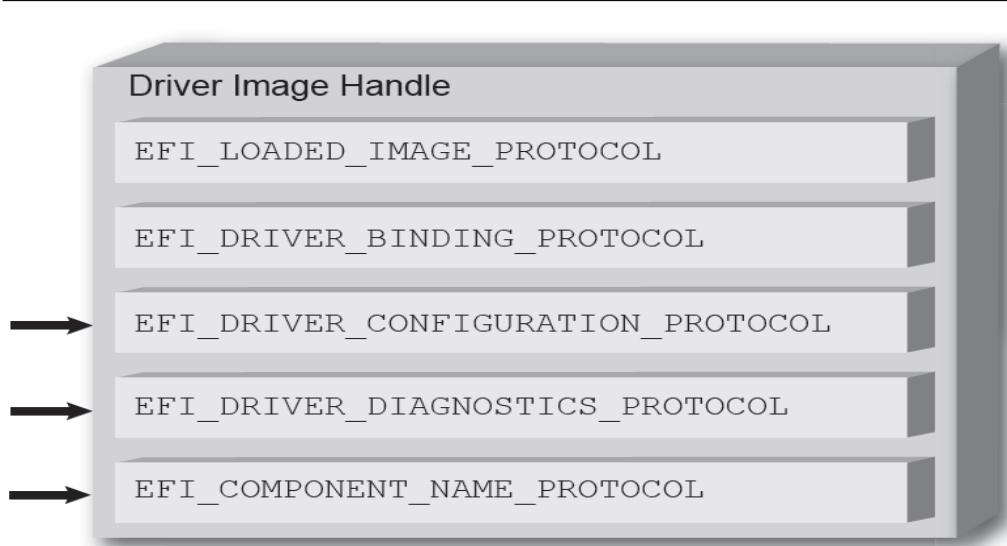


Figure 3.1 Image Handle

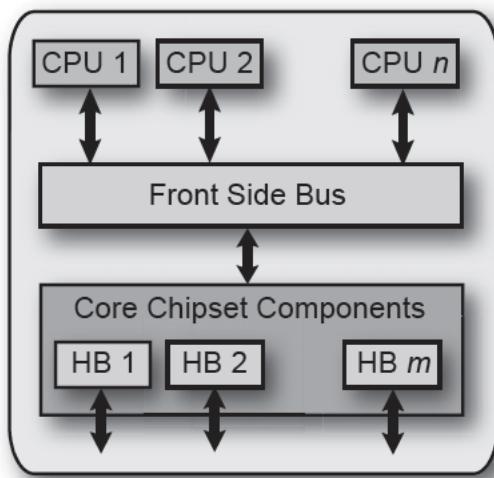
驱动被启动服务的 LoadImage() 加载之后，它需要通过启动服务的 StartImage() 进行启动，这适用于在 EFI 系统中能被加载和启动的各种类型的 EFI 应用程序和驱动。遵循 EFI 驱动模型的驱动的入口程序必须遵循一些严格的规定：首先，不允许涉及任何硬件的操作，仅仅允许选择性的安装一些协议例程到自己的 image handle 上，(遵循 EFI 驱动模型的驱动必须安装一个 Driver Binding Protocol 到自己的 image handle 上，另外，它还可以选择性地安装一些协议，) 例如 Driver Configuration Protocol，Driver Diagnostics Protocol，或者 Component Name Protocol。除此之后外，如果驱动希望可以被卸载，它可以选择性的更新 Loaded Image Protocol 以此来提供自己的 Unload() 接口。最后，如果驱动需要在启动服务的 ExitBootServices() 功能被调用的时候执行一些特殊的操作，它可以选择性的创建一个带有通知功能的事件，当启动服务的 ExitBootServices() 功能被调用的时候，该通知功能就会被触发。包含有 Driver Binding Protocol 例程的 image handle 被认为是 driver image handle。图 3.2 反映了图 3.1 中所示的 image handle 在启动服务 StartImage() 被调用后的可能配置。

**Figure 3.2** Driver Image Handle

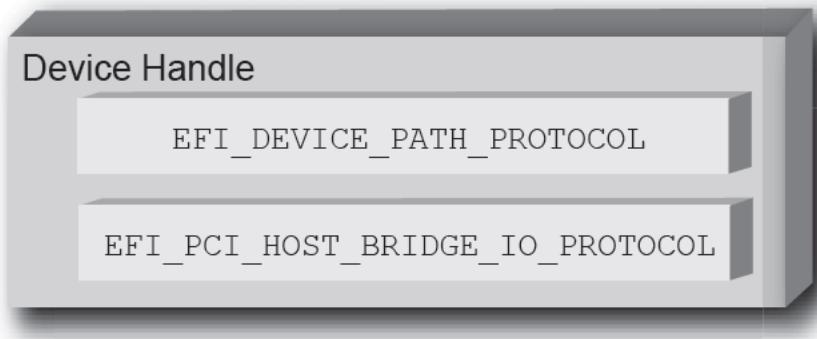
主机总线控制器

在驱动的入口程序中驱动不允许操作任何的硬件。因此，虽然驱动被加载和启动，但是他们都在等待被告知去管理该系统中的一个或者多个控制器。像 EFI 启动管理器这样的一个平台组件，会负责管理驱动和控制器之间的连接。可是，即使在第一个驱动和控制器被连接之前，某些最初的(固有的)控制器需要存在来被驱动管理，这些最初的(固有的)控制器就是大家知道的主机总线控制器。主机总线控制器所提供的 I/O 抽象是由 EFI 驱动模型范围之外的固件组件所产生的，主机总线控制器的 device handle 和 I/O 抽象必须由平台上的固件内核或者可能不遵循 EFI 驱动模型的 EFI 驱动产生。具体请参照 PCI Host Bridge I/O Protocol 规范上面对 PCI 总线 I/O 抽象的例子。

一个平台可以看作是一组 CPU 和一组可能包含一个或者多个主机总线的芯片组件的组合。图 3.3 展示了一个拥有 n 个 CPU 和包含 m 个主桥的芯片组的平台。

**Figure 3.3** Host Bus Controllers

在 EFI 中，每个主桥被表现为一个包含了 Device Path Protocol 例程的 device handle，和一个抽象出主机总线可以执行的 I/O 操作的协议例程。例如，PCI 主机总线控制器支持 PCI Host Bridge I/O Protocol。图 3.4 展示了一个 PCI 主桥的 device handle 的例子。

**Figure 3.4** Host Bus Device Handle

PCI 总线驱动能够连接到该 PCI 主桥，并且为系统中的每个 PCI 设备创造 child handles，然后 PCI 设备驱动需要(被)连接到这些 child handle 上，并且产生可用于启动 EFI 兼容的 OS 的 I/O 抽象。接下来的章节将会描述基于 EFI 驱动模型实现的不同类型的驱动。EFI 驱动模型是非常灵活的，所以这里不会讨论所有可能的驱动类型。我们在这里只讲述主要的几种驱动类型，这些驱动类型可以作为设计和实现其他类型驱动的基础。

设备驱动

设备驱动不允许创造任何的新的 device handle，相反它只是在现存的设备 handle 上安装其它的协议接口。大多数常见类型的设备驱动负责把 I/O 抽象挂到由总线驱动所创造的设备 handle 上，该 I/O 抽象可以被用于启动 EFI 兼容的操作系统，这些 I/O 抽象就包括例如 Simple Text Output, Simple Input, Block I/O 和 Simple Network Protocol。图 3.5 展示了一个设

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

备驱动被连接前后的 device handle。在这个例子之中，该 device handle 是 XYZ 总线的子设备 handle，所以它包含了 XYZ 总线支持的用于 I/O 服务的 XYZ I/O 协议。该 handle 也包含了一个由 XYZ 总线驱动创建的 Device Path Protocol。不是所有的 device handle 都必须有 Device Path Protocol，但对于用来表示系统中物理设备的 device handle 来说 Device Path Protocol 是必须的，而虚拟设备的 handle 是不包含 Device Path Protocol 的。

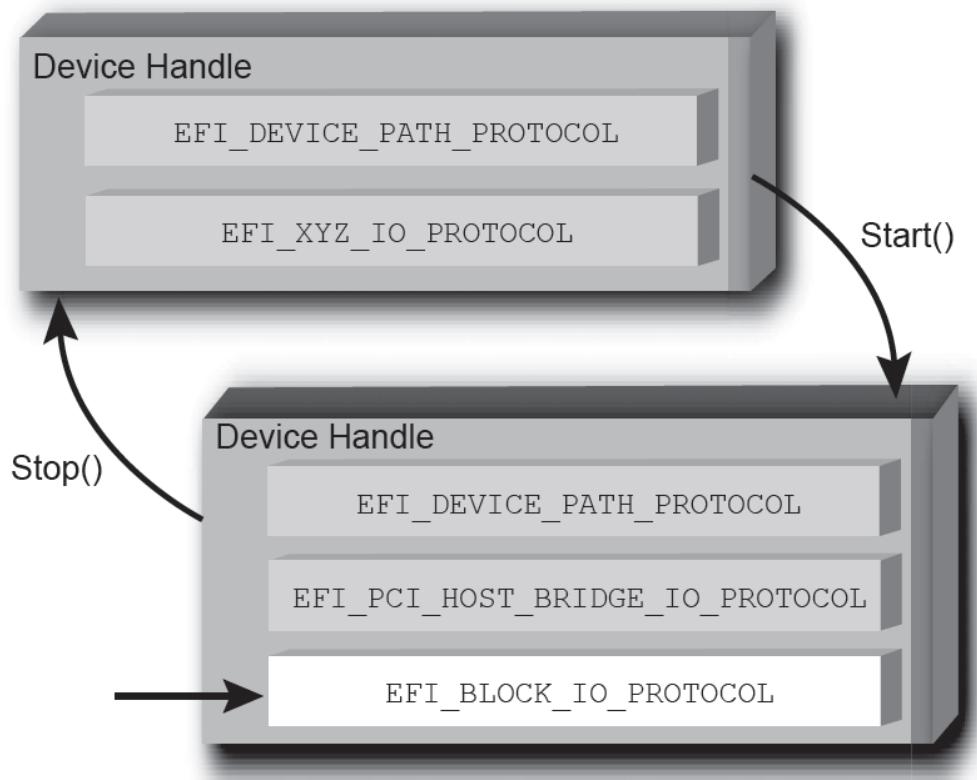


Figure 3.5 Connecting Device Drivers

图 3.5 所示的连接到对应 device handle 上的设备驱动必须已经在自己的 image handle 上安装上一个 Driver Binding Protocol。Driver Binding Protocol 包含有三个功能，分别为：Supported()，Start() 和 Stop()。Supported() 检测该驱动是不是支持一个给定的控制器。在该例子中，驱动将会检查 device handle 是不是支持 Device Path Protocol 和 XYZ I/O Protocol。如果驱动的 Supported() 功能检测通过，随后，才可以通过调用驱动的 Start() 功能将该驱动连接到控制器上。Start() 功能实际上就是将其他的 I/O 协议添加到 device handle 上。在该例子中，是 Block I/O Protocol 被安装到 device handle 上了。与之对应 Driver Binding Protocol 还有一个 Stop() 功能，使驱动停止管理该 device handle。这样设备驱动会卸载 Start() 功能中安装的所有协议。

在 EFI Driver Binding Protocol 的 Support()，Start()，和 Stop() 功能中，得到一个协议接口必须使用新的启动服务的 OpenProtocol() 函数，释放一个协议接口必须使用新的启动协议的 CloseProtocol() 函数。OpenProtocol() 和 CloseProtocol() 会更新由系统固件维护的 handle 数据库，以此去追踪哪些驱动在使用哪些协议接口。Handle 库之中的信息可以被用于获得关于驱动和控制器的信息。新的启动服务 OpenProtocolInformation() 可用于获得当前正在使

用某个特定协议接口的组件列表。

总线驱动

从 EFI 驱动模型的观点来看，总线驱动和设备驱动实质上是一样，唯一的差别在于总线驱动会为在该总线上发现的子控制器创造新的 device handle。因此，总线驱动比起设备驱动来会稍微有点复杂，但是反过来这也简化了设备驱动的设计和实现。总线驱动主要有两种类型，第一种是在第一次调用 Start() 时总线驱动会为所有的子控制器创造 handle，第二种是允许通过多次对 Start() 功能的调用来为所有子控制器创造 handle。第二种类型的总线驱动在支持快速启动的能力上是非常有用的，它允许每次创造几个甚至是一个子 handle，在需要花费很长时间去枚举所有的子控制器的总线上（譬如 SCSI 总线），这可以在平台启动过程中节省很多的时间。图 3.6 展示了 Start() 被调用前后的总线控制器的树结构。进入总线控制器结点的虚线代表了到总线控制器的父辈控制器的链接，如果总线控制器本身就是一个主机总线控制器，那么它没有父辈控制器。结点 A、B、C、D 和 E 代表总线控制器的 5 个子控制器。

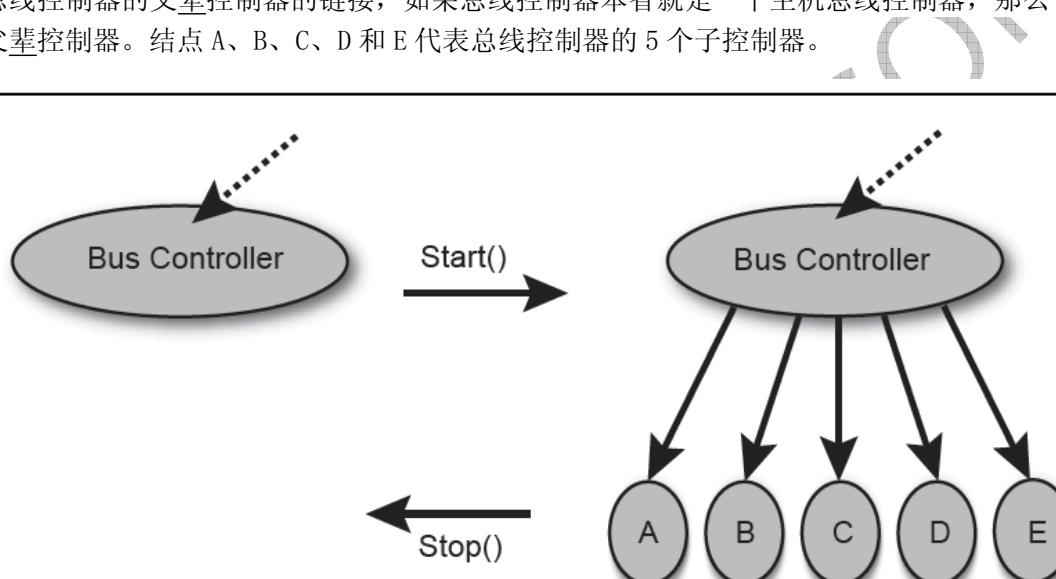


Figure 3.6 Connecting Bus Drivers

一个支持调用一次 Start() 然后创造一个子 handle 的总线驱动可能会选择首先为 C 子控制器创造 handle，然后是为 E，接下来是为 A, B 和 D，对于这种类型的行为，Driver Binding Protocol 的 Supported()、Start() 和 Stop() 三个功能是足够灵活的。

总线驱动需要为它创造的每个子设备 handle 安装协议接口，至少，它需要为子控制器安装一个协议接口，该接口提供了总线服务的 I/O 抽象。如果总线驱动创造了一个代表物理设备的子 handle，那么总线驱动必须在该子 handle 上安装一个 Device Path Protocol 例程。总线驱动可以为每个子 handle 选择性的安装一个 Bus Specific Driver Override Protocol，该协议在有多个驱动被连接到子控制器时被使用。新的启动服务 ConnectController() 会使用架构上定义的优先规则为给定的控制器选择最合适的驱动。在搜索驱动时，Bus Specific Driver Override Protocol 的优先级高于一般的驱动，低于 Platform Driver Override Protocol。PCI 总线上就有总线特定驱动选择的例子，PCI 控制器 option ROM 中的 PCI 总线驱动比平台其他地方的 PCI 总线驱动具有更高的优先级。图 3.7 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。

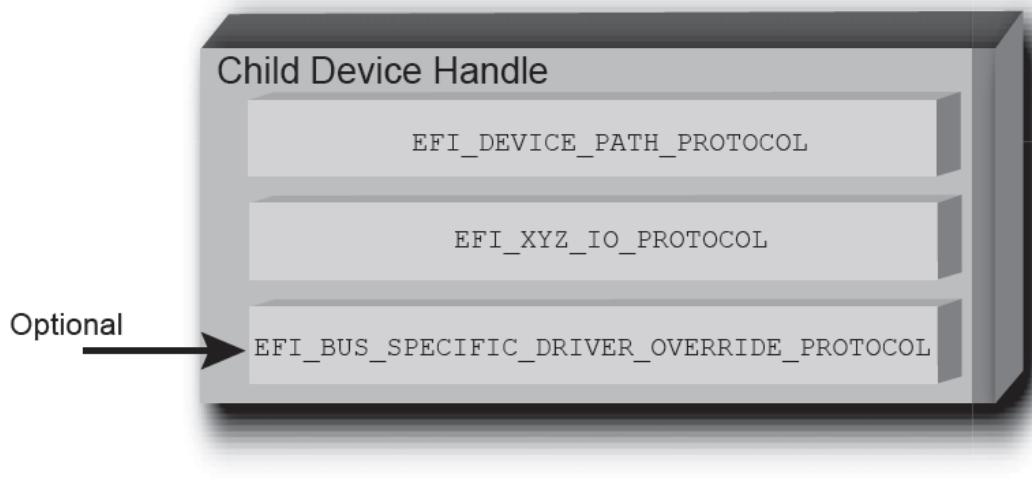


Figure 3.7 Child Device Handle with a Bus Specific Override

平台组件

在 EFI 驱动模型中，在某个平台上驱动和控制器之间的连接和断开的动作是处于平台固件的控制之下的，这通常是由 EFI Boot Manager 来完成，也可能有其他的方式。平台固件可以使用启动服务新的 `ConnectController()` 和 `DisconnectController()` 来判断哪些控制器已经被驱动，哪些还没有。如果平台希望执行系统的诊断或者是安装一个操作系统，那么它可以选择为所有可能的启动设备连接驱动。如果某个平台希望启动一个预安装的操作系统，它可以选择只为启动该操作系统所必需的设备连接驱动。EFI 驱动模型通过新的启动服务 `ConnectController()` 和 `DisconnectController()` 支持上述两种模式的操作。除此之外，由于负责启动平台的平台组件必须依靠设备路径来实现控制台设备和启动选项功能，所以在 EFI 驱动模型中所有的服务和协议其实已经通过设备路径优化过了。

平台可能也会选择产生一个叫做 Platform Driver Override Protocol 的可选协议，该协议与 Bus Specific Driver Override Protocol 是相似的，但是却拥有更高的优先权，这就给予平台固件最高的优先级去决定哪一个驱动被连接到哪一个控制器。Platform Driver Override Protocol 挂接在系统中的一个 handle 上，如果在系统中该协议存在的话新的启动服务 `ConnectController()` 将会使用该协议。

热插拔事件

在过去，在预启动环境中系统固件不必去处理热插事件。然而随着像 USB 这样的总线的出现，最终用户可以任何时间添加和移除设备，对于 EFI 驱动类型来说，确保能够描述这些类型的总线是非常重要的。对于一个支持热添加或者热移除设备的总线来说，其总线驱动要提供对这些事件的支持。对于这些类型的总线，一些平台的管理将不得不移入到总线驱动中。例如，在一个平台上当一个键盘被热添加到 USB 总线中，最终用户将会期望该键盘被驱动。USB 总线驱动应该能够检测该热添加事件，并且为该键盘设备创造一个子 handle。然而，由于除非 `ConnectController()` 被调用否则驱动将不会与控制器连接，该键盘将不会成为一个可用的输入设备。所以当一个热添加事件发生时，要使键盘驱动可用，USB 总线驱动要去调用

ConnectController()。另外，当一个热移除事件发生时 USB 总线驱动将必须调用 DisconnectController()。

设备驱动也会受这些热插拔事件的影响。像 USB 这样的总线，设备可以在没有任何预先通知的情况下被移动。这就意味着，USB 设备驱动的 Stop() 功能必须为一个系统中不再存在的设备终止驱动，所以，任何不能完成的 I/O 请求必须在不操作设备硬件的情况下被清除。

通常，添加对热插拔事件的支持大大的增加了总线驱动和设备驱动的复杂性。是否增加这种支持是由驱动编写人员决定的，因此必须在额外复杂度，驱动大小和预启动环境对这个特性的需求之间权衡。

下面的两个实例代码流程保证了设备驱动程序员如何查找驱动本身管理着的候选硬件设备。这些机制包括第一个例子中的查看控制器 handle 和第二个例子中的检查设备路径。

```
extern EFI_GUID gEfiDriverBindingProtocolGuid;
EFI_HANDLE gMyImageHandle;
EFI_HANDLE DriverImageHandle;
EFI_HANDLE ControllerHandle;
EFI_DRIVER_BINDING_PROTOCOL *DriverBinding;
EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath;
// 
// Use the DriverImageHandle to get the Driver Binding Protocol
// instance
//
Status = gBS->OpenProtocol (
DriverImageHandle,
&gEfiDriverBindingProtocolGuid,
&DriverBinding,
gMyImageHandle,
NULL,
EFI_OPEN_PROTOCOL_HANDLE_PROTOCOL
);
if (EFI_ERROR (Status)) {
return Status;
}
// 
// EXAMPLE #1
//
// Use the Driver Binding Protocol instance to test to see if
// the driver specified by DriverImageHandle supports the
// controller specified by ControllerHandle
//
Status = DriverBinding->Supported (
DriverBinding,
ControllerHandle,
NULL
);
```

```

if (!EFI_ERROR (Status)) {
Status = DriverBinding->Start (
DriverBinding,
ControllerHandle,
NULL
);
}

return Status;
// 

// EXAMPLE #2
//
// The RemainingDevicePath parameter can be used to initialize
// only the minimum devices required to boot. For example,
// maybe we only want to initialize 1 hard disk on a SCSI
// channel. If DriverImageHandle is a SCSI Bus Driver, and
// ControllerHandle is a SCSI Controller, and we only want to
// create a child handle for PUN=3 and LUN=0, then the
// RemainingDevicePath would be SCSI(3,0)/END. The following
// example would return EFI_SUCCESS if the SCSI driver supports
// creating the child handle for PUN=3, LUN=0. Otherwise it
// would return an error.
//
Status = DriverBinding->Supported (
DriverBinding,
ControllerHandle,
RemainingDevicePath
);
if (!EFI_ERROR (Status)){
Status = DriverBinding->Start (
DriverBinding,
ControllerHandle,
RemainingDevicePath
);
}
return Status;

```

伪代码

三种不同类型驱动的 Start() 功能的实现法则被呈现在这里。一个驱动的 Start() 功能的实现方法会影响到这个驱动的 Supported() 功能的实现方法。EFI_DRIVER_BINDING_PROTOCOL 中的所有服务需要共同工作以确保在 Supported() 和 Start() 中打开或者分配的资源在 Stop() 中被释放。

第一个的实现法则是一个简单的设备驱动，该驱动不会创造任何另外的 handle，它只是将

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

一个或者多个协议挂载在一个现存的 handle 上。第二个是一个简单的总线驱动，该驱动常常在第一次调用 Start() 时创造自己的所有子 handle，它并不会为总线控制器 handle 挂载任何的另外的协议。第三个是比较高级的总线驱动，该驱动可以在连续的对 Start() 调用时每次只创造一个子 handle，或者在一次调用 Start() 时它能够创造所有的子 handle 或者是剩余所有的子的 handle。再强调一次，它并不会为总线控制器 handle 挂载任何的另外的协议。

设备驱动

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_DRIVER 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_RESOURCES。

6. 使用 InstallProtocolInterface() 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。

7. 返回 EFI_SUCCESS。

第一次调用 Start() 时创造所有自己子 handle 的总线驱动

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_DRIVER 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 中任何一个对 OpenProtocol() 的调用返回错误，那么用 CloseProtocol() 关闭在步骤 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。

6. 如果总线要求，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 C。
8. 分配和初始化该驱动管理子设备 C 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 C 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 C 创造设备路径。
10. 初始化子设备 C。如果出现错误，那么用 CloseProtocol() 关闭在步骤 1 中打开的所有协议，并且返回 EFI_DEVICE_ERROR。
11. 为 C 创造新的 handle，并且为子设备 C 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 C 调用 OpenProtocol()。
13. 结束 for 语句。
14. 返回 EFI_SUCCESS。

在每次调用 Start() 时能够为自己所有或者其中一个子设备创造子 handle 的总线驱动：

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_DRIVER 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 中任何一个对 OpenProtocol() 的调用返回错误，那么用 CloseProtocol() 关闭在步骤 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. 初始化用 ControllerHandle 指定的设备。如果出现错误，用 CloseProtocol() 关掉在步骤 1 中打开的所有协议，并且返回 EFI_DEVICE_ERROR。
4. 如果参数 RemainingDevicePath 不为空，那么
5. C 就是由 RemainingDevicePath 指定的子设备。
6. 分配和初始化该驱动管理子设备 C 所需要的所有数据结构，这将包括公共协议的空间和与子设备 C 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 中打开的所有协议，并且返回 EFI_OUT_OF_RESOURCES。
7. 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 C 创造设备路径。
8. 初始化子设备 C。
9. 为 C 创造新 handle，并且为子设备 C 安装协议接口。这可能包括 EFI_DEVICE_PATH_PROTOCOL。
10. 用 EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER 的属性代表子设备 C 调用 OpenProtocol()。
11. 否则（如果参数 RemainingDevicePath 为空）
12. 查找由 ControllerHandle 指定的总线控制器的所有子设备。

13. 如果总线要求, 为 ControllerHandle 指定的总线控制器的所有子设备分配资源。
14. 循环 ControllerHandle 的每一个子设备 C。
15. 分配和初始化该驱动去管理子设备 C 所需要的所有的数据结构, 这将会包括公共协议的空间和与子设备 C 相关的任何的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。
16. 如果总线驱动为子设备创造设备路径, 那么基于附属于 ControllerHandle 的设备路径为子设备 C 创造设备路径。
17. 初始化子设备 C。
18. 为 C 创建新的 handle, 并且为子设备 C 安装协议接口。这可能包括 EFI_DEVICE_PATH_PROTOCOL。
19. 用 EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER 的属性代表子设备 C 调用 OpenProtocol()。
20. 结束 FOR 循环。
21. 结束 IF 语句。
22. 返回 EFI_SUCCESS。

第四章

你应该知道的 protocols

常识并不是说众所周知。

—Will Rogers

本章描述了 EFI 使用者无论是制作设备驱动，还是使用 EFI Pre-OS 程序，或者平台固件，都应了解掌握 protocols。这些 protocols 通过一些例子来介绍，以最常见的编程练习的“Hello world”开始。这里给出的例子程序都是尽量简单。它不依靠任何 EFI 库函数，所以 EFI 库函数不会被连接到生成的可执行文件中。这个测试的应用程序通过把 SystemTable 传入到入口点来访问 EFI 控制台设备。这个控制台输出设备通过用 SIMPLE_TEXT_OUTPUT_INTERFACE protocol 的 outputString() 函数来显示一组信息，并且这个应用程序等待来自控制台输入设备使用者的按键，该控制台输入设备使用 WaitForEvent() 服务程序和 SIMPLE_INPUT_INTERFACE protocol 中的 WaitForKey 事件。一旦有按键被按下，应用程序立即退出。

/*++

模块名称：

helloworld.c

摘要：

下面是一个简单的模块，用来展示一个基本的 EFI 应用程序的写法。

作者：

Waldo

版本历史：

--*/

```
#include "efi.h"

EFI_STATUS
InitializeHelloApplication (
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    UINTN Index;

    //
    // Send a message to the ConsoleOut device.
    //

    SystemTable->ConOut->OutputString (
        SystemTable->ConOut,
        L"Hello application started\n\r");
    //
```

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

```
// Wait for the user to press a key.  
//  
SystemTable->ConOut->OutputString (  
SystemTable->ConOut,  
L"\n\r\n\r\n\rHit any key to exit\n\r");  
SystemTable->BootServices->WaitForEvent (  
1,  
&(SystemTable->ConIn->WaitForKey),  
&Index);  
SystemTable->ConOut->OutputString (  
SystemTable->ConOut,L"\n\r\n\r");  
//  
// Exit the application.  
//  
return EFI_SUCCESS;  
}
```

如果执行一个 EFI 应用程序，可以在 `EFI shell` 命令行中键入程序名称。接下来的例子将会告诉你怎么样在 `EFI shell` 中去运行上面描述的测试应用程序。应用程序等待用户按键，然后即可返回 `EFI shell` 提示输入界面。以上是假定 `hello.efi` 是处在 `EFI shell` 环境的搜索路径下。

Example

```
Shell> hello  
Hello application started  
Hit any key to exit this image
```

EFI OS Loaders

这部分讨论当写 OS loader 时特别需要考虑到的几点。OS loader 是一种特殊类型的 EFI 应用程序，它负责将系统由 firmware 环境进入 OS 环境。为完成此任务，几个重要步骤必须执行：

1. OS loader 必须要确定从哪里被调用的。这样可以让 OS loader 从相同位置获取其他文件。

2. OS loader 必须要确定系统中 OS 存放在哪里。一般来讲，OS 驻留于硬盘驱动的一个分

区内。但是在 OS 存放的分区，可能没有使用 EFI 可识别的文件系统。在这种情况下，OS Loader 仅仅被作为一个 block device 且仅使用 block I/O 的操作方式来访问。OS Loader 这时将需要去实现或者读取文件系统驱动来访问系统分区上的文件。

3.OS Loader 必须建立物理内存资源的内存映射图，目的是使 OS kernel 知道它要管理的内存空间。因为系统下的某些物理内存必须被保留不被 OS kernel 改变，所以 OS Loader 必须使用 EFI 应用程序接口来重新获取系统当前的内存映射表。

4.OS 有将启动路径和启动选项以环境变量的形式存储在非易失性存储设备中的选择权。OS Loader 可能需要使用存储在非易失性存储设备中的环境变量。另外 OS Loader 可能也需要传递某些环境变量到 OS kernel 中。

5.下一步是去调用 ExitBootServices()。无论是 OS Loader 还是 OS kernel 都会完成这个调用。需要特别注意的是要保证在调用这个程序前大部分的当前内存已经释放并可以使用。一旦 ExitBootServices() 被调用，那么任何 EFI Boot Services 将不再被调用。在某个时候，调用 ExitBootServices() 之前或者之后，OS Loader 会把控制权交给 OS kernel。

6.最后，在调用 ExitBootServices() 之后，EFI Boot Services 调用就不再可用。这种可调用性的缺失意味着一旦 OS kernel 已经控制系统，OS kernel 就仅仅可以调用 EFI Runtime Services。

以下可以看到 OS loader 完整的应用程序示例列表。在接下来章节中的代码片段不会执行任何的错误检查。并且 OS loader 示例应用程序也使用了若干 EFI 库函数来简化执行过程。下面的输出信息由显示 OS loader 自己的设备路径和文件路径而开始的。它还显示出 OS loader 在内存中的位置，占用了多少字节。下一步，它将 OSKERNEL.BIN 文件载入内存中。OSKERNEL.BIN 文件可以像图 4.1 OS loader 描述那样从相同目录下重新获取。

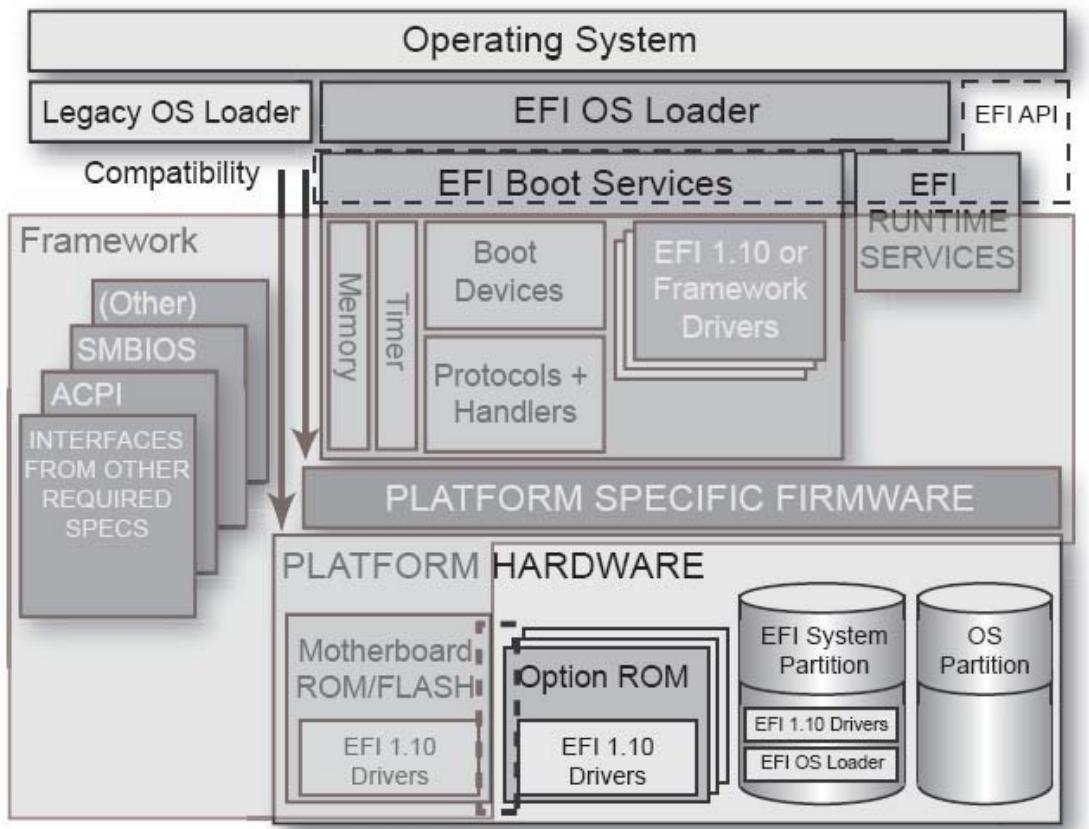


Figure 4.1 EFI Loader in System Diagram

下一部分输出结果说明了若干块设备中的第一分块。第一个是使用 FAT12 文件系统中软盘的第一分块。第二个是硬盘的主引导记录 (MBR)。第三个是在相同硬盘上大一点的 FAT32 分区的第一分块，第四个也是在相同硬盘上小一些的 FAT16 分区的第一分块。

最后一步列出了指向所有系统配置表，系统当前的内存映射表和所有系统环境变量的指针。所有步骤执行后，OS loader 调用 `ExitBootServices()`。

设备路径与 OS Loader Image 信息

接下来的代码片断演示获取 OS Loader 设备路径和文件路径的步骤。

首先调用 `HandleProtocol()`，从 `ImageHandle` 得到 `LOADER_IMAGE_PROTOCOL` 接口，传给 OS Loader 的应用程序。第二步调用 `HandleProtocol()`，得到 OS Loader Image 的 `device handle` 的 `DEVICE_PATH_PROTOCOL` 接口。这两个调用将 OS Loader Image 的设备路径、文件路径和其他 Image 信息交给 OS Loader。

```
BS->HandleProtocol(
```

```
  ImageHandle,
```

```
&LoadedImageProtocol,  
LoadedImage  
);  
BS->HandleProtocol(  
LoadedImage->DeviceHandle,  
&DevicePathProtocol,  
&DevicePath  
);  
Print (  
L"Image device : %s\n",  
DevicePathToStr (DevicePath)  
);  
Print (  
L"Image file : %s\n",  
DevicePathToStr (LoadedImage->FilePath)  
);  
Print (  
L"Image Base : %X\n",  
LoadedImage->ImageBase  
);  
Print (  
L"Image Size : %X\n",  
LoadedImage->ImageSize  
);
```

在 OS loader 的设备路径下访问文件

上一节演示的是怎样获取设备路径和OS Loader镜像路径。接下来的代码片断将演示怎样用这些信息来打开另一个叫作OSKERNEL.BIN的文件，这个文件与OS Loader在相同目录下。首先用HandleProtocol来获取从上一节得到的device handle的FILE_SYSTEM_PROTOCOL接口，然后，磁盘卷才可以被打开，文件存取成为可能，最终的结果是，变量CurDir成为OS Loader所在分区的一个文件handle。

```
BS->HandleProtocol(  
LoadedImage->DeviceHandle,  
&FileSystemProtocol,
```

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后24小时内删除，否则后果自负。

```
&Vol  
) ;  
Vol->OpenVolume (   
Vol,  
&RootFs  
) ;  
CurDir = RootFs;
```

下一步是为与OS loader image相同目录下的OSKERNEL.bin建立一个文件路径。一旦这个路径建立，文件handle CurDir可以用来对OSKERNEL.BIN文件进行Open(), Close(), Read()和Write()操作。接下来的代码段中建立了文件路径，打开文件，读取文件到已分配的缓冲器中，并关闭文件。

```
StrCpy(FileName, DevicePathToStr(LoadedImage->FilePath));  
for(i=StrLen(FileName); i>=0 && FileName[i]!='\\'; i--) ;  
FileName[i] = 0;  
StrCat(FileName, L"\\OSKERNEL.BIN");  
CurDir->Open (CurDir, &FileHandle, FileName,  
EFI_FILE_MODE_READ, 0);  
Size = 0x00100000;  
BS->AllocatePool(EfiLoaderData, Size, &OsKernelBuffer);  
FileHandle->Read(FileHandle, &Size, OsKernelBuffer);  
FileHandle->Close(FileHandle);
```

寻找 OS 分区

对系统的每个分区来说，EFI示例环境实现了一个BLOCK_IO_PROTOCOL实例。OS Loader能够依靠寻找所有的BLOCK_IO设备来搜索系统分区。下面的代码片断用LibLocateHandle()获得BLOCK_IO device handle 列表。这些handles用来检索每个BLOCK_IO设备的第一个block。HandleProtocol() API是用来获取每个BLOCK_IO设备的DEVICE_PATH_PROTOCOL和 BLOCK_IO_PROTOCOL的实例。变量BlkIo是一个使用BLOCK_IO_PROTOCOL接口的BLOCK_IO设备的handle。基于此，ReaddBlocks()可用来读取设备的第一个block。示例OS Loader仅仅是把block的内容显示出来了，实际上，OS Loader务必测试读取每个block，确保每个分区是可知的。在可知的分区找到之后，OS Loader能够实现一个简单的系统文件驱动，这个驱动会用EFI API ReadBlock()从分区中装载额外的数据。

获得当前的系统配置信息

系统配置信息可以通过SystemTable的数据结构得到（SystemTable会被传给OS

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后24小时内删除，否则后果自负。

loader)。操作系统Loader本身是一个EFI的应用，它起到沟通固件平台与操作系统之间的桥梁作用。System Table用来告知Loader: 可用的固件平台服务(例如从媒介装载OS Kernel binary要用到的block和控制台服务和装载OS驱动之前用户能够使用的block和控制台服务)和访问工业标准表，如ACPI, SMBIOS等。可用的5个表格的结构和内容在它们各自的spec中有阐述。

获取当前内存映射表

EFI功能库能够释放EFI所用的内存资源。在Loader运行的时候，内存由平台固件来管理，平台固件已经给固件划分了内存 (boot services memory)，也划分了其它需要一直保持到OS runtime的内存资源 (runtime memory)。直到装载程序把最后的控制权交给OS Kernel并调用 ExitBootServices()之前，EFI平台固件一直掌管着内存的分配。这就是说OS Loader和其他EFI应用程序可以通过内存映射服务来查明内存分配。

下面的代码片断演示一个查明内存映射并显示其内容的函数的用法。OS Loader必须特别注意MapKey这个参数，当每次EFI修改其所使用的内存映射时，MapKey就会增加。OS Loader需要将当前内存映射交给OS Kernel。如果在内存映射被释放和调用ExitBootServices()期间调用OS Loader功能，内存映射可能会被改动。一般来说，OS Loader应该在调用 ExitBootServices()之前释放内存。如果因为MapKey不匹配导致ExitBootServices()功能失败，OS Loader需要获得一份新的内存映射副本再重新尝试。

获取环境变量

接下来的代码段演示了如何选取出 EFI 环境下的环境变量。GetNextVariableName() API 贯穿整个列表。

过渡到 OS Kernel

当调用 ExitBootServices()时，所有的 EFI Boot Services 都将会被终止。从这点上讲，仅仅 EFI Runtime Services 可能被用到。一旦执行调用动作，OS Loader 就需要为过渡到 OS Kernel 做准备。假设 OS Kernel 拥有所有系统的控制权和一些其所需要的 EFI Runtime Services 功能，OS Loader 必须将 SystemTable 交给 OS Kernel，以便 OS Kernel 能方便调用 Runtime Services。OS Loader 过渡到 OS Kernel 的这种确切地机制具体跟实现相关 (就是说可以有不同的实现方法)。需要注意的是，OS Loader 可以在调用 ExitBootService()之前过渡到 OS Kernel，这种情况下，OS Kernel 在负责调用 ExitBootService()。

第五章

EFI Runtime

给一个已经延迟的软件项目增加人力，只会使它更加延迟。

--布鲁克法则

这一章描绘了在兼容 EFI 的系统里可以访问的基本服务。这些基本服务由一些接口函数所定义，运行在 EFI 环境的程序（包括管理设备访问的 Protocol 和拓展平台能力的 Protocol 等）可以使用这些接口函数。这一章讨论的焦点是运行时服务，运行时服务是指在 EFI 启动过程以及操作系统运行过程中都可以使用的函数。

在启动过程中，系统资源由固件所拥有，并且由多种表现为可调用的 API 的系统服务所控制。在 EFI 中有两种主要的服务：

■ Boot Services 启动服务 - 在启动目标（比如说是操作系统）运行之前，或者在 ExitBootServices () 被调用之前可以访问的函数。

■ Runtime Services 运行时服务 - 在启动目标运行之前，以及启动目标运行之后都可以使用的函数。

图 5.1 表明了一个平台在启动操作中所经历的各个阶段。

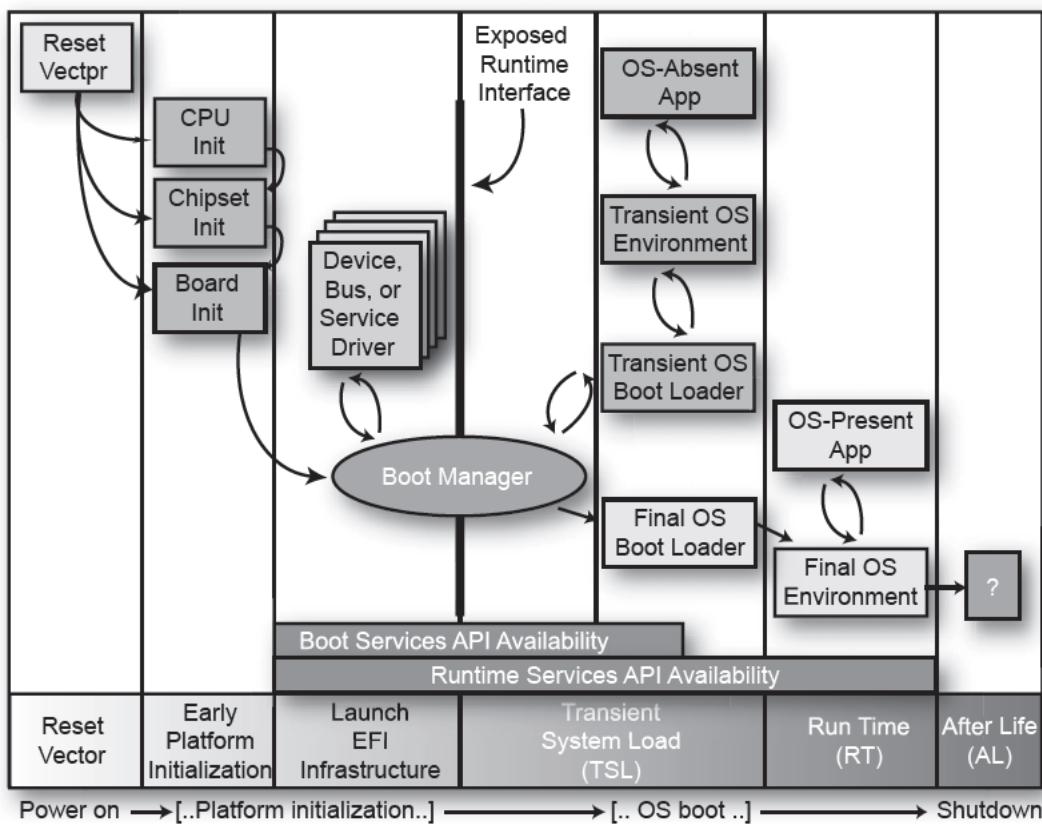


Figure 5.1 Phases of Boot Operation

从图 5.1 可以非常明显的看到，前面提到的两种服务（启动服务和运行时服务）在 EFI 架构运行的早期都可以访问的到，但是当固件栈中的剩余部分在把控制权释放给操作系统加载器以后，只有运行时服务还能被访问。一旦加载器把它自己的环境加载到足够能掌控系统的其他操作，它就会调用 `ExitBootServices()` 来结束启动服务。

原则上，`ExitBootServices()` 是设计给操作系统的，用以表明操作系统的 Loader 已经准备好控制 Platform 及它的所有资源管理。因此到这个时间点，启动服务帮助操作系统加载器准备启动操作系统的任务也就寿终正寝。一旦操作系统加载器控制了系统，并且完成了操作系统的启动过程，只有运行时服务可以被调用。然而，其他的不是操作系统加载器的程序，它们可以选择调用或者不调用 `ExitBootServices()`。这个选择部分取决于这些程序是否设计成要继续使用 EFI 启动服务或者启动服务环境。

难道不是只有一种内存吗？

当 EFI 内存分配以后，它就已经被按照某种分类方法划分成一种类型的内存了，这种分类方法指明了各种特定内存类型的一般用途。比如当我们需要一块缓冲中的数据直到平台操作的运行时阶段也能保持可用，那我们就可以选择用 `EfiRuntimeServicesData` 类型来对这块缓冲进行分配。在分配内存的时候，可能有人会想，为了以防万一，为什么不把所有的内存都以运行时内存类型来进行分配呢？原因在于，这种行为是危险的，因为当平台从启动服务阶段切换到

运行时阶段以后，所有以运行时类型分配的内存对操作系统来说都是被冻结，不能被使用的。有这么一个隐含假设，所有请求分配运行时内存的代码都知道它们在做什么。如果我们只是假定只有一种类型内存的分配的话，我们可以想象到内存泄露的扩散。基于这种考虑，EFI 建立了一套内存类型，这其中的每一种内存类型都有某种指定的使用惯例和它们相对应。

表 5.1 EFI 内存类型和 ExitBootServices()之前的使用方法

助记符	描述
EfiReservedMemoryType	未使用
EfiLoaderCode	加载的 application 的代码部分使用这种内存。(注意:EFI 操作系统加载器是 EFI application)
EfiLoaderCode	加载的 application 的数据部分,和 application 分配 pool 内存时的默认数据分配类型。
EfiBootServicesCode	加载的启动服务驱动的代码部分。
EfiBootServicesData	加载的启动服务驱动的数据部分,和启动服务驱动分配 pool 内存时的默认数据分配类型。
EfiRuntimeServicesCode	加载的运行时服务驱动的代码部分。
EfiRuntimeServicesData	加载的运行时服务驱动的数据部分,和运行时服务驱动分配 pool 内存时的默认数据分配类型。
EfiConventionalMemory	空闲(未分配)内存。
EfiUnusableMemory	探测到错误的内存。
EfiACPIReclaimMemory	保存 ACPI 表的内存。
EfiACPIMemoryNVS	预留给固件使用的地址空间。
EfiMemoryMappedIO	用来给系统固件请求内存映射 IO 区域，操作系统把内存映射 IO 区域映射到虚拟地址，这样这些区域就能被 EFI 运行时服务访问到了。
EfiMemoryMappedIOPortSpace	处理器用来把内存周期转化为 IO 周期的系统内存映射 IO 区域。
EfiPalCode	固件预留给处理器一部分程序的地址空间。

表 5.1 列出了各种内存类型以及它们在运行启动目标(比如说操作系统)之前的相应使用方法。大多数运行时驱动所使用的内存类型在助记符中都包含“runtime”关键字。

然而,为了更好的说明这些内存类型在平台启动的运行时阶段是如何被使用的,我们用表 5.2 来阐明在操作系统加载器调用 ExitBootServices() 之后这些 EFI 内存类型是如何被使用的,表明从预启动阶段到运行时阶段的转变。

表 5.2 EFI 内存类型和 ExitBootServices() 之后的使用方法。

助记符	描述
EfiReservedMemoryType	未使用
EfiLoaderCode	加载器和/或操作系统可以在它们认为合适的时候使用这种内存。注意: 调用 ExitBootServices() 的操作系统加载器使用了一个或者多个 EfiLoaderCode 区域。
EfiLoaderData	加载器和/或操作系统可以在它们认为合适的时候使用这种内存。注意: 调用 ExitBootServices() 的操作系统加载器使用了一个或者多个 EfiLoaderData 区域。
EfiBootServicesCode	一般使用可以访问的内存。
EfiBootServicesData	一般使用可以访问的内存。
EfiRuntimeServicesCode	这个范围内的内存是被加载器和处于工作状态以及 ACPI S1-S3 状态的操作系统所保护的。
EfiRuntimeServicesData	这个范围内的内存是被加载器和处于工作状态以及 ACPI S1-S3 状态的操作系统所保护的。
EfiConventionalMemory	一般使用可以访问的内存。
EfiUnusableMemory	有错误的内存,不能被使用。
EfiACPIReclaimMemory	在 ACPI 被使能之前,这种类型的内存被加载器和操作系统所保护。一旦 ACPI 被使能,这个范围里的内存可以用作一般用途。
EfiACPIMemoryNVS	这个范围内的内存是被加载器和处于工作状态以及 ACPI S1-S3 状态的操作系统所保护的。

EfiMemoryMappedIO	操作系统不使用这种内存。系统里所有的内存映射 IO 信息都来自于 ACPI 表。
EfiMemoryMappedIOPortSpace	操作系统不使用这种内存。系统里所有的内存映射 IO 端口空间信息都来自于 ACPI 表。
EfiPalCode	这个范围内的内存是被加载器和处于工作状态以及 ACPI S1-S3 状态的操作系统所保护的。这种内存也可以具有由其他的处理器实现所定义的属性。

在表 5.2 中，我们可以看到运行时内存类型是怎样被保护的，可以看到启动服务类型的内存是怎么被操作系统回收，当做自己可以使用的内存。

运行时服务是如何展示出来的？

在 EFI 中，固件服务是通过一套 EFI protocol 定义，一系列在某些特殊目的的服务表里的函数指针和 EFI 配置表展现出来的。在这些展现固件 API 的机制中，只有以下两种在进入计算机操作的运行时后还在继续使用。

- 运行时服务表 – EFI 运行时服务表包含所有运行时服务的指针。所有在 EFI 运行时服务表里的元素都是函数指针的原型，这些函数指针在操作系统调用 ExitBootServices() 接管平台以后就有效了。
- EFI 配置表 – EFI 配置表包含一套 GUID/指针对。这张表里的表项数目很容易随着时间而增长，所以才会使用 GUID 来识别各种配置表的类型。对于每一种配置表的类型而言，在配置表中至多只能存在一个实例。

运行时服务表中所展示出来的运行时服务，至少要具备 EFI 兼容平台的 Runtime API 能力的内核。这些函数最少要包括时间，虚拟内存和 Variable 服务。

通过 EFI 配置表展示出来的信息，将会在不同的平台实现之间表现出越来越大的差异性。然而需要注意的一个关键点是，与 GUID/指针对相关联的 GUID 定义了我们怎么去理解指针所指向的数据。指针所指向的内容可以是一个函数或者 API，可以是一张数据表，也几乎可以是其他的任何东西。举几个例子，通过 EFI 配置表展示出来的信息类型可以是 SMBIOS, ACPI 和 MP table，也可以是一块 UNDI 兼容网卡的函数原型。图 5.2 是一张反映 EFI 配置表和一个函数原型实例如何交互的例图。

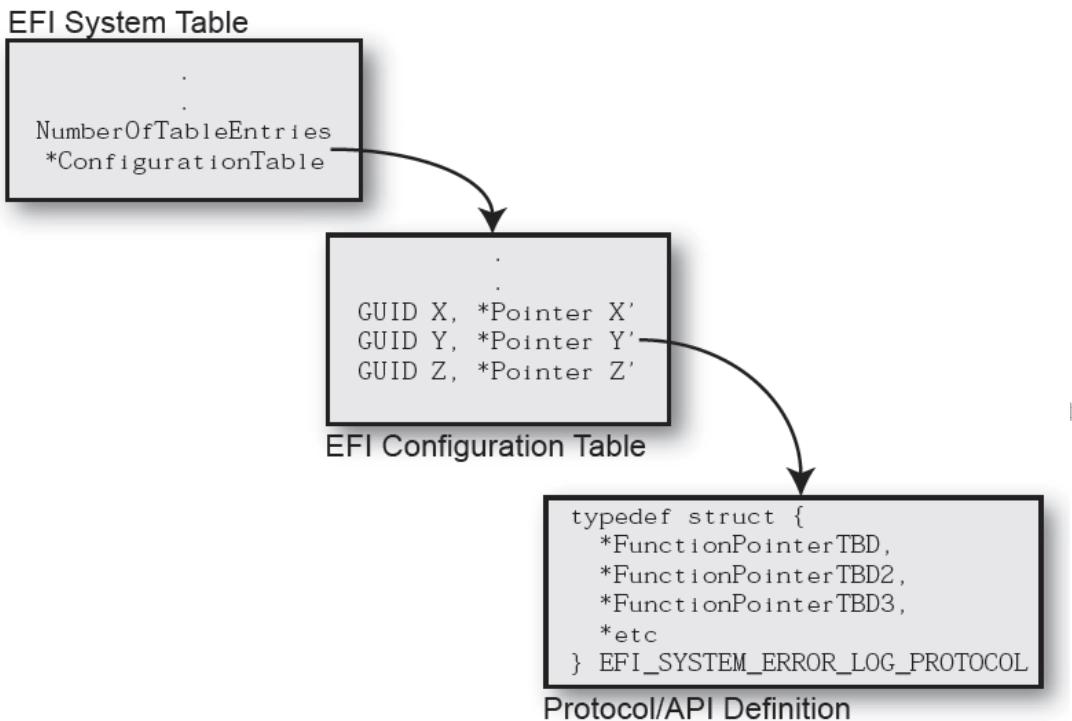


Figure 5.2 Interactions between the EFI Configuration Table and a Function Prototype

图 5.2 EFI 配置表和函数原型的互动

时间服务

这一节描绘的是用于时间相关的函数的核心 EFI 定义，这些函数是处于运行时的操作系统为了访问管理时间信息与服务的硬件所明确要求的。这些接口的目的是提供给在运行时想访问它们的消费者一个抽象，使得这些消费者免于直接去访问传统的硬件设备。表 5.3 中列出的函数存在于 EFI 运行时服务表中。

表 5.3 EFI 运行时服务表中基于时间的函数

名字	类型	描述
GetTime	运行时	返回当前的时间和日期, 以及平台的时间保持能力.
SetTime	运行时	设置当前的本地时间和日期信息.
GetWakeupTime	运行时	返回当前的唤醒警告时钟设置 clock.

SetWakeupTime

运行时

设置系统的唤醒警告时间.

为什么要抽象时间?

基于多种原因，人们可能会选择把访问平台实时时钟(RTC)的方法进行抽象。第一，访问平台 RTC 基本上不存在什么标准机制。许多传统中断也许服务于多个目的，但是一般不太可能抽象出足够多特别有用的信息。一个用户如果想直接和 RTC 进行对话，这个用户不会知道怎么处理除了标准的 IBM CMOS 指令之外的情况。最后，人们获取“现在是什么时候”这条基本信息的方法，随着时间的推移也在发生变化。记住这一点以后，人们就需要平台提供一套抽象，这样调用者就不必担心不同的对 RTC 编程以取得时间信息的方法产生异常的行为，也不必依赖那些使用不标准的，但是文档却又写的很少很差的传统中断来抽象这些同样的数据了。

获取时间

(Page76-85 未完成)

第六章

EFI 控制台服务

不要为你不知道如何掌控的错误情形做测试

—Steinbach's Guideline for Systems Programming

本章介绍了 EFI 如何扩展预启动阶段中控制台支持的传统界限，同时提供了广泛应用于符合 EFI 规范平台的一系列软件的分层方法。大部分平台都至少会有一个基于文本的控制台使用户能够在本地或远程与系统进行交互。在 EFI 中有很多种机制可以实现这种交流。不管是通过远程界面还是通过本地键盘和显示器，甚至通过远程的网络连接来实现，每种机制都有一个共同的基础，可被视为基本的 EFI 控制台支持。这种支持用于处理在 EFI 启动服务环境下运行代码时提供给系统用户的基于文本的输入输出信息。控制台定义可分为三类不同的控制台设备：输入一类，正常输出和错误各一类。

这些接口通过函数调用的定义加以详述，以便在实现中得到最大的灵活度。例如，一个符合规范的系统并不一定需要键盘或屏幕与其直接连接。只要函数的语义得以保留，它的实现就可以任意使用这些接口来引导信息流向，只要能够成功地将信息传递给系统用户即可。

EFI 控制台由两个主要协议组成，分别是 EFI 简单文本输入和 EFI 简单文本输出协议。这两个协议实现了一个基本的基于文本的控制台，它允许平台固件、EFI 应用程序和 EFI OS loaders 向系统管理员提交信息和从系统管理员接收输入。EFI 控制台由 16 位 Unicode 字符、一组简单的输入控制字符（即扫描码）以及一组可编程的面向输出（output-oriented）的可以提供等同于智能终端功能的接口组成。这组基于文本的接口并不原生支持指点设备输入或位图输出。

为保证最大的互操作性，EFI 简单文本输出协议至少应支持基本的可打印 Latin Unicode 字符集，以使标准终端仿真软件能够与 EFI 控制台共同使用。基本 Latin Unicode 字符集实现了一个 ASCII 码的超集，这个超集被扩展为一组 16 位的字符。通过使用外部终端仿真，避免了文本编码被降格转换为对应的 ASCII 码，从而提供了最大的互操作性。

EFI 有许多系统范围内的对控制台的引用。EFI 系统表（EFI System Table）包含六个控制台相关的入口：

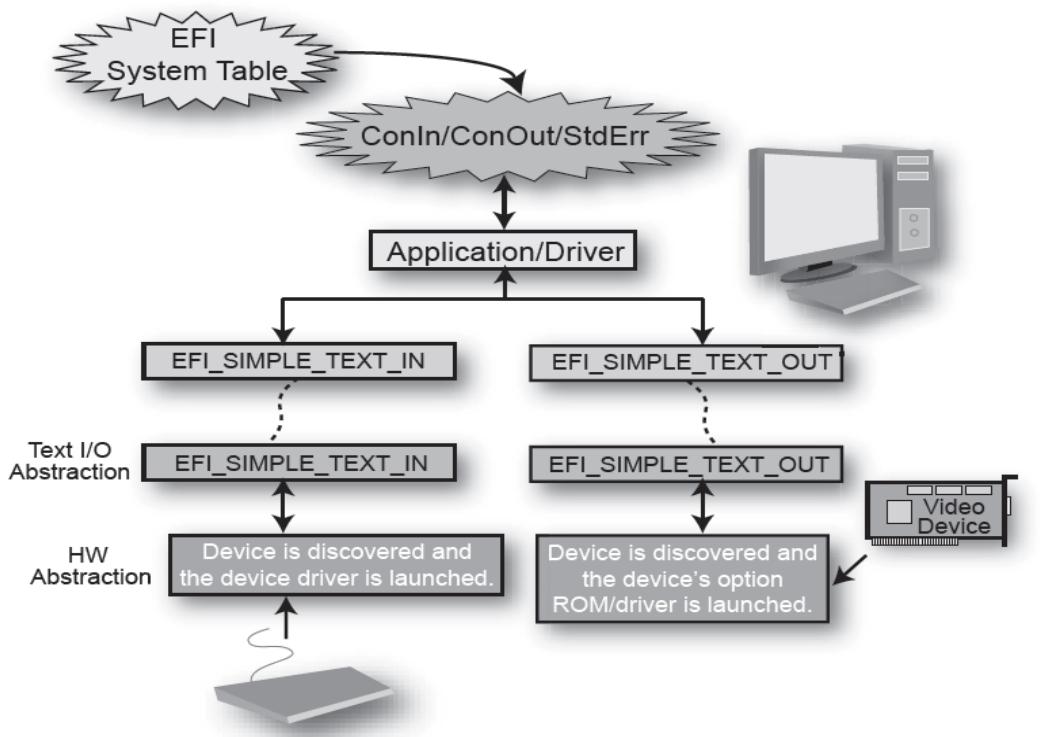
- `ConsoleInHandle` - 活动控制台输入设备的 handle。这一 handle 必须支持 EFI 简单文本输入协议。
- `ConIn` - 一个与 `ConsoleInHandle` 相关联的指向 EFI 简单文本输入协议接口的指针。

- ConsoleOutHandle - 活动控制台输出设备的 handle。该 handle 必须支持 EFI 简单文本输出协议。
- ConOut - 一个与 ConsoleOutHandle 相关联的指向 EFI 简单文本输出协议接口的指针。
- StandardErrorHandle - 活动标准错误控制台设备的 handle。该 handle 必须支持 EFI 简单文本输出协议。
- StdErr - 一个与 StandardErrorHandle 相关联的指向 EFI 简单文本输出协议接口的指针。

在 EFI 中，其它对控制台的系统范围的引用被包含在全局变量定义中。EFI 中一些与之有关的全局变量定义如下：

- ConIn - EFI 全局变量，包含默认的输入控制台的设备路径。
- ConInDev - EFI 全局变量，包含所有可能的控制台输入设备的设备路径。
- ConOut - EFI 全局变量，包含默认的输出控制台的设备路径。
- ConOutDev - EFI 全局变量，包含所有可能的控制台输出设备的设备路径。
- ErrOut - EFI 全局变量，包含默认的错误控制台的设备路径。
- ErrOutDev - EFI 全局变量，包含所有可能的控制台输出设备的设备路径。

图 6.1 所示的是目前为止我们所讨论过的软件分层方式。如果 EFI 应用程序或驱动程序需要通过文本界面进行通信，就可以使用 EFI 系统表里所示的活动控制台来调用支持合适的文本输入或文本输出协议的接口。在初始化过程中，系统表被传递给已启动的 EFI 应用程序或驱动程序，它们就可以立即开始使用所需的控制台。

**Figure 6.1** Initial Software Layering

为了进一步说明这些交互，我们有必要深入研究一下这些文本 I/O 接口究竟看起来是什么样以及它们能有效地负责哪些工作。

简单文本输入协议

简单文本输入协议定义了支持特定 ConIn 设备所需的最小输入。该接口为它的调用者提供了两个基本函数：

- **Reset** - 该函数将输入设备硬件复位。作为初始化过程的一部分，固件/设备进行一次快速但合理的尝试来验证设备是否工作正常。这种硬件验证过程是因不同的实现而异的，且它被留给固件和/或 EFI 驱动程序来实现。

- **ReadKeyStroke** - 该函数从输入设备读取下一个击键。如果没有后续击键，则该函数返回一个 EFI Not Ready 错误。如果有后续击键，则返回一个 EFI 键。一个 EFI 键由一个扫描码和一个 Unicode 字符组成。其中的 Unicode 字符是实际可打印的字符，或者若该键没有用可打印字符表示时则此 Unicode 字符为零，比如控制键或功能键。

当从 ReadKeyStroke() 函数读取一个键时，会生成一个 EFI Input Key。在传统固件中，所有 PS/2 键都有一个因硬件不同而异的扫描码，这也是固件唯一处理的东西。在 EFI 中，有些许变化以促进与本地及远程用户的该数据的合理交换。送回的数据有两个主要部分：

■ Unicode 字符 - 简单文本输入协议定义了一个包含 Unicode 字符的输入流。这个值代表了用 Unicode 编码的 16 位值，该 16 位值对应于用户所按下的键。一些 Unicode 字符有特殊含义，因而被定义为 EFI 支持的 Unicode 控制字符，如表 6.1 所示。

Table 6.1 EFI-supported Unicode Control Characters

Mnemonic	Unicode	Description
Null	U+0000	Null character ignored when received.
BS	U+0008	Backspace. Moves cursor left one column. If the cursor is at the left margin, no action is taken.
TAB	U+0x0009	Tab.
LF	U+000A	Linefeed. Moves cursor to the next line.
CR	U+000D	Carriage Return. Moves cursor to left margin of the current line.

■ 扫描码 - 输入流，除支持 Unicode 字符外还支持 EFI 扫描码。如果扫描码被置为 0x00，则 Unicode 字符有效且应当被使用。如果 EFI 扫描码被置为一个非 0x00 的值，则代表一个特殊的键，如表 6.2 所示。

Table 6.2 EFI-supported Scan Codes

EFI Scan Code	Description
0x00	Null scan code.
0x01	Move cursor up 1 row.
0x02	Move cursor down 1 row.
0x03	Move cursor right 1 column.
0x04	Move cursor left 1 column.
0x05	Home.
0x06	End.
0x07	Insert.
0x08	Delete.
0x09	Page Up.
0x0a	Page Down.
0x0b	Function 1.
0x0c	Function 2.
0x0d	Function 3.
0x0e	Function 4.
0x0f	Function 5.
0x10	Function 6.
0x11	Function 7.

EFI Scan Code	Description
0x12	Function 8.
0x13	Function 9.
0x14	Function 10.
0x17	Escape.

ReadKeyStroke 函数提供了当收到键值时触发一个 EFI 事件的功能。要使用该功能，程序员必须使用 WaitForEvent 或 CheckEvent 服务二者之一。传递给这些服务的事件如下：

- WaitForKey - 当调用 WaitForEvent() 来等待一个按键可用时所使用的事件。

简单文本输入协议所处理的内容与传统固件中的 INT 16h 服务所处理的内容非常相似。其主要区别在于，传统固件服务返回的是按键对应的 8 位 ASCII 码和因硬件不同而异（如 PS/2）的扫描码。

简单文本输出协议

简单文本输出协议用于控制基于文本的输出设备。它是将任何 handle 作为 ConOut 或 StdOut 设备用途时至少需要的协议。另外，EFI 支持此类设备的最小文本模式为至少 80 × 25 个字符。

一个只支持图像模式的视频设备必须能够仿真文本模式的功能。输出字符串中不允许包含除表 6.1 以外的其它任何控制码。光标置位只能通过 SetCursorPosition() 函数来完成。这里强烈建议将向 StdErr 设备输出的文本限制为串行字符串输出。换句话说，不建议在向 StdErr 输出的消息中使用 ClearScreen() 或 SetCursorPosition()，以使我们能够清晰地捕获或观看到输出信息。

简单文本输出协议中还包含了一个指向某种模式数据的指针，如图 6.2 所示。这种模式数据用于确定某指定设备的当前文本设置是什么。这些信息的大部分被用于确定当前的光标位置和给定的前景和背景颜色。此外，程序员还可规定光标是可见或不可见。

```
typedef struct {
    INT32                                MaxMode;
    // current settings
    INT32                                Mode;
    INT32                                Attribute;
    INT32                                CursorColumn;
    INT32                                CursorRow;
    BOOLEAN                               CursorVisible;
} SIMPLE_TEXT_OUTPUT_MODE;
```

Figure 6.2 Mode Structure for EFI Simple Text Output Protocol

简单文本输出协议还包含许多文本输出相关的函数。但是，本章只讨论最常用的几个：

- **OutputString** - 提供将以 NULL 结尾的 Unicode 字符串写入输出设备并显示的能力。所有输出设备还必须支持部分 EFI 1.1 规范中所列出的基本 Unicode 绘图字符。这是一个输出设备上最基本的输出机制。在输出设备上，字符串在当前光标处显示，光标根据表 6.3 所示的规则向后移动。

Table 6.3 Cursor Advancement Rules

Mnemonic	Unicode	Description
Null	U+0000	Ignore the character, and do not move the cursor.
BS	U+0008	If the cursor is not at the left edge of the display, then move the cursor left one column.
LF	U+000A	If the cursor is at the bottom of the display, then scroll the display one row, and do not update the cursor position. Otherwise, move the cursor down one row.
CR	U+000D	Move the cursor to the beginning of the current row.
Other	U+XXXX	Print the character at the current cursor position and move the cursor right one column. If this moves the cursor past the right edge of the display, then the line should wrap to the beginning of the next line. This is equivalent to inserting a CR and an LF. Note that if the cursor is at the bottom of the display, and the line wraps, then the display will be scrolled one line.

通过提供一种允许控制台设备（如视频驱动程序）产生文本接口的抽象，这可以与传统固件的 INT 10h 支持相媲美。简单文本输出接口的生成者负责将 Unicode 文本字符为该设备转换成正确的字型。当一个不能被识别的 Unicode 字符送到 OutputString() API 时，典型的结果是产生一个警告，提示这些字符将被忽略。

- **SetAttribute** - 该函数用来设置函数 OutputString() 和 ClearScreen() 的背景和前景色。EFI1.1 规范中定义了多种前景和背景色。即使设备处于一个无效的文本模式下，也可以设置颜色。能够支持不同种文本颜色的各种设备应在其能力允许的范围内尽量仿真规定的颜色。

- **ClearScreen** - 该函数能清除一个或多个输出设备的显示内容，并将其还原成当前选定的背景色。同时光标的位置将被置于坐标 (0, 0) 处。

- **SetCursorPosition** - 该函数用来设置光标当前位置的坐标。屏幕左上角被定义为坐标 (0, 0)

远程控制台支持

本章的前几节讲述了部分文本输入和输出协议，并使用了一些通过本地设备生成的例子。EFI 同样支持多种远程控制台。这种支持利用了已有的本地接口，但它实现了进出运行中的平台外设备的数据的路由。

一个远程控制台被实例化，通常是源于 EFI 构建一个控制台驱动程序与之挂接的 I/O 抽象时。在这种情况下，我们的讨论先专注于串行接口控制台。很多种控制台传输协议，如 PC-ANSI, VT-100 等，描述了送给和来自机器的数据的格式。

负责产生文本 I/O 接口的控制台驱动程序扮演了 I/O 过滤器的角色。例如，当一个远程键被按下时，可能需要从远程设备构建和传输的不同的数据块。同时在收到数据时，控制台驱动需要将这些信息翻译并转换成对应的 EFI 语意，如 EFI 扫描码和 Unicode 字符。对于所有在本地机器上运行的需要打印消息的应用程序，其机制也是完全一样的。消息由控制台驱动程序接收并被翻译成符合远程终端类型的语意。

表 6.4 举例说明了一个 EFI 扫描码如何映射到 ANSI X3.64 终端，PC-ANSI 终端，或 AT 101/102 键盘。PC ANSI 终端支持以 ASCII 字符 0x1b 开头，后接 ASCII 字符 0x5b（即 “[”）的转义序列。ASCII 字符定义的控制序列须跟在转义序列之后。转义序列不包含空格，但为了方便阅读，我们在表 6.4 中使用了空格。若需要得到 EFI 终端支持的更多信息，请查阅 EFI1.1 规范。

Table 6.4 Sample Conversion Table for EFI Scan Codes to other Terminal Formats

EFI Scan Code	Description	ANSI X3.64 Codes	PC ANSI Codes	AT 101/102 Keyboard Scan Codes
0x00	Null scan code	N/A	N/A	N/A
0x01	Move cursor up 1 row	CSI A	ESC [A	0xe0, 0x48
0x02	Move cursor down 1 row	CSI B	ESC [B	0xe0, 0x50
0x03	Move cursor right 1 column	CSI C	ESC [C	0xe0, 0x4d
0x04	Move cursor left 1 column	CSI D	ESC [D	0xe0, 0x4b
0x05	Home	CSI H	ESC [H	0xe0, 0x47
0x06	End	CSI K	ESC [K	0xe0, 0x4f
0x07	Insert	CSI @	ESC [@	0xe0, 0x52
0x08	Delete	CSI P	ESC [P	0xe0, 0x53
0x09	Page Up	CSI ?	ESC [?	0xe0, 0x49
0xa0	Page Down	CSI /	ESC [/	0xe0, 0x51

表 6.5 显示了 PC-ANSI 和 ANSI X3.64 的部分用来调整文本显示器的显示属性或文本显示属性的控制序列。

Table 6.5 Example Control Sequences that Can Be Used in Console Drivers

PC ANSI Codes	ANSI X3.64 Codes	Description
ESC [2 J	CSI 2 J	Clear Display Screen.
ESC [0 m	CSI 0 m	Normal Text.
ESC [1 m	CSI 1 m	Bright Text.
ESC [7 m	CSI 7 m	Reversed Text.
ESC [30 m	CSI 30 m	Black foreground, compliant with ISO Standard 6429.
ESC [31 m	CSI 31 m	Red foreground, compliant with ISO Standard 6429.
ESC [32 m	CSI 32 m	Green foreground, compliant with ISO Standard 6429.
ESC [33 m	CSI 33 m	Yellow foreground, compliant with ISO Standard 6429.
ESC [34 m	CSI 34 m	Blue foreground, compliant with ISO Standard 6429.

图 6.3 说明了具有文本 I/O 抽象的远程串行接口的软件分层结构。该图与展示了本地设备上的相同文本 I/O 抽象的图相比，主要的差别是本图多了一个软件驱动程序层。在前面的例子中，本地设备由一个已启动的代理发现，它将依次建立一套文本 I/O 抽象。在远程设备的情况下，本地设备是一个串行设备，其上有一层控制台驱动程序，并且它也会依次建立一套文本 I/O 抽象。

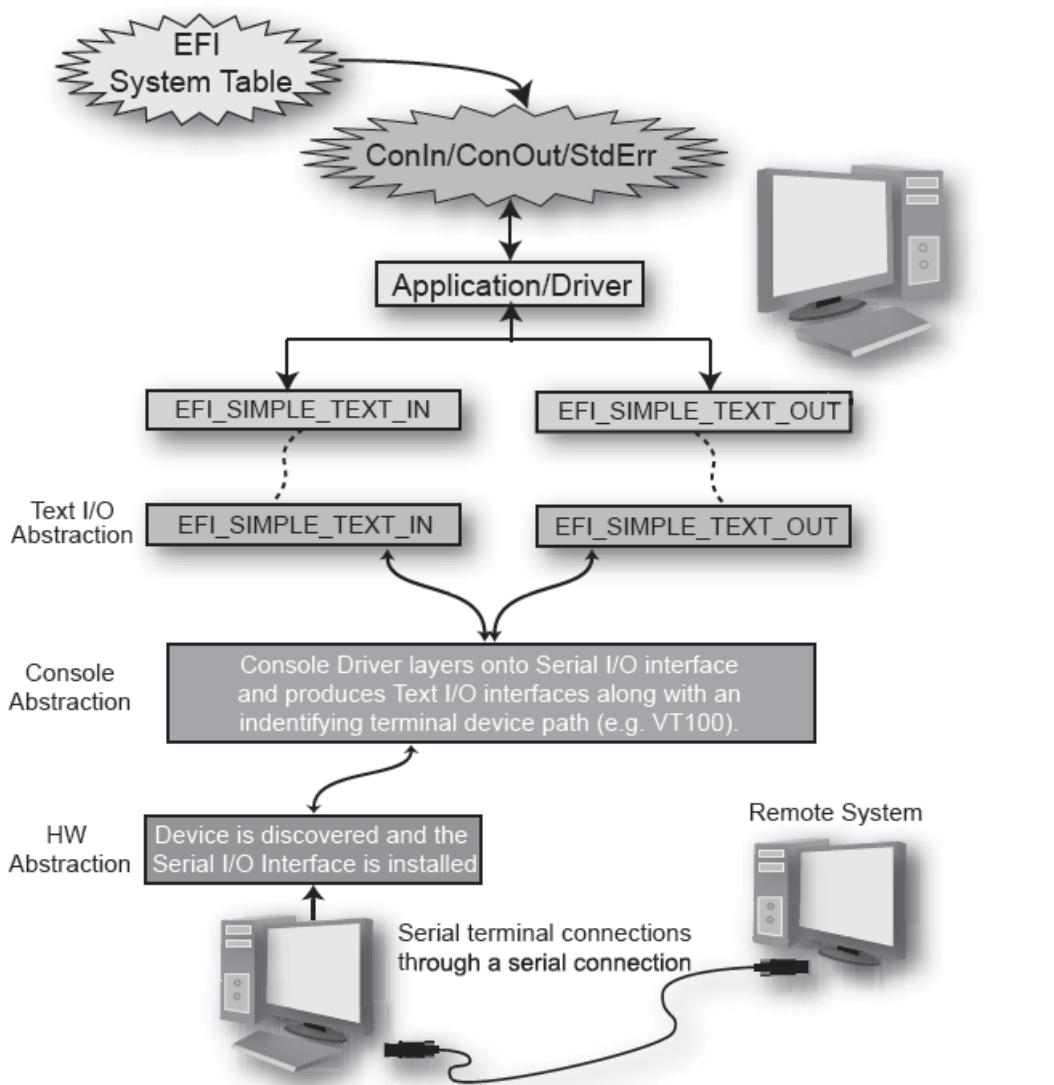


Figure 6.3 Remote Console Software Layering

控制台分离器

描述各种控制台设备的能力构成了新的有趣的可能性。在上一代固件中，程序员只能通过一种途径来描述文本 I/O 的来源和目的地。现在，规定哪些控制台是活动控制台的 EFI 变量由设备路径规定。在这种情况下，这些设备路径是多实例的，这意味着可以有多于一个的目标设备可作为活动输入或者活动输出。例如，如果程序员想让应用程序将文本同时打印到本地屏幕和远程终端的屏幕上，通过定制软件来实现这种特殊应用将是非常不切实际的。在 EFI 提供的方案中，通过控制台分离/合并能力，应用程序可以只使用 EFI 提供的标准文本接口，控制台分离器会将这些文本请求送至合适的单个或多个目标设备上。这种方法对输入和输出流都同样适用。

上述方法只有在如下条件下才能生效：在符合 EFI 规范的平台初始化后，控制台分离器将自己作为主活动控制台装入 EFI 系统表中。这样，它继而可以开始监控整个平台，同时其他 EFI 文

本接口作为协议被安装，且控制台分离器为给定的控制台变量保留一个用户选定的设备的运行标记，如 ConOut, ConIn, 或 ErrOut。

图 6.4 描述了一个应用场景，其中应用程序调用 EFI 文本接口，而 EFI 文本接口又依次调用 EFI 系统表控制台接口。这些接口是控制台分离器的组成部分，控制台分离器然后将来自应用程序的文本 I/O 请求送至依平台配置的控制台。

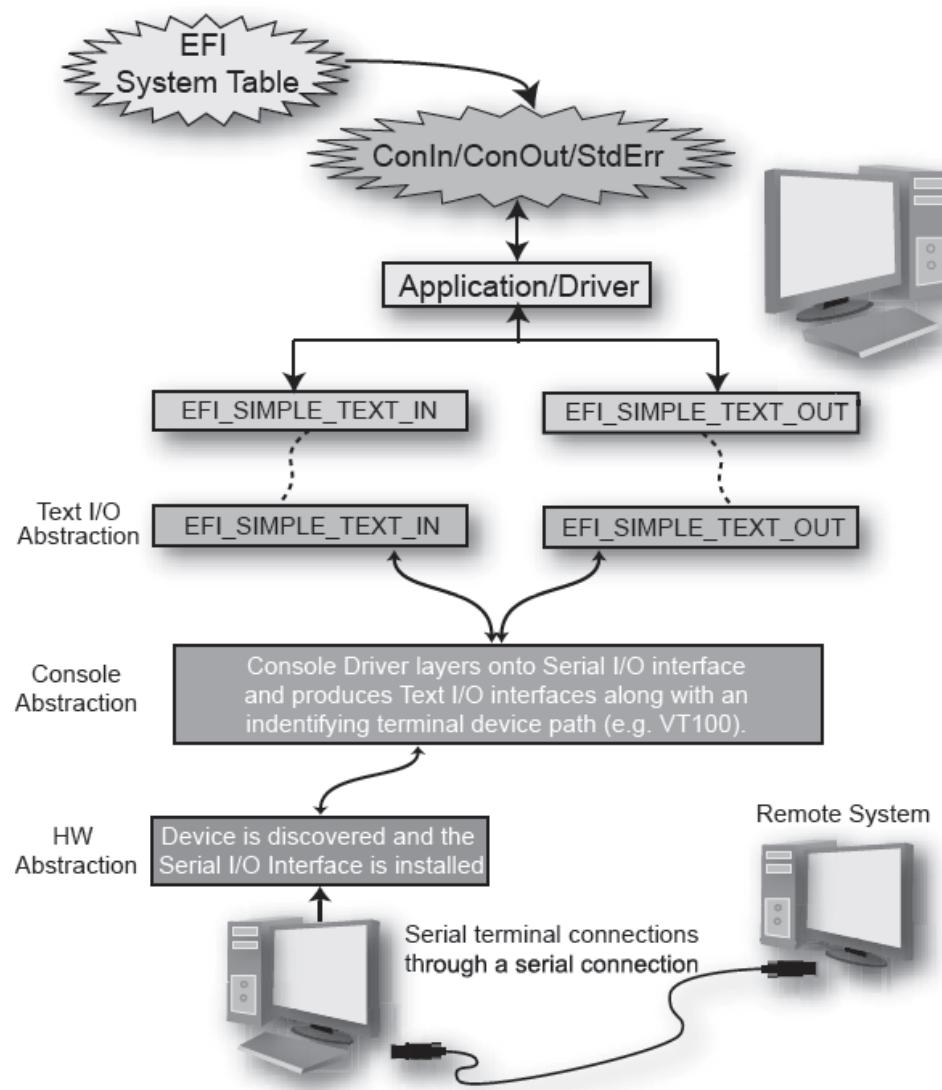


Figure 6.4 Software Layering Description of the EFI Console Splitter

网络控制台

EFI 也提供了与网络中远程平台建立数据连接的能力。只要安装了合适的驱动程序，程序员也可以让符合 EFI 规范的平台支持一套文本 I/O 的抽象。前面讨论过，硬件接口（例如，串行设备、键盘、视频、网络接口卡）都有一个抽象。与这一概念类似，其他部件建立在这种硬件

抽象的基础之上，来提供一个工作软件栈。

下面是一些 EFI 可以包含的网络组件

■ 网络接口识别 - 这是一个由通用网络驱动程序接口 (UNDI) 产生的可选协议，它用于生成简单网络协议。这个协议只有在底层网络接口是 16 位 UNDI, 32/64 位 S/W UNDI, 或 H/W UNDI 时才是必须的。它被用来获取底层网络接口的类型和版本信息。

■ 简单网络协议---这个协议给网络适配器提供了一个信息包一级的接口。另外它还提供了初始化网络接口、发送包、接收包和关闭网络接口的服务。

为了解释一个通用网络控制台的模型，你可以描述一个初始的硬件抽象，它与 UNDI 驱动程序产生的网络接口控制器 (NIC) 直接通信。这继而有一个位于 UNDI 之上的简单网络协议层。它能提供基本的网络抽象接口，如 Send 和 Receive。在此之上可能会安装一个传输协议（如 TCP/IP 协议栈）。对于大多数系统而言，一旦提供了一个确定的传输机制，我们可以在平台内搭建各种扩展程序，如 Telnet Daemon，可以让远程用户通过网络连接来登录系统。最终，这个守护程序会生成并负责处理已经在前面章节描述过的正常文本 I/O 接口。

图 6.5 描述了一个远程机器能够通过网络连接访问本地符合 EFI 规范平台的例子。将软件栈顶层 (EFI_SIMPLE_TEXT_IN 和 EFI_SIMPLE_TEXT_OUT) 作为应用程序与之对话的可互操作界面提供，这允许所有的标准 EFI 应用程序平滑的利用平台的控制台支持。再加上控制台分离和合并的固有能力，你就具备了通过一个更健壮的方式与平台交互的能力，而不再需要大量经过特殊调整的软件。

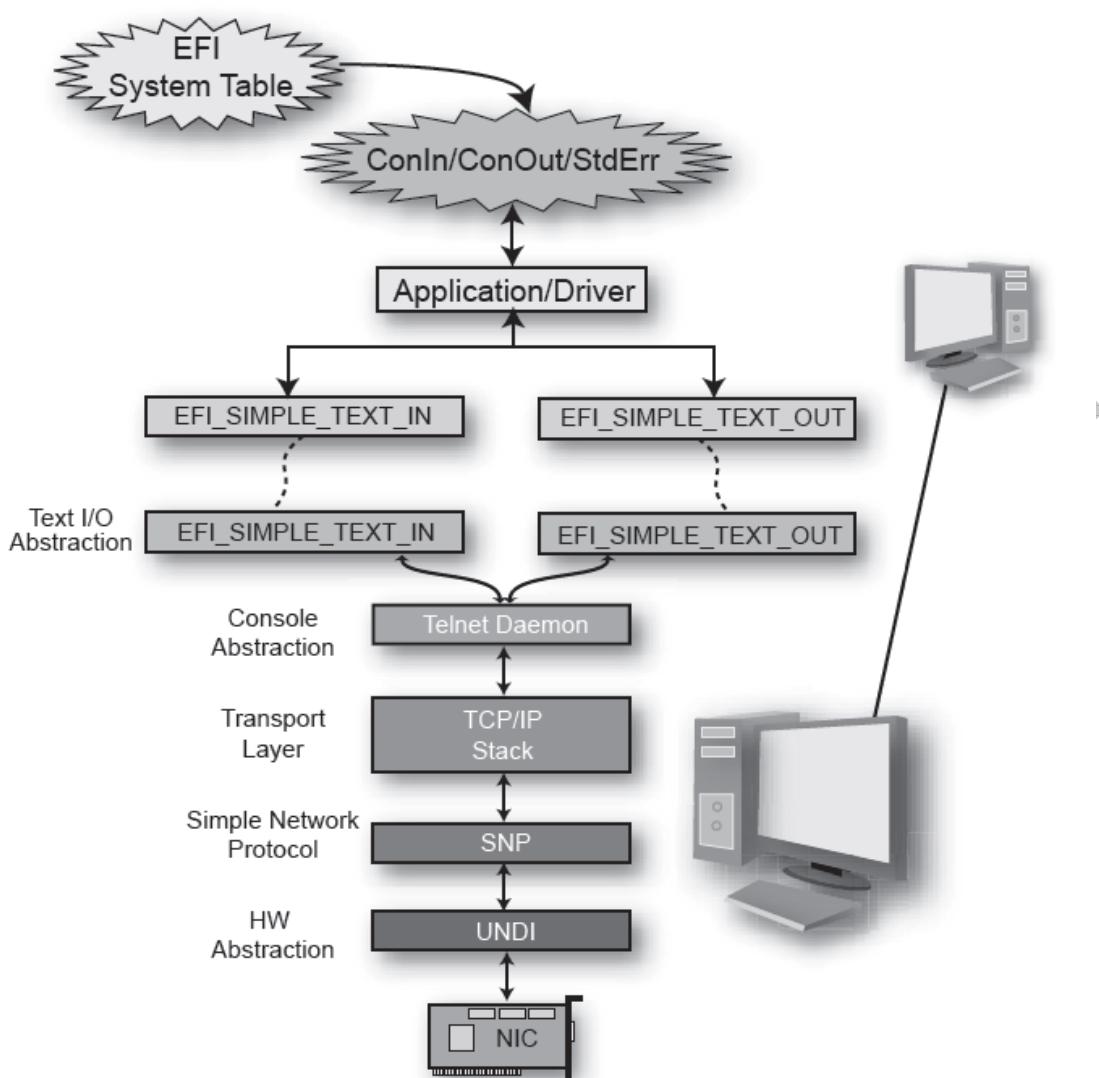


Figure 6.5 Example of Network Console Software Layering

第七章

不同类型的平台

多样性是生活的调料，它让生活丰富多彩。

----威廉·考柏

本章介绍了不同的平台类型和基于 Framework 的具体实现，如嵌入式系统，掌上电脑，台式机和服务器。除了提供一个人们熟知的个人电脑平台间的 BIOS replacement, Framework 架构同样还可以用于为服务器、手持设备、电视等构建引导和初始化设置的环境。这些设备可能包括在个人电脑中常见的 IA-32 处理器、具有低功耗特点的英特尔 XScale 处理器、以及安腾 Itanium 系统中的大型机处理器。此外，本章还探讨了用于构造标准 PC 平台所必需的 PEI 模块和 DXE 驱动程序。并且，描述了这些模块中用于仿真的部分和使用英特尔 Xscale 处理器的 PDA.

图 7.1 是一个典型的系统框图，展示了各种元件，集成北桥、南桥和 Super I/O, 和除此之外其他可能的元件。这些框图代表了系统主板制造的组分部分。每个芯片和平台上的组件都有与之相关联的模块或驱动程序来处理它们各自的初始化。除了主机板上的组成部分，该平台的初始内存映射也需要有特定的区域分配。图 7.2 展示了 PC 平台的内存映射。在这个平台的设置中，系统闪存的大小是 512K 字节的。为了让奔腾 4 处理器在系统复位后取得的第一条操作码来自系统闪存上，系统的闪存被安排到 32 位地址空间的最高端。系统复位的位置位于 32 位地址空间的最后的 16 个字节里。在 SEC 阶段，基于 Framework 的控制流允许 SEC 文件中的初始代码与平台固件相关。来自 SEC 的其他模块才会被执行。

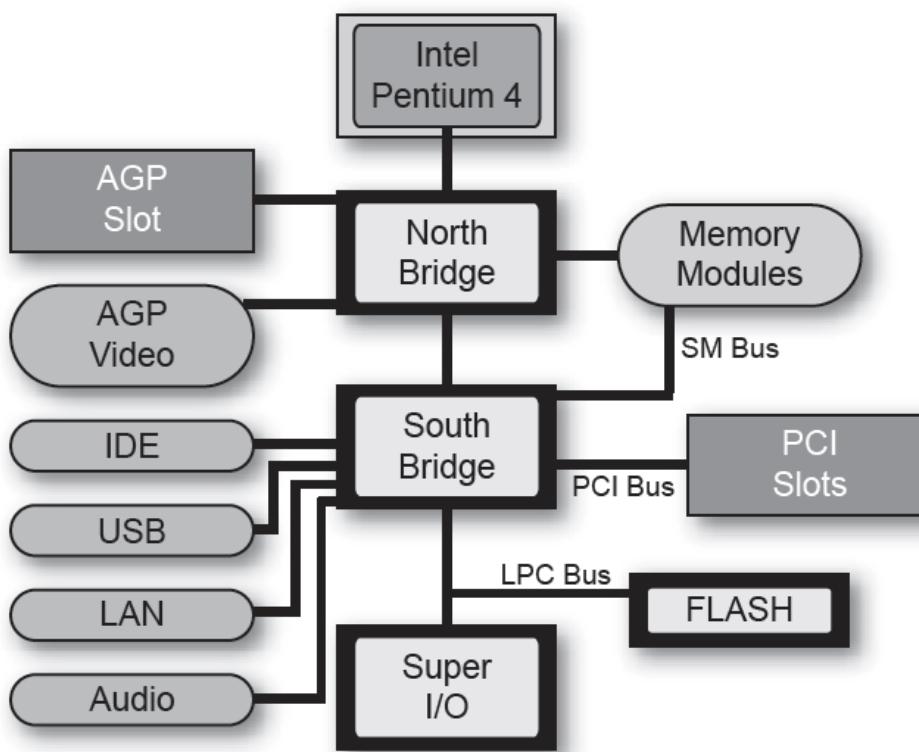


Figure 7.1 Typical PC System

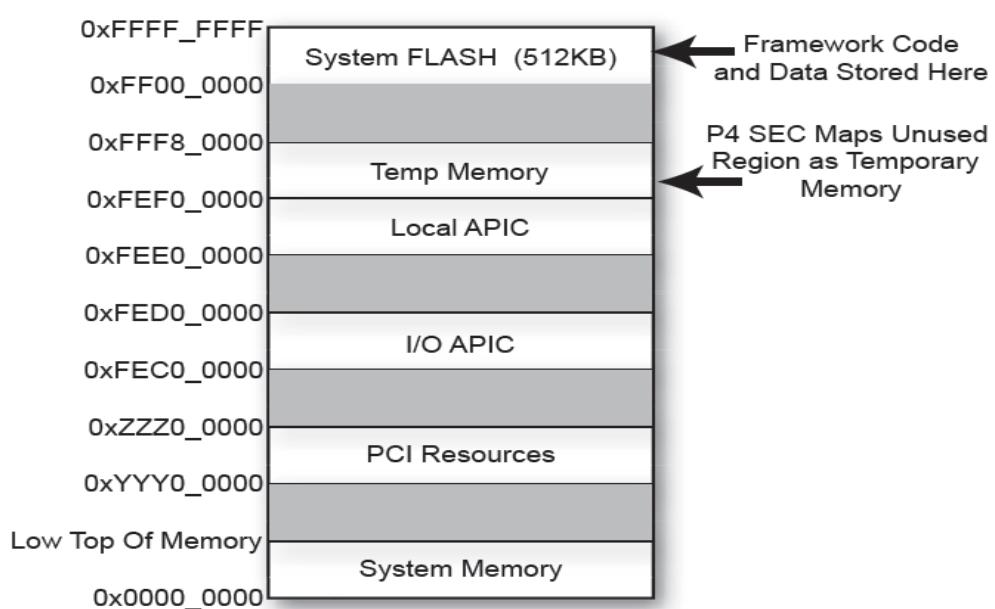


Figure 7.2 Address Map

在深入 PC 固件的各个组成部分的加载之前，先回顾一下其他一些平台。其中包括低功耗 IA32 的 CPU 或 Intel XScale 处理器的无线个人数字助理。然后再将平台扩大到服务器。如图 7.3 所示。

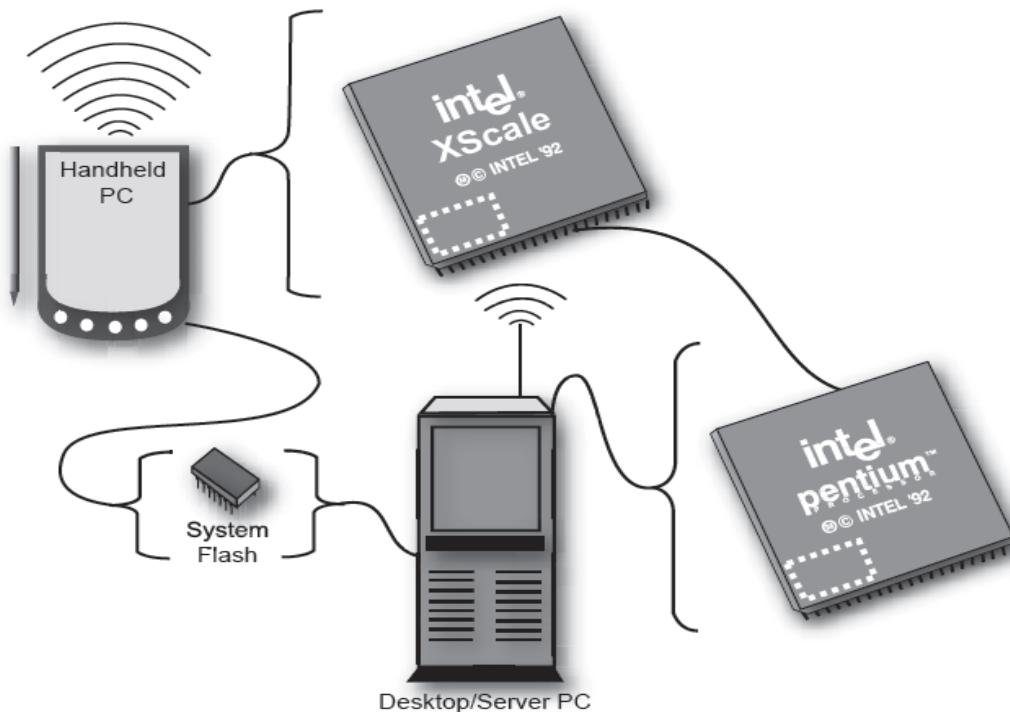
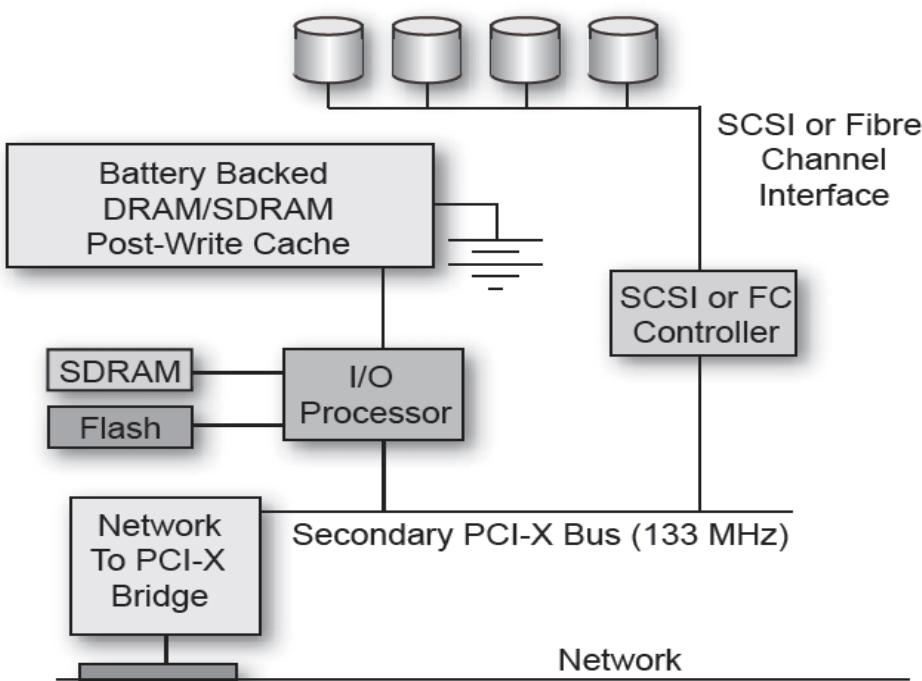


Figure 7.3 Span of Systems

图 7.4 显示了一个智能 I/O 板卡，即存储加速器，该 I/O 板卡是通过一个 PCI 总线插槽连接到一台服务器上的。当服务器重启时，该 I/O 板卡会独立重启。在这个板卡上，一颗特殊的 Intel Xscale 处理器会调用 PEI 模块（PEIMs）和 DXE 驱动去初始化本地 SCSI 控制器和 SDRAM 组件。然后，该 I/O 板卡可以引导到一个 EFI 版本的嵌入式操作系统，如 BSD Unix 等，也可以继续用 DXE 部件来作为 I/O runtime 来支持服务器主机的 CPU 复杂的读/写请求。这演示了怎样跨越如此多不同的拓扑结构的平台理念。这些拓扑结构包括典型的、开放式 PC 和无核的，封闭的嵌入式系统 I/O 板卡。

**Figure 7.4** An Intel XScale®-based System

现在，让我们更详细的探讨图 7.1 中电脑的各组件。在一个重启事件，比如上电复位，从休眠唤醒等后重启（比如带电的重启，休眠后的恢复等等）之后，PEI 阶段程序会被立刻执行。直到在主记忆体（如内存）被初始化后，PEI 模块都是在闪存存储器上运行的。

图 7.5 显示了基于 PC 平台的 PEIMs 模块组。不同的商业利益会提供不同的模块。例如，在“Lakeport”平台上，英特尔会提供 945 芯片组（图形内存控制器中心）内存控制器模块和 ICH6（I/O 控制器中心）模块。此外，针对连接到 ICH 的 SMBus（系统管理总线），提供了基于 ICH 的 SMBus 模块。“Status code”模块描述了一种可以用来作为平台相关的代码的调试方法，它可以用来发送调试信息，如向 I/O 端口 80H 丢字节的代码一样。

Pentium 4 CPU PEIM	Generic	Init and CPU I/O
DXE IPL PEIM	Generic	Starts DXE Foundation
PCI Configuration PEIM	PCAT	Uses I/O 0XCF8, 0XCFC
Stall PEIM	PCAT	Uses 8254 Timer
Status Code PEIM	Platform	Debug Messages
SMBUS PEIM	South Bridge	SMBUS Transactions
Memory Controller PEIMs	North Bridge	Read SPD, Init Memory
Motherboard PEIM	Platform	FLASH Map, Boot Policy

Figure 7.5 Components of PEI on PC

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

在图 7.5 中所列的用于 ICH 上的 SMBus 模块是一个标准 PEIM 至 PEIM 的接口 (PPI)，如图 7.6 所示。这样内存控制器模块就可以用 SMBus 读命令去得到实体内存的 DIMM (双列直插内存模块) 的 SPD (串行存在检测) 数据的信息了。SPD 数据包括内存的大小，时间，以及有关内存模块的其他细节。内存初始化模块可以使用 EFI_PEI_SMBUS_PPI，这样 GMCH 相关的内存初始化模块并不需要知道是哪个组件提供的 SMBus 读写功能。事实上，许多集成的 Super I/O (SIO) 部件也会提供一个 SMBus 控制器。在这种平台上就可以用 SIO SMBus 模块去取代 ICH SM SMBus 模块，而不需要更改内存控制器模块。

```

typedef
EFI_STATUS
(EFI API *PEI_SMBUS_PPI_EXECUTE_OPERATION) (
    IN      EFI_PEI_SERVICE           **PeiServices,
    IN      struct EFI_PEI_SMBUS_PPI  *This,
    IN      EFI_SMBUS_DEVICE_ADDRESS   SlaveAddress,
    IN      EFI_SMBUS_DEVICE_COMMAND   Command,
    IN      EFI_SMBUS_OPERATION       Operation,
    IN      BOOLEAN                  PecCheck,
    IN OUT     UINTN                  *Length,
    IN OUT     VOID                   *Buffer
);

typedef struct {
    PEI_SMBUS_PPI_EXECUTE_OPERATION Execute;
    PEI_SMBUS_PPI_ARP_DEVICE        ArpDevice;
} EFI_PEI_SMBUS_PPI;

```

Figure 7.6 Code fragment for a PEIM PPI

图 7.6 PEIM PPI 代码片段

除了我们之前介绍的 EFI_PEI_SMBUS_PPI 外，还有许多可能的实现方式。图 7.7 显示了一个前述的用于实现在 ICH 上的 SMBus 读操作代码片段。注意它使用的是 CPU I/O，从而来实现对 ICH 部件的 I/O 操作的抽象化。实际上用 C 编写的代码逻辑就意味着只要通过对目标架构的简单汇编就可以在英特尔 XScale 或者 Itanium 的系统上(使用相同的 ICH6)使用相同的源代码。

```

#define SMBUS_R_HDO 0xEFA5
#define SMBUS_R_HBD 0xEFA7

EFI_PEI_SERVICES           *PeiServices;
SMBUS_PRIVATE_DATA         *Private;?
UINT8   Index, BlockCount  *Length;
UINT8                           *Buffer;

BlockCount = Private->CpuIo.IoRead8 (
    *PeiServices, Private->CpuIo, SMBUS_R_HDO);
if (*Length < BlockCount) {
    return EFI_BUFFER_TOO_SMALL;
} else {
    for (Index = 0; Index < BlockCount; Index++) {
        Buffer[Index] = Private->CpuIo.IoRead8 (
            *PeiServices, Private->CpuIo, SMBUS_R_HBD);
    }
}

```

Figure 7.7 Code Fragment of PEIM Implementation

在 PEI 阶段之后，DXE 核心需要一系列的平台相关的，CPU 和芯片组特定的驱动程序，以提供一个完整的 DXE/EFI 支持服务。图 7.8 给出学习 PC 平台必须的部分架构协议。

Watchdog	Generic	Uses Timer-based Events
Monotonic Counter	Generic	Uses Variable Services
Runtime	Generic	Platform Independent
CPU	Generic	Pentium 4 DXE Driver
BDS	Generic	Use Sample One for Now
Timer	PCAT	Uses 8254 Timer
Metronome	PCAT	Uses 8254 Timer
Reset	PCAT	I/O 0xCF9
Real Time Clock	PCAT	I/O 0x70-0x71
Security	Platform	Platform Specific Authentication
Status Code	Platform	Debug Messages
Variable	Platform	Depends on FLASH Map

Figure 7.8 Architectural Protocols

事实上，DXE Foundation 并不用考虑任何与时间维持逻辑，中断控制器指令集等相关的问题，这样 DXE Foundation 的 C 代码可用于很多种类目标的平台，而无须改变 Foundation 代码。相反，Architectural Prorocols (APs) 的各式各样的集合会影响 Foundation 端口。

系统中需要被抽象处理的一个方面的就是时间的管理。在 PC/AT 兼容芯片组上的时间维持硬件，如 8254 定时器，与安腾处理器 CPU 集成定时器计数器或者 Intel XScale 处理器上的时间维持逻辑 (ITC) 是不同的。因此，为了实现一个独立的 DXE Foundation 的看门狗定时器的逻辑，访问 CPU /芯片组的上的特殊定时器具体是通过 Timer Architectural Protocol 来实现。此 AP

提供一系列的服务，如获取和设置系统时间。。设置系统时间将会在我们的平台引用类中来加以审查。

首先，图 7.9 提供了一套基于 NT32 平台的定时器的服务实例。NT32 是一个指令执行就象用户模式的进程一样的 32 位 Microsoft Windows System 虚拟的框架平台。这是一个“软”平台，该平台的功能是通过抽象 Win32 服务来实现的。因此，这个 AP 的实现就不需要访问 I/O 控制器或芯片组控制/状态寄存器。相反，该 AP 调用了 Win32 提供一系列的相互排斥和操作系统线程来模拟计时器的服务。

```

EFI_STATUS
TimerDriverSetTimerPeriod (
    IN EFI_TIMER_ARCH_PROTOCOL  *This,
    IN UINT64                  TimerPeriod
)
{
    .
    .
    gWinNt->EnterCriticalSection (&mNtCriticalSection);
    mTimerPeriod = TimerPeriod;
    mCancelTimerThread = FALSE;
    gWinNt->LeaveCriticalSection (&mNtCriticalSection);
    mNtLastTick = gWinNt->GetTickCount ();
    mNtTimerThreadHandle = gWinNt->CreateThread (
        NULL,
        0,
        NtTimerThread,
        &mTimer,
        0,
        &NtThreadId);
    .
    .
}

```

Figure 7.9 NT32 Architectural Protocol

NT32 的实现与一台 Framework 的裸机实现方式是截然不同的。图 7.10 给出了一个硬件实现的实例。其中英特尔 XScale 系统级芯片（SOC）内存映射寄存器是通过同一 AP 设置计时器接口来访问。DXE Foundation 是不能区分虚拟 NT32 平台服务和英特尔 XScale 处理上的实际硬件之间的不同的。

```

EFI_STATUS
TimerDriverSetTimerPeriod (
    IN EFI_TIMER_ARCH_PROTOCOL *This,
    IN UINT64                 TimerPeriod
)
{
    UINT64  Count;
    UINT32  Data;

    Count = DivU64x32 (MulU64x32 (TimerPeriod, OST_CRYSTAL_FREQ) + 5000000,
                        10000000, NULL);
    mCpuIo->Mem.Read  (mCpuIo, EfiWidthUint32, OSCR_BASE_PHYSICAL, 1, &Data);
    Data += (UINT32)Count;
    mCpuIo->Mem.Write (mCpuIo, EfiWidthUint32, OSMRO_BASE_PHYSICAL, 1, &Data);
    mCpuIo->Mem.Read  (mCpuIo, EfiWidthUint32, OIER_BASE_PHYSICAL, 1, &Data);
    Data |= (UINT32)1;
    mCpuIo->Mem.Write (mCpuIo, EfiWidthUint32, OIER_BASE_PHYSICAL, 1, &Data);
    mCpuIo->Mem.Read  (mCpuIo, EfiWidthUint32, ICMR_PHYSICAL, 1, &Data);
    Data |= (UINT32)(1 << SA_OSTO_IRQ_No);
    mCpuIo->Mem.Write (mCpuIo, EfiWidthUint32, ICMR_PHYSICAL, 1, &Data);
}

```

Figure 7.10 AP from Intel XScale®

最后，为了对 PC/AT 系统以及 20 世纪 80 年代中期的 ISA I/O 硬件提供支持，需要一个额外的 AP 服务来实现。图 7.11 图 7.11 显示了访问 8254 计时器时相同的设置计时器服务，同时注册一个 8259 可编程中断控制器（PIC）的中断。这种方式被称为 PC/AT 版的 AP，因为从个人电脑以来，所有 PC-Xt 都支持这些硬件接口。对于在这一章所讲 PC，其 ISA I/O 资源 ICH 部件都有支持，而对于先前的 PC，他们都有各自独立的元件给予支持。

```

EFI_STATUS
TimerDriverSetTimerPeriod (
    IN EFI_TIMER_ARCH_PROTOCOL *This,
    IN UINT64                 TimerPeriod
)
{
    UINT64  Count;
    UINT8   Data;

    Count = DivU64x32 (MulU64x32(119318, (UINTN) TimerPeriod) + 500000,
                        1000000, NULL);
    Data = 0x36;
    mCpuIo->Io.Write(mCpuIo, EfiCpuIoWidthUint8, TIMER_CONTROL_PORT, 1, &Data);
    mCpuIo->Io.Write(mCpuIo, EfiCpuIoWidthUint8, TIMERO_COUNT_PORT, 2, &Count);
    mLegacy8259->EnableIrq (mLegacy8259, Efi8259Irq0, FALSE);
}

```

Figure 7.11 AP for PC/AT

除了为了提供广度平台移植 AP 的实现多样，DXE 额外的功能就是支持各类目标平台。在 EFI 中，与平台互动是通过控制台输入和输出来实现的，一台 PC 控制台的输入设备是典型的 PS/2 或 USB 键盘，输出设备是一个 VGA 或增强型视频显示。不过，我们之前研究过 I/O 卡，是没有传统意义上的“核”或显示设备。这些特殊的嵌入式平台系统中可能只有一个简单的串口。有趣的是，同一台电脑的硬件再没有传统显示器也可以运行并且通过一个简单的串口与用户互动。图 7.12 显示一个 EFI 系统上的通过串口建立的控制台的层次图。

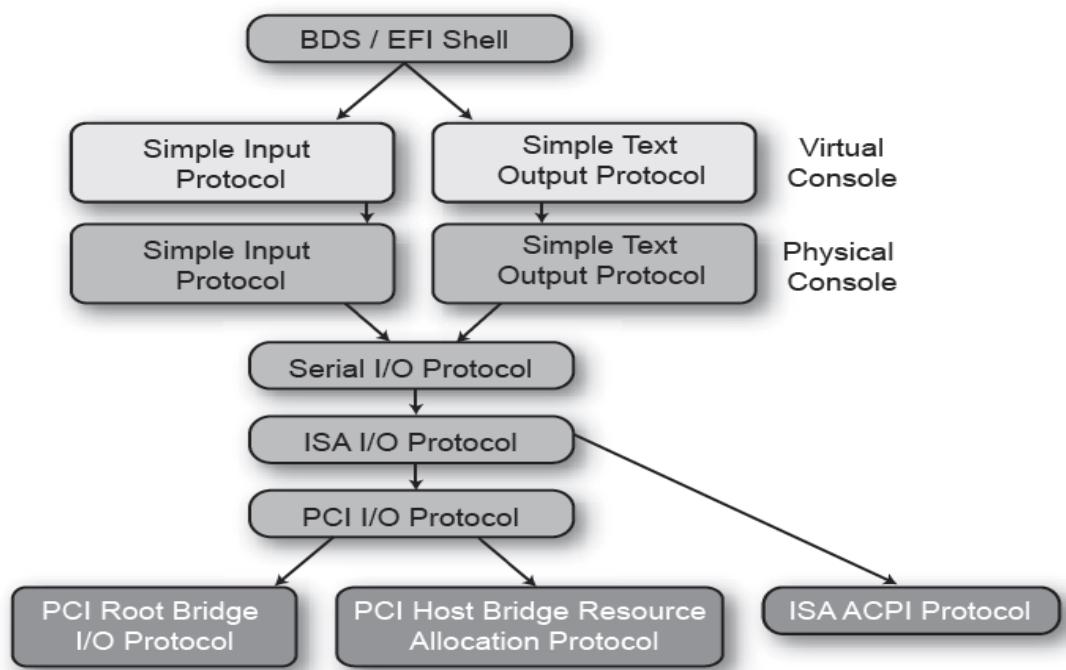


Figure 7.12 Console Stack on a PC

为了建立这个控制台堆，BDS 或 EFI shell 给用户提供了一个应用程序或命令行界面(CLI)。简单的输入输出协议是通过一个基于串行 I/O 协议的控制台驱动程序发布的。。针对于基于 PCI 的电脑，一个 PCI root bridge 协议允许访问串口的控制和状态寄存器;对于一个内部集成的 UART/串口的英特尔 XScale 平台，可能存在一个备用低级别的用于访问这些相同寄存器的协议，

在此平台上分层，图 7.13 中列出的组件，描述了建立这一控制台堆所需要的 DXE 和 EFI 组件。正如在 PEI 模块一样，不同的利益可以提供不同的 DXE 和 EFI 驱动程序。例如，超级 I/O 供应商可能提供 ISA ACPI 驱动程序，芯片厂商提供的 PCI root bridge (如 PC 中芯片组)，一个平台控制台，及一套基于 PC/AT 在 ISA 硬件可重用的组件。

BDS / EFI Shell	Generic	
Console Splitter	Generic	
Terminal	Generic	
ISA Serial	PCAT	
ISA Bus	Generic	
PCI Bus	Generic	
Console Platform	Platform	Platform Specific Policy
PCI Root Bridge	North Bridge	Work with Chipset Vendor
PCI Host Bridge	North Bridge	Work with Chipset Vendor
ISA ACPI	Super I/O	Work with Super I/O Vendor

Figure 7.13 Components for Console Stack

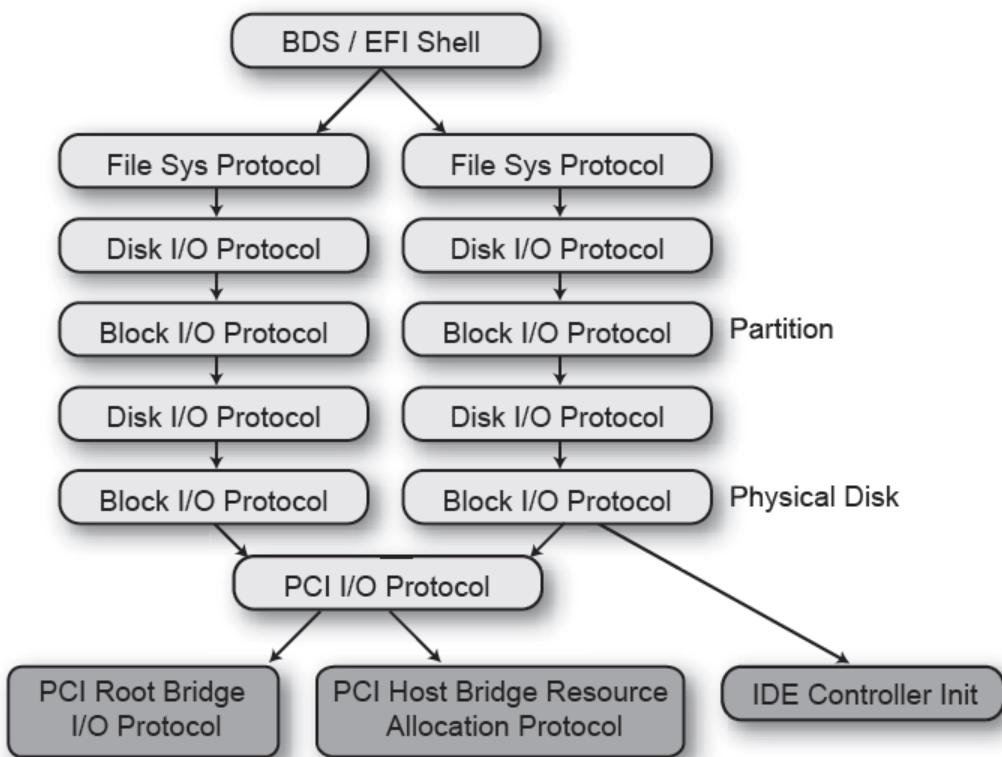
除了控制台组件，其他几个 PEI 模块和 DXE 组件也需要包在 firmware volume 中。图 7.4 上列出了这些用于提供其他功能的部件。包括与特定平台相关的 EFI 变量的存储，平台安全政策，和平台的配置。

Status Code	PEI	Platform
Memory Controller	PEI	North Bridge
SMBUS	PEI	South Bridge
Motherboard	PEI	Platform
Security	DXE	Platform
Status Code	DXE	Platform
Variable	DXE	Platform
Console Platform	DXE	Platform
PCI Root Bridge	DXE	North Bridge
PCI Host Bridge	DXE	North Bridge
ISA ACPI	DXE	Super I/O

Figure 7.14 DXE Drivers on a PC

EFI 变量可以存储在闪存的各个地方（或服务器上的一个服务处理器中），因此，需要一个驱动程序来抽象化变量的存储。出于安全考虑，供应商可以要求这方面的组件更新签署或模块派遣为一个可信平台模块（TPM）。安全驱动程序将抽象化的这些安全考量。

最后一个特色是描述对磁盘子系统支持的 DXE 驱动程序分层部件，即集成设备电子（IDE）和 EFI 文件系统。磁盘子系统的协议到文件系统的实例见图 7.15。

**Figure 7.15** IDE Stack

同样的，EFI shell 或 BDS 依然位于该协议分层的顶部。简单的文件系统（FS）协议的实例提供了应用程序读/写/开启/关闭文件的功能。FS 协议位于磁盘 I/O 协议之上。磁盘 I/O 提供字节级访问以扇面区域为导向的块设备。因此，磁盘 I/O 是唯一一个提供对硬件特定的块 I/O 抽象化映射的纯软件驱动器，磁盘 I/O 层继承了很多的块 I/O 的实例。块 I/O 协议是由相关块设备发布的，如 DXE 中抽象 ICH 中的串行-连接（SATA）磁盘控制器的 ICH 驱动程序。磁盘驱动程序使用了 PCI 块 I/O 协议来访问 ICH 部件的控制和状态寄存器。

用于该提供文件系统堆这些功能的组件可以在图 7.16 中看到。文件系统组成部分包括文件分配表（FAT）的驱动程序，和一个提供 FAT12/16/32 支持的驱动器。

随着时间的推移，用于早期的个人电脑上的 MS-DOS 的原始文件系统 FAT 已经被扩展，它在 32 位系统中最终演变成 Windows95 环境的 FAT32。此外，提供不同的性能选择的存储信道可以通过 IDE 控制器初始化组件来抽象化。从而提供了一个 API，这样一个平台安装/配置程序或诊断就可以通过它来实现 ICH 的设置。

BDS / EFI Shell	Generic
FAT	Generic
Partition	Generic
Disk I/O	Generic
IDE Bus	PCAT
PCI Bus	Generic
PCI Root Bridge	North Bridge
PCI Host Bridge	North Bridge
IDE Controller Init	South Bridge
	IDE Channel Attributes

Figure 7.16 Components for IDE Init

相同原理的串口控制台层次架构和 SATA 控制器的文件系统层次架构只能存在于一个有 PCI 抽象化的，有合适的组件支持的 ICH 组件上。因此，有相同的 ICH，或者与该芯片相当的集成版本放到另一种特定应用集成电路（ASIC）上，是可以重用其他系统（如 IA32 的桌面或 IA32 的服务器）相同的二进制文件的。除了 IA32 的平台类上的二进制重用外，C 代码本身也可以重用。对于 ICH 的使用，无论是文字部分或上述集成在 Itanium 处理器逻辑电路，都可以通过在 Itanium 处理器重新编译 C 代码来作为目标的二进制代码。

第八章 卷, 文件和区块

6号：你找什么？
2号：我们想要信息。
6号：你和谁一伙？
2号：这个不能说。我们需要消息……消息……消息。
6号：你们找不到的。
2号：千方百计，我们会的。

—Patrick McGoohan, *The Prisoner*

文件是计算机里最基本的概念之一, 这对于大多数用户来说是存储在存储媒体(比如硬盘, 光盘)上的一些数据的集合, 也是大部分程序员理解计算机工作原理的一个基本概念。文件是很基本的概念, 大多数程序员真的不用对它有太多的考虑. 如果确实需要考虑, 他们考虑的是文件创建, 读, 写和删除. 他们知道你可以在文件中存储数据, 一般来说是数据的矢量, 并且这些数据是非易失性的.

用户和程序员也知道文件是存储在硬盘, 软盘, U 盘或者其它存储媒体上的。但是固件(这里的固件指 bios)程序员通常很少使用传统的存储设备来存储他们的文件和数据。因此, 固件使用传统概念的文件来存储是非常少见的情形. 不过, 在 framework 中, 把程序和数据存储在文件中是一种最基本的结构. 需要特别注意的是, 这里所说的文件及文件系统的使用是不同于在 EFI 架构中用来管理引导分区的文件及文件系统的。

Framework 基于操作系统的原则定义了它使用的文件的基本机制. 但目标环境让这些文件和传统文件的概念又有所不同。

术语:

看起来这里的各种各样的术语就和操作系统里的差不多. 下面的术语使用在 framework 中. 重要的是要注意, 当下面的术语与现在你所使用的操作系统中的术语有可能相同, 但它们所描述的功能大多是不相同的.

固件设备(FD):任何可以存储固件的物理设备或设备的集合。如果是几个设备的集合, 那么在这里只能做为单个设备来使用。

固件卷(FV):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. 这里明确的禁止一个 FV 来自于多个的 FD(虽然我们记得, 一个 FD 可以来自于多设备). 不同于大多数操作系统, 所有在固件设备中的固件卷 (FV) 都可以只占用此固件设备有效空间中的一部分. FVs 在 FD 中的位置可以在编译的时候定义然后由驱动来描述!. 换一种说法, 它不需要固件卷 (FV) 可被发现的, 这位置可能只是提前知道并由其它驱动填充到此驱动程序的堆栈中。比如:在 FV0 里的驱动可以描述另外的 FVs. 从定义, 就已经知道 FV0 是引导向量的目标。知道自己的家在哪始终是

很重要的。, 在固件中找出 FVs (通常是由驱动或者 PEIM 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。

固件文件(File):存储在固件卷里并命名了的信息的集合。这里的文件名在 framework 里是 GUIDs (GUID 是全球唯一标识符, 它能保证文件名的唯一性)。文件名 (GUIDs) 在一个固件卷 (FV) 里必须是唯一的, 但不同的 FVs 里可能包含相同文件名的文件。因此, 这就需要我们参考一个文件时要以 FV 和文件名配对。这项规定与大多数操作系统里的文件系统规则是很相似的。和操作系统一样, 以相同后缀名命名的文件会提供相同的功能 (例如. TXT, . H), 但是在很多情形下, framework 中要求更加严格—举例来说, DXE 调度器只会派遣特定类型的文件。文件在 FVs 里只有一级目录, 这里是不会有子目录存在的。但是, 文件可能包含 FVs 在其中, 那起到类似的作用。这递归机制也提供了一个更加简洁的方法来产生可能由大量文件组成的单元的封装模块。对于文件的通用操作有: 检测 (固件卷) FV 以确定文件的类型和打开文件, 读取文件, 关闭文件等简单的操作。

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

容器区域块支持像压缩和安全等特殊特性。典型的操作包含扫描一个文件的特殊类型的区域块和读取这些区域块。

下面的图描述了 FV 更精确的层次。同样我们也可以把它看作一个节点树, 如图 8.1 (A) 所示。更多的用户把它想像为组件在容器里面的层次结构似乎是更直观的, 我们在后面的章节中将主要以这种层次结构来描述, 如图 8.1 (B)。本章剩下的内容就是讨论它们相互之间的作用和怎样去使用, 访问, 并实现。

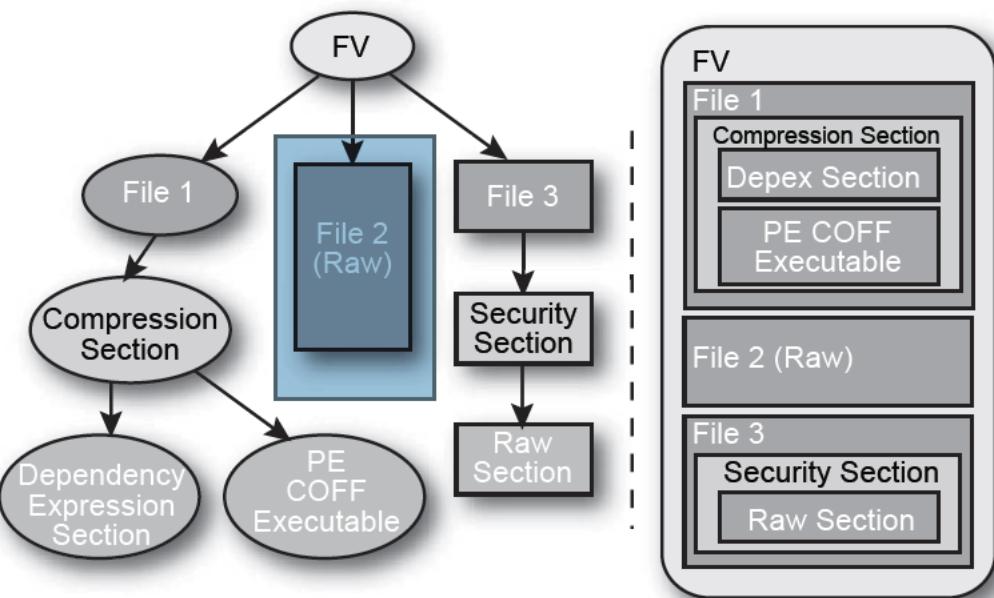


Figure 8.1 Hierarchical Views of FVs

FVs 的层次结构

● 设计约束:

固件的存储和访问机制强烈受固件的使用和固件有可能被储存的一些媒体的影响。 启动向量

启动向量指向固件设备的一个固定地址。文件系统必须能保证有一个文件或者其它结构数据在一固定地址, 以确保系统顺利的完成启动. 更复杂的处理器不会使用传统概念的启动向量, 但是仍然会确保至少有一个或者有可能几个固定区域在启动固件设备(像 FD0)中.

有趣的是, 在设计和后来的经验中调查发现, 启动向量是唯一需要知道的地址. 当初设计的时候的工作目标允许任意数量的固定地址文件被证明是不必要和增加了复杂度. 在特殊的情况下需要单独处理.

对齐

虽然很少要求文件在卷中的确切位置, 但是对于数据在文件中的特定边界地址方面确是有非常多的要求. 比如, 有些计算机架构, 必须从一个16字节边界地址开始运行, 这样也能使操作更加有效。.

简单的访问

文件必须在 PEI 周期的时候找到并读取, 在这个周期只有有限的 RAM 和其它可用的资源可以使用。这意思就是说内部文件的存储格式必须简单, 以使 PEI 能不用通过复杂的运算或构造复杂的数据结构而扫描到文件。

罕见的写入

写入固件设备不太像他们在操作系统传统写入, 往往是更加孤立的。在 EFI 中, 大多数在开机和运行时所需要写入数据主要是变量。事实上, 像后面描述的那样, 这就有充分的理由考虑存储变量而不使用传统概念的文件系统。

令人头痛的闪存

这些存储了固件的设备有他们特有的复杂性设置. 起初, 启动固件是存储在只读存储芯片 (ROM) 里的, 这种存储器是不可写的, 并且通常是没有分区的概念。

今天, 大多的固件都已经存储在闪存设备里。这种设备是可以随机读取的, 但对写入操作可能只能以一个区块为单位来写入。这样的限制似乎没有对硬盘驱动器的规定更麻烦, 硬盘只能以扇区为最小单位写入。

但是不幸的是，比起硬盘中扇区的大小来说，块的大小相对于此类设备来说都相当的大。比如：一些只有 512KB 的存储部分只分为 8 块。(按照这样的逻辑，我们想像一下管理一个 1.44MB 的软盘只有 24 个扇区而不是 2880)。典型的是 1MB 被分为 4096 块，相对于闪存的大小，使用这么大的尺寸的块会导致不能有效利用闪存的问题。特别是，传统的操作系统使用整数个扇区来存储文件的实践方法在这里已经变得不切实际。首先，文件的数量受到块的数量的限制，更重要的是对块的不完整使用而形成的内部碎片导致闪存利用效率低下。如图 8.2 所示：

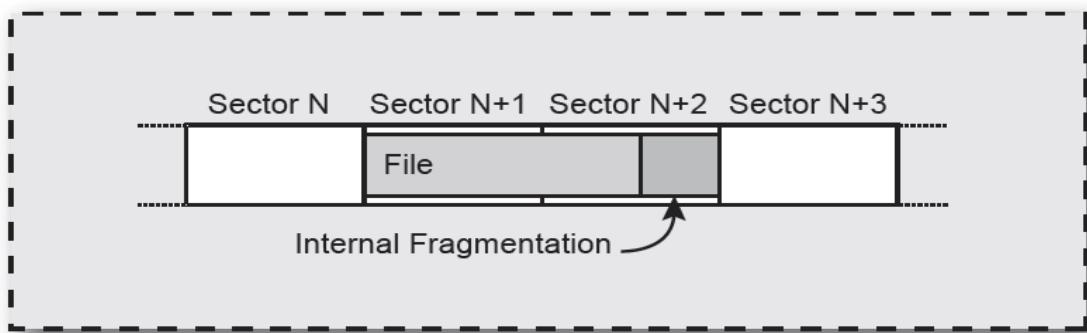


Figure 8.2 Internal Fragmentation

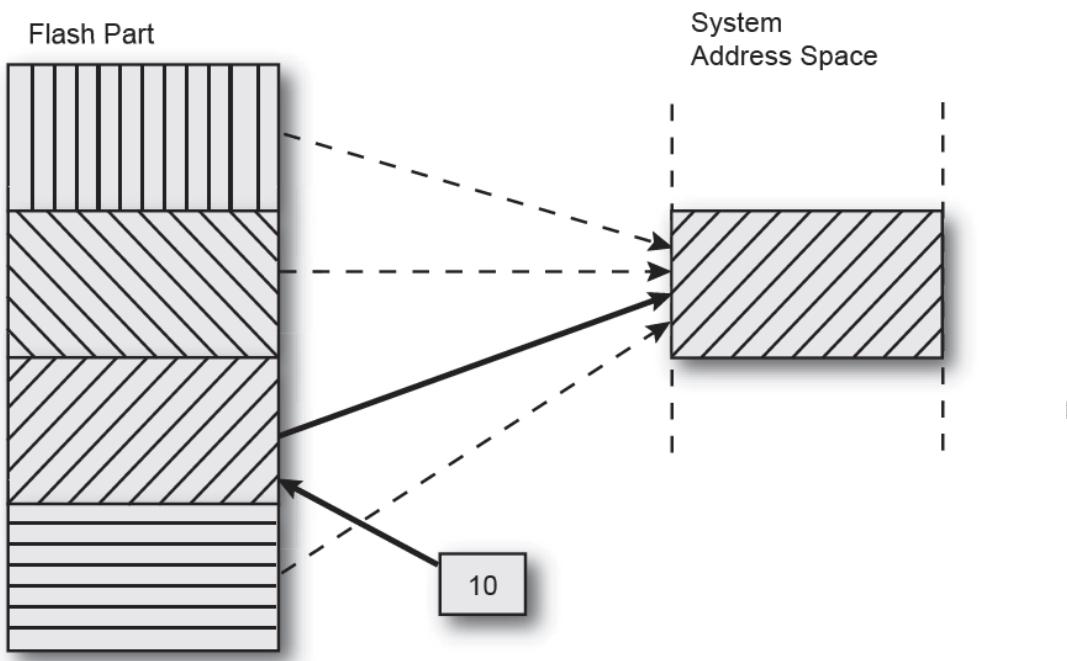
内部碎片

由于闪存每位的高成本，因此使用闪存的效率问题的压力可能要高于预期。容量比较小的闪存芯片是目前可用的存储器中成本比较高的一种。因此我们应该鼓励使用新的技术来更有效的使用闪存。对于操作系统的文件系统和 framework 的文件系统，也许最大的不同是 framework 的文件系统在文件和块之间缺少任何的对齐。

在 EFI 众多的执行环境中，驱动和程序文件更可能多的是从非传统的存储设备读取出来的，比如来自于网络环境或硬盘上的隐藏分区。这些文件不是由我们这里所说的固件文件协议来管理的，这也不是我们此章所要讨论的内容。

Bank Switching

闪存不总是以线性的形式出现在内存中，它通常是在服务器和其它系统中使用大量的闪存芯片中的比较少的一部分来与系统内存之间进行块交换，如图 8.3 所示。这里使用一个 I/O 端口(解码过的)来选择闪存中的一页映射到系统内存空间的一个窗口。这要想在任何地方执行是不太可能，所以 PEI 抽象层不使用这种机制。(It is not easily possible to execute out of any but the default bank in bank switched parts so PEI abstractions do not support this.) 在 DXE 里，设备是在背后被抽象的映射到一个点。

**Figure 8.3** Bank Switching

● 容错能力

如果已经确定固件的一部分存在错误，系统将无法启动，直到这存在错误的闪存芯片取下来并且换上已经写入程序数据的新的闪存芯片。这种机能是电脑中任何组件都无法比拟的。

错误是经常发生的事，但是，在更新数据的时候。当 FV0 需要更新，这闪存芯片（或者其它可存储媒体）必须先擦除原先写入的数据然后再写入新的数据。如果在擦除的时候突然掉电，系统将再也无法启动。通常情况下丢失 FV0 中的任何数据都将导致系统无法开机。因此只具备文件级的容错能力是不够的-卷级的容错能力将是必需的。

当我们有这卷级容错的能力，我们至少能让系统由外部的资源从致命错误中恢复，通常是在更新闪存的时候丢失电源。

这是容错更新的算法顺序。FVs 有两个相同内容的副本，并且再这两个副本之间相互候补。它的原理就是先更新没有使用的那一半闪存，然后再更新另外一半。如图 8.4

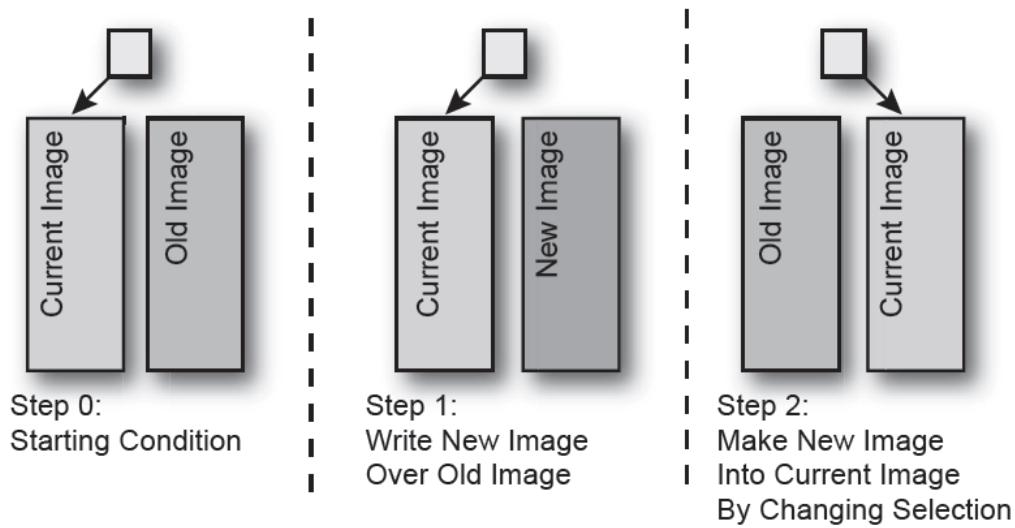


Figure 8.4 Dual Image Update

相对简单的解决方案是确定所使用的芯片组，比如intel 的ICH7，它支持一个特性就是TOP_SWAP。经选择，地址的第16位地址线在系统内存的高128KB是反向的。Upon selection, bit 16 of the address line to the top 128KB of memory is inverted. This has the effect of making the top 64KB the next lower 64KB and decoding the next to top 64KB as the top 64KB, as shown in Figure 8.5.

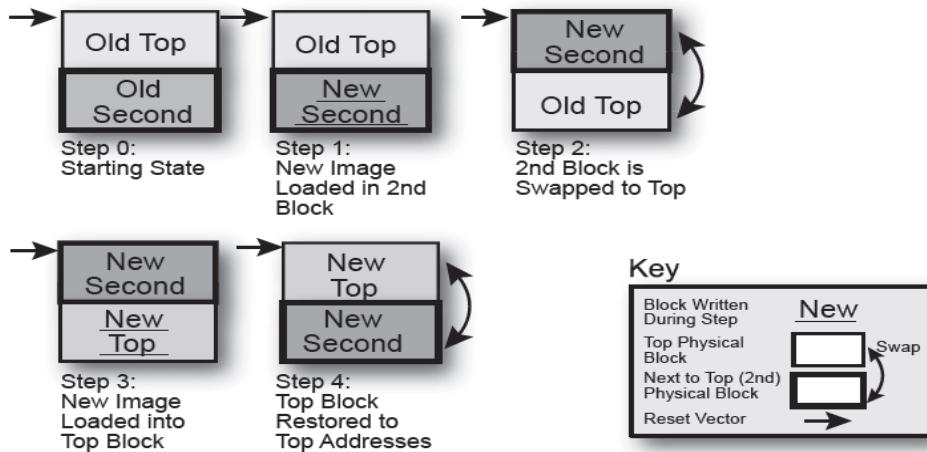


Figure 8.5 Top Swap Update

To File or Not to File

其中最不典型的改变是传统的操作系统的文件管理概念是管理所有媒体的空间而无需在卷

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后24小时内删除，否则后果自负。

里面管理！这同样是有用的，例如，同样的FD文件系统支持EFI变量但是存储在固件卷中的变量不一定支持。

这并不是FV文件系统的弱点。这是故意这样设计的。我们无法用一个机制来满足所有的需求，它是为大多数可能的需求而设计的。变量的属性通常与FD里的大多数数据是不同的：这里的数据是非易失性的，数据写操作比较频繁，而且每次写入的数据的数量都比较小（通常只有数百的字节）。

使用这种文件系统来支持变量并非不无可能，几次成功的实践都证明这是可行的。不过这并不是必需的。这是让系统设计员来选择正确的途径以满足系统的需求。

定义文件系统使用这一区域是为了根固件卷，也就是FV0和让其它的固件卷在PEI周期的时候可以访问到。这是因为PEI调度器是直接访问文件的，因此需要一个统一的存储布局。

● 文件类型

在Framework中使用文件类型与在传统的操作系统中使用文件类型都是出于同样的原因：都是为了区别各种各样的杂乱无序的文件。

少数文件是Framework所特有保留的类型。比如PEI ‘内核’，DXE加载器，和DXE ‘内核’。这里是用文件类型来指出文件的可执行性，PEIMs，DXE驱动，和EFI应用程序。

如上文中所述，在Framework中卷是不允许子目录存在的。相反的是，这允许文件包含卷，这使得一些分层次的文件系统不明白这种特性。请参阅本章“Capsules”小节的例子。

文件不确定定义为任何结构，所以可能有两种类型：freeform和raw。Freeform文件是以section为基本单位存储的，而raw却是连续的二进制文件的数据。使用这一种或两种格式是可让用户定义的。

这固件卷协议GetNextFile允许通过扫描卷来确定某些特定类型文件（或者任意类型）。

● Section 类型

大多数framework里的文件都由sections组成。大量section类型定义的不同的元素可能已经存储在了文件里。类似于GetNextFile，固件卷协议也提供了GetSections来扫描section。

Section分为两类，封装section和叶section。封装section包含其它section，而叶section没有。

在设计的时候，section的设计是非常简单的。只需要它的类型和它在存储器中的长度，有的section还有扩展头标。（特别是使用GUIDs）。

普通的封装块的(section)类型

压缩的卷(Volumes)和文件都不能(自, 建议去掉)直接存放(压缩)在 FFS 中, 但可以存储在压缩块里面。已经压缩的 volumes 在一个压缩块(的 section)中。压缩块(的 section 中)的头部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。把用于 DXE 的卷放到一起的一种通用做法就是用压缩块将它封装起来(一种通用(普通)的方法就是把用于 DXE 的卷与压缩的 section 放到一起), 然后储存在一个卷(Volume)文件里面。另外一种块(section)是叫做 GUIDed 块(section), 它通常会通过某种协议与另外的 section 联合。这个 section 的头部包含一个请求明确的 GUID。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, 必须要在使用这个 section 之前就定义好。当这个相应的 GUID 的 section 被使用时, 这个接口的协议就会被调用。举个例子(在实模式下执行), GUIDed 的类型可能会通过内部数字签名的方式去确认 sections。DXE dispatcher 将会决定在 dispatch 队列中的和这个 GUID 相应的协议去执行。

压缩的卷和文件都不能直接存放在 FFS 中, 但可以存储在压缩块里面。已经压缩的 volumes 在一个压缩块中。压缩块的头部描述了怎样去寻找已有的解压数据接口的协议。把用于 DXE 的卷放到一起的一种通用做法就是用压缩块将它封装起来, 然后储存在一个卷文件里面。

另外一种块叫做 GUIDed 块, 它通常会通过某种协议与另外的 section 联合。这个 section 的头部包含一个由应用确定的 GUID。在遇到这个块之前, 必须要有以这个 GUID 命名的协议(例如 GUIDed 块抽取协议)存在, 这样的协议具有确定的接口。当遇到相应 GUID 的块的时候, 这个协议就会被调用。举个例子(实际上已经实现了), 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。

普通的叶子块类型

最后统计, 已经定义了超过 10 种 leaf section 类型。它们可分为几类:

- 一种叶的类型是为了每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)。还有一个专用的 section 类型是为兼容模块 (CSM) 的可执行程序保留的。
- 可执行文件类型也可能有 DXE 依赖性表达以及 PEI 依赖性表达的块。
- 虽然不是必需的, 为了更加用户友好方式的管理的文件, 定义了一种版本和文件名块。
- 固件卷映像是一种能够把卷存储在文件中的块类型
- 自由形态的子类型 GUIDed 和裸的块和文件中的意义相同。

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。

分派器如何试用卷、文件和块？

DXE 分派器的主要工作是为了寻找和分派驱动。分派器使用固件文件基础架构来完成这些动作。通过这样做，它提供了关于如何使用 FVs、文件和块的一种更复杂的例子。系统知道的每一个 FV 都是通过一个协议来表达的。基于这个事实，分派器借由等待一种新版本的协议（通过 EFI 的 RequestProtocolNotify 服务）来寻找新卷。PEI 阶段中发现的卷，通过 HOB 传递给分派器的，分派器会在初始化自身的时候进行处理。当分派器得到一个新卷出现的通知时，它使用 Firmware volume 的协议来扫描 DXE 驱动类型的文件。当它找到了这些文件，它就会搜索依赖性关系块。他使用这些块的信息为这个驱动设定一个分派顺序。当分派器决定分派一个驱动执行，它从依赖表达式相关的那个文件中读取 PE/COFF 映像块。然后分派这个驱动执行。

内部的格式

虽然 Framework 的目的是支持将任意数量的文件系统，它在开机的时候必须依赖于一个的文件系统，才可能找到告诉它如何访问其他文件系统的模块。这就是所谓的 Framework 文件系统格式，也就是 FFS.

基本格式

固件卷被定义为包含一些文件的一些线性数据块。文件有一个头，数据体，和一个“尾巴”。固件卷以头部在最低地址开始，这个头部包含了一个 GUID 、卷大小以及其他相关数据的属性。由于卷的头部开始于最低的地址，所以能够简单和明确地找到它的位置。

文件在卷中从卷头部结束的位置开始连续地存储。文件头包含名字（GUID）、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。

文件系统得以通过完整性校验来确认文件是否出错。属性则包含跟这个文件相关的各种标志。最重要的标志是文件的数据部分的对齐（不是文件头）。这实际上显示了除了头以外需要跳过的字节数，这使得数据区以边界开始开始。某些直接映射到硬件的文件需要这样的方式。11 中对齐方式涵盖了从 1 字节和 64K 字节的大部分 2 的幂次。

状态信息是将被用作更新操作的一部分。更新操作本质上来说就是删除和写操作。状态信息的目的是保证这个过程的原子性。也就是说，在更新中的任何状态下，老的文件和新的文件有且只有一个有效的。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。

FV 的管理

文件在卷中从底部（最低地址处）开始，然后一个与一个连接（允许对齐而产生空隙），直到顶部的文件。新的文件会写到之前刚写入的文件位置之上。删除文件，不会立刻引起上面的文件位置发生从上到下的变动。文件会改变头部的删除标志位——从擦除状态变为非擦除状态（译注：写入文件时不会写擦除标志位，即该状态位在的初始状态是擦除状态，通常 Flash 的某一位擦除时该位回到 1，只有写入 0 时才变为 0）。

当空间不足而不能响应写的请求，合并的操作便会产生。这有点类似于早期的文件系统(1960s vintage)，有着简单，可预期的大小限制，低区块/卷率的特点。没有一种单一的专门方式去合并 FVs。在选择一种机制的时候，设计者必须权衡可用资源、容错能力要求和其它因素。一个相当简单的模式就是拷贝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。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，以“兵-兵”的方式来使用这两个备份，这是一种容错的方式（假设在此种情况下驱动知道哪个卷是新的），但是成本太高了。开发者们可以想出很多种其它的方案来。FFS 状态的数据可以用来实现安全的更新或者刷写的机制。

系统文件之下

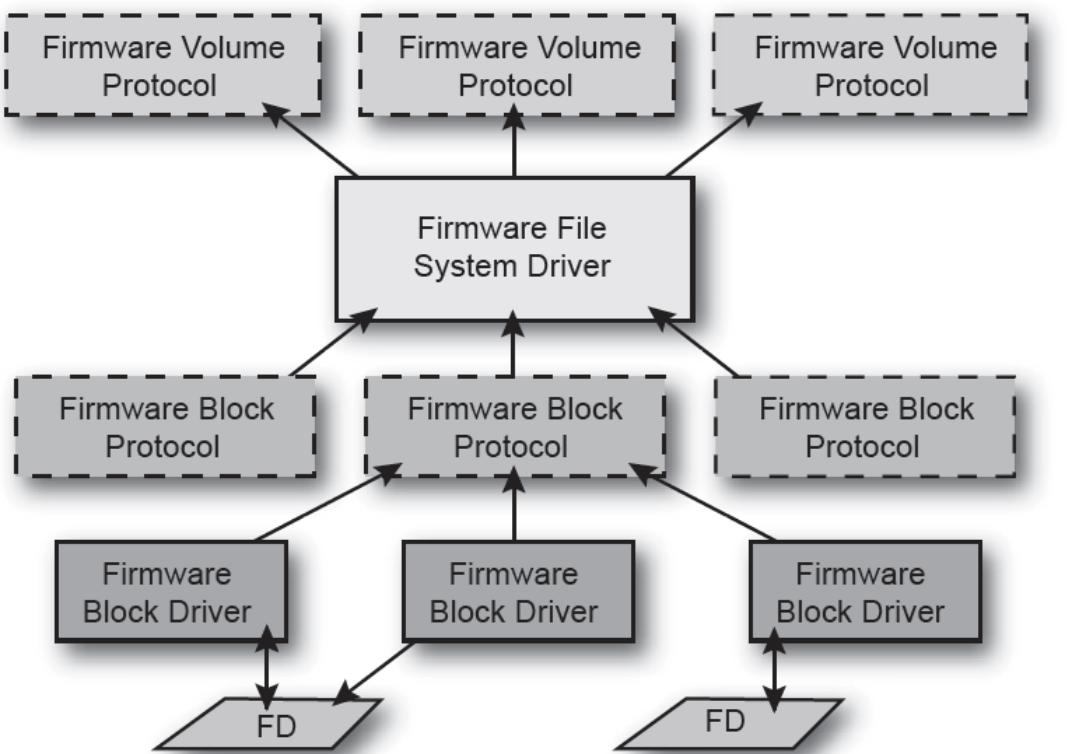
支持操作系统的文件系统的驱动程序能够将按照块设备映射成为的文件的方式管理。这个框架(Framework)也是一样的。

驱动的堆栈

一个固件卷是作为固件卷协议 (Fireware volume protocol) 来显示的。任何文件系统都可以显示这个协议，从而创建一个新的文件卷。

Framework FFS 定义了一套接口，它允许建立一个驱动堆栈，如图 8.6。使得显示文件卷堆栈的驱动程序不需要每一个都重新实现。

这个架构没有从定义机制去发现一个新的 FD。相反，硬件相关的 DXE 驱动程序编写的时候需要在编写的时候就知道固件设备，或者有某种机制可以发现它。一旦设备被检测出，驱动就必须提供一个 Firmware Block Protocol 的接口。FFS 驱动会监视这个协议的建立。该协议的创建会触发 FFS 驱动建立一个固件卷协议 (Firemware volume Protocol)。这个协议优惠触发分派器 (Dispatchers) (PEI 和 DXE) 寻找相应的模块去执行。在 PEI 和 DXE 中发现新设备都可以通知框架(Framework)。PEI 的方式就简单一些。设备是内存映射的，并且是只读的。对这样的设备，接口只需要提供这个设备的基址地址。

**Figure 8.6** FFS Driver Stack

固件卷的块接口

在本章中 Framework 接口很类似于操作系统的接口，特别是 FVB 接口就更接近于操作系统上的分区，只不过这里是块的概念。

FVB 接口定义了 7 中接口：

- 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，GetPhysicalAddress 允许驱动去获取卷的物理地址。
- 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。
- 擦除：合并操作的一个部分就是使用 EraseBlocks 擦除区块。

变量

EFI 的规范定义了对变量的支持，这些变量的存取和功能很类似于 UNIX 和现在的大多数操作系统的环境变量。环境变量是 SHELL 级别的数据数据，程序主要用它来设置系统和应用程序的默认值。

EFI 的变量以 GUID 和 UNICODE 的方式命名，它通常用来存储任意数量数据。EFI 规范并没有定义变量容量的大小。变量（注意这个跟 C 这样的程序设计语言的变量不同）与固件卷（FV）的不同之处在于固件卷（FV）很少写，而变量却要经常被反复写入。（在某种意义上讲，在用于启动计数的计数器就通常被定义为一个变量）。

如上所述，变量的实现可以采取两种基本形式。存储在 FV 和不储存在 FV 里。即使变量存储在 FVs 的内部，在系统管理这个 FV 的方式可能会与其他的 FVs 完全不同。它可能不含有任何文件，而且向 Framework 声称它已经满了，使用 FV 的机制仅仅是为了定位和管理。

存储变量的 FV 能够以区别与其它块(Block)不同的方式进行备份，因为 EFI 规范中要求的变量具有容错能力，而对于应用程序它应该是透明的。变量的实现方式必须能够支持几乎无限数量的变量。比较简单的方式是使用一个块用于变量的存储而另外一个 block 用于备份，这分为两种情况。第一种情况是，2 个块被交替的标记为“活动”和“备份”block。另外一种情况是，在更新数据的时候，只有一个块被定义为“活动”的，另外一个被定义为“备份”的。如果发现一个活动的块正处于升级的过程中（译注：说明该块升级失败了），那么就会复制备份块中的内容到活动 block 中。

Capsule

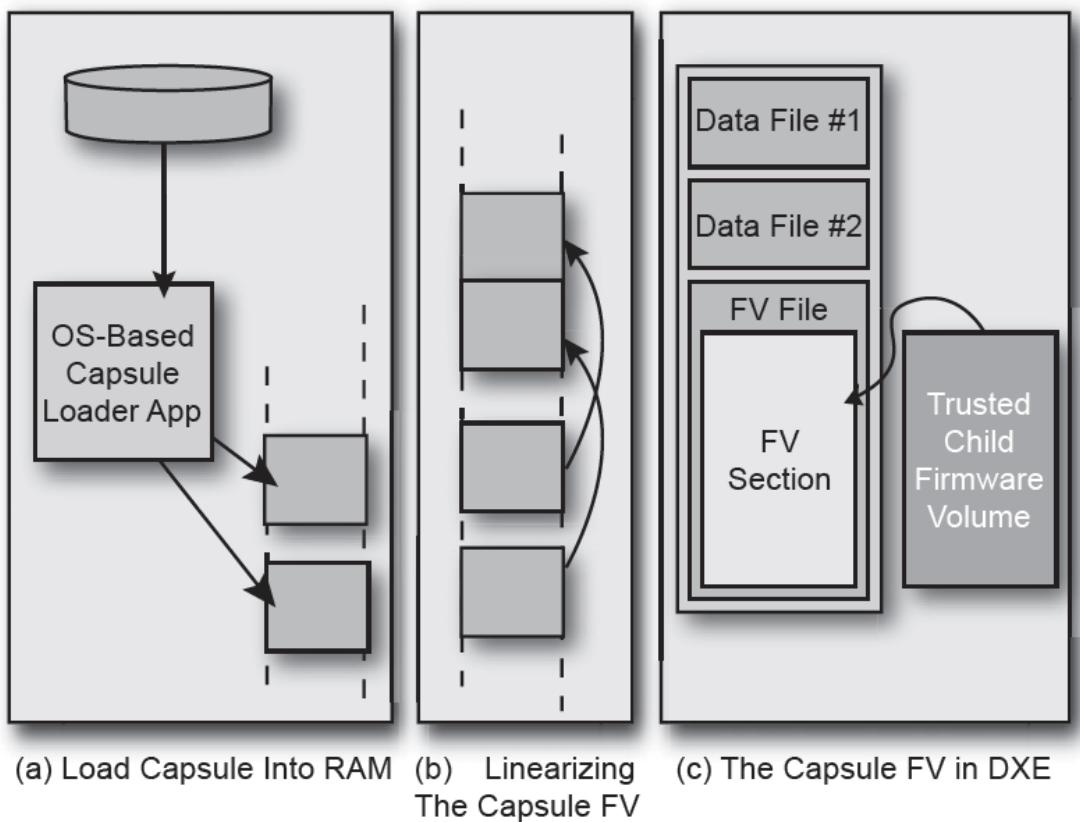
Capsule 是一种机制，允许操作系统为下次启动提供数据。开发 capsules 的动机是为了创建一种机制用来刷新固件，但是架构本身还不能满足这个需求。事实上，UEFI 2.0 的规范中设想使用 Capsules 可以把从操作系统传递的数据在重启期间转变为稳定的版本，用以系统崩溃的报告和分析。而操作系统通常认为数据在重启阶段是不稳定的。

Capsules 的结构

Capsules 由一个或多个头和固件卷组成，只有固件卷会参与重新引导的过程。头（标记位）是用于创建通用的 Capsules 处理程序。因此，它包含所在 capsule（能够被任意数量的语言定位）的入口（例如关于内容的短或长的 UNICODE 描述）和机制允许 capsule 被分割成多个块后能够重组。

重要的一点，在使用外接设备时，Capsules 包括一个对设备描述（设备的描述不包括路径信息）的描述。这是很有必要的，可以让开发人员不依靠知道在载有该驱动程序 option ROM 卡插槽的位置进行更新。设备描述符标识了设备，更新程序可以关联到一个或者多个系统中正要更新的设备。

升级程序通过在 capsule firmware volume (CFV) 以已知格式存储的文件来负责设备的更新。

**Figure 8.7** Capsule Loading

通过重启传递

一旦剥掉他的头部，CFV 将被存储在一个不可交换的内存区域（为了保证在重启的时候存在在 RAM 里），如图 8.7 (b)。

不可交换的内存区域在大多数操作系统中是用于硬件的 DMA 内存，比如网卡，USB 控制器，以及硬盘适配器。如果系统足够可靠，在重启的时候可以不擦除这个内存的区域的。由于可能没有足够的不可以交换的连续内存空间用来分配，capsules 定义了一种机制用来存储那些没用空间分配不连续的块的列表。也有同样的机制来一次传递一个或者更多的 Capsule 到预启动的空间。

Framework 定义了一个 EFI reset 类型用于传递 capsule 数据结构的基址物理地址，UEFI 2.0 定义了一系列的 runtime services 来管理 capsules，这些方法同时工作又不互相干扰。

当 reset 请求到来时，整个系统的控制从操作系统回传给 EFI firmware。所以，操作系统要运行的操作等同与 S0 到 S5 ACPI 转换所做的操作（软启动不保留内存数据）。现在 firmware 拥有了整个系统。

它要通过一个reset 保持 capsule 内存。但是，在一个非常高端高效的系统中，不需要 full reset。实现一般需要操作系统的合作，不属于本书讨论的范围。

用于产生 reset 的方法不是 结构性的，在支持 ACPI S3 状态的系统中（“保存在内存中”），这是个显而易见的方法，特别的如果有一个预设唤醒事件的机制，那么 reset 实质上是瞬间完成的，它当然不是唯一的一个。

用于传递 capsule 数据结构地址的方法也不是 architectural 的。因为 Framework 一般具有对实时时钟 CMOS 的需要，这里的例子是假设地址存储在那。例如，一个小于 0xFF 的值的意思是当前没有 capsule，并且它替代了包含状态数据的值。任何其他的值被认为是一个地址。另外，还有许多其他的机制。

另外的一种方式的重启

假设 S3 机制被用来 reset 和 cmos 使用来储存 capsule 地址，那么 reboot 将继续执行直到内存中的数据被恢复，就好像正常的 S3 一样。如果地址值小于 0xFF，reset 被当作是正常的 S3。

假设找到一个地址，结构地址是有效的。如果是无效的，则报告一个状态并且与 S5 等效 reset 开始了。

实现的最复杂部分是 PEI 过程。它把分散的多个 capsule 重新组成整个的一块。首先要寻找一块足够大而没有被占有的内存，capsule 被复制到这些自由内存中。然后内存保留这些 capsule 并且通过 HOB 报告给 DXE。

Capsule firmware volume 仅存储在内存中，访问它需要一个 FV Block 驱动，这个驱动可以给 FFS 提供接口来创建 FV protocol。它同样需要。

The Capsule in DXE

乍一看，capsules 像是一种操作系统向 pre-boot space 注射病毒的方法。如果不小心，事实就真的会这样，firmware volume 层级结构和 Framework 的安全层级的一些特性可用于解决这个问题，如图 8.7 (c) 所示。

负责管理 capsule 的 DXE Driver 把 volume 标记为不可靠的，这就意味着 DXE dispatcher 将不会从它执行。在一个优秀的设计中，capsule 中的一个或多个文件都是一个 firmware volume 文件。这个 firmware volume 文件被一个 GUIDed section 包装起来。Driver 公布相应的 GUIDed section 的提取协议，然后验证 volume 的内容。如果有效，它把更小的 volume 的属性设置为可靠的，这样就可以继续执行下去。

当在执行的过程中，他们可以找到本身解压的那个父卷。父卷是 CFV 本身，并且包括一些更新程序写入的数据文件。

在只有数据的 capsules 中是不需要验证的，例如，当操作系统要进行下一次 reboot 时。因为只有数据的 capsules 没有执行代码。他们只能草草做成 chain。

有趣的是，在 framework 启用 capsule 只需要相当少的代码。这个实现包括作需要包括一小段的重启程序及在 S3 状态下的特殊处理、Capsule 连接代码，为 Capsule 构造 FVB 协议的驱动。确认的代码完成这个请求。所有的这些工作都由 firmware volume 的服务来完成，就像对待内部的 volume 一样。

经验表明为 32 位的 intel Pentium 4 所编译的是大约 4K 大小（未压缩过）。这是因为实际上更新的代码可以进入到 Capsule 的内部而不是保存在 FLASH 里面。

www.BIOSREN.COM

第九章

DXE 基础：基础框架、调度器和设备驱动

我不害怕计算机，我害怕的是如果没有了计算机。

——艾萨克·阿西莫夫

本章描述了设备驱动执行环境(DXE, Driver Execution Environment)的结构，以及在平台初始化过程中它是如何运行的。此外，本章还详细描述了底层组件的工作机制，以及数据在系统启动过程中的不同阶段是被如何传递的。底层组件的工作机制的描述还详细提供了启动指令的内部结构，因为它与传统BIOS中的POST table有很大的不同。

DXE阶段是依据EFI 1.1 标准中的定义实现的。因此，DXE的基础架构和DXE的设备驱动在EFI映像属性方面有很多共同之处。大多数的系统初始化都在DXE阶段执行的。PEI阶段则是负责初始化物理内存以供DXE阶段代码的加载和执行。在PEI阶段结束时，系统状态信息是通过HOBs被传递到DXE阶段的，HOBs(Hand-Off Blocks)是一个存储系统信息的数据结构，它在内存中的位置不是固定的。DXE阶段包括一下几个部分：

- DXE基础框架
- DXE调度器
- DXE设备驱动

DXE基础框架创建了一系列的启动服务、运行时服务和DXE服务。DXE调度器负责发现并按照正确的顺序执行DXE的设备驱动。DXE设备驱动包括初始化处理器、芯片组、以及平台组件，甚至包括供控制台和启动服务使用的抽象出来的软件。所有这些部分共同协作完成平台的初始化并提供OS启动所需的各种服务。DXE和BDS(Boot Device Selection)阶段一起建立一个可供操作系统启动的平台。DXE阶段一直持续到OS成功开始启动——也就是说，直到BDS阶段开始。只有在DXE基础框架或DXE驱动部分中创建的运行时服务才不会被停止，它们最后被保存在OS运行时环境中。

图9.1显示了一个Framework的固件平台在冷启动过程所经历的各个阶段。本章包括以下内容：

- PEI阶段到DXE阶段的过渡
- DXE阶段
- DXE阶段和BDS阶段的合作

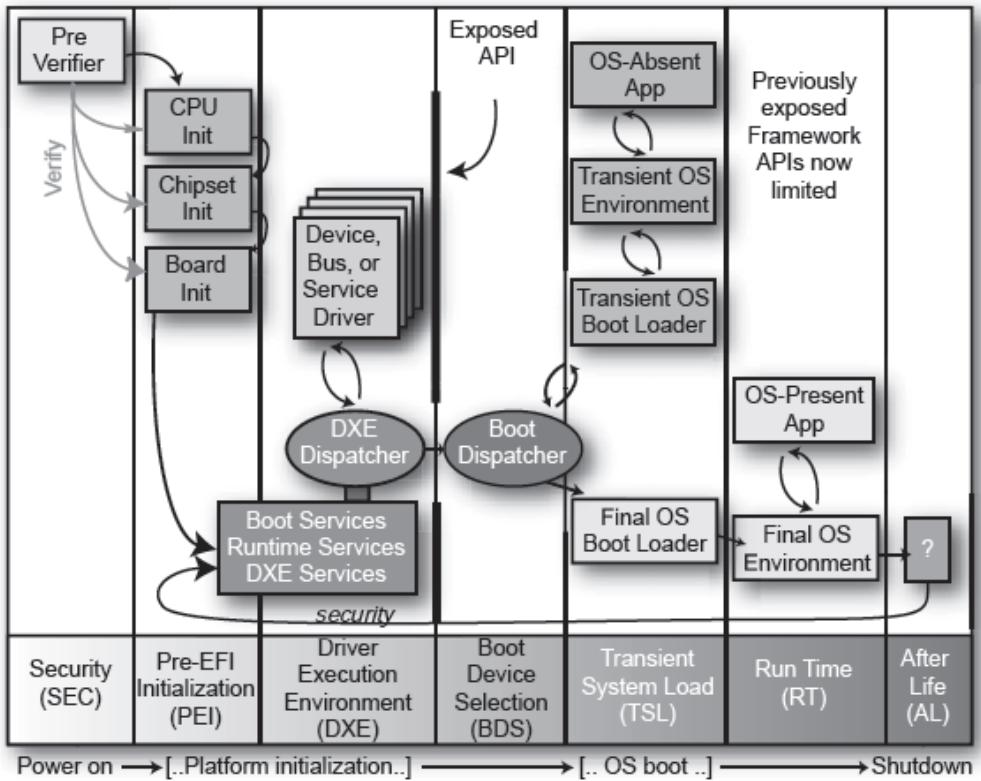


图 9.1 Framework Firmware Phases

DXE 基础框架

DXE 基础框架具有可移植的特点，它不依赖于特定的处理器、芯片组或者单个平台。它具有以下特征：

■DXE 的基础框架的初始化仅仅依赖于 HOB 链表，这种简单的依赖关系意味着它不会调用任何以前的阶段中创建的服务，因此一旦 HOB 链表被传递给 DXE，之前的所有阶段中的数据和服务都会被释放。

■DXE 阶段不使用硬编码的地址空间。因此 DXE 的基础框架可以在物理内存的任何位置被加载，而且无论在内存的任何地方，无论固件卷（firmware volume）位于处理器的任何地址空间，DXE 的代码都能被正确执行。

■DXE 的基础框架不包括任何特定处理器、特定芯片组、特定平台的信息。因为，DXE 的基础框架是从系统硬件抽象出来，通过一系列 protocol 接口架构实现的。这些 protocol 接口由一系列 DXE 设备驱动创建出来的，这些驱动接口会被 DXE 调度器调用。

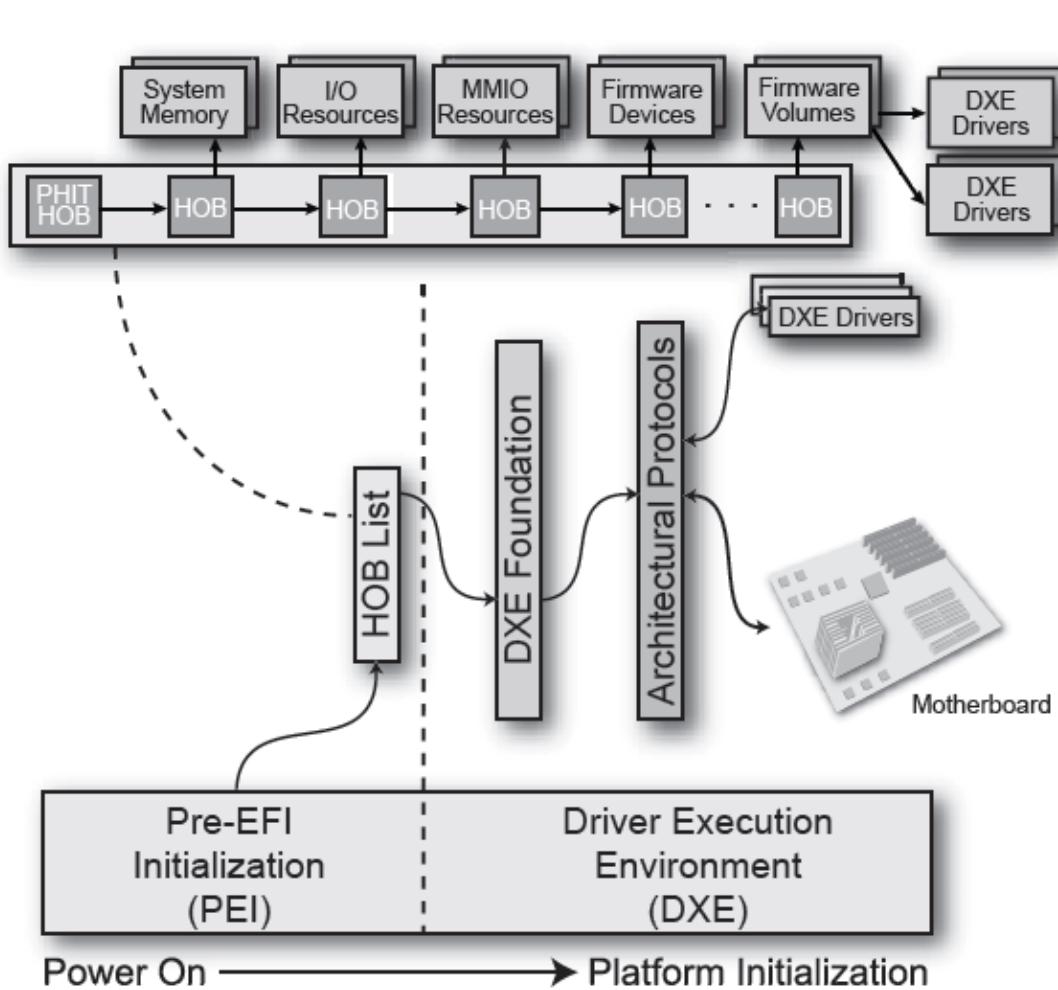


图 9.2 Early Initialization Illustrating a Handoff between PEI and DXE

DXE的基础框架创建EFI系统表以及一系列的EFI启动服务和EFI运行时服务。DXE基础框架也包括了DXE调度器，调度器的主要作用是发现并执行存储在固件卷(firmware volume)中的DXE设备驱动。DXE设备驱动的执行顺序取决于一组可配置的依赖关系描述文件（参考DXE调度器一节）以及驱动中的依赖检查。固件卷文件格式允许依赖表达式随同可执行的DXE驱动映像一起被打包。DXE驱动沿用PE/COFF映像格式，因此，DXE调度器也必须包括PE/COFF格式解释器用来加载并执行DXE驱动。

DXE框架结构也必须管理一个句柄数据库。句柄数据库是一个或多个句柄的列表，同时，句柄是一个或多个protocol GUIDs，这个protocol GUIDs都是独一无二的。一个句柄是一系列服务的软件抽象。一些句柄来自I/O设备，其他的抽象自一系列的通用的系统服务。一个典型的句柄包括一系列APIs和一些数据段。每个protocol都用一个GUID来标识。DXE框架结构提供了一些服务用来向句柄数据库中注册protocol。因为DXE调度器会执行DXE设备驱动，一些包括平台特性信息的protocol可以被添加进句柄数据库，并最终同DXE Architectural Protocols一起被DXE调度器调用。

Hand-Off Block(HOB) 链表

HOB 链表中包括了各种信息，这些信息被 DXE 框架结构用于创建基于存储器的服务。HOB 链表中包括启动模式信息，处理器指令集，以及在 PEI 阶段发现的系统内存。同时它也包括了在 PEI 阶段初始化的系统信息描述，以及 PEI 阶段发现的固件设备信息。固件设备信息包括了系统内存在固件设备和固件卷中的位置。固件卷包括 DXE 设备驱动和 DXE 调度器，调度器负责加载并执行 DXE 设备驱动，这些设备驱动都被存储在固件卷中。最后，HOB 链表还包括了 PEI 阶段找到的 IO 资源和内存映射 IO 资源。

图9.3是一个HOB链表的例子。HOB链表的头节点总是PHIT HOB(Phase Handoff Information Table)，PHIT HOB存储着启动模式信息。剩下的HOB节点的顺序不是固定的。从示例可以看出，各种不同类别的系统资源都在同一个HOB链表中描述出来。对于DXE框架结构来说，HOB链表中最重要的部分是描述了系统内存和固件卷的HOB节点。一个HOB链表总是以一个特定的HOB节点结束(end-of-list HOB)。还有一种HOB节点类型没有在示例中给出，带有GUID的扩展节点，这类节点用于从PEIM向DXE阶段传送私有数据。特定的DXE设备驱动可以通过GUID的扩展HOB来辨识并解析出节点中存储的数据。整个HOB链表可以被存储在内存中的任意地址。这种不指定地址的特性允许DXE框架结果将HOB链表转储在任意指定的位置。

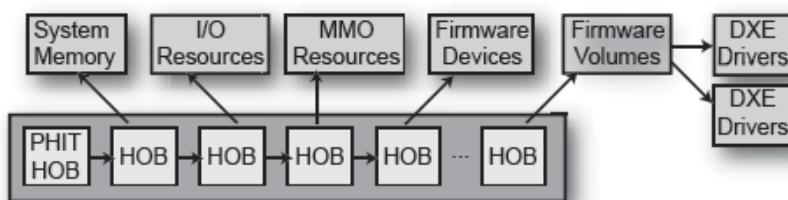


图 9 3 HOB List

DXE 架构 Protocols

DXE的框架结构是从平台硬件抽象出来，并由一系列DXE架构protocol组成的。DXE框架结构将这些protocol分成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protocol。从这个设计中我们可以看出，DXE框架结构在一开始必须包含足够的服务用于加载和启动DXE设备。

DXE框架结构的HOB链表必须包含系统内存的描述信息和至少一个固件卷。HOB链表中的系统内存描述信息被用作初始化EFI服务。整个系统必须运行在单处理器下，并且没有中断服务的平坦物理内存模式(flat physical mode)。通过HOB被传递给DXE调度器的固件卷必须包括一个只读的FFS驱动，这个只读FFS驱动用来解析并搜索指定固件卷中的依赖描述文件和DXE设备驱动。当一个驱动被找到，并且依赖关系被满足时，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会被早期的DXE驱动创建出来，以便在稍晚的DXE基础结构能创建出全部的EFI启动服务和EFI运行时服务。图9.4显示了一个被传递给DXE阶段的HOB链表。DXE基础架构使用DXE架构protocol创建了EFI系统表，EFI启动服务表和EFI运行时服务表。

图9.4包括了DXE阶段所有的主要组件。左边的EFI启动服务表和DXE服务表被分配在EFI启动
声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后
24 小时内删除，否则后果自负。

服务内存中。这种分配方式表明，EFI启动服务表和DXE服务表在操作系统启动后会被释放掉。右边的EFI系统表和EFI运行时服务表是分配在EFI运行时服务内存中的，它们会在操作系统运行阶段驻留在内存中。

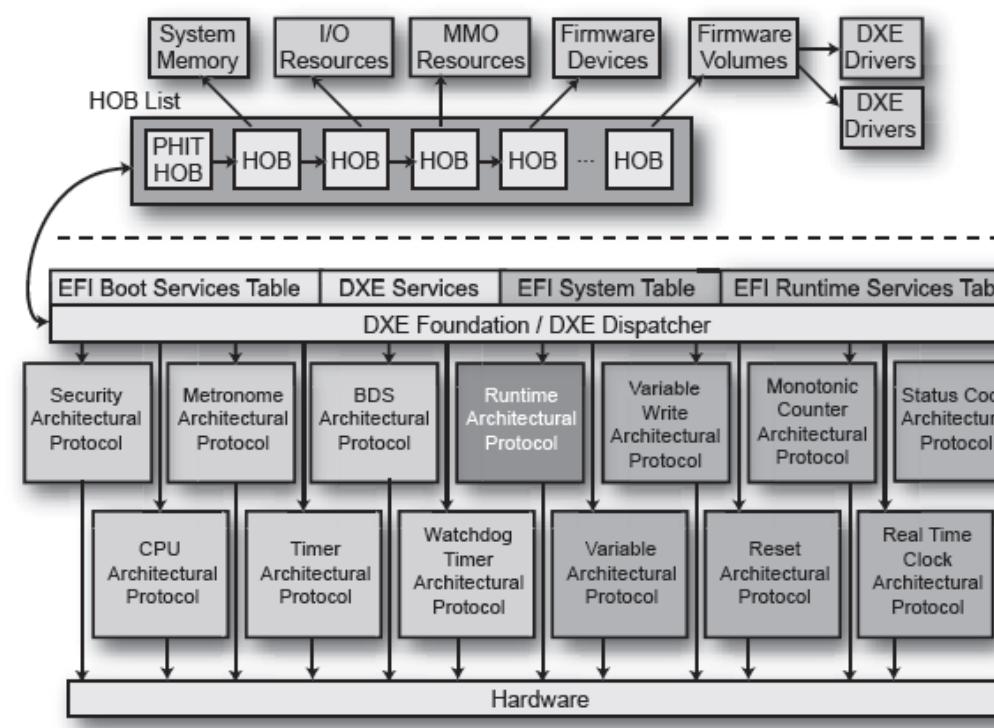


图 9.4 DXE Architectural Protocols

图9.4中左部的DXE架构protocol是用来创建EFI启动服务的。DXE基础结构，DXE调度器，以及左边的protocol会在操作系统运行时被释放掉。右边的DXE架构protocol是用来创建EFI运行时服务的。这些protocol会在操作系统运行阶段仍然驻留在内存中。中间的那个运行时架构protocol比较特殊。这个protocol会在POST到操作系统运行阶段，用于将运行时服务的地址从物理地址转换到虚拟地址空间。当地址转换完成后，这些运行时服务对于POST过程中的BIOS来说将是不可访问的。

以下是一些DXE架构protocol的简要介绍：

- 安全架构protocol：用于DXE基础架构验证固件卷中的文件是否是可用的。
- CPU架构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。
 - 节拍器架构protocol：提供让处理器暂停并等待一段时间的服务。
 - 时钟架构protocol：提供在DXE基础结构中使用的heartbeat timer中断服务。
- BDS架构protocol：提供一个在DXE阶段结束时的入口地址，它会在所有DXE设备驱动完成加载后被调用。它负责系统运行从DXE阶段到BDS阶段时，建立控制台并使启动服务开始运行。
 - Watchdog Timer架构protocol：提供用于启动和中止平台监视时钟的服务。
 - 运行时架构protocol：提供运行时服务和从物理地址转换到虚拟地址的服务的运行时驱动。

动。

- 变量架构protocol: 提供环境变量读取和临时变量的存储服务。
- 写变量架构protocol: 提供永久保存变量的存储服务。
- 单一计数架构protocol: 提供用于DXE基础结构的一个64位计数器。
- 重置架构protocol: 提供用于重置和关闭系统的服务。
- 状态码架构protocol: 提供处理DXE基础结构和DXE设备驱动返回码的服务，返回的状态码可以被存储在日志中或者发送给其他设备。
- 实时时钟架构protocol: 提供设置当前时间和日期的服务，同时也包括可选择的用于唤醒功能的计时器

EFI 系统表

DXE基础结构创建了EFI系统表，系统表是由所有DXE设备驱动和BDS阶段调用的可执行映像共同调用的。它包括被DXE基础结构和所有DXE设备驱动调用的所有组件和服务。图9.5存储在EFI系统表的各种可用组件。

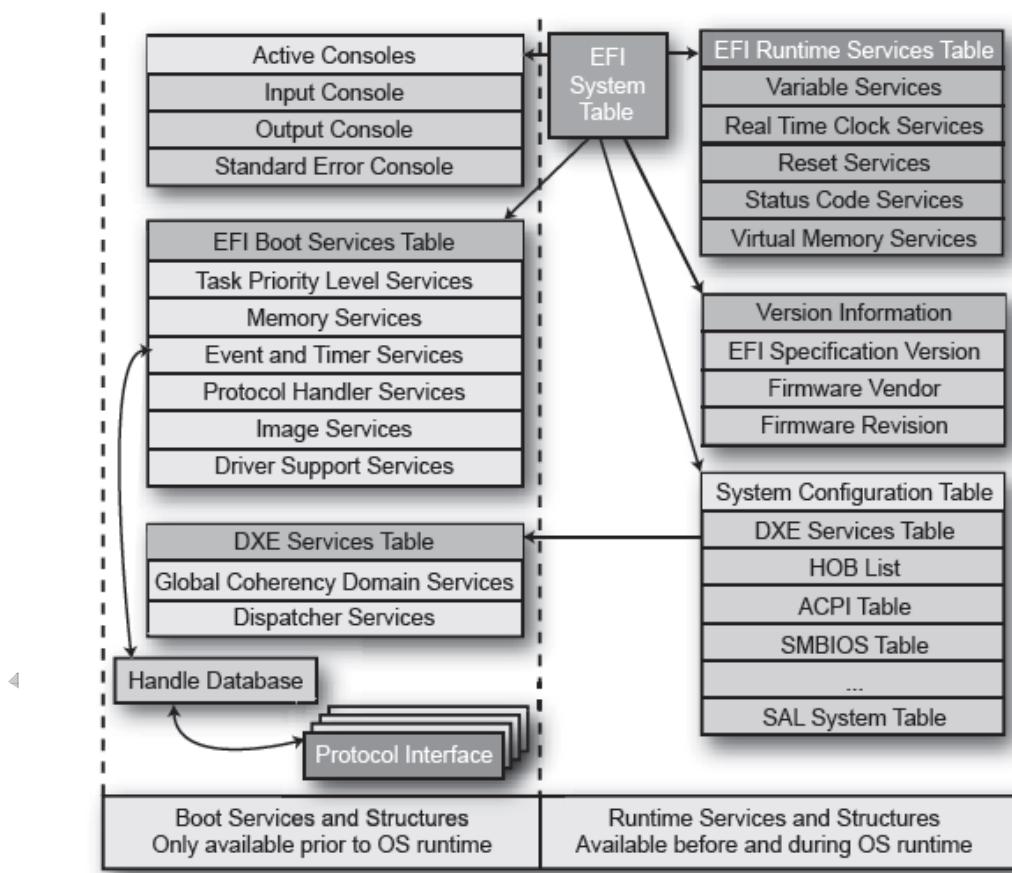


图 9.5 EFI System Table and Related Components

DXE基础结构创建了EFI启动服务，EFI运行时服务，以及基于DXE架构protocol的各种服务。

EFI系统表提供平台上所有可用设备访问方法，以及一系列的EFI配置表。EFI配置表是一个可扩展的表集，这些表描述了平台上的配置信息，包括DXE服务，HOB链表，ACPI，SMBIOS和SAL系统表。这个表集还可以在未来扩展各种新的类型的表。与此同时，任何可执行映像可以通过EFI启动服务表中的Protocol Handle Services访问句柄数据库和所有已注册在DXE驱动表中的protocol接口。

当操作系统进入运行时状态以后，这些句柄数据库，现有的控制方法，EFI启动服务和DXE驱动提供的各种服务都会被中止。释放的内存空间会留给操作系统使用，EFI系统表，EFI运行时服务表，以及系统设置表会存在并仍然被用于操作系统的运行时环境中。当然你也可以选择将所有的EFI运行时服务从物理地址转换成操作系统管理下的虚拟地址。地址转换只能运行一次。

EFI 启动服务表

以下是EFI启动服务表的各种服务的简要介绍：

- 任务优先级服务：提供的服务用于提高或降低当前的任务优先级别。这种优先级机制可以用于实现简单的锁以及使计时器中断一小段时间。这个服务依赖于CPU架构protocol。
- 内存服务：提供分配和释放4K的页面，以及分配和释放以字节为最小单位的内存池管理服务。它还提供了获取当前平台的物理内存分配映射表。
- 事件和计时器服务：提供用于创建事件，触发事件，查询事件状态，等待事件和关闭事件的服务。其中一类事件是计时器事件，计时器事件支持各种频率的周期性计时器，和一小段等待时间的事件。这些服务依赖于CPU架构protocol，计时器架构protocol，节拍器架构protocol和时钟监视架构protocol。
- Protocol句柄服务：提供了向句柄数据库中添加和删除句柄的服务。它同时也提供了向句柄中添加和删除protocol的服务。此外还包括用于其他组件在句柄数据库中查找，打开和关闭句柄的服务。
- 映像服务：提供加载，启动，退出和卸载PE/COFF格式映像的服务。这些服务依赖于安全架构protocol。
- 驱动支持服务：为平台设备提供启用和停用驱动的服务。BDS阶段这些服务被用于启用所有设备或者仅启用最小数量的设备，以便完成控制和启动到操作系统。这种最小化启用策略被用于快速启动机制中。

EFI 运行时服务表

以下是EFI运行时服务表的各种服务的简单介绍：

- 环境变量服务：为永久存储的环境变量提供查找，添加，删除的服务。这些服务依赖于变量架构protocol和写变量架构protocol。
- 时间日期服务：提供读写当前时间和日期的服务。同时也提供读取和设置可选的唤醒计时器的服务。这些服务依赖于时间日期架构protocol。
- 重置服务：提供用于重置和关闭系统的服务。这些服务依赖于重置架构protocol。
- 状态码服务：提供处理状态码的服务，可以发送状态码到日志或者一个输出设备。这些服务依赖于状态码架构protocol。
- 虚拟内存服务：提供DXE的运行时服务从物理地址转换到虚拟地址的映射服务。这个服务只能在物理地址空间中被调用一次。一旦转换完成，被转换地址的服务将不能再被调用了。这些服务依赖于运行时架构protocol。

DXE 服务表

以下是DXE服务表的各种服务的简单介绍:

- GCD管理服务(Global Coherency Domain Services): 提供用于管理平台上I/O资源, MMIO资源, 以及系统内存资源的服务。这些服务被用于向处理器的GCD中动态增加和删除资源。
- DXE调度器服务: 提供管理被调度器加载和运行的DXE设备驱动的服务。

全局共享服务

全局共享(GCD)服务被用来管理处理器启动过程中的可用内存和I/O资源。这些资源通过两个图来管理:

- GCD内存空间分配图
- GCD I/O空间分配图

GCD内存空间分配图和I/O空间分配图在内存和I/O资源被增加、减少、分配、释放时会被动态更新。同时GCD服务也提供了恢复这两个资源分配图的方法。

GCD服务可分为两个部分。一个用来管理处理器启动过程中的可用内存资源, 另一个用来管理可用I/O资源。由于并不是所有的处理器都支持使用I/O资源, 所以用于管理I/O资源的服务并不是必须的。无论如何, 由于系统内存资源和MMIO对于DXE环境是必要的, 所以用于管理内存资源的服务是必须的。

GCD 内存资源

GCD服务用于管理内存的方法主要有以下几种:

- AddMemorySpace()
- AllocateMemorySpace()
- FreeMemorySpace()
- RemoveMemorySpace()
- SetMemorySpaceAttributes()

GCD服务中用于恢复GCD内存空间分配图有以下两种:

- GetMemorySpaceDescriptor()
- GetMemorySpaceMap()

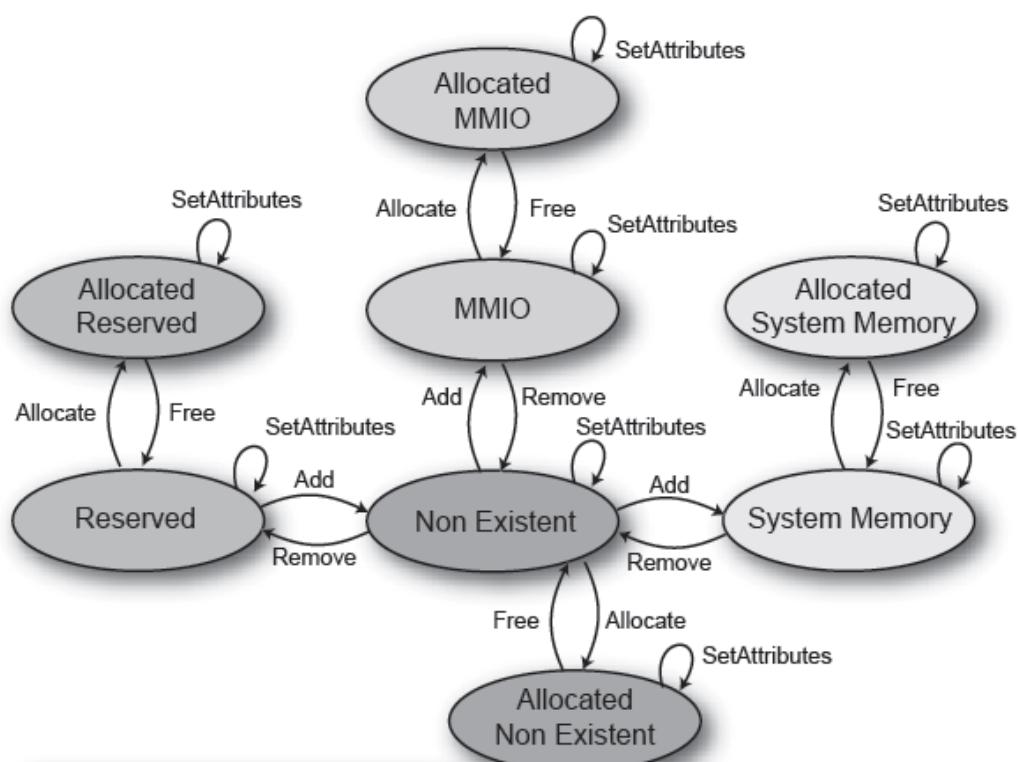
GCD内存空间分配图的初始化所必须的信息是来自DXE基础框架中的HOB链表。链表中有一类HOB节点用来描述内存地址总线的, 地址总线可以被用来访问内存资源。这些信息是用来初始化GCD内存空间分配图的状态。任何超出地址总线的内存对于GCD内存资源管理服务来说都是不可用的。GCD内存空间分配图使用一个64位地址管理可用内存。GCD内存资源管理的所有内存区域都是以比特为最小单位访问。还有一类HOB节点用于描述系统内存的位置, 内存映射I/O资源的位置, 固件设备的位置, 固件卷(firmware volumes)的位置, 预留空间的位置, 以及分配给DXE框架结构使用的系统内存的位置。DXE框架结构会解析HOB链表里面的这些信息, 并且保证这些用于DXE框架结构的地址段是可用的。所以, GCD内存空间分配图必须按照HOB链表中的描述完成地址空间的映射。GCD内存空间分配图为DXE框架结构中的内存管理函数提供必要的信息, 内存管理函数例如AllocatePages(), FreePages(), AllocatePool(),

FreeePool(), GetMemoryMap()。

GCD内存空间分配图中的内存区域有四种状态：

- Nonexistent memory (不可用内存)
- System memory (系统内存)
- Memory-mapped (I/O内存映射I/O)
- Reserved memory (保留内存)

内存块可以由DXE环境下的DXE驱动分配和释放。另外，DXE设备驱动还可以指定内存块的缓存属性。图9.6显示了内存块在GCD内存空间分配图中的状态转换。由GCD服务管理的内存可以从一种状态转换成另一种状态。GCD服务必须能够移动相同状态的内存块并将其合并成一个新的内存块，这样可以减少GCD内存空间分配图的内存块数量。



Operation	GCD Service
Add	AddMemorySpace()
Remove	RemoveMemorySpace()
Allocate	AllocateMemorySpace()
Free	FreeMemorySpace()
SetAttributes	SetMemorySpaceAttributes()

图 9.6 GCD Memory State Transitions

GCD I/O 资源

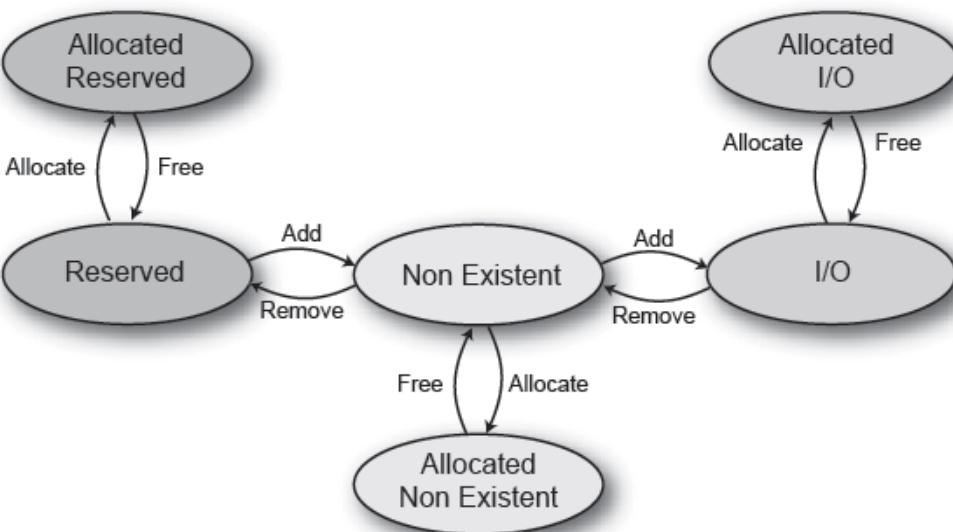
GCD服务用于管理I/O资源的方法主要有以下几种：

- AddIoSpace()
- AllocateIoSpace()
- FreeIoSpace()
- RemoveIoSpace()

GCD服务中用于恢复GCD I/O空间分配图的方法有以下两种：

- GetIoSpaceDescriptor()
- GetIoSpaceMap()

GCD I/O空间分配图的初始化所必须的信息是来自DXE基础框架中的HOB链表。有一类HOB节点描述了用于访问I/O资源的地址线数量。任何超出这个地址线的I/O地址对于GCD服务来说都是不可用的。GCD I/O空间分配图使用一个64位地址描述I/O地址信息，GCD I/O空间分配图中的I/O块最小单位是字节。



Operation	GCD Service
Add	AddIoSpace()
Remove	RemoveIoSpace()
Allocate	AllocateIoSpace()
Free	FreeIoSpace()

图 9.7 GCD I/O State Transitions

GCD I/O空间分配图的I/O块有三种状态，包括不可用I/O，I/O，以及保留I/O。I/O资源可以由DXE环境下的设备驱动分配和释放。图9.7显示了I/O块的状态转换。由GCD服务管理的I/O可以从一种状态转换成另一种状态。GCD服务必须能够移动相同状态的I/O块并将其合并成一个新的I/O块，这样可以减少GCD I/O空间分配图的I/O块数量。

DXE 调度器

在DXE基础框架完成初始化以后，控制权被转交给DXE调度器。DXE调度器负责加载和调用固件卷中的DXE驱动。DXE调度器根据HOB链表中的描述信息在固件卷中查找驱动。随着程序执行，其他的固件卷会被相继找到。当新的固件卷被找到以后，DXE调度器也会搜索新的固件卷中的所有驱动。

当一个新的固件卷被找到以后，我们会根据一个优先级描述文件来查找驱动。这个优先级描述文件使用一个固定的文件名并且包括了应该被优先加载并执行DXE驱动的加载顺序。每个固件卷最多只能有一个优先级描述文件，但是也允许没有任何优先级描述文件。在较高优先级的DXE驱动被加载以后，相关的依赖表达式会被更新并找出下一个被加载的DXE驱动。优先级描述文件会提供一个固定执行顺序的DXE驱动列表，这些列表中的驱动的加载不使用依赖表达式。依赖表达式提供给剩下的优先级较低的DXE驱动使用。在每个DXE驱动执行前，都必须经过安全架构protocol的验证。这种验证是为了禁止加载其他不明来源的DXE驱动程序。

在优先级描述文件中的驱动和所有满足依赖表达式的DXE驱动都被加载以后，控制权从DXE调度器被移交给BDS架构Protocol。BDS架构Protocol负责创建控制台设备并尝试启动操作系统。在控制台设备和用于启动操作系统的驱动都被创建完成以后，可能会有新的固件卷被发现。如果BDS架构Protocol不能启动控制台，或者不能取得启动操作系统的权限，DXE调度器会被重新调用。重新启动的DXE调度器会加载并执行最近一次新发现的固件卷中的设备驱动。在DXE再次加载完所有可用驱动以后，控制权会重新被移交给BDS架构Protocol，以便继续启动操作系统。图9.8描述了DXE调度器、DXE驱动和BDS的执行顺序。

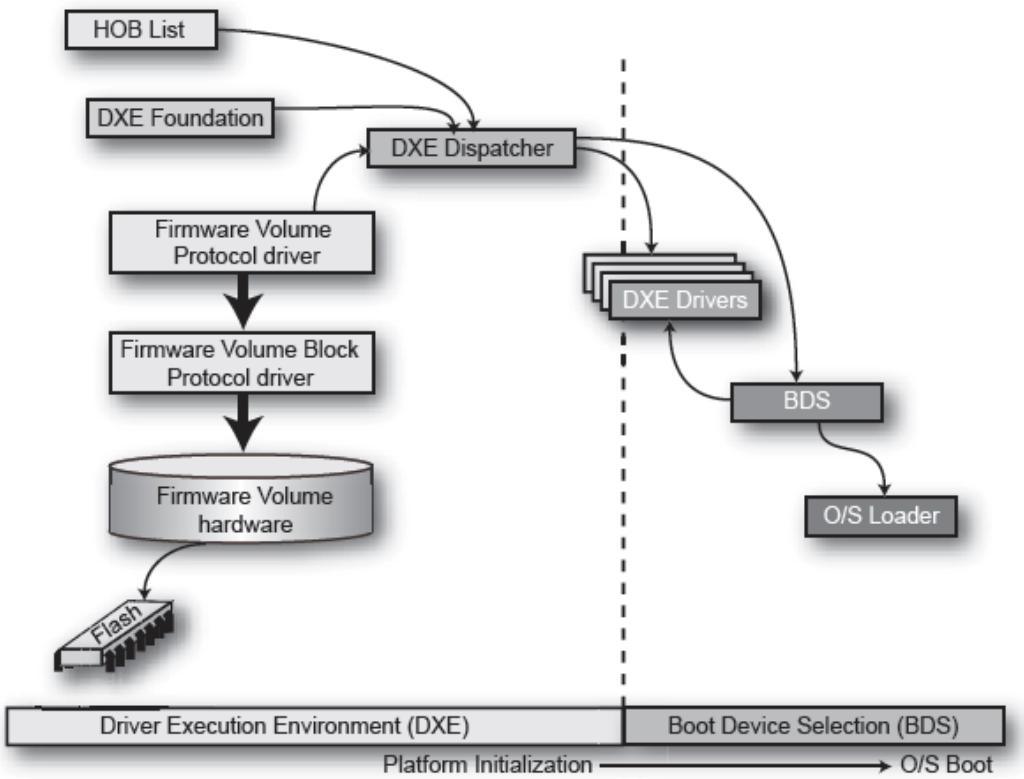


图 9.8 The Handshake between the Dispatcher and Other Components.

优先级文件

优先级文件是固件卷中可能包含的一个特别的文件。每个固件卷中最多可以包括一个优先级文件。优先级文件使用已知的GUID文件名，所以DXE调度器总是能通过这个GUID来找到优先级文件。每当新的固件卷被发现以后，DXE调度器会首选尝试查找这个优先级文件。优先级文件中包括了优先级较高的一个DXE驱动列表，这些驱动必须在其他较低优先级驱动被加载前被执行。优先级文件中的驱动列表是按照执行顺序排列的。如果这些驱动包含有依赖表达式，那么这些依赖表达式会被忽略掉。

优先级文件的目的是为了提供一个固定执行顺序的DXE驱动列表。其它根据依赖表达式执行的DXE驱动优先级较低，也就是说，这些驱动在系统启动过程中的加载顺序不是固定的。当然，有时候这些驱动也需要一个固定的执行顺序。例如用于提供debug DXE驱动的服务有时候就必须被加入优先级文件中。否则如果使用依赖表达式的话，这些服务可能会在很晚才被加载。如果能在更早的时候加载debug服务，我们就可以debug更多的DXE基础架构和DXE驱动。另外一个例子是使用优先级文件来忽略依赖表达式。有些嵌入式平台可能只需要少量的DXE驱动按照固定顺序加载。优先级文件就可以提供一个严格按照固定顺序来加载的方法，而不使用依赖表达式的顺序加载。当然依赖表达式描述了之前必须加载的固件驱动，这种方法可以保护固件空间不被破坏。优先级文件的最主要目的是为了使固件平台的设计更加灵活。

依赖性语法

DXE驱动是以文件的方式存储在固件卷中，它占用一个或多个区域。其中一个区域必须是PE/COFF映像。如果这个DXE驱动包括一个依赖表达式，那么这个依赖表达式会被存储在一个特定区域。DXE驱动可能还包括额外的区域用于压缩和加壳。并且，DXE调度器根据DXE驱动的类型来辨识不同的驱动。DXE调度器可以通过查找DXE驱动附属区域来查询依赖表达式。附属区域包括由一种数据流类型的操作数和操作码组成的依赖表达式。

表 9.1 Supported Opcodes in the Dependency Expression Instruction Set

Opcode	Description
0x00	BEFORE <File Name GUID>
0x01	AFTER <File Name GUID>
0x02	PUSH <Protocol GUID>
0x03	AND
0x04	OR
0x05	NOT
0x06	TRUE
0x07	FALSE
0x08	END
0x09	SOR

为了节省空间，存储在附属区域中的依赖表达式必须被设计的简短。另外，它们还应该能被很快解析，以便降低执行时间。为了达到这两个目的，我们必须设计一种简短，基于栈结构的依赖表达式编码。DXE调度器必须实现一种用于解析依赖表达式的解释程序。表9.1给出了依赖表达式支持的一些操作码。

由于在同一时刻可能有多个依赖表达式值都为true，所以即使在完全相同的平台和固件中，在启动时，DXE驱动的执行顺序都可能是不同的。这种不确定性就是为什么说使用依赖表达式执行的DXE驱动的顺序是不固定的。

DXE 驱动

DXE驱动可以分为两类：

- DXE阶段早期执行的DXE驱动。
- 按照EFI 1.1驱动模型创建的DXE驱动。

第一类在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EFI 1.1驱动模型创建的DXE驱动。

当第二类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

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后24小时内删除，否则后果自负。

Driver Binding Protocol接口注册到handle数据库中。在BDS阶段，Driver Binding Protocol会被用于创建控制台和访问启动设备。第二类驱动可以提供控制台设备和启动设备的软件支持。

DXE驱动需要调用EFI启动服务和EFI运行时服务来执行他们的功能。应该注意到，此时DXE架构Protocol可能没有完成注册。DXE驱动必须使用依赖表达式来确保当前驱动所依赖的其他服务都已经是可用的。

在DXE阶段，第二类DXE驱动会被注册到handle数据库中，然后这些驱动的Driver Binding Protocol会在BDS阶段被执行，此时所有的DXE基础架构protocol都已经被加载了。如果DXE调度器已经执行完所有的DXE驱动，但是DXE Architectural Protocols并没有完全加载，BIOS必须发出一个严重错误并halt住系统。

BDS 阶段

BDS Architectural Protocol 是在DXE阶段被找到，并且在BDS阶段被执行的，protocol在执行前必须满足两个条件：

- 所有DXE Architectural Protocols都已经注册到handle数据库中。这个条件要求DXE Foundation 已经创建出完整的EFI启动服务和EFI运行时服务。
- DXE调度器已经将所有满足加载条件的驱动都加载完成了。这个条件要求，所有在优先级描述文件中的驱动和所有满足依赖表达式为真的驱动都被加载和执行完成了。

BDS Architectural Protocol查找并加载在许多应用，这些应用程序可以在启动前环境中执行。这些应用程序可以创建一个传统的OS boot loader，或者一个扩展服务以便用来加载最后需要启动的OS。这些扩展启动前服务可以包括setup配置、扩展诊断服务、flash升级支持、OEM服务，或者OS启动代码。

像IBV、OEM、ISV等供应商可以选择使用推荐的方式，在reference代码基础上实现他们自己的平台；或者是在现有代码的基础上修改完成BDS Architectural Protocol中的各种应用。

BDS阶段会执行一系列任务。在不同的平台上，用户界面和响应方式可能完全不同；但是BDS阶段的启动规则却非常相似。这些任务包括：

- 基于ConIn、ConOut和StdErr这些环境变量初始化控制台设备。
- 尝试加载环境变量中列在Driver####和DriverOrder中的所有设备
- 试图从Boot####和BootOrder环境变量中列出的设备开始启动系统。

如果BDS阶段连接控制台设备失败，加载驱动或者启动失败，DXE调度器会被重新调用。这次调用可能是必须的，因为在上次完成调用后，可能有新的固件卷被找到。这些新找到的固件卷也许包括有用来管理控制台设备和启动设备的DXE驱动。在所有新发现的固件卷中的驱动都被加载以后，控制权会被重新交给BDS阶段。如果BDS阶段还是不能连接控制设备或者是启动设备，那么BDS会启动失败。BDS启动失败后会自动尝试从下一个控制台设备启动。

控制台设备

控制台设备是使用简单文本输入输出协议的设备。在基于Framework的平台上，任何能支持简单文本输入/输出协议的设备都能被作为一个控制台设备使用。下列几种类型的设备都可以作为控制台设备使用：

- VGA适配器：按照简单文本输出协议在适配器上基于文本显示输出。
- 通用图形适配器(UGA)：这些适配器有一个支持块转换操作(BLT)的图象处理接口。

UGA可以通过BLT操作向frame buffer中发送Unicode图象来实现简单文本输出协议。

■ 串口终端：串口终端设备可以同时实现简单文本输入输出协议。串口中断非常灵活，它支持各种协议例如PC-ANSI, VT-100, VT-100+, 还有VTUTF8。

■ Telnet：telnet可以同时实现简单文本输入输出协议。和串口一样，telnet支持的协议包括PC-ANSI, VT-100, VT-100+, VTUTF8。

■ 远程图象显示(HTTP)：远程图象显示也可以实现单文本输入输出协议。比如使用HTTP协议实现，因此可以基于Framework的平台，使用标准Internet浏览器管理

启动设备

Framework支持的启动设备包括：

- 使用FAT格式文件系统，且支持Block I/O Protocol 设备
- 支持File System Protocol的设备
- 支持Load File Protocol的设备

磁盘设备一般支持Block I/O Protocol，网络设备一般支持Load File Protocol。

Framework也可以选择使用传统的兼容设备。这些设备会提供用于启动到传统的OS的服务，BDS阶段应该必须支持启动到传统OS。

中止启动服务

在OS加载和成功启动完成后，BDS阶段就结束了。OS加载器或者OS kernel可以通过调用ExitBootServices()服务来中止BDS阶段。调用结束后，所有的启动服务都会释放其使用过的资源。调用返回后，系统进入Runtime(RT)阶段。

第十章

一些常见的 EFI 功能

不要让将来的事困扰你，因为如果那是必然要发生的话，
你将带着你现在对待当前事物的同样理性走向它们。

—Marcus Aurelius Antoninus

我 EFI 提供了大量的功能函数用于驱动和应用程序与底层的 EFI 部件进行通信。许多之前设计的接口都有很多局限性的，因为我们不能预见技术改变的趋势。让我们举一个这样的例子，当时在设计硬盘接口时，普遍认为一个硬盘可用空间永远不会大于 8GB。这往往是很难预测技术可能会提供什么样的变化。许多著名论断曾说过个人的计算机永远不会现实的出现，或者保证 640kb 的内存已经足够读者使用了。由于过去贫乏的推断缺乏预见性，人们可能尝试去学习现在看起来是很不错的“错误”知识和设计接口，然后努力从今天这些常见的实践中尝试预见将来怎样使用现在设计的这些接口。本章描述了一个存在 EFI 以及框架中的常见的接口选择：

- 这是一组协议从 EFI 驱动程序和应用中抽象的硬件平台。这些协议一般情况下只是使用在 DXE 的阶段，并且在一个框架的基础上的执行。在目前的形式中这些协议只是在框架中被介绍，并没有在当前 EFI 规范中被定义。
- PCI 协议。这些协议是与底层 PCI 总线各方面的相互作用，列举，总线以及资源分配。这些接口在 EFI 中介绍，并且将在 EFI 和框架的实现中存在。
- I/O 模块：这个协议用于抽象的大容量存储设备，允许代码运行在 EFI 启动服务环境中访问这个协议，并且不用设备类型和管理设备控制器的特殊知识。这些接口在 EFI 中介绍，并且将在 EFI 和框架的实现中存在。
- 磁盘 I/O：这个协议用于抽象的块访问 I/O 模块协议，是更加普通的长度偏移量协议。固件负责添加此协议到任何 I / O 模块接口，这些模块存在没有磁盘 I/O 协议的系统中。文件系统和其他磁盘访问代码利用磁盘 I / O 协议。这些接口在 EFI 中介绍，并且将在 EFI 和框架的实现中存在。
- 简单的文件系统：这个协议允许代码运行在 EFI 启动的服务环境，以获得基于文件访问设备。简单文件系统协议，用于打开一个设备卷并且返回一个 EFI_FILE 的句柄，用于提供在设备卷上访问文件的接口。这些接口在 EFI 中介绍，并且将在 EFI 和框架的实现中存在。

协议结构举例

大量的结构协议存在于平台中。这些协议功能就像在各种方式调用中的其他协议。唯一不同的是，这些协议是由平台的核心服务调用，然后其余的驱动和应用程序依次调用这些核心服务以各种方式在平台上激活运行。一般来说，结构协议的唯一用户是核心服务的本身。结构协议在预启动环境时，在系统中唯一的与硬件进行沟通的抽象硬件。其他一切在服务对任何硬件的请求都会通过与核心服务沟通。图 10.1 说明了这种高层次的软件沟通。

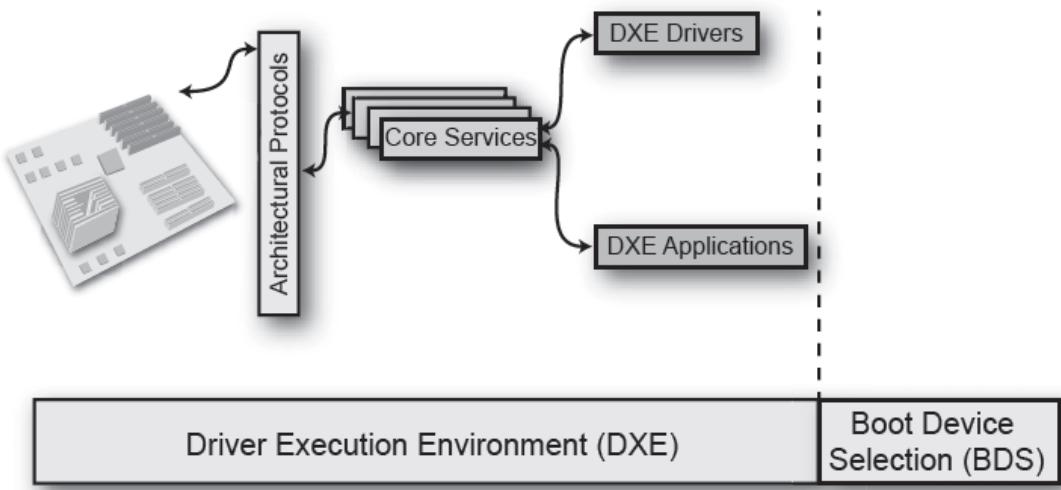


图 10.1 Platform Software Flow Diagram

为了更清楚地表明，在这些结构协议的设计和运作方式，几个关键的例子，将被说明在进一步的细节中。请注意，下面的例子不是一套完整的结构协议，但是可以用来说明一些典型的功能。对于全套说明，请参考相应的 DXE 规格。

CPU 的构造协议

CPU 的构造协议在 D X E 基础上用于抽象的特殊处理器功能。这包括缓存运用，启用和禁用中断，调用中断向量和异常向量，读内部处理器定时器，复位处理器，并确定处理器的频率。这一协议必须由启动服务或者运行时的 DXE 驱动程序产生，并且可能只能在 DXE 基础上，和产生的此结构协议的 DXE 驱动程序上运行。通过允许由一个引导服务驱动程序产生协议，很显然，这种抽象协议不会持续存在，当平台的启动服务被引导到一个启动目标，例如操作系统而终止时。

GCD 内存空间映射是在 D X E 基础上基于 HOB 列表的内容被初始化的。HOB 列表包含不同的内存区域的功能，但是它不包含它们当前的属性。DXE 驱动程序负责维护处理器能够访问到得当前内存区域的属性。

这意味着产生 CPU 构造协议的 DXE 驱动程序必须记录下对处理器能够访问到的内存所有区域的初始状态的属性的 GCD 内存空间映射。DXE 服务函数 SetMemorySpaceAttributes() 允许修改内存范围的属性。DXE 服务函数 SetMemorySpaceAttributes() 通过 CPU 构造协议的服务函数 SetMemoryAttributes() 来执行。

为了初始化在 GCD 的内存空间映射的属性状态，DXE 驱动程序产生的 CPU 构造协议必须调用 DXE 服务程序 SetMemorySpaceAttributes() 传递当前的属性为处理器能够可见的所有不同的内存区域。这反过来将会回调 CPU 的构造协议的 SetMemoryAttributes() 服务程序，并且自从在 DXE 基础上会只要求对内存区域的属性被设置为当前的设置，所有的这些调用必须返回 EFI_SUCCESS。在 GCD 内存空间映射中强迫将当前的属性被设置为当前的设定。在这些初始化完成以后，下一步调用 DXE 服务程序 GetMemorySpaceMap() 将正确地显示所有的内存区域目前的属性。此外，今后任何调用 DXE 服务程序 SetMemorySpaceAttributes() 将会依次调用 CPU 的

构造协议，所以将会看到如果这些属性被修改，GCD 内存空间映射将会相应的被更新。

CPU 的构造协议使用下面的协议定义：

```
Protocol Interface Structure
typedef struct _EFI_CPU_ARCH_PROTOCOL {
    EFI_CPU_FLUSH_DATA_CACHE    FlushDataCache;
    EFI_CPU_ENABLE_INTERRUPT    EnableInterrupt;
    EFI_CPU_DISABLE_INTERRUPT   DisableInterrupt;
    EFI_CPU_GET_INTERRUPT_STATE GetInterruptState;
    EFI_CPU_INIT                Init;
    EFI_CPU_REGISTER_INTERRUPT_HANDLER
        RegisterInterruptHandler;
    EFI_CPU_GET_TIMER_VALUE     GetTimerValue;
    EFI_CPU_SET_MEMORY_ATTRIBUTES SetMemoryAttributes;
    UINT32                      NumberOfTimers;
    UINT32                      DmaBufferAlignment;
} EFI_CPU_ARCH_PROTOCOL;

■ FlushDataCache—刷新范围内的处理器的数据缓存。如果处理器不包含数据缓存，或者数据缓存是完全一致的，那么这个功能会返回EFI_SUCCESS。如果处理器不支持刷新在数据缓存中地址范围，那么数据缓存中的全部必须被刷新。这功能被调用在根桥I / O抽象的DMA操作刷新数据缓存。
■ EnableInterrupt—处理器启用中断处理。见EnableInterrupt()功能描述。这个功能被Boot Service RaiseTPL()和RestoreTPL()函数调用。
■ DisableInterrupt—通过处理器禁用中断处理。见DisableInterrupt()功能描述。这个功能被Boot Service RaiseTPL()和RestoreTPL()调用。
■ GetInterruptState—得到该处理器的当前的中断状态。
■ Init—处理器产生一个INIT中断。此功能被用于根据指定的启动路径重置结构协议。如果处理器不能以编程方式生成一个没有外部硬件帮助的INIT，那么此功能返回EFI_UNSUPPORTED。
■ RegisterInterruptHandler—在处理器的中断向量表中查找一个相应的中断服务进程。此功能典型的被调用在计时器结构协议在系统中调用计时器中断时。它也能够在查找调用异常向量表时被调用。
■ GetTimerValue—返回一个处理器内部的计时器的值。
■ SetMemoryAttributes
■ NumberOfTimers—给一个处理器中可用的计数器的数量。在CPU结构协议被安装之后，在此字段中的值是一个常数，不能修改。所有使用者必须视它为只读字段。
■ DmaBufferAlignment—以字节为单位提供分配DMA缓冲区排列所需的大小。这是在平台中典型的数据缓存行的大小。通过在平台中所有现存的缓存中查找这个数据缓存行的大小并且返回最大值而决定此值。这些被根桥I/O抽象的协议使用，以保证没有2个DMA缓冲区共享同一个缓存行。这个值是一个常量，在安装完CPU结构协议以后是不能被改变的。所有的使用者必须视它为只读字段。
```

实时时钟结构协议

实时时钟结构协议为访问一个系统的实时时钟硬件提供服务。这一协议必须是在运行DXE驱动程序时被产生，并且可能只能由DXE基础上会调用。

这一协议必须是一个运行的DXE驱动程序产生。这个驱动程序负责初始化在EFI运行时的服务表的四个领域GetTime(), SetTime(), GetWakeupTime(), and SetWakeupTime()。请参阅“第5章时服务的有关这些服务的细节”。在EFI运行时服务表中的四个领域已经被初始化以后，驱动程序必须在一个新的指向空接口的句柄上安装实时时钟构造协议。这一协议的安装通知DXE阶段与实时时钟相关的服务可以应用，而且在DXE基础上必须更新EFI运行时服务表的32位CRC。

计时器构造协议

计时器构造协议提供的服务，包括初始化一个周期性计时器中断，并存储一个为每次计时器中断调用的句柄。它也可以提供服务来调整计时器中断周期的大小。当一个计时器中断发生时，这个句柄记录从上次发生计时器中断到这次的时间量。该协议打开使用SetTimer()启动服务。这一协议必须由启动服务产生或运行的DXE驱动程序，可能只能在DXE基础上运用，或者DXE驱动程序产生其它的DXE阶段的结构协议。通过允许一个由引导服务驱动器产生的协议，很显然，这种抽象协议不会持续存在，当平台的启动服务被引导到一个启动目标，例如操作系统。

Protocol Interface Structure

```
typedef struct _EFI_TIMER_ARCH_PROTOCOL {
    EFI_TIMER_REGISTER_HANDLER RegisterHandler;
    EFI_TIMER_SET_TIMER_PERIOD SetTimerPeriod;
    EFI_TIMER_GET_TIMER_PERIOD GetTimerPeriod;
    EFI_TIMER_GENERATE_SOFT_INTERRUPT GenerateSoftInterrupt;
} EFI_TIMER_ARCH_PROTOCOL;
```

■ *RegisterHandler*(寄存器句柄)——寄存器记录一个每次发生计时器中断调用的句柄。计时器周期定义为每次计时器中断之间最小的时间差，所以计时器周期也是每次调用寄存器句柄之间的最短时间差。

■ *SetTimerPeriod*(设置计时器周期)——设置计时器的中断期是以100纳秒为单位的。这个功能是可以选择的并且可能返回一个EFI_UNSUPPORTED。如果这个功能是被支持的，那么计时器周期四舍五入至最接近支持的计时器时期。

■ *GetTimerPeriod*(取得计时周期)——得到这个计时周期是以100纳秒为单位的。

■ *GenerateSoftInterrupt*(产生软终端)——产生一个计时器软终端，模仿计时器中断的产生。如果计时器中断已在一段时间内被屏蔽了那么这个服务可以用来调用寄存器句柄。

复位构造协议

复位构造协议提供复位所需的服务平台。这一协议必须由运行时的DXE驱动程序产生，并且可能只能在DXE基础上运用。这个驱动程序负责初始化EFI运行时服务表的ResetSystem()领域。在EFI运行时服务表中的这个领域已经被初始化以后，驱动程序必须在一个新的指向空接口的句柄上安装复位构造协议。这一协议的安装通知DXE阶段与复位相关的服务可以应用，而且在DXE

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后24小时内删除，否则后果自负。

基础上必须更新EFI运行时服务表的32位CRC。

选择引导设备构造协议：

引导设备选择构造协议，将DXE的控制转移到由操作系统或者系统工具控制，正如图10.2的说明。这一协议必须由启动服务或者运行时的DXE驱动程序产生，并且可能只能在DXE基础上运用。通过允许一个由引导服务驱动器产生的协议，很显然，这种抽象协议不会持续存在，当平台的启动服务被引导到一个启动目标，例如操作系统。

当此协议用来访问系统要求某个启动的设备时，如果此时还没有足够的驱动程序被初始化好，那么这儿协议将会添加驱动程序到调度的队列中并且将返回控制权回到这个调度程序。一旦这个被要求启动的设备可以用了，那么这个引导启动设备可以用来装载并且调用操作系统或者一个系统实用程序。

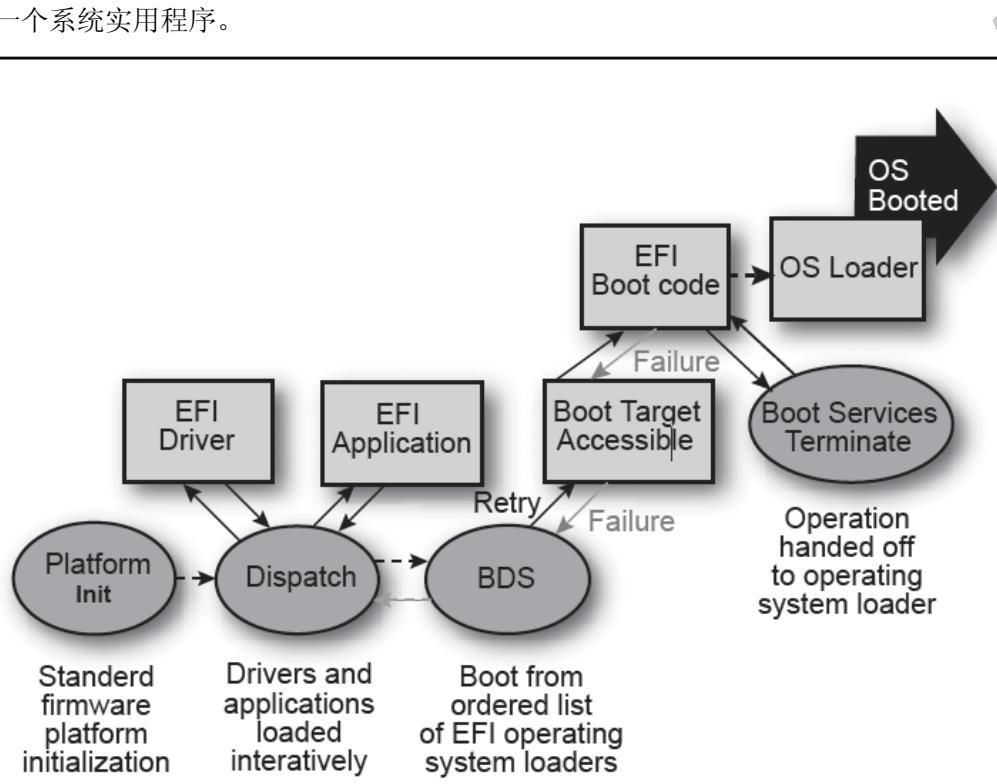


图 10 2 Basic Dispatch and BDS Software Flow

Protocol Interface Structure

```
typedef struct _EFI_BDS_ARCH_PROTOCOL {
    EFI_BDS_ENTRY Entry;
} EFI_BDS_ARCH_PROTOCOL;
```

Entry -这是指向引导设备选择的入口。见Entry()函数功能描述。这个函数调用没有任何参数，并且它的返回值也可以被忽略。如果它有返回值，那么这个调度程序必须重新被调用，如果它没有返回值，这是操作系统或者系统实用程序将会被调用。

变量结构协议

此变量结构协议是为服务提供所要求得到的和设置的环境变量。这一协议必须由运行时的DXE驱动程序产生，并且可能在DXE基础上被运用。这个驱动程序负责初始化EFI运行时服务表的

GetVariable(), GetNextVariableName(), 和 SetVariable()三个领域。请参阅第5章“变量服务”有关这些服务的详情。在EFI运行时服务表中的这三领域已经被初始化以后，驱动程序必须在一个新的指向NULL的句柄上安装变量结构协议。这个安装的协议通知在DXE基础上只读的和可变的环境变量相关的服务可以应用，并且在DXE基础上必须更新EFI运行时服务表为32位CRC。直到此协议和可写变量结构协议被安装，这时全部的环境变量服务才能应用。DXE驱动程序需要只读访问或读/写访问可变的环境变量时，必须在表达式中依赖这些结构协议。DXE驱动程序需要写访问不可变的环境变量时，必须在表达式中依赖可写变量结构协议。

看门狗计时器结构协议

看门狗计时器结构协议用于看门狗计时器程序，并且当引用看门狗计时器时存储一个句柄。这一协议必须由启动服务或者运行时的DXE驱动程序产生，并且可能只能在DXE基础上，或者产生其它DXE结构协议的DXE驱动程序上运行。当看门狗计时器终止时，如果一个平台，希望执行特定平台的行为时，那么包含执行BDS结构协议的DXE驱动程序将会使用此协议的RegisterHandler()服务。

此协议为执行启动服务-SetWatchdogTimer()提供所需求的服务。它提供的服务用来设置在启动看门狗计时器之前等待的时间量，它也提供服务用来存储当启动看门狗计时器时调用的一个句柄。此协议可以通过使用事件和计时器器启动服务来执行看门狗计时器，也可以使用自定义硬件执行看门狗计时器。当看门狗计时器被启用，那么控制权将交给之前被存储的句柄。如果之前没有存储句柄，或者存储的句柄被返回，那么系统将要调用运行时服务程序ResetSystem()来复原。

Protocol Interface Structure

```
typedef struct _EFI_WATCHDOG_TIMER_ARCH_PROTOCOL {  
    EFI_WATCHDOG_TIMER_REGISTER_HANDLER      RegisterHandler;  
    EFI_WATCHDOG_TIMER_SET_TIMER_PERIOD     SetTimerPeriod;  
    EFI_WATCHDOG_TIMER_GET_TIMER_PERIOD     GetTimerPeriod;  
} EFI_WATCHDOG_TIMER_ARCH_PROTOCOL;
```

RegisterHandler-存储句柄，当启用看门狗计时器时，存储被调用的一个句柄。

SetTimerPeriod-设置计时器周期，设置在启用看门狗计时器之前等待的时间以100纳秒为单位。如果设置的功能被支持，那么看门狗计时器周期被四舍五入到被支持的最接近的看门狗计时器周期。

GetTimerPeriod-得到计时器周期，取得在启用看门狗计时器之前，系统将要等待的时间以100纳秒为单位。

未完 Page174-194

第 11 章

信息传递

最终，种子种下了，在澳大利亚的土地上出现了直线和理性的曲线。直线的办公室和旅馆。高速路得曲线和歌剧院的平滑的弧线。最错综复杂而又精妙的几何实体。

—Michael Frayn

本章主要关于基础架构部分，关于Framework同等的管道，电力，废料和信息系统盘绕成的现代化世界。与书中其他章节不同的是，不同的构造是由概念组成的，而不是功能。他们也拥有由环境需求所产生的相似的体系。他们之间的不同在于被告知的需求和被处理的数据的区别：大部分房屋都有电路和水管介入，但是将两者混在一起则有害无利。

本章将讨论的构造包括：

- 变量，作为系统数据的非易挥发存储和与操作系统通信的通道。
- 握手单元(HOBs)，PEI到DXE转换的主要管道。
- 数据中心，作为暂存用于累积和传播数据，主要是预期的外部消耗。
- 人机交互架构，用于用户和其他构造配置模块的集中管理。

相似和差异

这里所描述的构造都有以下特性

都是数据管道。可调用接口（比如协议成员函数）本身只能作为启用数据传输和模块间的管理。在传输过程中没有太多的数据编辑。他们可能支持其他构造来做编辑动作，视构造而定。这正如电力部门会为了提高效率，从发电厂到住宅传输途中改变电压。都为数据很好地定义其产生者和使用者。这意味着，例如，相关的路由选择信息的提供能够使使用者清楚地确定数据包的适用性。

有助于减少模块之间的约束力。例如，内存的初始化PEIM。在其工作的路径中，会发现关于可用的内存插槽和系统中内存类型的有用的信息。没有任何其他模块使这些数据变成可用的，作为详细目录的一部分对系统管理员来说，是非常宝贵的。使用HOB，PEIM可以缩小到DXE差距。使用者可以从DXE传输数据到数据中心。数据中心封装的使用者可以通过所需要的操作系统或预操作系统接口提供数据。

有助于放宽产生者和使用者所必须遵守的标准。比如内存初始化PEIM，这些数据能够被任何相互抵触的系统管理标准所使用。内存PEIM没有必要知道最终的使用者是IPMI，SMBIOS或者一些新的接口。

都能够正常工作而不需要调用基于基础协议的服务。PEI中的模块和后面的模块可以利用为这个阶段定义的接口而及早地运行。在许多情况下，就难以创建服务，并且利用基础协议，这个属性比较容易消失。另一方面，仍然是非常重要的。早期的DXE驱动能够提供其配置信息数据给HII数据库。例如，驱动能够提供数据，优先于接受开始请求。一旦驱动的确收到开始请求并列举其相关硬件，它可以更新配置信息来描述。关于在设计的部分确实需要长远考虑，无论

是通用的或更具体的版本让最终用户更能够理解。

都是位置独立，不受区域限制。所提供的数据都可以在内存各个位置移动。反过来，这也影响到所要传输的数据结构和程序接口与验证测试不能够执行的需求。尤其是指针的形式

```
MY_STRUCT *MyPointer;
```

在同一数据结构中从一个点到另一个点是被禁止的。该指针会变得无效如果该结构被移到另一区域。相反的，结构应该使用能够被拷贝的自相关的偏移和长度区域。这个要求也许有些不可思议。一个HOB入口描述的是指针形式的RAM的顶端，这一类指针是允许的因为它指向的是本数据结构外的不可变的位置。

既然这些构造都具有这么多的共同点，为什么他们没有被整合到一个单独的架构呢？当然存在一些理由，这大致上和一个大城市里为什么不止一种单一类型的物理基础设施的理由差不多。

有一些历史因素。自从EFI最初的设计，“变量”已经成为EFI规范的一部分。另外的构造是为了Framework而发明的，间接的表现在EFI中或者根本不。大部分普遍使用的“变量”是永久不变的（即使没有电源，该值也保持有效）而其他的实例则会每次启动的时候重新创建。

这些构造表现出不同的功能。HOBs 跳过单独的阶段边界。PEI空间施加的限制不再需要在DXE中表现出来，“数据中心”是最一般通用的构造。他通用的本质是其好处但也使其不适合于HII数据，HII数据为更加约束和更可编辑的数据类型提供支持。

变量

“变量”是EFI和操作系统之间消息传递的实体，它也有可能用于Framework本身。与本章其他机制不同，“变量”无论在ExitBootService前后都是可以被访问的。

为了理解怎样使用“变量”，首先理解为什么他们被称为“变量”是很有帮助的。Unix 提供用户所谓的环境变量，环境变量是任意字符和其他任意（ASCII）标签结合而成的。然后应用程序能够寻找标签和找到字符串。其后，环境变量是Fortran（福特兰语言）逻辑单元构造更友好的版本（例如，5永远是标准输入，6永远是标准输出）。

EFI“变量”用于与环境变量一样的目的和相似的方式。他们的命名（keys）既是GUID也是Unicode字符串。都是名子的一部分，也都是必须提供的。

GUID在EFI和Framework与其他地方有一样的目的：保证其独一无二性和避免冲突。然而，以同样名字命名的几个“变量”仍然是有效的。例如，某平台也许会选择其设置配置数据存储在某一以“Setup”命名的确定的GUID变量，同时存储其默认值在另外一个以“Defaults”命名的相同的GUID变量。

“变量”数据（值）结构是由数据产生者和使用者共同定义的。

“变量”的使用

“变量”可以由SetVariable 创建或者更新，通过GetVariable 重新得到和通过GetNextVariableName发现。这些都是实时（gRT）功能。

“变量”的可变性

“变量”以两种类型存在：易挥发的和非易挥发的。非易挥发的“变量”在系统重启中保存下来而易挥发的则丢失。这完全取决于系统设计者来确保非易挥发的“变量”会以有效的容

错方式来更新，以至于一旦 SetVariable 成功返回，该变量就会以稳定存储的方式保存下来和在接下来启动的中有效。

大小的考量

大部分“变量”都是非易挥发的。这将给系统设计者带来一个问题。总体上除了非常复杂的系统，在系统建立之初，系统用于存储非易挥发数据的空间是固定的。反过来，应用程序不能使用任意大的和任意数量的“变量”。唯一的解决方法是拒绝超过变量存储空间的请求，因此必须在调用时有效地使用有限的资源。

EFI 规范没有具体定义系统所需要变量空间的实际数量。最重要的是，规范限制定义在系统运行时必须可用的数量。这样的目的是因为特定类型的系统很可能比其他的更庞大，正如一年被认为是“庞大”的，在若干年后可能是“相当的渺小”。

在文档中有一点不明确的是谁将使用变量。尤其是，使用变量对于可选 ROMs 是很具吸引力的，但他们有另人信服的理由，在他们的配置区域存储自己的数据。

首先，如果该卡所包含的可选 ROM 被移除，在该可选 ROM 留在变量的任何数据已经失效。没有办法能够确定变量的数据只是被刚移除的卡所使用。

第二，如果卡的配置信息现在存储在变量空间，一旦该卡移到一个新的系统，卡会丢失其所有配置，这样对用户不是特别友好。

此外，可选 ROMs 保存他们的数据到变量给系统设计者提出一个没有明显答案的问题：应该给可选 ROMs 分配多少空间？最常用的方法是假设他们不需要任何空间。现在，两个负面的结果成为可能，或者可选 ROM 不能存储其配置由于变量空间已经满了，或者可选 ROM 能够成功存储其配置而操作系统不能。

普通变量

遍及 EFI 规范都是操作系统和 EFI 之间通信的特定变量的定义。大部分变量都是能够使操作系统有效传送其已经发现的或者需要在接下来启动过程中反馈给 EFI，这包括启动顺序和语言数据。

摘要

谨慎使用变量。

桥接阶段差距：握手单元（HOBs）

Framework 定义了几个启动阶段。某些阶段之间的边界比其他的更固定。边界之间最明显的分歧是在 PEI 和 DXE 阶段之间。PEI 是 RAM 限制的环境而 DXE 是可以使用合理内存数量的环境。基本的概念是完成在 PEI 过程中该做的，去获得内存，然后到 DXE 的更广阔的空间。

PEI，在内存初始化之前和过程中，累积数据，这些数据是 DXE Framework 作为初始化部分所需要的和 DXE 驱动作为其执行过程所需要的。数据必须完成从 PEI 到 DXE 的有效传送。

然而 PEI 受其低内存环境的限制，需要一个合适的内存存储和原始数据传送机制。握手单元（HOBs）是为此服务的一个机制。

遵守 PHIT

握手单元（HOB）数据结构是被组建为一个连续内存范围，包含一个已知的叫做阶段握手信息表（PHIT）的头文件和接下来其他 HOBs。每个 HOB 拥有一个公共头文件，该文件包含一个允许简单 HOB 扫描的长度。PHIT 和所有当前的 HOB 一起被统称为 HOB 列表。

PHIT 本身的位置通过 PEI Framework 调用（GetHobList）而来的。PHIT 显示了作为存储 HOBs 最大可用内存的地址和 HOB 表的当前末端。添加 HOB 是通过调用 CreateHob 来实现的。该服务处理调整 PHIT 末端指针和复制新的 HOB 数据的工作

除了 PHIT 中的指针，HOBs 不应该包含指向其他 HOBs 的指针。这是因为部分从 PEI 到 DXE 的 HOBs 的处理过程，涉及到将 HOB 列表复制到真正内存中新的位置，这些位置大部分当然是在不同的地址。在复制的过程中，HOBs 之间的指针不会被更新，所以指针表现为无效的。指向外部 HOBs 区域的指针通常是有效的。例如，PEI 过程发现的 RAM 的首地址是一个指针，从一个地方到另一地方复制 HOB 不会使地址无效。

HOBs 的使用

HOBs 能够被用作两种通信类型。最明显的是用在介于在 PEI 阶段的产生者和在 DXE 中的使用者。第二种是用在 PEI 阶段的产生者和使用者之间。因此 HOBs 结束在 PEI 空间中作为有限 RAM 虚拟内存分配的计划。因为这目的而再分配 HOBs 使用将会涉及到转移其他 HOBs，所以是不大安全的。移动其他 HOBs 将使指向被其他 PEIMs 所拥有的移动 HOBs 的指针无效。

作为他们传统工作的一部分（PEI 到 DXE 的通信），很多类型的 HOBs 规定为 Framework 的一部分，包括内存、CPU、资源和 FV。这些 HOBs 的一部分是被用于 DXE 初始化代码用来初始移植 GCD（Global Coherency Domain）表。

HOBs 也有可用于 PEIM 到 DXE 驱动间的通信。这样允许 PEIM 更小和避免可能的代码重复。识别 HOB 的 GUID 只需要被 PEIM 和驱动之间所知道和承认。

DXE 中的 HOBs

DXE 载入体需要使用 HOB 列表。他使用 CPU、FV 和 HOBs 资源来移植最初的 GCD 表和 PHIT 表去确保在使用的过程中不会破坏 HOB 列表。其他的 HOBs 可以通过 EFI 配置表格来获取，EFI 配置表格从 EFI 系统表中获取。HOBs 只能在 DXE 中使用的，而不是在 DXE 中产生的。

摘要

节约使用“HOBs”。

人机交互架构 (HII)

计算机只能识别英语（不是非常好）的时代已经结束。计算机应该被假定为用户友好或者给终端用户好的体验。他们应该假定为（更甚）可管理和可信赖。

BIOS 从来没有以一个好的用户界面为人们的所认识。ROM 容量太有限了，如果支持复杂的图形界面，显示接口有太多的不可预见性。

许多 HII 涉及的领域已经被操作系统的同志们研究和解析了，而且他们有价值的发现已经形成的 Framework 的支持的基准。

Framework 将配置支持分为以下 4 种类型：

键盘：全操作系统等效键盘抽象概念是超出了固件领域。另一方面，Framework 提供该支持需要建立在本地化键盘固件层面的支持。

字体：和 BIOS 不一样，EFI 视频接口本身不带有产生字符的机制。

字符串：字符串本地化方式在当今的语言环境是被很好的认识的和普遍的。通过结点关联字符串是很普遍的。也有例外（像日期，时间，当地货币）是广为人知的。

表格：随着 HTML 和随后的 XML 及其后续的普及，标记语言已经为许多人多熟悉。然而这些语言既不是明显的空间高效率的，也没有把他们本身本地化。Framework 提供内部表格表示法，这种语言能够在被转化为普通标记语言后但仍然保持其对固件环境的特性。

HII 架构定义了数据的结构，描述这些数据通过动态创建 HII 数据库和关联协议来贡献和访问这些数据。

以上这些结合起来提供一个很好的架构，这个架构既能够满足终端用户的需求同时也满足了管理的需求。

术语

国际化：图符和其他非本地特殊化的构造的使用是为了尝试创造一个所有语言都能够使用的接口。比如停止符号图符，蓝色信息的使用和鼠标的使用，都是国际化的。适合一些简单的接口（比如选择语言和时区）。

本地化：为了特定语言或者地区而使用的构造来改写接口。

标志图形：非音标书写语言，像中文，日语和韩语（“CJK”）都是使用一个字一个标志，而不是一个声音或音节一个标志。

配置模型

如图 11.1 所显示的配置，是环绕的模型。启动之初，驱动读取它们非易挥发数据来配置其硬件。也提供到 HII 数据库的数据包。如果 setup 运行，它更像浏览器访问网站一样访问数据库。然后它将生成的配置发送到驱动的 NVRAM。

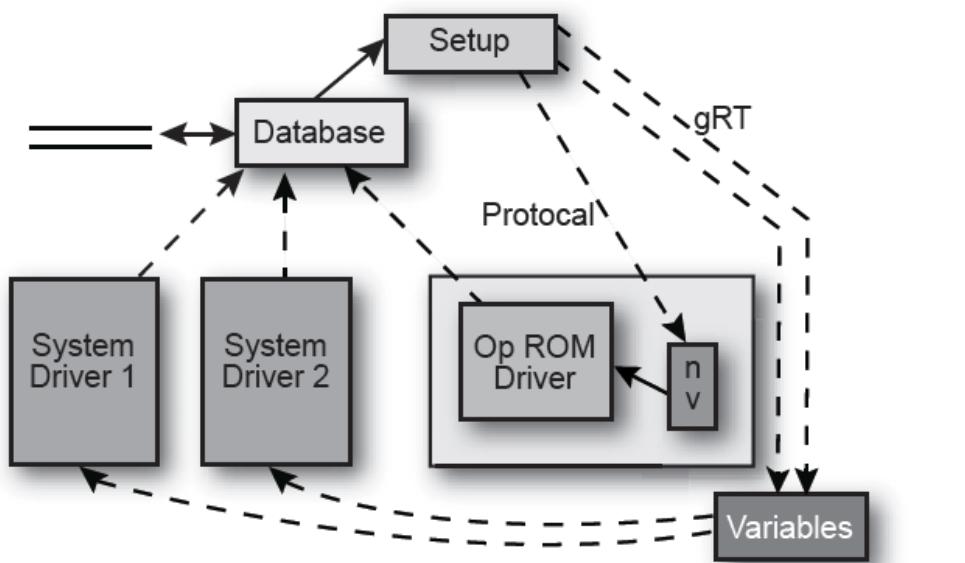


图 11.1 Configuration Flow

驱动可以自己预创建的数据，然而大多数驱动需要报告动态获取数据。HII 数据库很好地支持该要求。最后，某些数据可能如此地动态以至于每秒都会改变（最明显的是实时时钟，实际上是作为 HII 特别的例子。系统的温度是另一个例子）。在这种情况下，IFR 为了给驱动提供实时信息支持调用返回，虽然有这个功能，不鼓励到处都用。

不鼓励使用调用返回的理由很重要。这个模型允许 HII 数据库的其他用户。与远程启动应用程序交互的驱动也能够通过 HII 数据库（“提取”命令以标准的形式获得数据）发送和接收更改的配置返回数据。

如图 11.2 所描述，提取的数据能为基于操作系统的应用程序所使用，或者是通过操作系统内的浏览器程序，或者是通过为了管理而使用的脚本语言来表现。

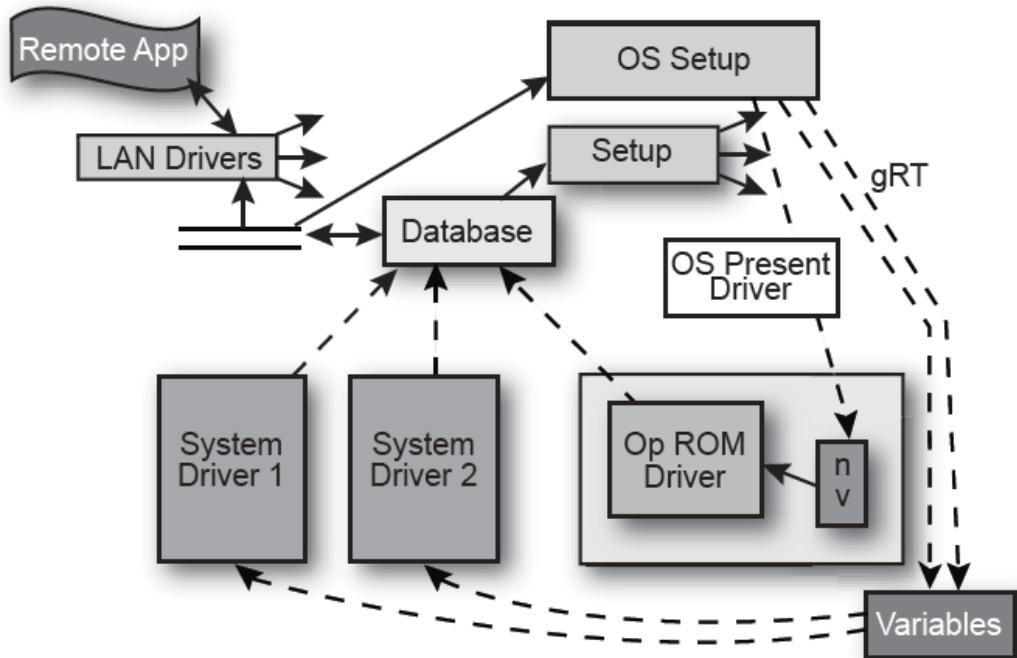


图 11.2 More Configuration Flow

循环会变得更加复杂，但不是为了这个驱动，从这种水平的支持，驱动和存储是很彻底的抽象。

关键字

Framework 在全键盘本地化的复杂性和用户友好界面的需求之间建立平衡。

问题

Framework 的默认键盘是 U.S. 英语键盘。大部分终端用户没有意识到这一点，因为大部分 setup 引擎已经非常国际化（游标键，Fn 键，Tab 键，数字键）以满足这些需求。

十六进制数字，密码，万能钥匙等的输入使国际化不可能。检查 U.S. 英语键盘和法语键盘，我们可以发现 Q, A 和 Z 键在不同的位置。传统的（“PS/2”）键盘和 USB (HID) 键盘都是根据键盘位置返回键盘输入数据而不是根据键帽的字符。这就是，PS/2 键盘的“4”或者 USB 键盘的“31”在英语键盘都表示“a”，但是在法语键盘都表示“q”。用错误的转换表输入十六进制数字几乎是不可能的。

在图形文字语言中，复杂度将会放大很多倍。例如，在日语中，Kana 字符是语音的，但是 Kanji 却不是。叫做 IMEs 的复杂工具就是用来监控为 Kanji 等效的 Kana 输入。根据和用户的约定，将它转化为 Kanji。

基本架构

Framework 解决方案取决于在假设每个人使用 U.S. 英语和 IME 支持（需要 I/O 数据流支持的先进特性）之间。

每一个键可能都有四种转化：没有转化的按键和伴随 Shift, Control, 或 Alt (Gr) 同时按

下的按键。每个按键的敲击都对应一个 Unicode 字符。每种转化都是从 U.S. 英语键盘以三角形式保存，所以拉丁-1 语言是比较小的，国际化的按键（Fn，游标键，等等）都是不转化的。

字体

协议提出在 EFI 中对显示接口支持，但是不提供字符生成功能的支持。也就是他们不支持字体。另一方面，EFI 声明它对支持拉丁-1（一般是指西欧）语言的显示支持。

EFI 和 Framework 需要公用的字符大小和表示方法的同时支持，以便调用程序能够知道什么是有用的，以便 Cyrillic, CJK, Arabic 和 Hebrew 的扩展，例如能够建立起来。

HII 是基于很简单的计算来定义标准的。在标准 CRT 上的 480 像素被 25 条线等分的最小高度大概是每个字符 19 像素。为了存储效率，8 个像素宽度表现出相当好的比率。标记图形字符是不能以 8 像素来读取的，所以加倍它们的宽度到 16 像素。8 像素宽度字符被认为是窄字符，而 16 像素的字符是宽字符。结果是字符在 800×600 的像素下都是很容易被读取的，成为 EFI 的推荐显示支持的一个必须的屏幕分辨率，提供 31 行的 100 窄字符。

数据结构为了简化分割在重叠视窗的字符就大致上以两个 8×19 字符来存储 16×19 字符。和每个字符结合，不论大小，都是 UnicodeUTF-16 高度（例如：0x0037 作为“7”）和一个标记字。标记字节主要用来表示字符是否是非空格，一个非常聪明也很狡猾的 Unicode 发明。

一个非空格字符就是以下列模式 OR'd 的位模式是有效的这是用于使某些语言(像越南)不常用的重音模式有效。这个想法是聪明的。结果却不怎么好：非空格字符会遍及 Unicode 的高度不可预见的出现。这里的解决方案是让字符阐述它自己。

这些数据结构以窄列表然后宽列表提供。每个列表必须以 Unicode 高度从最低到最高被排列，使其在固定大小（22 和 44 字节）的结构支持二进制查询。

注意每个使用非拉丁-1 字体的驱动必须提供那些字体。然而，字体库只需要保持每种字体高度和大小（窄和宽）的一个例子，因此驱动可能以使用它的或者是其他驱动一些符号（以单一高度和大小为实际代表的 HII 名字）结束。

UEFI 委员会正在考虑扩展和更新这种格式，使其对其他高度和多种字体形式提供支持。

字符串

在系统中，字体和键盘是为所有驱动所共享的。字符串和表格是对拥有它们的驱动来说是本地的。

支持本地化最普遍的方法是标记字符串索引，然后为每个片段创建转化。在 HII，结点是 16 位的，文本的片段被称为字符串。

结点将会以从 1 (1, 2, 3, 4...) 单调的增长方式被分配。结点 0 是为不存在的字符串预留的。

字符串数据结构由一个头文件，一个数组索引和字符串本身组成，空字符（Unicode 高度 0x0000）结束。

头文件以 ISO-639-2 格式（或者 3166，取决于 UEFI 的兴致）和该语言中的文本来描述这语言。（例如“Cymraeg,” 不是“Welsh,”）

数组是通过两倍的结点值来索引的，因此为每个驱动允许 65,535 字符串—在本书中比这个更多的字符串是句子。16 位的偏移允许总共超过 65,535 字符。

虽然在驱动被编译时，大部分字符串很可能是被认知的，一些字符串只有在驱动初始化时才开始是很普遍的。HII 数据库协议拥有更新和创建字符串的功能。

UEFI 工作组和他下面的小组又一次考虑扩展和更新主要是为了提高压缩率。

表格

表格是大致上考虑那些处理用户输入的标记语言子集。当购买内容，填写调查等等，互联网表格会被使用。

EFI 的目标是简化访问固件的用户接口。在 BIOS 中，用户必须得知道任何不同的热键来为系统（系统 BIOS 和可选 ROMs）不同部分的各种设置程序获得访问权。目标是拥有一种单一的机制来访问启动管理器，这能够以一种更加简单的访问使用格式的方式显示所有的配置。当然，可选 ROMs 将没有必要附带它们自己的设置引擎，这个引擎是复制系统设置引擎的基本功能。表格是寻址的一种方式。

表格是基于其他 HII 片段建立的：键盘，字体和字符串。特别地，HII 表格相对于嵌入式字符串使用字符串结点。

一般格式

表格是以内部表格表示法 (IFR) 而著名的语言的代表，这种语言最接近的模拟是复杂指令集 (CISC) 处理器的二进制语言：一个被称之为“指令”的长度可变的二进制结构的整齐顺序。另外一种语言，可视化表格表示法 (VFR) 已经被定义为相当于 IFR 的人类可读取等价物，它是通过 IFR 编译和汇编的。技术层面上，不是 Framework 的一部分。以下例子就是允许复杂部分被忽略的 VFR 事务。

每个结构的第一个字节是操作符，第二个是不包括操作符和长度的指令字节的长度。因此，最短的指令是两个字节长度，其中包含操作符和二进制字节零。指令和当中的所有区域都是字节对齐和遵循 EFI 代码指导方针 (little endian 等等)。取决于浏览器重新对齐任何有需要的区域。

字符串索引是通过 HII 字符结点表现出来的。

存储索引是通过两种方式。第一，偏移宽度形式提供基于零字节偏移和以标准 EFI 变量的字节方式的宽度，如本章前面所阐述的。变量 GUIDs 的名字是通过单独的操作符来保存空间。第二，名字的值为形成标准 HTML/XML 名字 (=值&名字=值&...) 而提供名字或者值索引。偏移宽度类型的偏移可以作为字符串结点重复使用，因为两者都是 16 位的方式来描述名字。值可能被设置成非数字内容。数字内容以十进制格式返回。

HTML 和 IFR

经历过 HTML，第一次使用 HII 表格的人可能很快上手但是会感觉有点勉强因为新的环境所限制。

IFR 支持许多中 HTML 表格的构造除了一种子集层次。例如，代替头文件的 8 种类型的只有一个。代替各种不同的枚举（在许多可能性中选择一种）输入类型，IFR 只提供一种。

HTML 提供了停止脚本语言（像 JavaScript）的出口，如果问题变得严重。IFR 必须独自解决。

支持，同时丢弃，修改和创造标记语言特性的决定，是被规划了的环境系统推动的。使用 Framework 可能也能发现它们自己。

使用字符串结点而不是嵌入式字符串（像 HTML 的），是被本地化时节约空间的需求激发的。

像头文件这样构造数量的减少是基于实际上很少使用的目标复杂表格体验。将多种标签种类转变为较少的操作符的减少和结合过程，也是为了收缩浏览器的大小。更重要的是，这个过程通过控制它们本身来使浏览器形成更多的陈述责任。这个对今后的可扩展性是至关重要的。表格也可能在不寻常的条件（比如通过脚本或者每 2 线有 16 字符的方式为前台服务）下

出现。

Form 体系

表格是由页面组成，而页面是由指令组成。

指令被分为以下几种类型：

文本类型：标题，副标题和文本。这些指令允许文本输出。

索引类型：超文本跳转允许页面跳转到其他任何页面的开始。

问题类型：这些指令实际上是请求数据和描述怎样存储。所有问题支持默认状态和边界检查的定义。问题支持提示和上下文帮助字符串结点。所有问题支持调用返回。以下几种问题类型是支持的：

复选框：类似于一个布尔值，这个指令请求一个二进制值：如果 选中是 1，不选是 0。

列表框：类似于一种枚举类型，它要求从已知的选项中选择一个单一的值。列表框是唯一的多操作符问题：一个选择和每个选项一一对应。

数字：类似于一个整型类型，他要求是数字区域（16 位限制）。该指令包含范围，步骤和默认值。

日期，时间：这些是特别情况因为它们通常在显示的时候会改变。

密码：一个原始密码的方案是默认设置的。调用返回是需要复杂代码方案的。

布尔表达式：反向波兰表达式允许问题的结果与彼此和常量进行比较。大小比较（比如小于）和布尔操作（与，或，非）都是支持的。

条件类型：三种指令类型组成了布尔表达式。第一种，灰色处理，开始包括问题在内用来告诉浏览器回收处理插入文本的区块（以最后操作符结束）。第二种，删除处理，开始一个最后结束区块，这个区块用于如果布尔表达式是真，告诉浏览器删除包括问题在内的围绕的文本。第三种，允许表格进行复杂的多问题确认，作为一种强制的检查和提供一个诊断消息如果对比失败。

例子

思考这个经典的设置顺序：传统的串口配置

OneOf SerialPort1, SerialPort1Store, SerialPort1Prompt

SerialPort1Help

Option SerialPort1Off, SerialPort1OffText, 0

Option SerialPort13F8, SerialPort3F8Text, 1

Option SerialPort12F8, SerialPort2F8Text, 0

OneOf SerialPort2, SerialPort2Store, SerialPort2Prompt

SerialPort2Help

Option SerialPort2Off, SerialPort1OffText, 0

Option SerialPort13F8, SerialPort3F8Text, 0

Option SerialPort12F8, SerialPort2F8Text, 1

InconsistentIf SerialPortConflict,

SerialPort1 == SerialPort2 and SerialPort1 != 0

可能显示如下：

Serial Port 1 Address

Off

At 3F8

At 2F8

Serial Port 2 Address

Off

3F8

2F8

混杂的 VFR 定义了两个简单枚举入口，在这里表现为单选按钮。每个列表框紧接着都是他的选项。在列表框/选项顺序中另外唯一有效指令是收回处理和取消。

例如，选项线根据选项定义等效值，文本（“At 3F8”）和标记，包含 1 作为默认值。

不符合条件线定义了警告字符串。该表达式是会被编译进 RPN。

HII 数据库接口

应该（假设）只有一个 HII 数据库协议的实例。它是被一个驱动（或者一系列支持的驱动）特别地支持。这些驱动应该是只需要内存的方式实际。协议的功能是方便以上的设计。

基本功能考虑到了 HII 数据的提交和提取。数据包提交的时候，它们被分配到在数据库中其它剩下时间所涉及的句柄。这些句柄为这类字符串结点定义了有效范围，所以字符串对一个配对（句柄，字符串）是唯一的。

该数据库已经被设计成最大可能地避免调用返回。例如，这个协议使在数据包在被提交到数据库之后再编辑字符串成为可能。要想明白这个功能怎么使用，假设一个驱动是负责一个单一的 SCSI 主控制器的配置，这个 SCSI 驱动在编译的过程中知道很多该配置信息，但是它不知道一些信息，例如连接的是什么类型的设备。SCSI 驱动能够提交一个普通的数据包，然后根据发现的接入主控制器的设备类型，更新用来描述连接外围设备的相关字符串。

字符串和脚本

假设表格参考字符串和字符串能够被不同语言本地化，那么使像编译脚本语言这些非人类语言的本地化的智力跳跃也显得不那么困难。这就需要消费者方面支持将 HII 数据库翻译成本地语法支持。或许需要支持两种语言：一种是每个标准脚本语言支持，另一种是生产组织为它自己制造和自动化测试的本地化。

别无选择，每个固件必须拥有相互独立的机制来支持每一种脚本配置语言和独立的配置数据的描述。这种方式必然会增加容量和增加在不同驱动之间试图显示相同信息而不准确的可能性。

驱动设计的注意点

HII 字符串和表格对单一驱动是本地化的。然而这并不意味每个驱动必须提供独立的 HII 包。比如，一个单一驱动（或者可能是一些）期望为主板提供 HII 数据的。这些驱动是被期望成为主板特有的，它们其中一项责任是为其他驱动分配配置数据。

强制执行这种设计有以下一些理由，特别是，让平台提供用户接口允许设计者去控制那些询问用户实际来获得答案和那些保存用其他方式获取的常量。外围驱动的设计者必须提供为了保证驱动属性而大致上需要的普遍问题的超集。然后平台驱动就能够根据需要来编辑。

和摘录问题一样，这种方法允许平台设计者在文本样式和协调保持一致性。这也有益于节约字体空间。

数据中心

启动固件获得大量的数据作为它的附属工作。各种基于操作系统的使用者对其中一些这种数据感兴趣。数据中心是一种机制，通过这种机制，建立数据的驱动对建立接口来提供给使用者没有什么约束力。数据中心依附 DXE 而存在，而且每次启动就重建，尽管某些提供给它的数据可能是非易挥发的数据。它的设计是生产者/使用者模型的经典例子。这个接口直到 ExitBootServices 一直存在。

如果你开始认为这像事件日志，那你离成功就不遥远了。世界上存在如此多的工业标准通过事件日志来定义其意义，Framework 用一个不同名字。

微弱捆绑的声明是很重要的。数据中心的目的是从它们的使用者来抽象化数据的产生者。使用这个模型，产生者没有必要知道系统需要遵循什么样的管理标准，也不需要拥有每一种需要的接口和伴随其的大小的影响。使用者也能够更加简单地编辑适合他们需要的数据。

消息顺序

不是所有生产者都是在其使用者之前运行的，这是很有可能的。如图 11.3 所示，数据中心的数据是作为数据记录（数据块是从产生者发送到数据中心本身）日志保存的。这些数据都必须以记录保留的接收顺序（从最旧的到最新的）保存，实际上是时间和单一的标签。

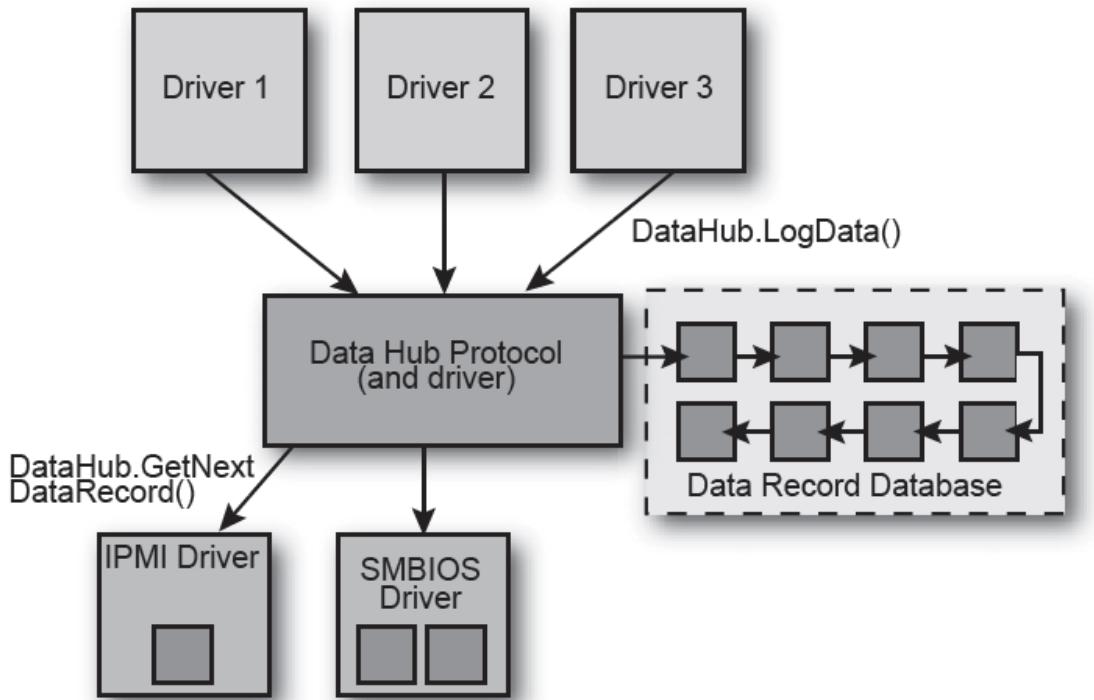


图 11.3 Data Hub

记录是通过数据中心协议从数据中心提供和获取的。当新的记录传到数据中心的时候,使用者可能将它们自己定义为过滤驱动来获得通知(和因此而生的控制)。

当使用者注册要从数据中心接收数据,它首先通过数据中心为获取所有当前保留的数据承担责任。然后它才能够接收它们产生的数据。因为定义的数据不是时序至关重要的,这就意味着使用者不能够区分其需要的数据是在注册之前产生的或者之后或者中间。

虽然不鼓励,但是为了生产驱动而提供关于一些特定内容或者事件和以后更新那些获取的更加详细的数据仍然是有效的。同样的,数据中心不编辑同类别的旧的消息,但是处理所有要求的消息。虽然不是通过数据中心执行,系统表示的进展是类似于内存 HOB 从 PEI 传递到 DXE。DXE 驱动可能更新内存使用映射(在 Global Coherency Domain)作为初始化他们周边的设备。在系统中的内存使用映像变得越来越像是启动过程的延续。

可编辑的类

一个主要的问题是数据中心数据的产生者作为通过工业标准和内部使用情况所必须的,过滤数据使之成为可管理的空间。相同数据的不同组合可能被代码支持的管理接口所使用,像 SMBIOS, IPMI, 也包括评价和制造支持的内部使用者。每个使用者将理论上只看到它关心的数据。

数据中心设计目标定位于要求产生者通过“类”来标记数据。使用类是相当普遍(比如处理器,内存和外围设备)是作为 Framework 规范的一部分定义的。Headroom 是在类定义中剩下地用来支持在定义的类之外的数据。使用者可以注册通过类接收消息和完成它本身所有的

编辑。在使用之前，基于使用的驱动多遵循的规范来提供给使用者的数据可能需要从新格式化（比如，从兆赫兹到千兆赫兹）。

处理代码

数据中心一个不怎么明显的用法是作为处理代码的抽取过程。存在更加先进的 POST Codes（或者 80 端口代码，从卡的标准地址开始）的版本。处理代码是被提供用于使用相同数据中心机制作为其他数据中心数据记录，实际上，是“类”。

通过数据中心驱动处理代码解决一些用其他办法相当棘手的问题。首先，如果不同的驱动有不同组织开发，处理代码（Port 80 卡是限制在 256 个值：00 到 FF）将怎样在驱动之间分配？诊断过程中，一个驱动可以根据需要使用许多状态代码。然而，对于产品驱动，所有驱动要把它们的处理代码发送到数据中心。然后状态驱动能够被记录，这个驱动映射这些记录到平台感兴趣的报告。第二，日志也可能作为系统表格的一部分被获取和提供，例如，通过 post-boot 分析的驱动。

第十二章

DXE 驱动和 EFI 驱动之间的区别

将酒掺起来喝也许是个错误，但新老智慧却完美地融合在一起。

—Bertolt Brecht

EFI 可以以多种方式执行，一种是通过 Framework 的基础部分 DXE 执行，这样，DXE 代表了一种特殊类型驱动，这类驱动能够和 EFI 驱动组合在一起放到指定的 Firmware Volume 里面。

有两种基本类型的 DXE 驱动。第一种是在 DEX 早期阶段执行的驱动，DXE 驱动的执行顺序取决于依赖关系表达式的值。通常，这些早期的 DXE 驱动包含了基础服务，处理器的初始化代码，芯片组和平台的初始化代码。同时这些驱动为 DXE 核心产生完整的 EFI 启动服务和运行时服务提供所需要的架构协议。为了提供尽可能最快的启动时间，很多的初始化要不同于符合在 EFI1.10 规范中描述的 EFI 驱动模式的 DXE 的驱动，大部分平台和芯片组驱动属于这种类型。值得注意的是，当这类 Drivers 执行时，并不是所有的 services 都是可用的；它们通过依赖表达式来确保所需 protocols 以及 services 可用。

第二种 DXE 驱动由那些符合 EFI 驱动模式之驱动构成。这些驱动在初始化的时候不操作任何硬件资源。相反，它们注册一个驱动绑定协议接口到句柄数据库中。BDS 阶段会使用驱动绑定协议来连接驱动到那些需要建立控制台和提供访问来启动设备的设备上。最终，遵循 EFI 模式之 DXE 驱动为控制台设备和启动设备提供一种软件抽象——当然，只有明确地要求才会提供该抽象。所有的总线驱动(包含 PCI 总线枚举)和 EFI ROMs 属于此类范畴。

除了以上这些驱动类型，DXE 驱动使用了两种 EFI 驱动完全不会用到的 service，确切地说就是 Global Coherency Domain (GCD) 和基于依赖表达式的分发器(Dependency Expression-Depex)，对 EFI 驱动来说，前者是毫无意义的，因为 EFI 的内存映射是完全实例化的；而后者，所有的必须 service 都是可用的，而一个 EFI 驱动明确地包含不存在的分发器 (Depex) ——即认为 EFI 驱动在任何一个 DXE 驱动拥有非空 Depex 时就被分发了。Depex 的缺点在于可能造成 EFI 驱动混乱，因为，EFI 驱动可能早于 Depex 存在或者 EFI 驱动没有意识到 DXE 驱动的 Framework 实现是在特定的硬件环境中。

GCD 和基于依赖的派遣的使用情况对与区分 EFI 驱动和 DXE 驱动很重要的，后面的部分将做详细描述。

GCD 服务：

GCD 服务的简述：

GCD 服务用来管理启动处理器的所能掌握的内存和 IO 资源。这些资源分成 2 种映射：

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

GCD 内存空间映射

GCD IO 空间映射

如果有内存或者 IO 资源的添加，删除，分配和释放，那么 GCD 的内存和 IO 空间映像就要更新。GCD 服务也提供了获取这两种资源的映像的服务。

GCD 服务分成 2 类，第一类管理启动处理器可见的内存资源，第 2 类是理启动处理器可见的 IO 资源。并不是所有的处理器都有 IO 资源，所以并不一定需要有对 IO 资源的管理。但是执行 DEX 的环境是需要系统内存和内存映射资源的，所以对内存资源的管理总是必须的。

GCD 中用来管理内存资源的服务有：

AddMemorySpace()

AllocateMemorySpace()

FreeMemorySpace()

RemoveMemorySpace()

SetMemorySpaceAttributes()

GCD 中用来获取 GCD 内存空间映射的服务有：

GetMemorySpaceDescriptor()

GetMemorySpaceMap()

GCD 的内存空间映射的建立来自于传递给 DXE Foundation 的入口参数 HOB list。其中一种 HOB 类型描述了用来访问内存资源的地址线数量。这个信息用来初始化 GCD 内存空间映射的状态。所有不在初始化范围（由 HOB 类型指定的地址线数量决定）内的内存区间对于任何用来管理内存资源的 GCD 服务来说，都是不可用的。GCD 内存空间映射是被设计用来描述多达 2^{64} bit 的内存空间。在 GCD 内存空间映射里面，每个内存区间都能够以单一 byte 对齐。另外一种 HOB 类型描述了系统内存的位置，IO 映射的内存位置，固件设备的位置，FV 的位置，保留区域的位置，以及事先为执行 DXE Foundation 所分配的系统内存的位置。DXE Foundation 必须解析 HOB list 上的内容，以确保预先保留给执行 DEX Foundation 的内存区间是有效的。其结果就是，GCD 内存空间必须反映 HOB list 中所描述的内存区域。GCD 内存空间映射给 DXEFoundation 初始化内存服务如 AllocatePages(), FreePages(), AllocatePool(), FreePool(), GetMemoryMap() 提供了必需的信息。关于这些内存服务的详细定义请参见 EFI1.10 的 spec。

在 GCD 内存空间映射中的一段内存区间可以是如下几种状态：

Nonexistent memory

System memory

Memory-mapped I/O

Reserved memory

这些内存区间，在 DXE 环境中可以由 DXE 驱动来分配和释放。此外，DXE 驱动 能够尝试去调整一段内存区域的 cache 属性。图 21.1 展示了 GCD 内存空间映射中内存的每个 byte 的状态转换可能。这种转换被打上 GCD 服务的标记，GCD 服务能够将一个 byte 从一种状态变为另一种状态。

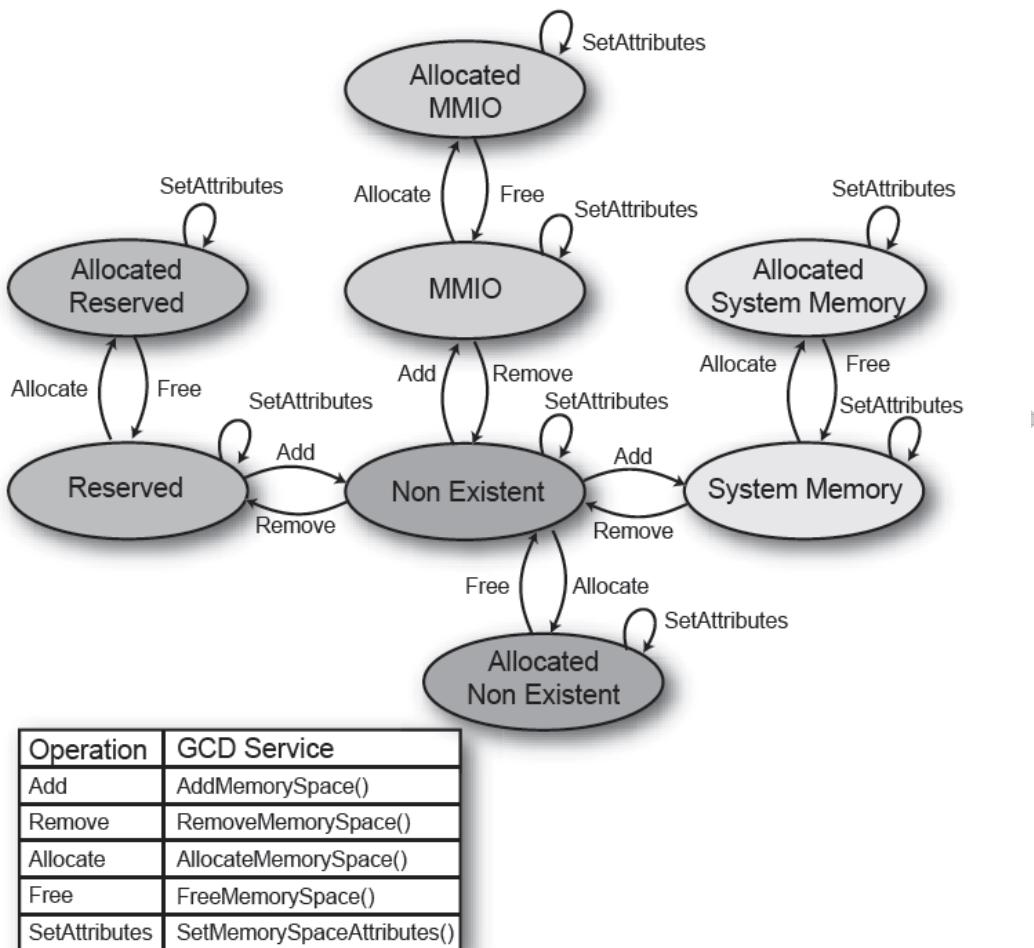


图 12.1 GCD Memory State Transitions

GCD 服务得把那些临近的相似内存区间合并，并给它们以单一的内存描述符，这样便减少了 GDC 内存空间映射中内存描述符之数量。

GCD IO 资源

用来管理 I/O 资源的 GCD 服务：

AddIoSpace()
AllocateIoSpace()
FreeIoSpace()
RemoveIoSpace()

用来获取 GCD I/O 空间映射的 GCD 服务：

GetIoSpaceDescriptor()
GetIoSpaceMap()

GCD 的 I/O 空间映射的建立来自于传递给 DXE Foundation 的入口参数 HOB list。其中一种 HOB 类型描述了用来访问 I/O 资源的地址线数量。这个信息用来初始化 GCD I/O 空间映射的状态。所有不在初始化范围（由 HOB 类型指定的地址线数量决定）内的 I/O 区间对于任何用来管理 I/O 资源的 GCD 服务来说，都是不可用的。GCD I/O 空间映射是被设计用来描述多

达 2^{64} bit 的 I/O 地址空间。在 GCD I/O 空间映射里面，每个区间都能够以单一 byte 对齐。

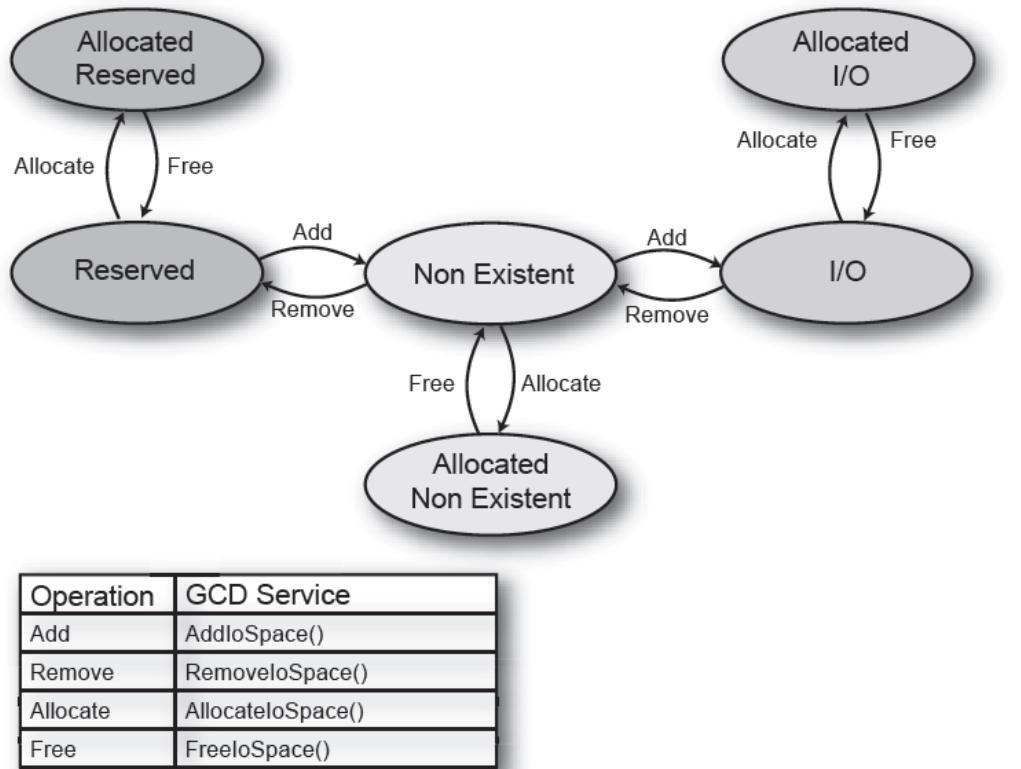


图 12.2 GCD I/O State Transitions

一个由 GDC I/O 空间映射描述的 I/O 区间有这样几种不同的状态：不存在的 I/O，I/O 和保留的 I/O。这些 I/O 区间，在 DXE 环境中可以由 DXE 驱动来分配和释放。图 12.2 展示了 GCD I/O 空间映射中 I/O 的每个 byte 的状态转换可能。这种转换被打上 GCD 服务的标记，GCD 服务能够将一个 byte 从一种状态变为另一种状态。GCD 服务得把那些临近的相似 I/O 区间合并，并给它们以单一的 I/O 描述符，这样便减少了 GDC I/O 空间映射中 I/O 描述符之数量。

表 12.1 Global Coherency Domain Services

Name	Type	Description
<u>AddMemorySpace</u>	Boot	Adds reserved memory, system memory, or memory-mapped I/O resources to the global coherency domain of the processor.
<u>AllocateMemorySpace</u>	Boot	Allocates nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
<u>FreeMemorySpace</u>	Boot	Frees nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
<u>RemoveMemorySpace</u>	Boot	Removes reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
<u>GetMemorySpaceDescriptor</u>	Boot	Retrieves the descriptor for a memory region containing a specified address.
<u>SetMemorySpaceAttributes</u>	Boot	Modifies the attributes for a memory region in the global coherency domain of the processor.
<u>GetMemorySpaceMap</u>	Boot	Returns a map of the memory resources in the global coherency domain of the processor.
<u>AddIoSpace</u>	Boot	Adds reserved I/O, or I/O resources to the global coherency domain of the processor.
<u>AllocateIoSpace</u>	Boot	Allocates nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.
<u>FreeIoSpace</u>	Boot	Frees nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.
<u>RemoveIoSpace</u>	Boot	Removes reserved I/O, or I/O resources from the global coherency domain of the processor.
<u>GetIoSpaceDescriptor</u>	Boot	Retrieves the descriptor for an I/O region containing a specified address.
<u>GetIoSpaceMap</u>	Boot	Returns a map of the I/O resources in the global coherency domain of the processor.

在某一平台上，那些组成 GCD 服务的函数在 preboot 阶段被调用，以达到添加、删除、分配、释放和提供系统内存映射、I/O 内存映射以及 I/O 资源。这些服务和内存分配服务一起，赋予了平台管理所有内存和 I/O 资源的能力。表 12.1 列出了 GCD 服务。

Dispatcher 服务

表 12.2 Dispatcher Services

Name	Type	Description
<u>Dispatch</u>	Boot	Loads and executes DXE drivers from firmware volumes.
<u>Schedule</u>	Boot	Clears the Schedule on Request (SOR) flag for a component that is stored in a firmware volume.
<u>Trust</u>	Boot	Changes the state of a file stored in a firmware volume from the untrusted state to the trusted state.
<u>ProcessFirmwareVolume</u>	Boot	Creates a firmware volume handle for a firmware volume that is present in system memory.

组成 Dispatcher 服务的函数在 preboot 阶段被调用，以调度驱动之执行。在驱动依赖关系表达式中，驱动可以有选择的将 SQR(Schedule On Request) flag 置位。被置位之驱动不

会被加载直到显示的请求到来。从 FV(Firmware Volumes)中加载的文件可能被安全架构协议(Security Architectural Protocol)置为 untrusted 之状态。Dispatcher 服务有能力在 DXE 驱动的依赖关系表达式里清除 SOR Flag 以及将从 FV 中加载的文件由 untrusted 状态提升至 trusted 状态。表 12.2 列出了 Dispatcher 服务。

依赖关系表达式之逆波兰表达式(RPN)

实际的等式由 DXE 驱动以一个易于计算的形式——即，后缀——来表示。下面的一段语法符合 BNF(巴科斯范式)编码。更多详情请参加依赖关系表达指令集。

```
<statement> ::= SOR<expression> END | BEFORE<guid> END | AFTER<guid> END |  
<expression> END  
<expression> ::= PUSH<guid> | TRUE | FALSE |  
<expression> NOT |  
<expression> <expression> OR |  
<expression> <expression> AND
```

DXE Dispatcher 状态机

DXE Dispatcher 的职责在于，从 DXE 驱动在 FV 中被发现一直到 ExitBootServices()被调用来终止 DXE Foundation 这段时间内，追踪 DXE 驱动的状态。在这段时间内，每个 DXE 驱动都可能有数种不同状态。在图 12.3 中展示了用来追踪 DXE 驱动的 DXE Dispatcher 状态机。

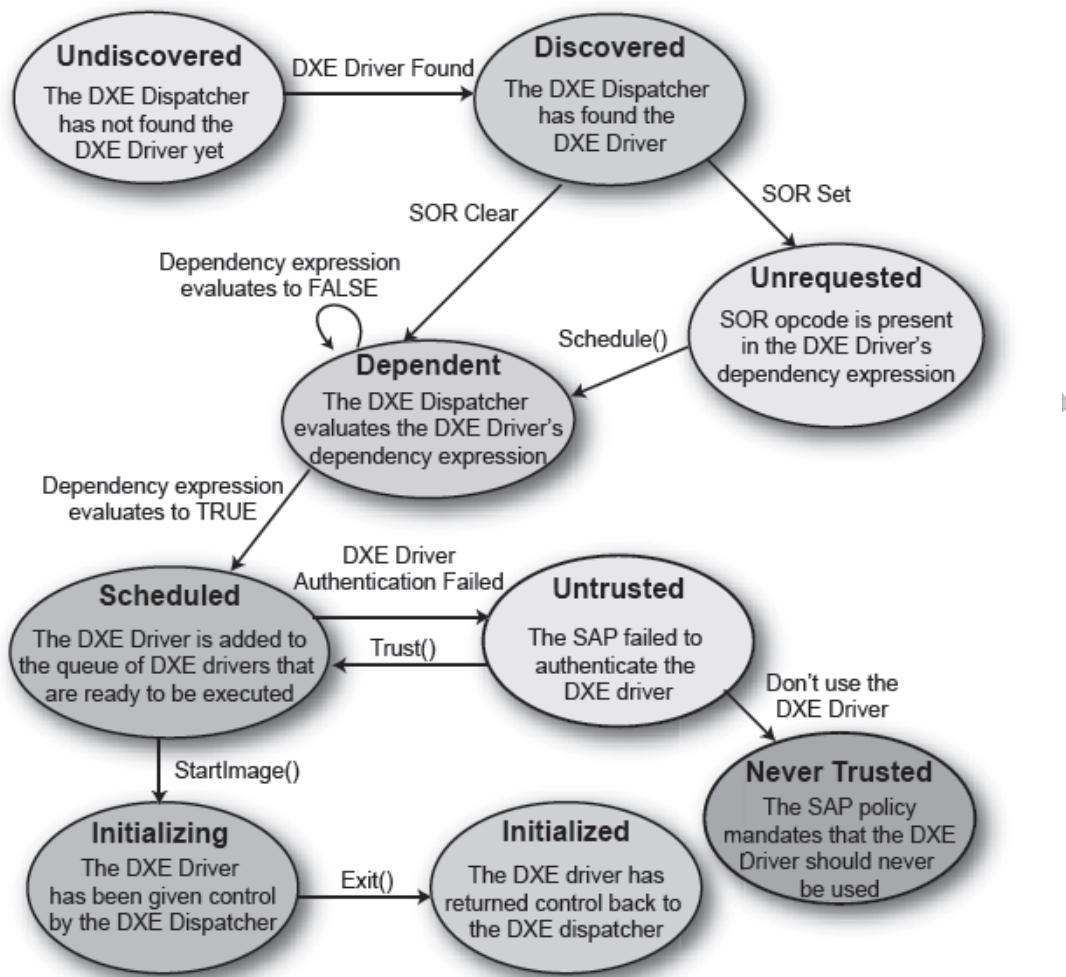


图 12.3 DXE Driver States

一个 DXE 驱动由 Undiscovered 状态开始，也就是说，DXE 驱动存在于 FV 中，而 DXE Dispatcher 尚未知其存在。当 DXE Dispatcher 发现了一个新 FV，任何在这 FV 的 priori 文件中列出的 DXE 驱动会被立即加载和执行。在 priori 文件中列出的 DXE 驱动立即被提升至 Scheduled 状态。接下来，DXE Dispatcher 会搜索 FV，找出那些未列出在 priori 文件中的 DXE 驱动，任何被找到的驱动，其状态将从 Undiscovered 提升至 Discovered。每个 DXE 驱动的依赖关系表达式都将被计算。如果在依赖关系表达式中，SOR 被置位，那么这个 DXE 驱动被置为 Unrequested 状态；如果 SOR 未被置位，那么这个 DXE 驱动被置为 Dependent 状态。一旦某个 DXE 驱动处于 Unrequested 状态，它只能通过调用 DXE 服务的 Schedule() 函数来将状态提升至 Dependent。

一旦某个 DXE 驱动处于 Dependent 状态，DXE Dispatcher 就会计算该驱动的依赖关系表达式。如果该驱动未包含依赖关系表达式，那么所有架构协议的依赖关系表达式相与在一起就假定为该驱动的依赖关系表达式。如果依赖关系表达式的值为 FALSE，那么该驱动就停留在 Dependent 状态；如果依赖关系表达的值永远不会为 TRUE，那么该驱动将永远不会脱离 Dependent 状态。而一旦依赖关系表达式的值为 TRUE，那么该驱动将被提升至 Scheduled 状态。

一个被提升至 Scheduled 的 DXE 驱动将被添加到一个由其它处于 Scheduled 状态之驱动组成的队列末尾。当这个 DXE 驱动到达了队首，DXE Dispatcher 必须使用 SAP(Security Authentication Protocol, 安全认证协议)服务来检查该驱动的认证状态。如果 SAP 认为该驱动违背了具体平台之安全策略，那么该驱动将被置于 Untrusted 状态。并且，SAP 能够告诉 DXE Dispatcher 该驱动应该被置于 Never Trusted 状态，永远不被执行。如果一个 DXE 驱动被置于 Untrusted 状态，那么它只能通过调用 DXE 服务的 Trust() 函数将状态提升至 Scheduled。

一旦某个 DXE 驱动到达了调度队列首部，并且该驱动通过了 SAP 的认证检测，那么该驱动就被启动服务的 LoadImage() 函数装载进内存。届时，控制权将经由 DXE Dispatcher 通过启动服务之 StartImage() 函数传递给该驱动。当 StartImage() 被调用了，该驱动的状态就被提升至 Initializing。通过启动服务的 Exit() 函数，驱动将控制权返还给 DXE Dispatcher。当一个 DXE 驱动已经将控制权返还给 DXE Dispatcher 了，该驱动就处于最终状态，称之为 Initialized。

DXE Dispatcher 有责任清空由处于 Scheduled 状态的 DXE 驱动组成的队列。一旦队列为空，DXE Dispatcher 必须评估所有处于 Dependent 状态的 DXE 驱动，以确定它们中是否存在需要被提升至 Scheduled 状态的驱动。当一个或多个 DXE 驱动被提升至 Initialized 状态，这一评估动作都需要执行，因为这些驱动可能产生了那些处于 Dependent 状态的驱动一直等待的协议接口。

排序例子

由 DXE Dispatcher 装载和执行的 DXE 驱动次序是一个优先级高低混合的排序情况。优先级高的是由 priori 文件指定的；优先级低的是由驱动的依赖关系表达式指定的。图 12.4 展示了一个样例 FV 的组成，包含有如下内容：

DXE Foundation 映像

DXE 驱动映像

一个 Priori 文件

这些映像出现在 FV 中的次序是随机的。DXE Foundation 和 DXE Dispatcher 不可对 FV 中文件位置作任何假设。首先出现在 FV 中的是一个包含所有将被装载和执行的 DXE 驱动的 GUID 文件名的 priori 文件。在 FV 中，各个 DXE 驱动引出的依赖关系表达式和协议都紧挨着各自 DXE 驱动映像。

基于图 12.4 中 FV 的组成内容，Security 驱动、Runtime 驱动和 Variable 驱动总是被先执行。这是一个由于 priori 文件而优先处理的例子。而后，DXE Dispatcher 会评估剩下的 DXE 驱动之依赖关系表达式的值，以确定它们将要被执行的顺序。基于每个 DXE 驱动产生的依赖关系表达式和协议，将会有 30 个有效的处理顺序供 DXE Dispatcher 选择。因为 BDS 驱动和 CPU 驱动的依赖关系表达式的值为 TRUE，它们并列成为接下来被调度的驱动。依赖关系表达式的值为 TRUE 意味着该驱动不需要任何其它协议接口条件就能够被执行。DXE Dispatcher 选择任意一个依赖关系表达式值为 TRUE 的驱动先被调用。Timer 驱动、Metronome 驱动和 Reset 驱动都依赖于 CPU 驱动产出的协议条件。一旦 CPU 驱动被装载和执行，Timer 驱动、Metronome 驱动和 Reset 驱动能够以任意次序被调度。一个合理的 DXE Dispatcher 实现，总是一如既往的为既定系统配置产出同样的驱动调用序列。任何情况下，一旦系统配置被改变(包括存储在 FV 中文件顺序变动)，那么一种新的调度顺序将会产生，但是，这一新的调度顺序在系统配置再次改变前都是固定的。

Firmware Volume

A Priori File	
Security Driver	
Runtime Driver	
Variable Driver	
Runtime Driver	Depex = TRUE END Produces: EFI_RUNTIME_ARCH_PROTOCOL
CPU Driver	Depex = TRUE END Produces: EFI_CPU_IO_PROTOCOL, EFI_CPU_ARCH_PROTOCOL
Timer Driver	Depex = EFI_CPU_IO_PROTOCOL AND EFI_CPU_ARCH_PROTOCOL END Produces: EFI_TIMER_ARCH_PROTOCOL
Metronome Driver	Depex = EFI_CPU_IO_PROTOCOL END Produces: EFI_METRONOME_ARCH_PROTOCOL
Variable Driver	Depex = TRUE END Produces: EFI_VARIABLE_ARCH_PROTOCOL, EFI_VARIABLE_WRITE_ARCH_PROTOCOL
Reset Driver	Depex = EFI_CPU_IO_PROTOCOL END Produces: EFI_RESET_ARCH_PROTOCOL
DXE Foundation	
BDS Driver	Depex = TRUE END Produces: EFI_BDS_ARCH_PROTOCOL
Security Driver	Depex = TRUE END Produces: EFI_SECURITY_ARCH_PROTOCOL

图 12.4 Sample Firmware Volume

第十三章

启动设备选择

我只是发明，然后等着有人来这里找我创造的东西。

—R. Buckminster Fuller

随着时间的推移，EFI 逐渐演化成了一种用来建立 firmware 决策引擎的基本模型。这种设计的理念早在那种建立在单一引导管理器的 Framework (EFI 1.1 甚至更早) 的早期都已经发展起来了，而后的唯一目的仅仅是为了执行一些结构上已定义好的全局变量 NVRAM 所确立的策略。随着这种架构体系的发展，诸如 SEC, PEI, DXE, BDS, Runtime 和 Afterlife 等这些启动阶段都已经定义好了，其中，BDS(Boot Device Selection)阶段却变成了一种独特的类似于 Boot Manager 的阶段。本章中，将重点讲述 Boot Manager 组件，那些控制策略内容为最终形成的 BDS 阶段打下了一个大体的架构基础。

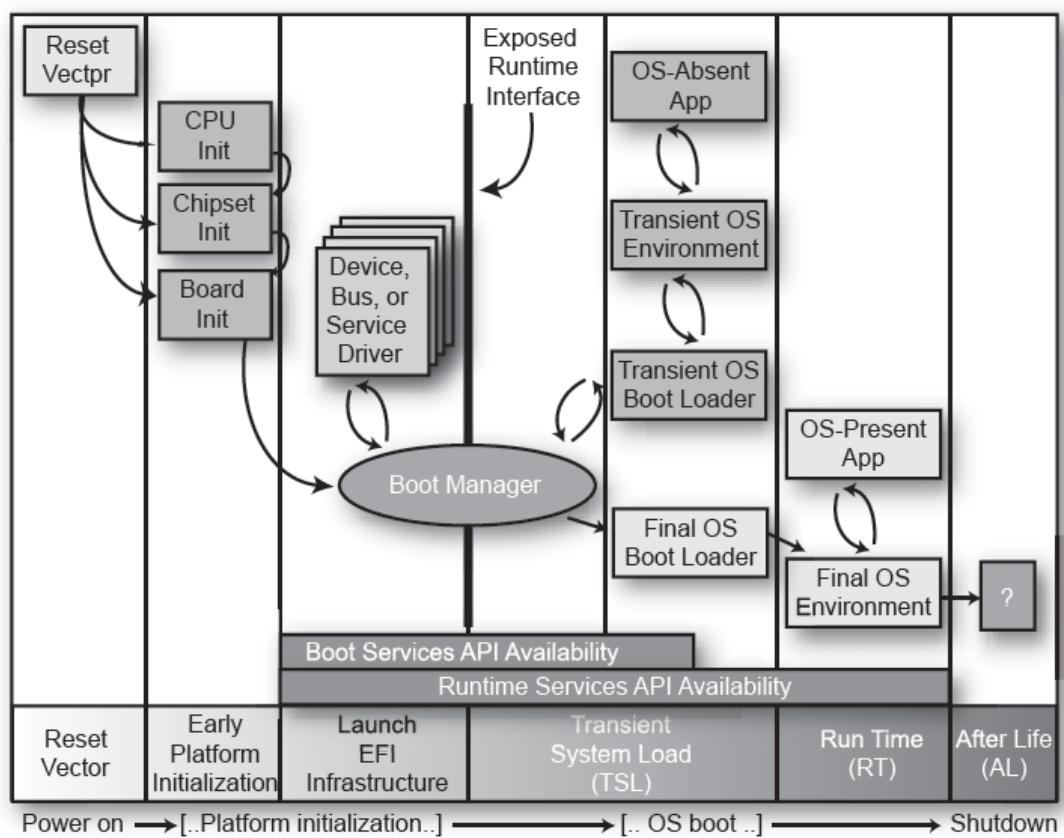


图 13.1 EFI with Boot Manager Component

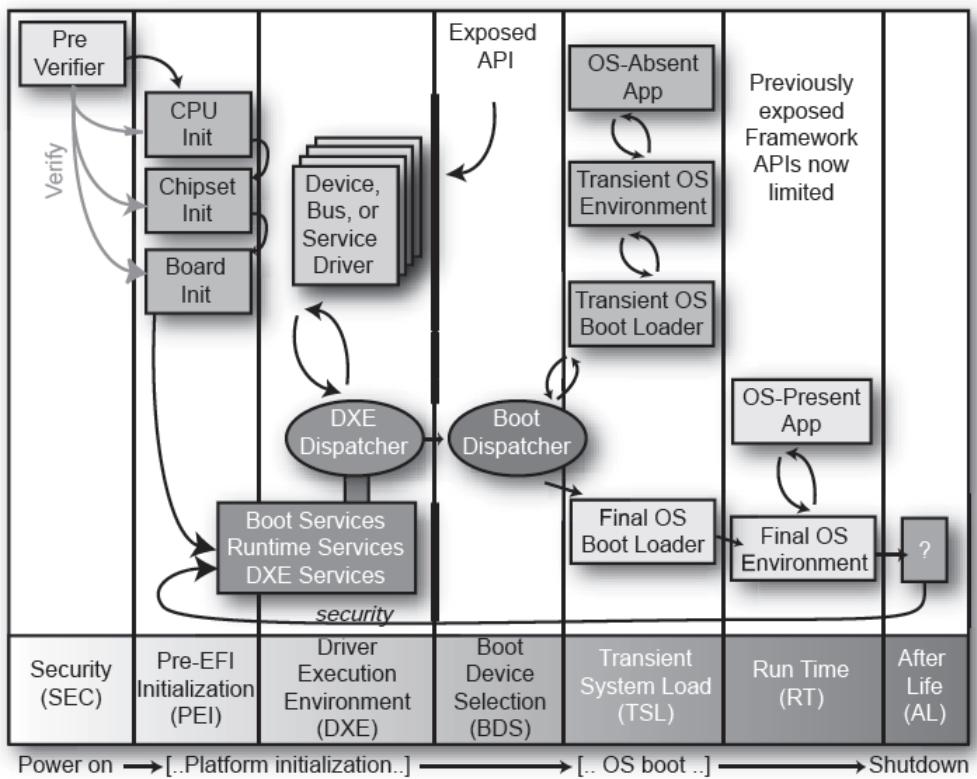


图 13.2 Framework with BDS Component

实际上,关于目前大家所知的 Pre-Framework 方案的 Boot Manager 和 Framework 方案的 BDS 之间的不同之处, 是非常容易阐述的。图 13.1 是一个 EFI1.1 兼容(pre-Framework)环境下的流程图, 而图 13.2 是一个 Framework 兼容环境下的流程图。

通过比较这两张图, 可以发现它们有许多相似之处, 而本章中所描述的和第九章深入阐释的就是这种体现, 它们同样被包含于 BDS 阶段中。

EFI boot manager 是一个可以通过修改已定义的全局变量 NVRAM 来进行配置的 Firmware 决策引擎。它试图通过 NVRAM 所拟定好的顺序来加载 EFI 驱动和 EFI 应用程序(包括 EFI OS boot loaders), 所以为能够正常启动, Platform Firmware 必须遵循这种启动顺序。另外, Platform Firmware 也需要在启动顺序列表中增加一些引导选项或删除一些无效引导选项。

当然,如果在 Firmware 启动过程中发生了一些异常状况,Platform Firmware 也需要在 boot manager 中加入一些附加特性。举个例子, 当第一次加载某个 EFI 驱动时启动失败, 那么这种附加特性就会记录这种状态, 并且以后将不会加载该驱动。

另一个例子是启动过程中出现严重错误时, 将会进入到 OEM 定制的诊断环境中。

EFI 启动顺序描述如下:

Platform firmware 从 NVRAM 中读取启动顺序列表, 该表定义了一系列有关引导信息的 NVRAM 变量。而每个变量定义了一个 Unicode 名字, 它就是用户所看到的启动选项。

这些变量也包含指向硬件设备的指针, 以及指向硬件设备所要加载的 EFI Image 的文件指针。

这些变量可能也包含操作系统所在分区的路径, 目录以及其他特定配置的目录。

当然, NVRAM 也包含直接传递给 EFI image 的加载选项。但是, Platform Firmware 却并不清楚加载选项里有什么内容。在设置 Platform Firmware 的引导决策时, 只有优先级最

高的软件才能修改 NVRAM 变量去设定这些选项。当 OS Kernel 的位置与 EFI OS Loader 的位置不同时，通过这种方式就可以很好的界定 OS Kernel 的位置。

Firmware Boot Manager

Boot manager 是 EFI firmware 的一个组件，它用于决定在何时，哪个 EFI 驱动或 EFI 应用程序应该被加载。一旦 EFI Firmware 初始化好了，Boot Manager 就取得了控制权。然后它就决定该加载什么，以及做出这样的行为所需要的与用户的交互。实际上，此类行为更多的是由 Firmware 开发者来决定的，至于具体的操作细节并不在本规范考虑范围之内。有可能的实现包括以下这些，有关启动的任何输入输出接口，集成的启动选择管理器，可能的其他内部应用程序，或者通过 boot manager 集成在系统里的恢复驱动。

与 Boot Manager 的可编程式交互是通过全局变量来实现的，在初始化时，Boot Manager 就可以从这些 EFI 环境变量当中读取包含所有已列举的加载选项的值，并且通过调用 SetVariable() 函数也可以对含有这些环境变量的数据进行修改。

每一个加载选项入口都驻在一个 Boot#### 变量或 Driver#### 变量，其中#### 是一个唯一的选项号，用数字 0-9，大写字符 A-F 形成的可打印的 16 进制表示。#### 必须始终是 4 个数学符号，因此，最小的数字必须是 0 开头的。接着，加载选项就可以被由需要的顺序编码的一系列选项号有序的逻辑推算。有 2 个这样的有序选项列表。第一个是 DriverOrder，把 Driver#### 加载选项有序排成它们的加载顺序。第二个是 BootOrder，把 Boot#### 加载选项变量有序排成它们的加载顺序。

例如，添加一个新的启动选项，一个新的 Boot#### 变量将被添加。接着，新的 Boot#### 的选项号将被添加到已经排好的 BootOrder 表中，并且 BootOrder 变量会被重写。要改变一个已经存在的启动选项，只需要重写这个 Boot####。与此类似，可以完成添加，删除，修改驱动加载表。

如果通过 Boot#### 启动，返回一个 EFI_SUCCESS 状态，boot manager 就停止操作 BootOrder 表变量，展示一个 boot manager menu 给用户。如果通过 Boot#### 启动，返回一个有别于 EFI_SUCCESS 的状态，启动就失败，而且，下一个在 BootOrder 表的变量 Boot#### 会被测试，直到所有的可能都试完。

Boot manager 可能会自动维护变量数据库。例如，它可能删除没有用到的加载选项的变量，任何不可解析或不可加载的加载选项，以及重写没有响应的加载选项变量。另外，boot manager 可能在它需要时为了放置它自己的加载选项来自动更新顺序表。Boot manager 也可以，根据它自己的平台特殊的行为，提供手动的维护操作。例如包括在所有加载选项里选择顺序，激活或禁用加载选项，等等。

Boot manager 被要求在启动加载选项入口前，先执行驱动加载选项。Boot manager 也要求初始化一个启动选项，而这个启动选项是要特别把 BootNext 变量在下次启动并且只在下次启动时作为第一启动选项。如果 BootNex 选项启动失败，启动流程继续使用 BootOrder 变量。如果 BootNext 选项返回 EFI_SECCCESS 表明启动成功，boot manager 将不会再继续使用 BootOrder 变量。

Boot manager 必须调用 LoadImage()，该函数为了处理加载项，至少要支持 SMIPLE_FILE_PROTOCOL 和 LOAD_FILE_PROTOCOL。如果 LoadImage() 成功，在调用 StartImage() 前，Boot Manager 必须先调用 SetWatchdogTimer() 服务激活 Watchdog Timer 5 分钟。当启动项将控制权交给 Boot Manager 时，boot manager 也必须还要调用

SetWatchdogTimer()服务来禁用 watchdog timer。

若函数 LoadImage()没有加载 Boot Image 失败, Boot Manager 就会搜寻默认的应用程序来启动, 一般这种应用程序的类型是属于可插拔或固定媒介的。当启动项的引导 Image 设备路径直接指向 SIMPLE_FILE_SYSTEM 设备而又没有指定具体的文件去加载时, 就会发生这种情况。本章稍后的“Default Behavior for Boot Option Variables”部分将会详细的介绍这种文件搜寻方式。对于目标设备路径而言, 除了 SIMPLE_FILE_SYSTEM 外, 缺省媒介的引导事件也可以由 LOAD_FILE_PROTOCOL 来处理, 至于 Boot Manger 就没必要去做这件事了。

Boot Manager 还必须能够支持从短格式的设备路径去引导, 这种路径的开头部分是一个硬盘驱动媒介设备路径。另外, Boot Manager 也需要使用硬盘驱动设备路径上的 GUID 或标签及分区号对应到系统的设备上去。如果驱动支持 GPT 分区格式, 那么硬盘驱动媒介设备路径的 GUID 就会与 GUID 分区入口中的 UniquePartitionGuid 区域相匹配; 如果驱动支持 MBR 格式, 那么硬盘驱动媒介设备路径的标签就会与 Legacy Master Boot Record 的 UniqueMBRSignature 区域相匹配。要是标签匹配了, 分区号也必须相匹配。当然, 硬盘驱动设备路径也可以被添加到对应的硬件设备路径中, 在正常启动情况下就可以使用。一旦有多个设备匹配同一驱动设备路径, Boot Manager 将会从中随机选择一个。因此, 操作系统必须确保硬盘驱动标签的唯一性, 才能保证预期的启动行为。

每个加载项变量都包含了一个 EFI_LOAD_OPTION 描述符, 该描述符可以看成是长度可变的字节封装区域。由于此类区域的某些部分的长度是可变的, 致使 EFI_LOAD_OPTION 不能用标准的 C 语言数据结构来表示。因此, 该描述符的定义有一定的要求, 具体定义如下:

```
UINT32 Attributes;  
UINT16 FilePathListLength;  
CHAR16 Description[];  
EFI_DEVICE_PATH FilePathList[];  
UINT8 OptionalData[];
```

Attributes - 加载项入口属性。根据 EFI Spec 的要求, 未被使用的位必须为 0, 这样做是为了将来可能会用到它们, 具体请参考“Related Definitions”。

FilePathListLength - FilePathList 的字节长度。OptionalData 从描述符 EFI_LOAD_OPTION 的 sizeof(UINT32) + sizeof(UINT16) + StrSize(Description) + FilePathListLength 处开始的。

Description - 对于用户而言, 它是可读的描述符, 且以一个 Null 的 Unicode 字符为结束标志。

FilePathList - EFI 设备路径的封装数组。该数组的第一个元素是 EFI Device Path, 它是用来描述加载项的设备和位置的。FilePathList[0]指的是设备类型, 而其他的 Device Paths 可能存在于此 FilePathList 中, 但它们仅限于用于 OSV。此数组中每个成员的长度都是可变的, 且都是以 Device Path End Structure 结束的。由于描述符 Description 的长度是可变的, 所以不能将这种数据结构体对齐到 Natural Boundary, 不过在使用它之前把它拷贝到已对齐的 natural boundary 处是可以的。

OptionalData - 加载项描述符 Description 的剩余部分就是一个二进制数据 Buffer, 是用来传给已加载的 Image 的。如果该 Buffer 长度为 0, 那么就会传递 Null Pointer。OptionalData 的字节数可以用 EFI_LOAD_OPTION 的总字节数减去 OptionalData 的起始位置偏移来得到。

Related Definitions

The load option attributes are defined by the values below.

```
//  
// Attributes  
  
#define LOAD_OPTION_ACTIVE 0x00000001  
#define LOAD_OPTION_FORCE_RECONNECT 0x00000002
```

调用SetVariable()可创建一个加载选项。加载选项的大小和SetVariable()创建variable的参数DataSize的大小一样。当创建一个新的加载选项时，所有未定义的236个属性bit的值必须为0。当更新一个加载选项，所有未定义的属性bit必须保留。如果一个加载选项没有被标记为LOAD_OPTION_ACTIVE，bootmanager就不会自动加载这个选项。这就提供了一种简便的关闭或打开加载选项的方法，而不必去删除或重新添加它们。如果一个Driver####加载选项被标记为LOAD_OPTION_FORCE_RECONNECT，则所有的系统中的EFI驱动设备都将在最后一个加载选项执行完后，断开连接，再重新连接。这就允许以Driver####加载的EFI驱动设备，去覆盖之前加载过的一个EFI驱动设备，以执行EFIbootmanager。

全局定义的 Variables

这一节定义一组有架构性明确定义意义的variables。除已经定义的数据内容外，每一个这样的variables都有一个架构性明确定义的属性，来指明何时这个数据variable可以被访问到。这些variables有一个属性是NV，即nonvolatile非易失性。它的意思是说这些值在reset和power cycles都会保留。任何情况下，没有这个属性的variable的值都会在系统断电后消失，并且保存在内存中的firmware的状态也不保留。Variables的属性BS只在ExitBootServices()被调用之前有效。它的意思是这些环境Variable只能在preboot 环境下被重建或修改。它们对操作系统不可见。环境Variable的RT属性表示在ExitBootServices()被调用前或被调用后，它们都有效。这种环境Variable既可以在preboot环境下，也可以在操作系统中被重建或修改。所有架构性定义的variable都使用EFI_GLOBAL_VARIABLE VendorGuid:

```
#define EFI_GLOBAL_VARIABLE \  
{8BE4DF61-93CA-11d2-AA0D-00E098032B8C}
```

为避免和未来会定义的全局variable名称冲突，其他未在此定义的内部firmware数据variable，必须用不同于EFI_GLOBAL_VARIABLE的唯一的一个VendorGuid来保存。表13.1列出这些全局variables。

表 13.1 Global Variables

Variable Name	Attribute	Description
LangCodes	BS, RT	The language codes that the firmware supports.
Lang	NV, BS, RT	The language code that the system is configured for.
Timeout	NV, BS, RT	The firmware's boot managers timeout, in seconds, before initiating the default boot selection.
ConIn	NV, BS, RT	The device path of the default input console.
ConOut	NV, BS, RT	The device path of the default output console.
ErrOut	NV, BS, RT	The device path of the default error output device.
ConInDev	BS, RT	The device path of all possible console input devices.
ConOutDev	BS, RT	The device path of all possible console output devices.
ErrOutDev	BS, RT	The device path of all possible error output devices.
Boot####	NV, BS, RT	A boot load option, where ##### is a printed hex value. No 0x or h is included in the hex value.
BootOrder	NV, BS, RT	The ordered boot option load list.
BootNext	NV, BS, RT	The boot option for the next boot only.
BootCurrent	BS, RT	The boot option that was selected for the current boot.
Driver####	NV, BS, RT	A driver load option, where ##### is a printed hex value.
DriverOrder	NV, BS, RT	The ordered driver load option list.

LangCodes Variable 包含一组由 3 个字符（8 位的 ASCII 字符）组成的 Firmware 能支持的 ISO-639-2 Language Codes。在初始化时，Firmware 解译所支持的语言，创建一个 Data Variable，由于 Firmware 在每次初始化时都会产生一个只读的数值，所以就不需要在非易失性内存中储存它的值。

Lang Variable 由 3 个字符（8 位的 ASCII 字符）的 ISO-639-2 Language Codes 所组成，其值可以转换成 LangCodes 所能支持的任何值。尽管如此，这种转换只有在下一次启动时才生效。如果 Language Codes 被设成不支持的值，那么 Firmware 就会选择一个支持的初始默认值，并将 Lang 设成支持的值。

Timeout variable 是一个 UINT16 的二进制数据，它提供在初始化最早的默认启动选择时，firmware 暂停的秒数。值 0 表示默认的启动选择将在启动时立即被初始化。如果值不存在，或者值是 0xFFFF，则 firmware 将会在启动前等待用户输入。这表明 firmware 将不会自动执行开始默认的启动选择。

ConIn, ConOut, 以及 ErrOut variables，每个 variable 都含有一个 EFI_DEVICE_PATH 描述符，这个描述符定义了默认在启动时使用的设备。修改这些值，改动在下次启动才能生效。如果 firmw 不能解析设备路径，允许自动把这些值替换成需要的值，用来给系统提供一个控制台。

ConInDev, ConOutDev, 以及 ErrOutDev variables，每个 variable 都含有一个 EFI_DEVICE_PATH 描述符，这个描述符定义了所有可能的默认在启动时使用的设备。这些 variable 是易失性的，并在每次启动被动态设置。通常，ConIn、ConOut 和 ErrOut 是 ConInDev、ConOutDev 和 ErrOutDev 的真子集。

每个 Boot#### variable 含有一个 EFI_LOAD_OPTION。每个 Boot#### variable 都是由名称”Boot”，再加上一个唯一的，4 个 16 进制符号的数。例如：Boot0001, Boot0002, Boot0A02,

等等。

BootOrder Variable 是一个 UINT16 数组，它构成 Boot ##### 项的次序列表。数组的第一个元素是第一个逻辑启动项，第二个元素是第二逻辑启动项，依次类推下去。Boot Manager 一般是将 BootOrder 次序列表作为默认启动顺序的。

BootNext Variable 是一个 UINT16 数，它定义了 Boot##### 项的下次启动引导的首选项。只有 BootNext 启动项处理完，才能正常的使用 BootOrder 列表。为避免重复，Boot Manager 在将控制权交给预启动项前需要删除此 Variable。

BootCurrent Variable 也是一个 UINT16 数，它定义了当前启动时所使用的 Boot##### 项。

Driver##### Variable 含有一个 EFI_LOAD_OPTION，每个 Variable 的名字都是由“Driver”加上特定的数字组成的，比如 Driver0001, Driver0002, 等等。

DriverOrder Variable 由一个 UINT16 数组组成，该数组构成了 Driver##### Variable 的次序列表。数组的第一个元素是第一逻辑驱动加载项， 第二个元素是第二逻辑驱动加载项，依次类推。对 EFI Drivers 而言，Boot Manager 会把 DriverOrder 列表当做默认加载次序依序的加载。

Boot Option Variables 的默认行为

全局 Variables 的默认状态值是 Firmware 厂商自己设定的。尽管如此，在发生一些异常情况譬如 Platform 上没有有效的启动项时，就需要一个默认的标准行为判据。这种行为判据在 BootOrder Variable 不存在或指向一个并不存在的启动项的任何时候就会被调用。

当不存在有效的启动项时，Boot Manager 在枚举完所有的固定 EFI 媒介设备后枚举所有的可插拔的 EFI 媒介设备。由于每组的顺序是不确定的，所以这些新的默认启动项并不会存储到非易失性存储器里。然后 Boot Manager 试图从每个启动项启动。如果设备支持 SIMPLE_FILE_SYSTEM Protocol，就会执行可插拔的媒介启动行为。否则，Firmware 就会试图通过 LOAD_FILE Protocol 来引导设备。

通常，这种默认启动会加载操作系统或维护程序。当加载的是操作系统安装程序，在随后的启动中，Platform Firmware 需要设置一些必要的 Environment Variables，除此之外，它可能还要根据具体情况决定恢复或设定已知启动项。

启动机制

EFI 能从一个使用 SIMPLE_FILE_SYSTEM Protocol 或 LOAD_FILE Protocol 的设备启动。一个支持 SIMPLE_FILE_SYSTEM Protocol 的设备必须为这个设备实例化一个文件系统，以使其可启动。如果一个设备不支持一个完整的文件系统，它可能产生一个 LOAD_FILE Protocol，来允许它直接实例化一个 image。Boot manager 将会先尝试使用 SIMPLE_FILE_SYSTEM Protocol 去启动。假如失败，就会使用 LOAD_FILE Protocol。

从简单文件 Protocol 启动

当从一个简单文件 Protocol 启动时，FilePath 将从一个设备路径开始，这个设备路径指

向支持SIMPLE_FILE_SYSTEM Protocol的设备。FilePath的下一部分将会指向文件名称，包括含有可启动image的子目录。如果文件名称是一个空的设备路径，要在媒介设备上找到文件名称，必须使用这样的规则，这个规则是带有模糊文件名称的可拔除设备定义的。(见"Removable Media Boot Behavior"节)。

EFI spec含有EFI特有的文件系统的格式。然而，firmware必须要产生一个能理解EFI文件系统的SIMPLE_FILE_SYSTEM Protocol，任何文件系统都可以被这种SIMPLE_FILE_SYSTEM Protocol接口抽象。

可拔除媒介的启动行为

在可拔除媒介设备上，FilePath不可能含有文件名称及子目录。平台中，存储在非易失性存储器上的FilePath不可能和一个随时可以改变的媒介保持同步。一个可拔除设备的FilePath将会指向支持SIMPLE_FILE_SYSTEM Protocol的设备。这个FilePath将不会包含文件名称和子目录。

一个默认文件名，加上这样的格式\EFI\BOOT\BOOT{ machine type short-name }。EFI，添加到一个可拔除设备FilePath，系统firmware会尝试从这样的FilePath去启动。其中，machine type short-name定义了一个PE32+ image的格式结构。每一个文件只包含一个EFI image类型，并且，系统可能支持从一个或多个这样的image格式启动。表13.2列出EFI image的类型。

表 13.2 EFI Image Types

Architecture	File name convention	PE Executable machine type*
IA-32	BOOTIA32.EFI	0x14c
Itanium architecture	BOOTIA64.EFI	0x200

注意：这种PE Executable Machine Type包含在COFF文件头的Machine区域中，就像Microsoft Portable Executable and Common Object File Format spec 6.0版中所定义的那样。

简单的使每一个可能的机器类型有一个\EFI\BOOT\BOOT{machine type short-name}。EFI文件，一个媒介可能支持多个结构。

从 LOAD_FILE Protocol 启动

当从LOAD_FILE Protocol启动，FilePath是一个指向支持LOAD_FILE Protocol设备的设备路径。Image会从支持LOAD_FILE Protocol的设备上直接加载。FilePath的剩余部分将会包含此设备特有的信息。EFI firmware把这个设备特有的数据传给已加载的image，但是不会用它们来加载image。如果FilePath的剩余部分为空，则已加载的image负责去实现找到正确的启动设备的决策。

LOAD_FILE Protocol被用来给不直接支持文件系统的设备用。Network设备通常在这种模式下启动，其image的实例化不需要文件系统。

Network Booting

Network Booting在Preboot Execution Environment (PXE) BIOS Support规范中有所讲述，该规范是The Wired Management Baseline Specification的一部分。PXE定义了UDP, DHCP, and TFTP Network Protocols，Boot Platform可以用它们与智能System Load Server通讯。EFI还定义了用于处理PXE的特定接口，这些接口包含于EFI规范所定义的

PXE_BASE_CODE Protocol中。

Future Boot Media

自从EFI在操作系统和Platform之间定义了一个抽象层， 随着技术的发展， EFI的loader就可以添加新类型的启动媒介。这样， OS Loader就没有必要为支持新的引导类型而被迫作出改动。不过， EFI Platform Services的实作可能会做些修改， 但共用的Interface并不会改变。操作系统需要Driver去支持新类型的启动媒介， 以便于完成从EFI Boot Services到操作系统对该媒介控制的过渡。

www.BIOSREN.COM

第十四章

引导流程

树林里分出两条岔路.....

—Robert Frost, "The Road Less Taken"

系统重启允许很多可能性，或者说，引导程序的执行路径不是固定的。对一颗给定的 CPU，引起它重启的原因有很多，并且环境状态也有差异。这些原因包括：闪存存储更新固件请求；电源管理事件恢复请求；系统初始化启动请求；以及其它可能的情形。这一章将介绍一些可能的引导流程以及框架（Framework）是怎样来处理这些事件的。

首先，在框架（Framework）中，正常的代码流程需要顺序经过以下几个阶段：

1. SEC
2. PEI
3. DXE
4. BDS
5. Runtime
6. Afterlife

这一章会介绍一个按照这个流程的引导方案，请参考图 14.1。

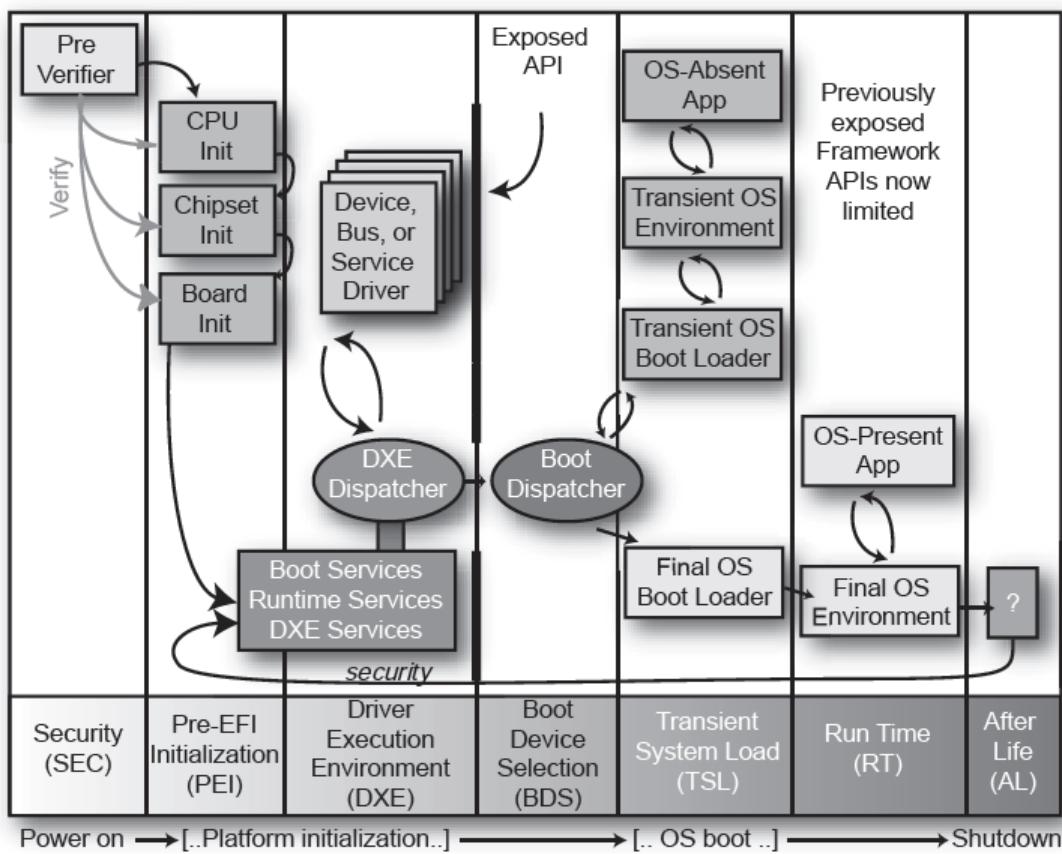


图 14.1 Ordering of Framework Execution Phases

PEI 基础模块（Foundation）并不知道该系统所需要的引导路径。它通过 PEIM 来确定引导模式（R0, R1, S3 等），并根据引导模式来做相应的动作。为了确定引导模式，每一个 PEIM 都可以通过 PEI Service SetBootMode()（见 15 章）来设定模式。注意：PEIM 不能根据引导模式来改变其被分派（dispatch）的顺序。

Defined Boot Modes 已定义的引导模式

在下一节会列出可能的引导模式和其相应优先级的清单。当需要定义新的模式时，框架结构避免定义一个明确的升级路径，新的启动模式需要加以界定。这样做的必要性在于，防止新增的引导模式与已定义的引导模式一起工作或者两者起冲突。

Priority of Boot Paths 引导路径的优先级

在一个给定的 PEIM 内，引导模式的优先级必须是可见的，如图 14.2. 引导模式的优先级顺序应当如下（从最高到最低）：

1. BOOT_IN_RECOVERY_MODE

2. BOOT_ON_FLASH_UPDATE
3. BOOT_ON_S3_RESUME
4. BOOT_WITH_MINIMAL_CONFIGURATION
5. BOOT_WITH_FULL_CONFIGURATION
6. BOOT_ASSUMEING_NO_CONFIGURATION_CHANGES
7. BOOT_WITH_FULL_CONFIGURATION_PLUS_DIAGNOSTICS
8. BOOT_WITH_DEFAULT_SETTINGS
9. BOOT_ON_S4_RESUME
10. BOOT_ON_S5_RESUME
11. BOOT_ON_S2_RESUME

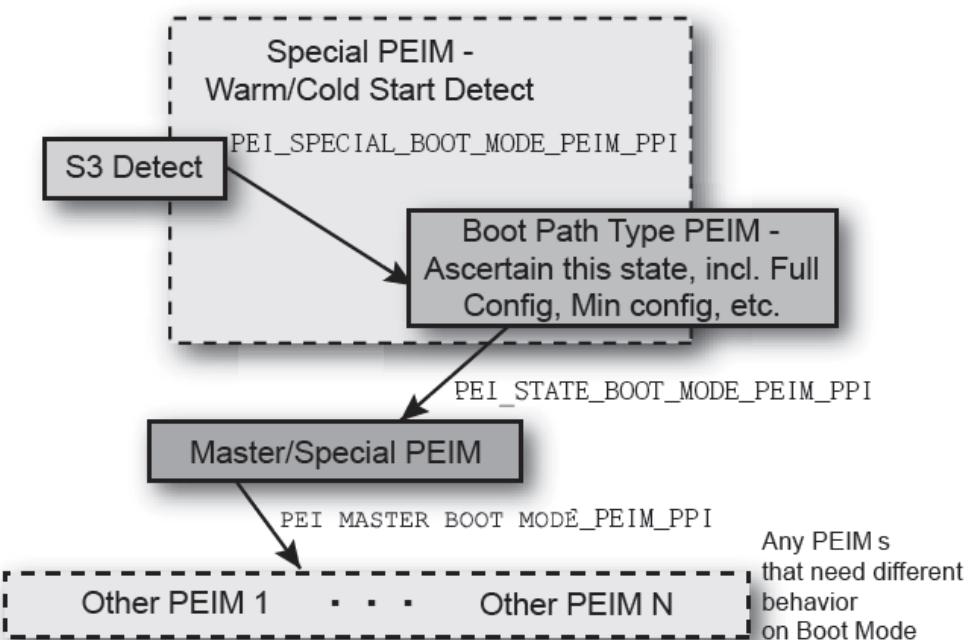


图 14.2 Priority of the boot modes

表 14.1 Boot Path Assumptions

系统状态	描述	假定情况
R0	冷启动 (Cold Boot)	不能假定之前存储的数据是有效的
R1	热启动 (Warm Boot)	可以假定之前存储的数据是有效的
S3	ACPI Save to RAM Resume 挂起到内存概要	之前存储的数据和内存都是有效的。在内存被使用之前，它的配置必须从非易失性存储器（NVS）还原。固件可以只更改之前预留的部分内存。预留内存有两种类型：一种等同于 INT15h, E820h 的 Type4 存储器，这类是预留给固件使用的；建议增加的另一种是，允许操作系统在运行（Runtime）时使用，但是可以在恢复的时候被改写。
S4, S5	Save to Disk Resume, “Soft Off” 挂起到磁盘 概要，软关闭	从 PEIM 的角度来看，S4 和 S5 是一样的。它们的区别体现在后续的阶段。整个系统必须重新初始化，但是 PEIM 可以假定之前的配置是有效的
基于固件更新的启动		这种引导模式可以源于 INIT, S3, 或者其它引起系统重启的原因。例如，如果同时有 S3 和 固件更新，那固件更新会有更高的优先级。具体行为取决于平台代码，比如由内存初始化 PEIM 来确定重启的原因和要执行的相应动作（也就是说，究竟是做 S3 的数据还原还是做 INIT 的初始化）

Reset Boot Paths 复位时引导路径

接下来的小节介绍系统在几种不同类型的复位时的引导路径。

Intel® Itanium® Processor Reset 英特尔安腾处理器复位

英特尔安腾架构包含足够的钩子 (hooks) 来验证处理器厂商发布的 PAL-A 和 PAL-B 代码。处理器芯片内部的微代码在 PG 信号复位时启动，并通过启动固件卷 (BFV) 内在架构层定义的指针找到位于启动固件卷内的 PAL-A (the first layer of processor abstraction code)。微代码负责验证处理器厂商提供的 PAL-A 代码层未被篡改过。如果通过了验证，控制权就会交给 PAL-A 层，PAL-A 又会去验证处理器的另一层抽象代码 (PAL-B)，然后再把控制权交给 PAL-B。除了对为架构特有的验证，SEC 阶段还负责找到 PEI Foundation 的位置，并且检验它的可靠性。

在一个基于 Itanium 的系统内，BFV 的固件模块必须有结构的组织起来，以满足 PAL-A 被放置在容错区 (Fault-tolerant region) 这个最低要求。不同的处理器有不同的 PAL-A 代码，PAL-A 用来验证 PAL-B，但是 PAL-B 通常被放置在不支持容错更新 (fault-tolerant updates) 的固件区域。在开机的时候，PAL-A 和 PAL-B 的二进制组件，对所有的同一节点 (node) 上的处理器都是可见的；没有必要去对系统架构进行初始化。

Non-Power-on Resets 非上电复位

非上电复位可以有很多原因引起。一些 PEI 和 DXE 系统服务复位整个平台，包括所有的处理器和设备。很重要的一点是，建立一套标准的引导路径来处理如下的情形：

- 复位处理器以改变频率设定
- 硬件重启以完成芯片组初始化
- 对异常处理的相应

这种复位也用于 CVDR (Configuration Values Driven through Reset) 的配置。

Normal Boot Paths 正常的引导路径

传统的 BIOS 执行 POST 的情形有：冷启动 (G3 到 S0)；电源管理恢复；一些特殊的情况，比如 INIT。EFI 不仅涵盖了所有这些情形，并且提供更丰富 (richer)、更标准的运行环境。

环境改变时，基本的代码流程也需要改变。引导路径变量即满足了这个要求。引导模式的初始值由一些较早的 PEIM 定义，但是它能被后续的 PEIM 更改。所有的系统都必须支持一个基本的 S0 引导路径。典型的情况是，一个系统拥有一系列的引导路径，包括 S0 引导路径，S-State 引导路径，一条或多条特殊的引导路径。

多条引导路径的架构可以带来如下的好处：

- PEI Foundation 不需要知道特定系统的要求，比如 MP 和各种电源状态。这样可以在以后做灵活的扩展。
- 支持多条路径只是增大了 PEI Foundation 的大小
- 需要支持多条路径的 PEIM 分担了系统的复杂度

注意：启动模式寄存器在过渡期到 DXE 阶段时，变成了一个变量。DXE 阶段也可以有一些额外的修改符来改变引导路径，而不是仅仅局限于 PEI 阶段。这些增加的 modifier 可以表明系统是否处于工厂模式，机箱防盗，没有 AC 电源，或者允许默认启动。

除了引导路径的类型，modifier bits 也可以存在。当任何一个 PEIM 检测到自己的数据有误，必要的恢复修饰被设置。

Basic G0-to-S0 and S0 Variation Boot Paths 基本的 G0 到 S0 以及 S0 衍生引导路径

基本的 S0 引导路径是 Boot with full configuration。这种设定告诉 PEIM 要做全部的配置。系统必须支持基本的 S0 引导路径。

框架架构也定义了几个可选的 S0 衍生引导路径，对衍生路径的支持依赖于以下几点：

- 可支持的特性的多样性
- 平台是开放的还是封闭的
- 硬件平台

例如，一个封闭的系统或者一个支持机箱防盗的系统，可以支持这样一条引导路径：假定从上次开机到这次开机没有任何配置变化，从而让系统迅速启动。默认不支持 S0 衍生引导路径。如下是已定义的一些衍生路径：

- Boot with minimal configuration：只配置启动系统所必需的硬件

- Boot assuming no configuration changes: 使用上一次开机的配置数据
- Boot with full configuration plus diagnostics: 除了硬件的全部配置外，还要执行诊断程序
- Boot with default settings: 使用默认的数据去去设定硬件

S-State Boot Paths S-State 引导路径

从 S3, S4, S5 恢复的时候，引导路径如下：

- S3 (挂起到内存恢复): 支持 S3 的平台必须保存和还原内存和一些关键的硬件配置。
- S4 (挂起到磁盘): 一些平台也许要执行缩短到 PEI 和 DXE 阶段
- S5 (软关闭): 一些平台也许希望 S5 off 启动和正常开机不一样，例如，非电源按钮可以唤醒系统。

我们要再详细地解释一下 S3 恢复，因为它需要 G0 到 S0 引导路径和 S3 恢复路径的合作。G0 到 S0 引导路径需要将 S3 恢复所需要的硬件编程信息保存起来。这些信息被保存在硬件结果保存表，并使用预定义的数据结构执行 I/O 或存储器写入。数据被保存在等效于 INT15h E820 type 4 (固件保留内存) 的一块区域或一块预留的固件设备区。在内存被还原之后，S3 恢复路径的代码可以访问这块区域。

Recovery Paths 恢复路径

如果系统检测到恢复是必须的，那么之前描述的所有启动路径都可以被修改或失效的。恢复是一个系统固件设备被破坏而重新建立的过程。这种破坏可能有多种机制引起。大部分位于非易失性存储设备 (flash, disk) 上的固件卷以块的形式被管理。如果系统在一个块或语义意义上的块更新时断电，这块存储器可能失效。另一方面，存储设备可能被一个错误的程序或硬件破坏。系统设计者必须基于他们能够想到的这些可能发生事件的和结果来决定所能支持的恢复层次。

以下是系统设计者可能选择不支持恢复的一些原因：

- 一个系统固件卷存储介质可能不支持更改当被制造好后。它可能是功能类似于 ROM 的设备。
- 大部分恢复机制的实现需要额外的固件卷空间，这对于一些特殊应用可能昂贵。
- 一个系统可能有足够的固件卷空间并且固件卷在硬件特性上可以实现彻底的容错，这样恢复机制就没有必要。

Discovery 发现

发现需要恢复可以由一个 PEIM (例如，检测一个“强制恢复”的跳转) 或者 PEI Foundation 来实现。PEI Foundation 可能发现一个特殊的 PEIM 没有通过正确的验证或者整个固件被破坏。

General Recovery Architecture 常规恢复体系

恢复的定义就是要保留足够的系统固件以便系统可以启动到它可以做到以下两点的阶段：

- 从被选择外围设备丢失的数据读取一份备份
- 重新用这些数据编写固件卷

可恢复固件的保存是管理固件卷存储方面的一个功能，这是在本书之外普遍存在的一个观点。这段描述的目的是希望 PEIMs 和其它需要恢复的固件卷内容要被标记。固件卷存储体系必须保存标记部分，或使它们不可被改变（可能需要硬件支持），或用一种容错的升级过程来保护它们。注意如果一个 PEIM 是恢复必须的或另一个 PEIM 依靠它来恢复，那它必须放在一个容错区域。这个体系结构同样也假定很容易公平的去判断哪些固件卷损坏。

然后 PEI 调度器正常运行。如果它遭遇到已经损坏的 PEIM（例如，接收到一个错误的 hash 值），它自己必须改变启动模式去恢复。一旦被设置成恢复模式，其它的 PEIMs 必须不能改变系统到其它状态。在 PEI 调度器发现系统处于恢复模式，它会重启，只调度那些恢复需要的 PEIMs。一个 PEIM 也可以检测灾难条件然后产生一个强制恢复事件去通知 PEI 调度器去进行一个恢复调度。一个 PEIM 可以通过和现在的启动模式进行或操作 BOOT_IN_RECOVERY_MODE_MASK 位来警告 PEI 基础模块去开始恢复。然后 PEI 基础模块重启使启动模式进入 BOOT_IN_RECOVERY_MODE 然后调度以 BOOT_IN_RECOVERY_MODE 为当前唯一模式的开始阶段。

一个 PEIM 可以被建立去处理部分恢复工作来初始化支持恢复工作的外围设备（包括他们位于的总线），然后从外围设备去读取新镜像来升级固件卷。

PEI 然后进入 DXE 阶段，因为 DXE 被设计来处理与外围设备的通信。这种进入额外的好处就是如果 DXE 发现一个设备损坏，它可能不用把控制权交给 PEI 阶段来开始恢复。

如果 PEI 基础模块没有一个关于调度内容的列表，它如何知道在固件卷里一个无效的区域是否包含一个 PEIM？PEI 基础模块在它搜索 PEIM 时它可以发现大部分的损坏，这看起来像是一个附加结果。因此，如果 PEI 基础模块在完成它的调度过程后并没有发现足够的静态系统存储空间去开始 DXE 阶段，然后它会进入到恢复阶段。

Special Boot Path Topics 特殊启动路径

本章剩下的部分将讨论对所有的处理器或对特殊的 intel 安腾处理器可用的特殊启动路径。

Special Boot Paths 特殊启动路径

以下是在 Framework 架构中的特殊启动路径。其中一些路径是可选的，而另一些是用于特殊处理器系列的。

- 强制恢复路径：一个跳转或一个类似的机制指明了一个强制的恢复
- intel 安腾架构启动路径：见下节
- 胶囊升级：这种启动模式可以用于一个中断、S3、或其它重启机器的方式。例如，如果是 S3，这个胶囊将代替 S3 重启。类似于内存初始化 PEIM 的平台代码的任务就是决定一个精确的原因并且来执行正确的行为，这就是相对于中断行为的 S3 状态恢复

Special Intel Itanium® Architecture Boot Paths

这个架构需要以下的特殊启动路径:

- 中断后启动: 一个中断发生了
- MCA 后启动: 一个机器检测体系 (MCA) 事件发生

intel 安腾处理器拥有几个独特的启动路径, 它们可以唤醒位于系统抽象层入口点 SALE_ENTRY 的调度器。处理器中断和 MCA 是两个启动安腾系统 sec 代码/调度器的异步事件。efi security 模块对于除了发生在硬启动的恢复检测外所有的代码路径是透明的。PEIM 或 DXE 驱动处理这些事件是架构级设计的并且不将控制权返回核心调度器。它们调用它们各自架构的位于操作系统的处理程序。

Intel Itanium® Architecture Access to the Boot Firmware Volume intel 安腾架构访问启动固件卷

图 14.3 显示了 intel 安腾处理器执行的重启路径。图 14.4 显示了启动流程。

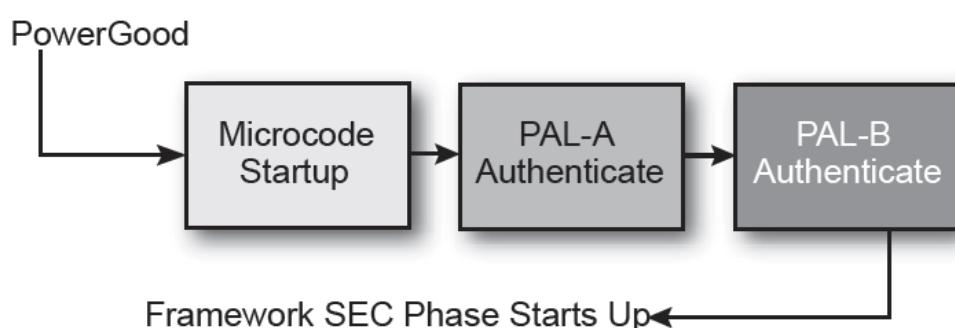


图 14.3 Intel® Itanium® Architecture Resets

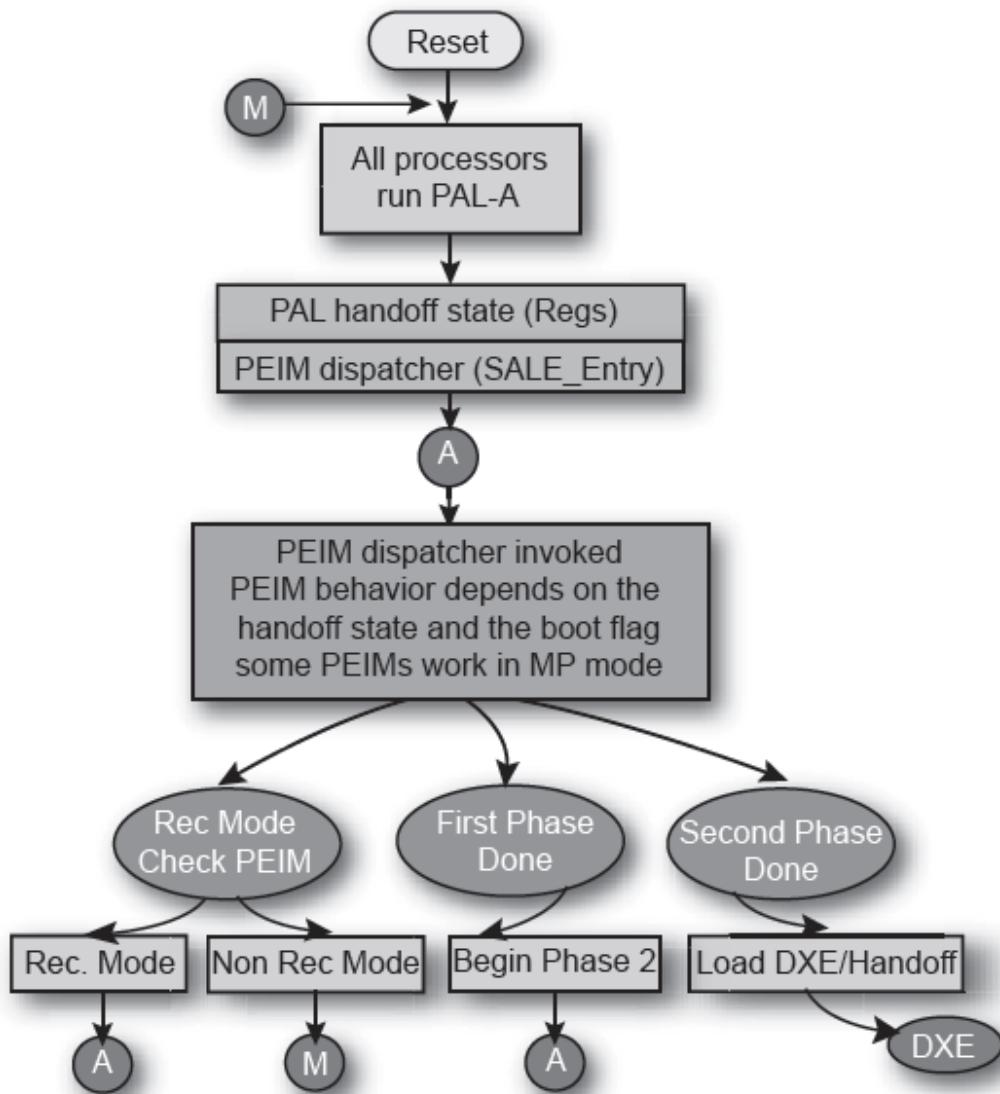


图 14.4 Intel® Itanium® Processor Boot Flow (MP versus UP on Other CPUs)

在 intel 安腾架构里，处理器制造者提供的位于启动固件卷的微码启动 PAL 代码的第一层。代码最小化初始化处理器然后发现并鉴别 PAL 代码的第二层（称作 PAL-B）。PAL-A 和 PAL-B 的位置可以被发现通过查找在 4G 区域附近的 ROM 中的架构指示器或者查找在 ROM 中的固件接口表 (FIT) 指示器。PAL 层使用一个单独的名叫 SALE_ENTRY 的入口点与 OEM 启动固件交流。

intel 安腾架构定义了以上的初始化描述。另外，安腾系统使用 Framework 架构必须做以下动作：

- 一个“特殊”的 PEIM 必须位于 BFV (启动固件卷) 来提供其它固件卷的布局信息。PEI 基础模块将位于 BFV(启动固件卷)的 SALE_ENTRY 的入口点。intel 安腾架构 PEIM 可能位于 BFV (启动固件卷) 或其它的固件卷，但是一个特殊的 PEIM 必须位于 BFV (启动固件卷) 来提供其它固件卷的启动信息。
- BFV (启动固件卷) 的一个特殊结点必须可以被运行在这个结点的处理器访问。

在每个结点的所有的处理器启动并且执行 PAL 代码然后进入 PEI 基础模块。BFV（启动固件卷）的一个特殊结点必须可以被运行在这个结点的处理器访问。这个区别同样意味着一些在 intel 安腾架构启动路径上 PEIM 可以被多处理器识别。

- 在启动时，在 BFV（启动固件卷）上的被 PAL-A、PAL-B、FIT 等二进制文件组织的固件模块必须对在一个结点上的所有处理器可见。这些二进制文件在系统结构没有进行任何初始化时必须是可见的。

```
/*
** EFI_BOOT_MODE
*/
typedef UINT32    EFI_BOOT_MODE;
#define  BOOT_WITH_FULL_CONFIGURATION      0x00
#define  BOOT_WITH_MINIMAL_CONFIGURATION   0x01
#define  BOOT_ASSUMING_NO_CONFIGURATION_CHANGES 0x02
#define  BOOT_WITH_FULL_CONFIGURATION_PLUS_DIAGNOSTICS 0x03
#define  BOOT_WITH_DEFAULT_SETTINGS        0x04
#define  BOOT_ON_S4_RESUME                0x05
#define  BOOT_ON_S5_RESUME                0x06
#define  BOOT_ON_S2_RESUME                0x10
#define  BOOT_ON_S3_RESUME                0x11
#define  BOOT_ON_FLASH_UPDATE             0x12
#define  BOOT_IN_RECOVERY_MODE            0x20
0x21 – 0xF..F Reserved Encodings
```

表 14 2 Boot Mode Register

REGISTER BIT(S)	VALUES	DESCRIPTIONS
MSBit-0	000000b	Boot with full configuration
	000001b	Boot with minimal configuration
	000010b	Boot assuming no configuration changes from last boot
	000011b	Boot with full configuration plus diagnostics
	000100b	Boot with default settings
	000101b	Boot on S4 resume
	000110b	Boot in S5 resume
	000111b-001111b	Reserved for boot paths that configure memory
	010000b	Boot on S2 resume
	010001b	Boot on S3 resume
	010010b	Boot on flash update restart
	010011b-011111b	Reserved for boot paths that preserve memory context
	100000b	Boot in recovery mode
	100001b-111111b	Reserved for special boots

Architectural Boot Mode PPIs 架构启动模式 PPI

在 PEI 章节里，PPI 的思想在这个操作阶段被作为一个协同模块被介绍。PEI 模块可以知晓启动模式通过 GetBootMode 服务被调度，但是一个系统设计者可能不想让一个 PEIM 运行除非是在一个特定的启动模式。可以从各种不同制造者的启动模式抽象出一个启动模式 PPI 的分层。各种启动模式可以被设置成一个按优先级排列顺序的层次。PPI 和它们各自的 GUID 在 Required Architectural PPI 中定义，在 PEI 阶段这些 PPI 可以位于 PEI 核心接口和可选的架构 PPI 中。这个层次包括一个主 PPI 和一些辅助 PPI，主 PPI 会发行一个依赖于适当 PEIM 的 PPI。对于那些最终可以知道启动模式的 PEIM 来说，主启动模式 PPI 可以被作为一个依赖。

表 14.3 Architectural Boot Mode PPIs

PPI Name	Required or Optional?	PPI Definition in Section...
Master Boot Mode PPI	Required	Architectural PPIs: Required Architectural PPIs
Boot in Recovery Mode PPI	Optional	Architectural PPIs: Optional Architectural PPIs

Recovery 恢复

这一段主要讨论平台固件恢复。恢复是用来提供一个更高的 RASUM（可靠性、实用性、可用性、可管理性）的可选方式。恢复是一个系统固件设备被破坏而重新建立的过程。这种破坏可能有多种机制引起。大部分位于非易失性存储设备（例如闪存和硬盘）上的固件卷以块的形式被管理。如果系统在一个块或语义意义上的块更新时断电，这块存储器可能失效。另一方面，存储设备可能被一个错误的程序或硬件破坏。系统设计者必须基于他们能够想到的这些可能发生事件的和结果来决定所能支持的恢复层次。

以下是系统设计者可能选择不支持恢复的一些原因：

- 一个系统固件卷存储介质可能不支持更改当被制造好后。它可能是功能类似于 rom 的设备。
- 大部分恢复机制的实现需要额外的固件卷空间，这对于一些特殊应用可能昂贵。
- 一个系统可能有足够的固件卷空间并且固件卷在硬件特性上可以实现彻底的容错，这样恢复机制就没有必要。

Discovery 发现

发现需要恢复可以由一个 PEIM (例如, 检测一个“强制恢复”的跳转) 或者 PEI Foundation 来实现。PEI 基础模块可能发现一个特殊的 PEIM 没有通过正确的验证或者整个固件被破坏。

注意: 在这种观点下, 系统地物理性重启并没有发生。PEI 调度器仅仅清除状态信息和重启它自己。

一个 PEIM 可以被建立去处理部分恢复工作来初始化支持恢复工作的外围设备 (包括他们位于的总线), 然后从外围设备去读取新镜像来升级固件卷。

PEI 然后进入 DXE 阶段, 因为 DXE 被设计来处理与外围设备的通信。这种进入额外的好处就是如果 DXE 发现一个设备损坏, 它可能不用把控制权交给 PEI 阶段来开始恢复。

如果 PEI 基础模块没有一个关于调度内容的列表, 它如何知道在固件卷里一个无效的区域是否包含一个 PEIM? PEI 基础模块在它搜索 PEIM 时它可以发现大部分的损坏, 这看起来像是一个附加结果。因此, 如果 PEI 基础模块在完成它的调度过程后并没有发现足够的静态系统存储空间去开始 DXE 阶段, 然后它会进入到恢复模式。

第十五章

PEI 初始化

小就是美

---- E.F. Schumacher

在平台运行期间 PEI 阶段有两个主要的任务：确定重新启动的来源和为相继运行的 DXE 阶段提供少量的固定内存。“少的”，“最小限度的”，等术语经常用于 PEI 代码，因为硬件资源的约束限制了编程的环境。特别地，PEI 阶段为处理器、芯片组和系统板提供了标准的加载和调用特殊初始配置程序的方法。PEI 阶段发生在 SEC 阶段后，运行该阶段代码的首要目的是初始化好足够多的系统，允许 DXE 阶段能够运行。最起码，PEI 阶段要负责决定系统启动的路径、初始化和描述一个最小数量的系统内存及包含 DXE Foundation 和 DXE 结构协议的 FV。如无必要，勿增实体，最少量的动作应该被精确的加载到这个阶段执行，不多不少刚刚好。

作用域

PEI 阶段负责为后续阶段提供一个稳定的基础而初始化足够多的系统。它也负责从失效的固件储存空间检测和恢复并提供重起原因（启动模式）。

现今的计算机一般都在一个非常原始的状态执行，纵观启动固件，例如 BIOS 或框架。处理器可能需要更新它们的内部微码；芯片组（芯片组提供了处理器和系统其它主要器件之间的接口）需要相当多的初始化。PEI 阶段就是负责初始化这些基本的子系统。PEI 阶段打算提供一个简单的基础结构，通过它一些有限的为了转移到更先进的 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那些小部分任务；换而言之，它应当只需要履行那些为启动 DXE 所需的那部分最小的任务。因为硬件的改进，这些任务中的部分可能从 PEI 阶段移出来。

基本原理：

设计 PEI 本质上就是设计一个微型版的 DXE，它们有很多相同的问题。PEI 阶段包含几个部分：

- 一个 PEI 基础；
- 一个或多个 PEI 初始化模块

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，加入它们将也起不到作用。

PEI 不像 DXE，DXE 阶段存在合理数量可用的固态系统内存，PEI 阶段仅有有限的临时内存存在并且这些临时内存存在固态系统内存被初始化后可能在 PEI 阶段中被重新配置用做他用。正因如此，PEI 阶段没有 DXE 阶段那样丰富的特征。下面是它们之间特征不同的最明显例子：

- DXE 有丰富的 loaded image 数据库和与绑定在 image 上的协议
- PEI 没有像 DXE 驱动模型这样丰富的模块层次结构。

概述

PEI 阶段由一些 Foundation 代码和像 PEIM 这样的特殊驱动组成，PEIM 可定制平台 PEI 阶段所需的操作。Foundation 代码负责为插入的 PEIM 按顺序命令调配并提供基本的服务。PEIM 类似于 DXE 驱动，一般对应于器件的初始化。PEIM 预期是器件供应商提供可能以原代码形式统一实行的 PEIM，这样客户能快速地调试整合造成的问题。

实现 PEI 阶段比其它 Foundation 更多地依赖于处理器的结构。特别是，处理器提供越多其初始化或接近初始状态的资源，PEI 环境就会变得越丰富。同样地，下面有几个关于结构需求注意事项的讨论部分，但是，这是处理器结构的一些特殊部分，因此不是完整的定义。

PEI 阶段能够从时间和空间的两个角度看待。图 15.1 提供了启动过程的总体结构。图 15.2 是 PEI 从空间上看的结果。这张图描述了框架中各器件的层次。这张图通常被说成是“H”型结构。PEI 折衷“H”的下半部分。从时间的角度看就是等到 PEI Foundation 和它相关的模块执行的时候。图 15.3 中标记的是图 15.1 的一部分，这部分就包含着 PEI。

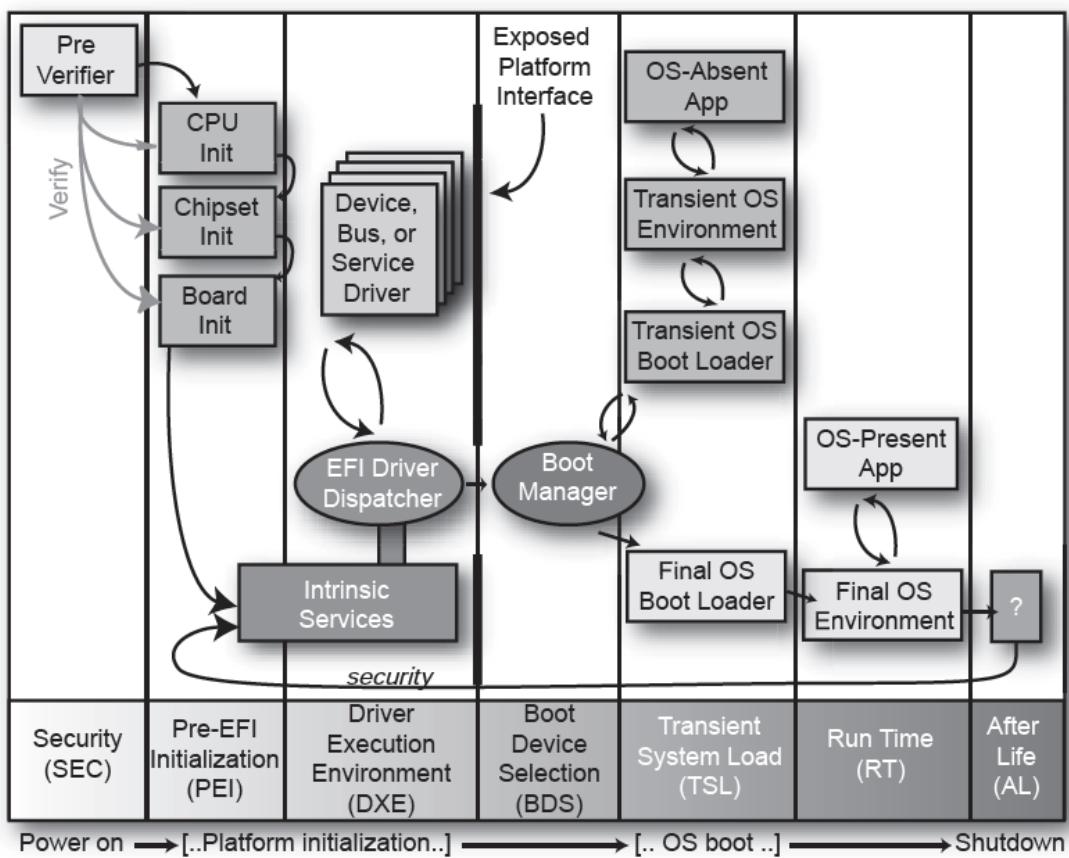


Figure 15.1 Overall Boot Flow

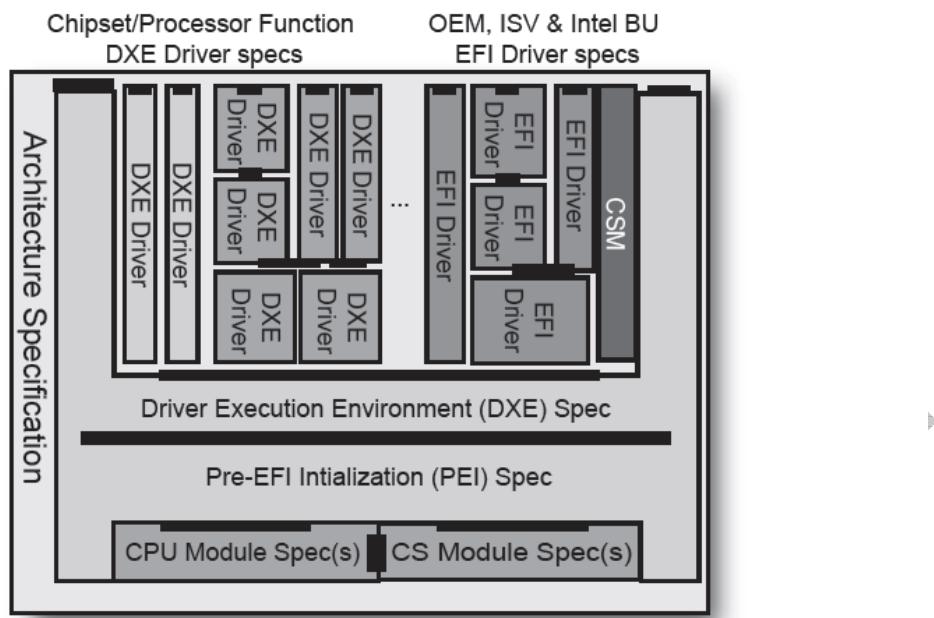


Figure 15.2 System Components

KJJSWZ

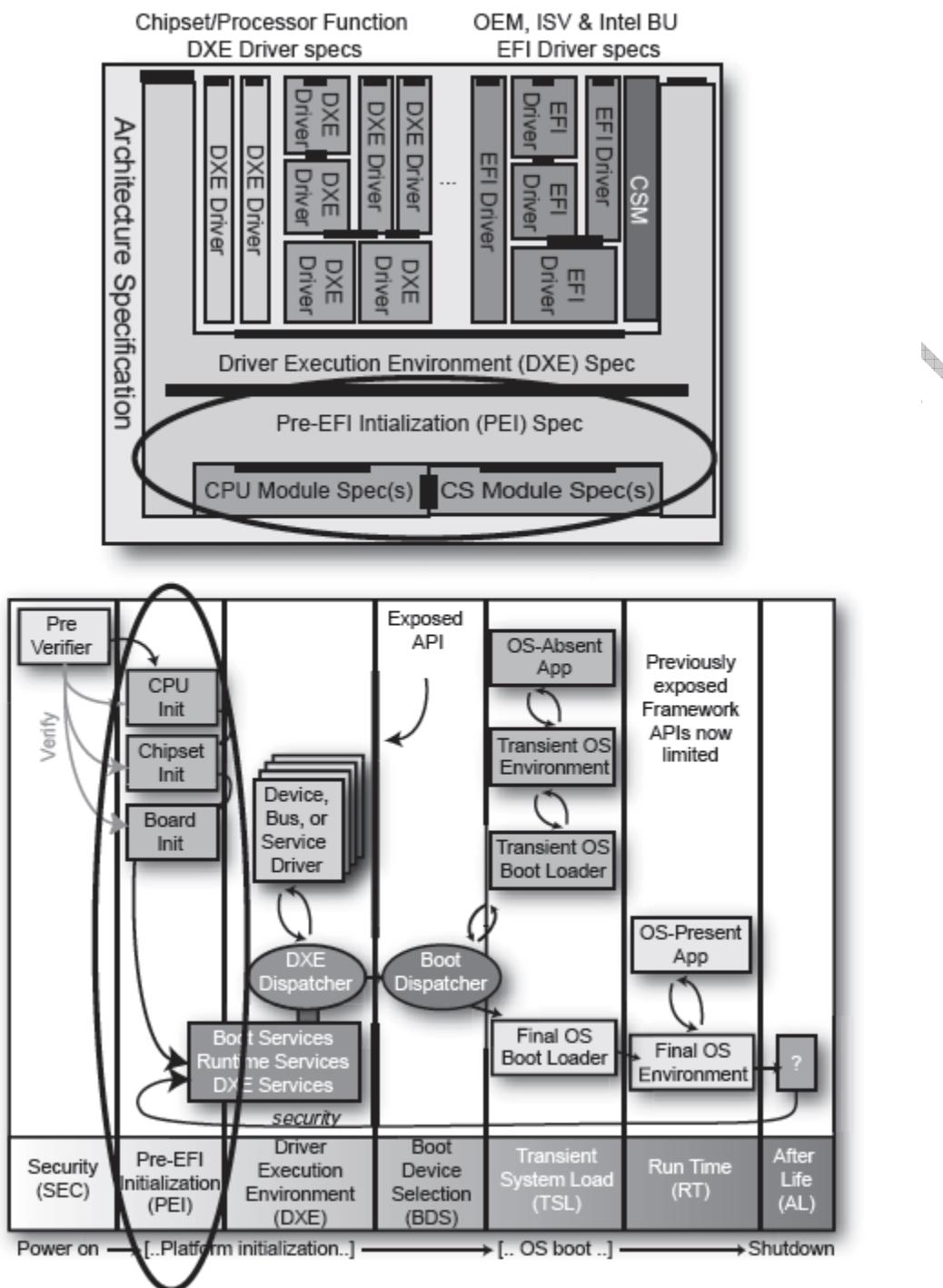


Figure 15.3 Portion of the Overall Boot Flow and Components for PEI

此阶段要求

下面描述了 PEI 阶段能成功完成所必要的部分：

临时的 RAM

PEI Foundation 要求在系统内存被完全初始化前 SEC 阶段应初始化好最小数量的暂存式 RAM 用于 PEI 阶段数据存储。这个暂存式 RAM 应该能像常规 RAM 那样被访问—例如：前端总线通过存储周期访问。系统被完全初始化好后，临时 RAM 可能会被重新初始化给别人使用。典型地，被提供用作临时 RAM 的是处理器的内部 Cache。

启动 FV (BFV)

启动固件卷包含了 PEI Foundation 和 PEIM 模块。BFV 必需出现在不需要固件预先干涉的系统内存空间并且典型地包含具有处理器结构的复位矢量。

BFV 的内容遵循 EFI flash 文件系统的格式。PEI Foundation 遵循 EFI flash 文件系统的格式在 BSV 中寻找 PEIM。平台中一个特殊的 PEIM 可能会将系统中其他 FV 的位置通知给 PEI Foundation，这就允许 PEI Foundation 去其他 FV 中找 PEIM。PEI Foundation 和 PEIM 命名通过 EFI flash 文件系统中的统一 ID。

Recovery (恢复) 要求 PEI Foundation 和一些 PEIM 既能锁定在一个不能被更新的 BSV 中也能使用一种容错机制被更新。EFI flash 文件系统提供了出错 Recovery (恢复)；如果系统终止在任何一个点，不管是预先更新好的 PEIM 或是最新更新好的 PEIM 都能完全地有效，并且 PEI Foundation 能确定哪个是有效的。

安全性元素

SEC 阶段提供了一个用来执行验证操作的接口到 PEI Foundation。为了延续根部信任，PEI Foundation 将用这个机制去使各个 PEIM 有效。

概念

下面几个部分描述了设计 PEI 阶段的一些概念。

PEI Foundation

PEI Foundation 是一个用每个处理器结构编译成函数的一个简单可执行的二进制文件。它履行两个主要的功能：

- 调度 PEIM
- 提供一组通用的核心服务给 PEIM 使用。

PEI 调度器的工作是有序地传递控制权给 PEIM。通用核心服务是通过一个服务程序表提供，这个表通常称做 PEI 服务程序表。这些服务程序主要做如下事情：

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

- 辅助 PEIM 与 PEIM 通讯。
- 抽象化管理临时 RAM
- 提供如下通用函数帮助 PEIM:
 - 在 FFS 中查找其它文件
 - 报告状态码
 - 为传递状态给框架的下一个阶段做准备。

当 SEC 阶段完成，SEC 调用 PEI Foundation 并提供几个带参数的 PEI Foundation:

- BFV 的大小和位置，因此，PEI Foundation 知道到哪里去找初试设置好的 PEIM。
- PEI 阶段能用的最小数量临时 RAM
- 一个回调验证服务，用来允许 PEI Foundation 验证在调度 PEIM 之前这些 PEIM 已经被确认发现了

PEI Foundation 辅助 PEIM 之间互相通信。PEI Foundation 为所有 PEIM 维护一个已注册的接口数据库，图 15.4 所示。这些接口被叫做 PEIM 到 PEIM 之间的接口（PPI）。PEI Foundation 提供了允许 PEIM 注册 PPI 并且当另一个 PEIM 建立了 PPI 时会通知（反馈）的接口。

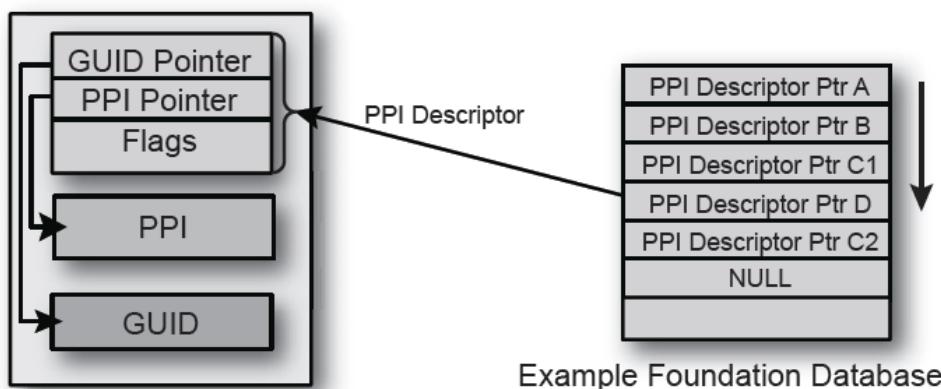


Figure 15.4 How a PPI Is Registered

PEI 调度器由一个单一阶段组成。在调度阶段 PEI Foundation 检查所有包含在 FV 中类型为 PEIM 的文件。它在每个固件文件中检查依赖性表达式来决定这个 PEIM 是否可以运行。用于 PEIM 的二进制编码条件判断表达式与 DXE 阶段驱动相关联的条件判断表达式是一样的。

PEI 初始化模块

PEIM 可包含处理器，芯片组，设备，或者其它特殊平台功能的可执行二进制文件。PEIM 可抽象成可提供 PEIM 或 PEI Foundation 与 PEIM 或硬件间通信的接口。PEIM 是单独编译成的二进制模块，它典型地驻留在 ROM 中且没有被压缩。可能存在一小部分 PEIM 为了性能原因在 RAM 中运行，这些 PEIM 以压缩格式放在 ROM 中。放在 ROM 中的 PEIM 是内置式可执行的模块，这些模块可能是由位置独立代码或位置相关代码与重定位信息一起组成的。

PEI 服务程序

PEI Foundation 在系统中建立一个可让所有 PEIM 可视化的系统表，这个表被命名为 PEI Services Table。一个 PEI 服务程序可以被定义成一个函数，命令，或者其它功能，当需要初始化成哪种服务时，PEI Foundation 就能把它表现出来。因为直到 PEI 阶段的末端此阶段没有可用的固定存储器，所以 PEI 阶段能被创建的服务程序的范围不能像 PEI 后阶段一样丰富。因为在编译的时候不知道 PEI Foundation 和其临时内存的位置，所以一个指向 PEI Services Table 的指针被放进每个 PEIM 和 PPI 中。PEI Foundation 提供了下面几种类型的服务：

- PPI 服务：管理 PPI 促进 PEIM 模块间的调用。在临时 RAM 中维护接口被安装和被跟踪信息的数据库。
 - 启动模式服务：管理系统的启动模式（S3, S5, 正常启动, 诊断, 等）。
 - HOB 服务：建立叫 Hand-Off Blocks 的数据结构，这用于传递信息给整个框架中的下一个阶段。
 - Firmware Volume 服务：在 Flash 的 FV 中扫描各 FFS 寻找 PEIM 和其它固件文件。
 - PEI Memory 服务：在固定存储器被发现的前后，提供存储器管理服务程序集。
 - 状态码 Services：统一的进展和错误代码报告服务程序，用来从 80 口或串口丢简单字符调试用。
 - Reset Services：提供一个统一的方式重起系统。

PEIM 到 PEIM 间的接口 (PPI)

PEIM 可以通过 PPI 接口调用其它 PEIM。为了允许模块的独立开发和接口间没有名字冲突，这些接口使用 GUID 给自己命名。GUID 是一个 128 位的值，在启动服务中用于区分服务程序和结构。PPI 被定义成结构，它可以包含函数，数据，或者这两者的联合体。PEI Foundation 管理了注册 PPI 的数据库，PEIM 必须要使用它来注册 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。两种类型的 PPI 是：

- 服务程序
- 通知

PPI 服务程序允许这个 PEIM 给另外一个 PEIM 提供函数或数据使用。PPI 通知允许注册一个 PEIM 回调，当另一个 PPI 被 PEI Foundation 注册。

简单的堆

在固定系统存储器被安装前，pei Foundation 使用临时 RAM 提供一个简单的堆存储。PEIM 可以请求从堆中分配存储单元，但是没有从堆中释放内存的机制存在。一旦固定存储器被安装好，堆就会被重新定位到固定系统内存中，但是 PEI Foundation 不会处理堆中存在的数据。因此，当目标是堆中其它数据时某 PEIM 不能将指针存入此堆中。正如链表。

传递下去的信息块 (HOBs)

HOBs 是 Framework 结构中从 PEI 阶段传递系统状态信息到 DXE 阶段的结构机制。HOB 是内存中简化的数据结构（单元），它包含一个头和数据部分。头的定义对所有 HOB 都是统一的且允许任何代码使用这个定义去了解两个条款：

■ 数据部分的结构

■ HOB 总大小

固定内存被安装好后，HOB 在内存中被顺序地分配，PEIM 可以用到这些 HOB。一系列核心服务促成了这个内存中 HOB 的顺序列表，常被称做 HOB 列表。HOB 列表中的第一个 HOB 必须是 PHIT HOB，它描述了 PEI 阶段所使用的内存和这期间所发现启动模式。

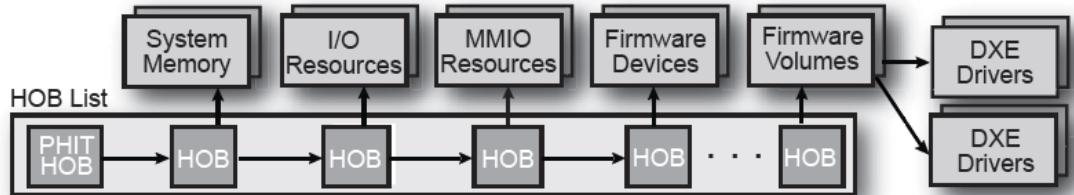


Figure 15.5 The HOB List

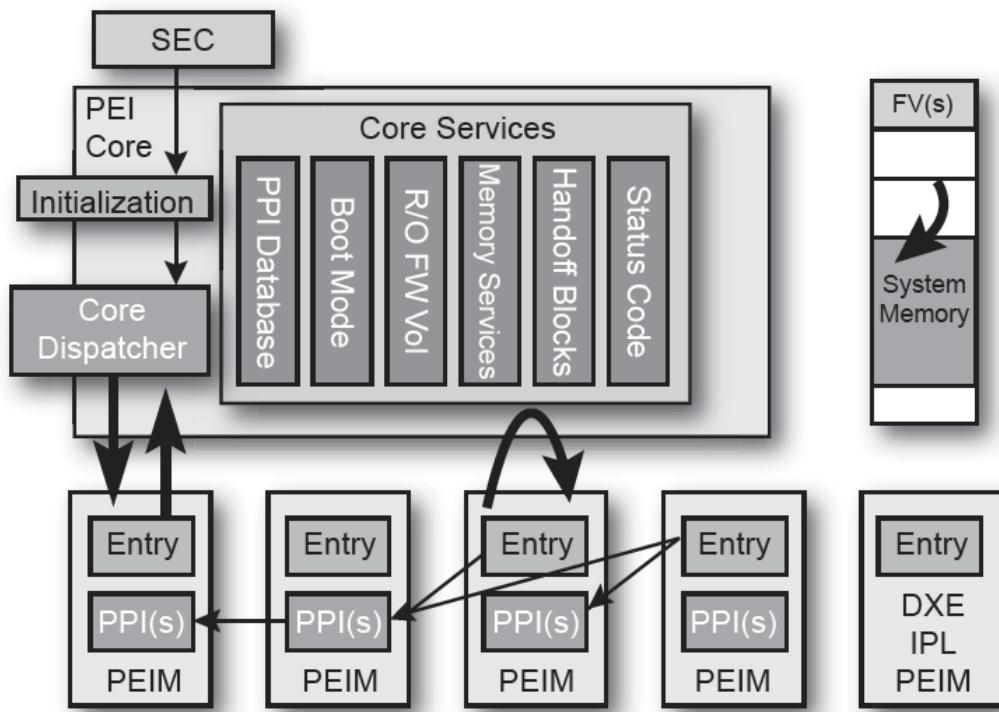
只有 PEI 阶段的元件才允许增加或更改 HOB。一旦 HOB 列表被传递到 DXE，DXE 元件只能对它有效地读。DXE 阶段中从只读 HOB 列表中读到的衍生物是握手信息，例如，启动模式，必需以一种统一的方式处理；如果 DXE 将造成一个 recovery 条件，它将不会更新启动模式，而是使用一种特殊类型的复位调用来代替执行动作。HOB 列表包含 PEI 阶段传递到 DXE 阶段时所包含的系统状态数据，但不代表 DXE 阶段期间系统当前的状态。DXE 阶段的器件应使用 DXE 定义的那些服务代替通过分析 HOB 列表来获得当前系统的状态。作为指导方针则期望 PEI 和 DXE 阶段之间将仿照一个生产者和一个消费者的模型传递信息。换而言之，即在 PEI 阶段一个 PIEM 会产生一个 HOB，并且 DXE 驱动能使用那个 HOB 并将与那个 HOB 相关联的信息传递给 DXE 阶段中其它需要这个信息的器件。DXE 驱动提供给其它 DXE 器件的 HOB 信息所使用的方法应仿照 DXE 结构所定义的机制。

操作

PEI 阶段的操作包含：调用 PEI Foundation，顺序地调度所有的 PEIM，发现和调用下一个阶段。PEI Foundation 初始化期间，PEI Foundation 初始化那些需要给 PEIM 提供通用服务的内部数据区域和函数。PEIM 调度期间，PEI 调度器遍历所有所有的 FV，根据 flash 文件系统定义发现 PEIM。然后，如果下面的条件满足的话 PEI 调度器调度所有的 PEIM：

- PEIM 还没有被调用过。
- PEIM 文件已经被正确的格式化好。
- PEIM 可信赖
- PEIM 的依赖性条件满足。

调度完一个 PEIM 后，PEI 调度器继续在 FV 中查找其它的 PEIM，直到，（既）发现的所有 PEIM 都被调用过，同时也没有能够被调用的 PEIM，因为没有一个 PEIM 能满足上述要求。一旦这个条件被满足，PEI 调度器的工作就完成了，然后它调用一个（类似）PPI 结构来启动整个框架的下一个阶段，DXE 初始化程序加载这个 PPI (IPL)。

**Figure 15.6** PEI Boot Flow

Dependency 表达式

执行 PEIM 的顺序通过评估每个 PEIM 相关联的条件判断表达式来决定。这个二进制表达式描述了每个 PEIM 运行所必要的条件，它强调了 PEIM 的弱序化。在这个弱序化中，PEIM 可以被以任何顺序初始化。

在 Dependency 表达式中参考（引用）PPI 和文件名中的 GUID。Dependency 表达式是操作中一个有代表性的语法，它能够在多数 Dependency 中执行，来判断 PEIM 是否可以运行。PEI Foundation 从内部运行的 PEIM 反向估算 Dependency 表达式并注册 PPI。那些估算操作可以在 Dependency（包括逻辑与、逻辑或、逻辑非和之前、之后的顺序操作）上完成。

验证/鉴别

PEI foundation 对于安全性没有固定形式。相反，安全性决定是安排给平台特殊器件的。两个值得注意的关于安全性抽象化的器件是 Security PPI 和验证 PPI。验证 PPI 的目的是检查每个给予的 PEIM 的鉴别状态。用在那里的机制包含数字签名验证，简单的校验和，或者一些其它 OEM 特殊机制。这个验证的结果返回到 PEI Foundation，它轮流传送结果给 Security PPI。Security PPI 决定是否延迟执行 PEIM 或者现在执行。另外，Security PPI 提供者可以选择产生一份已调度 PEIM 的证明日志入口或者提供一些其它安全性异常事件。

执行 PEIM

当 PEIM 被 PEI Foundation 调用时就会跑去执行。每个 PEIM 只能被调用一次，并且必须完成它自己的工作（调用和安装其它 PPI，允许其它 PEIM 在必要的时候能够调用到它）。如果必要的时候 PEIM 也可以注册一个通知 callback 用于其它的 PEIM 已经运行完后，这个 PEIM 需要再次获得控制权。

发现内存

发现内存是 PEI 阶段中一个重要的结构事件。当一个 PEIM 成功地发现、初始化和测试了一个连续范围的系统内存时，它会把这个 RAM 报告给 PEI Foundation. 当那个 PEIM 退出时，PEI Foundation 把 PEI 使用的临时内存转移到真实系统 RAM 中，这涉及到下面两个任务：

- PEI Foundation 必须把临时 RAM 中使用的堆栈转换到固定的系统内存。
- PEI Foundation 必须把 PEIM (包括 HOB) 分配好的简单堆转移到真正的系统 RAM 中。

一旦这个过程完成了，PEI Foundation 就会安装一个 architectural PPI 通知任何相关的（感兴趣的）PIEM，告诉它们真正的系统内存已经建立好了。这个通知允许 PEIM 在内存建立好之前 (ran to be called back) 被回叫，因此它们能完成必要的任务——例如，在实际的系统内存中为 DXE 阶段建立 HOB。

Intel 安腾处理器 MP 考虑事项

这部分给出了 Intel Itanium 处理器 MP 系列系统在 PEI 阶段所需要的特殊考虑。基于安腾的系统里，所有处理器在系统起动处同时执行由处理器供应商提供的 PAL 初始化代码。然后，处理器被一个请求调用到 Framework 的起动代码中来检查 recovery。起动代码给每个活动的处理器分配大量的临时内存并设置堆栈和在分配临时内存中重填指针。临时内存可能是处理器 Cache 的一部分 (CAR)，这个能被调用 PAL 配置。启动代码然后在每个处理器上开始调度各个 PEIM。运行在多处理器模式下的前期的 PEIM 会选择一个处理器作为启动绑定处理器 (BSP) 来运行启动阶段的 PEIM。

BSP 继续运行 PEIM 直到它找到固定内存和 PEI Foundation 的内存建立好为止。然后，BSP 唤醒所有的处理器来决定它们的 health 和 PAL 兼容状态。如果这些检查都没有授权固件复原，各处理器将返回到 PAL 做更多的处理器初始化因正常启动。

在基于安腾的系统中不管 INIT 或 MCA 事件发生与否，Framework 启动代码照样被触发。在这些条件下，PAL 代码丢出状态码，同时一个缓冲区调用最小状态缓冲区。一个特殊的 Framework 指针指向附着在这个最小状态缓冲区中的 INIT 和 MCA 代码区，它包含了在 INIT 和 MCA 上执行

event 的详细资料。这个缓冲区也保持了启动代码在给这些特殊硬件事件做决定时所需要的重要变量值。

Recovery

恢复是当固件被破坏时重新建立一个系统固件设备的过程。这个错误（失效）可由不同的机制引起。大多数 FV 在非易失性存储设备中以块的型式被管理。当一个 Block 或 semantically bound blocks 正在更新的时候系统掉电，这个储存器可能会变得无效。另一方面，固件设备可能是由于编程错误或者硬件错误引起失效的。假设 PEI 在容错 block 中还是活着的，它能支持一个 recovery 模式的调度。

PEIM 或 PEI Foundation 自身能够发现是否需要 recovery. 有个 PEIM 能检查“强制复原”跳线，例如，检查 recovery 需求。PEI Foundation 可能能发现某个特殊的 PEIM 不是恰当地有效，或者整个 FV 已经变得无效了。

Recovery 后的概念是有足够的系统固件被保留，以至于系统可以启动到一个可以读到从所选择外围设备中丢失的数据副本的点，然后用那些副本数据对 FV 编程。

recovery 固件的保存是FV存储器管理方式的一个函数。在EFI flash文件系统中，recovery 所需求的 PEIM 就被标记成那样。FV 中用于储存的结构体必须预留已标记的条款，既可以把它标记成不能变更的（可能需要硬件的支持）也可以用一个容错更新进程保护它们。

到 recovery 模式被发现后，PEI 调度器才能正常的继续下去。如果 PEI 调度器遇到 PEIM 已经被破坏（例如，通过接收一个错误的无用信息），它必须改变 recovery 的启动方式。一旦真正开始 recovery，其它的 PEIM 不能改变它们的状态。PEI 调度器已经发现系统已处于 recovery 模式后，它将重新启动自己，仅仅去调度那些 recovery 所需要的 PEIM。它也可能是一个去检测灾难性条件的 PEIM 或者是一个强制性 recovery 检测的 PEIM，并且通知 PEI 调度器（这个 PEIM）要继续执行一个 recovery 调度。当一个 PEIM 在 recovery 媒介上找到 FV 且 DXE Foundation 已经从那个 FV 上被启动后，Recovery 调度才算完成。在那个 DXE FV 中的驱动能够完成恢复过程。

S3 返回

PEI 阶段 S3 返回不同于其它正常启动的几个基本方式。下面是不同之处：

- 内存部分被恢复成睡眠前的状态而不是初始化它。
- 系统内存归属于 OS，不能被 PEI Foundation 或者 PEIM 使用。
- 返回的时候 DXE 阶段不能被调度，因为它将破坏内存。
- PEIM 将正常地调度 DXE 阶段代替使用特殊的 Hardware Save Table 去恢复基本硬件到启动时的配置。恢复完硬件后，PEIM 将控制权交给 OS 提供的返回矢量。

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

■正常启动期间 DXE 和其后续阶段在 Framework 预留的内存或 FV 中保存了足够的信息，供 OS 用于恢复硬件设备的状态。这个保存的信息就位于 Hardware Save Table。

系统示例

当检查一个具体的平台时所有 PEI 相关的概念能被综合。下面的列表代表一个具有所有相关系统组件的 865 系统。这个相同的系统也在图 15.7 中展示，它包含了各种实际的 silicon 组件。这些组件在后面的图中有对应的 PEIM 去抽象化组件的初始化和其服务程序。这些 Component 中的每一个，在几个 PEIM 中可能有一个被分配去抽象化该特定 component 的行为。这些 component 可包含的例子有：

- Pentium4 处理器 PEIM: 初始化 CPUIO 服务程序
- PCI 配置 PEIM: PCI 配置 PPI
- ICH PEIM: ICH 初始化和 SUNBUS PPI
- 内存初始化 PEIM: 通过 SMBUS PPI 读 SPD, 初始化内存控制器, 报告 PEI 可用的内存
- 平台 PEIM: Flash Mode 的创建, 检测启动模式
- DXE IPL: 用于建立起 DXE 的特殊服务, 调用 S3 或者恢复流程

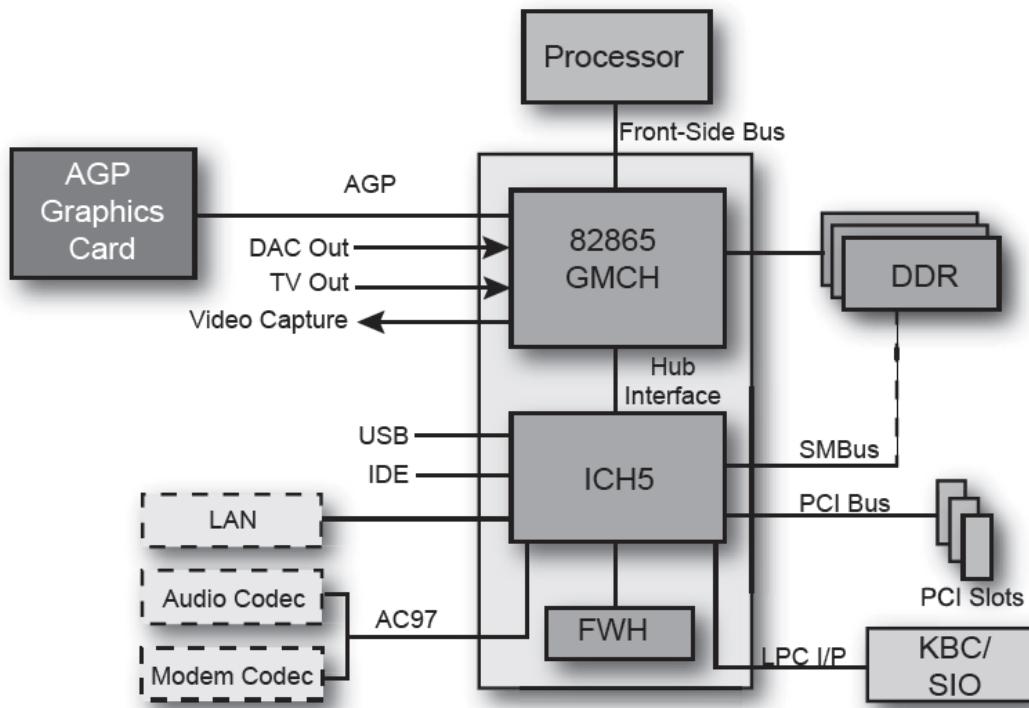


Figure 15.7 Specific System

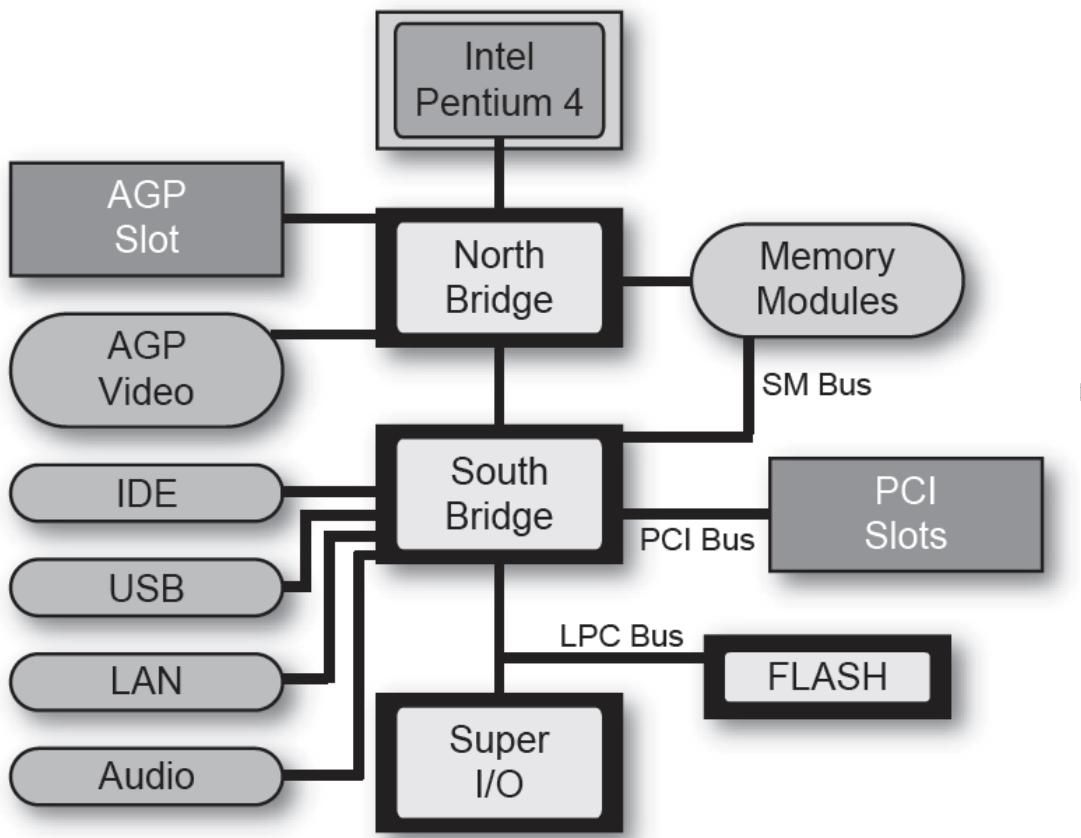


Figure 15.8 Idealization of Actual System

```

typedef
EFI_STATUS
(EFIAPI *PEI_SMBUS_PPI_EXECUTE_OPERATION) (
    IN      EFI_PEI_SERVICE           **PeiServices,
    IN      struct EFI_PEI_SMBUS_PPI  *This,
    IN      EFI_SMBUS_DEVICE_ADDRESS   SlaveAddress,
    IN      EFI_SMBUS_DEVICE_COMMAND   Command,
    IN      EFI_SMBUS_OPERATION       Operation,
    IN      BOOLEAN                  PecCheck,
    IN OUT     UINTN                  *Length,
    IN OUT     VOID                   *Buffer
);
typedef struct {
    PEI_SMBUS_PPI_EXECUTE_OPERATION Execute;
    PEI_SMBUS_PPI_ARP_DEVICE        ArpDevice;
} EFI_PEI_SMBUS_PPI;
  
```

Figure 15.9 Instance of a PPI

图 15.9 PPI 事例

创建一个 PPI 需要注意的地方是，PPI 像一个 EFI protocol 它有服务程序成员和（或）静态数据。PPI 通过一个 GUID 号命名并且有数个例程。例如，SMBUS PPI 能实现在 ICH/其它供应商集成在 SI0/或其它元件中的 SMBUS 控制器。图 15.10 阐述了 Intel 南桥中 SMBUS PPI 的一个例程。

```
#define SMBUS_R_HD0 0xEFA5
#define SMBUS_R_HBD 0xEFA7

EFI_PEI_SERVICES          *PeiServices;
SMBUS_PRIVATE_DATA        *Private;
UINT8 Index, BlockCount   *Length;
UINT8                      *Buffer;

BlockCount = Private->CpuIo.IoRead8 (
    *PeiServices, Private->CpuIo, SMBUS_R_HD0);
if (*Length < BlockCount) {
    return EFI_BUFFER_TOO_SMALL;
} else {
    for (Index = 0; Index < BlockCount; Index++) {
        Buffer[Index] = Private->CpuIo.IoRead8 (
            *PeiServices, Private-
>CpuIo, SMBUS_R_HBD);
    }
}
```

Figure 15.10 Code that Supports a PPI Service

图 15.10 支持 PPI 服务程序的代码

第十六章

集成在一起 - 固件仿真

专家就是一个犯了他那个狭小领域所有可以犯的错误的人。

—Niels Bohr

在前面的章节里，固件初始化的不同阶段已经介绍了。另外，还描述了在目标平台上可以实现的一些模块（组件）。还有，很明显的，大多数的 EFI 固件接口并没有直接与硬件打交道，取而代之的是与管理硬件的底层组件交互。传统上来说，固件开发不可能离开电路仿真器(ICE)或者硬件调试器而进行的。考虑到 EFI 的固件中只有非常少的组件直接与硬件打交道，我们就可以在一个标准的操作系统环境内对固件中的绝大多数组件进行仿真开发。

在 EFI 的可执行实例中，有一个叫做“NT32”目标平台。（目标平台是 EFI 编译时候的一个选项，一般都可以做为独立的 flash，烧写到 bios 中去。但是这个 nt32 不是用来烧写的，而是一个仿真器。）在这个环境中，我们可以把大多数固件模块作为一个应用程序在操作系统中执行一样来运行，并且可以很方便进行开发和调试工作。大多数的固件代码最初就是利用这个环境下的编译器调试器在没有真实硬件的情况下写成的。当然这个仿真环境也有其局限性，毕竟还有一些模块必须与硬件交互，这些模块很难在仿真环境中实现，我们在本章的后面会进行相关的讨论。图 16.1 显示一个固件仿真环境下的 EFI Shell 运行在某个操作系统环境中的情形。

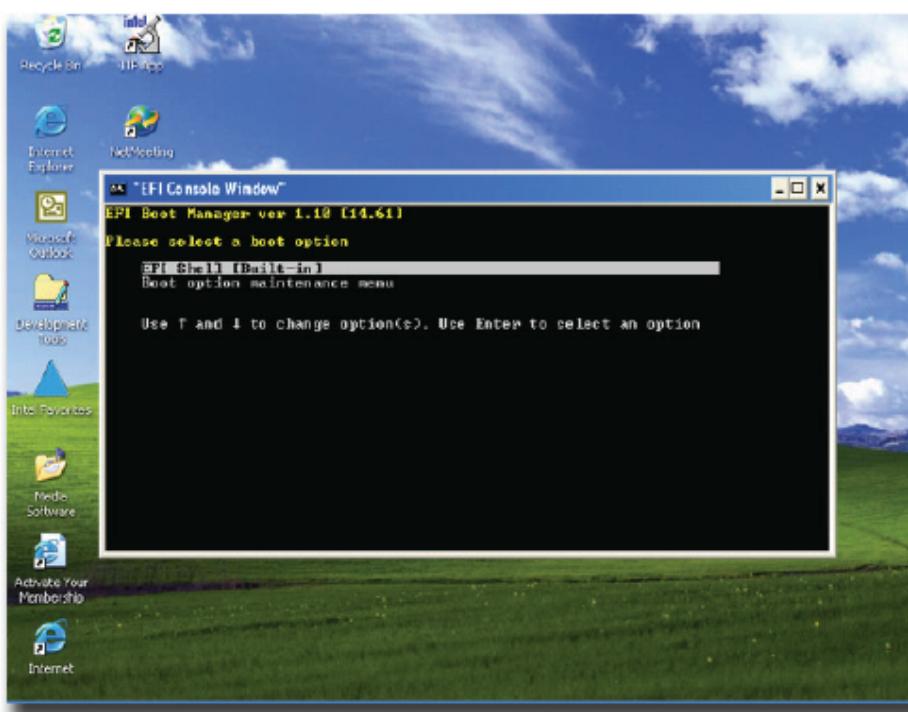


图 16.1 An Emulation Environment Contained within an Operating System Environment

虚拟平台

NT32 平台不需要知晓硬件细节，而是使用操作系统的应用程序接口来实现对硬件平台的抽象。图 16.2 显示了这个平台是如何启动的。它作为正常启动进程的一部分，最终以启动应用程序的方式启动固件仿真平台。对多数开发者来说，创建和执行这个 NT32 仿真环境变得非常简单，不过是在一个标准的硬件平台上启动一个熟知的操作系统，然后象创建和执行普通的应用程序一样来启动这个仿真环境。

www.BIOSREN.COM

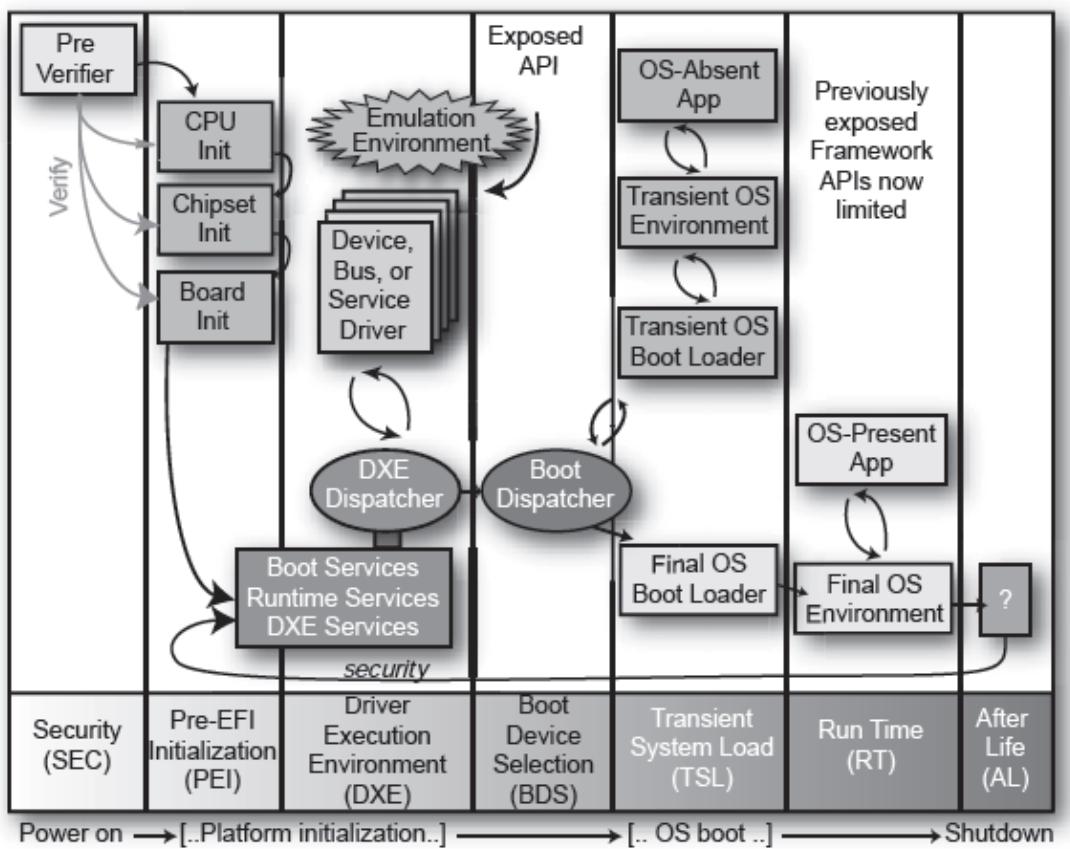


图 16.2 The Normal Boot Process Launching an Operating System that Will Launch the Emulation Environment

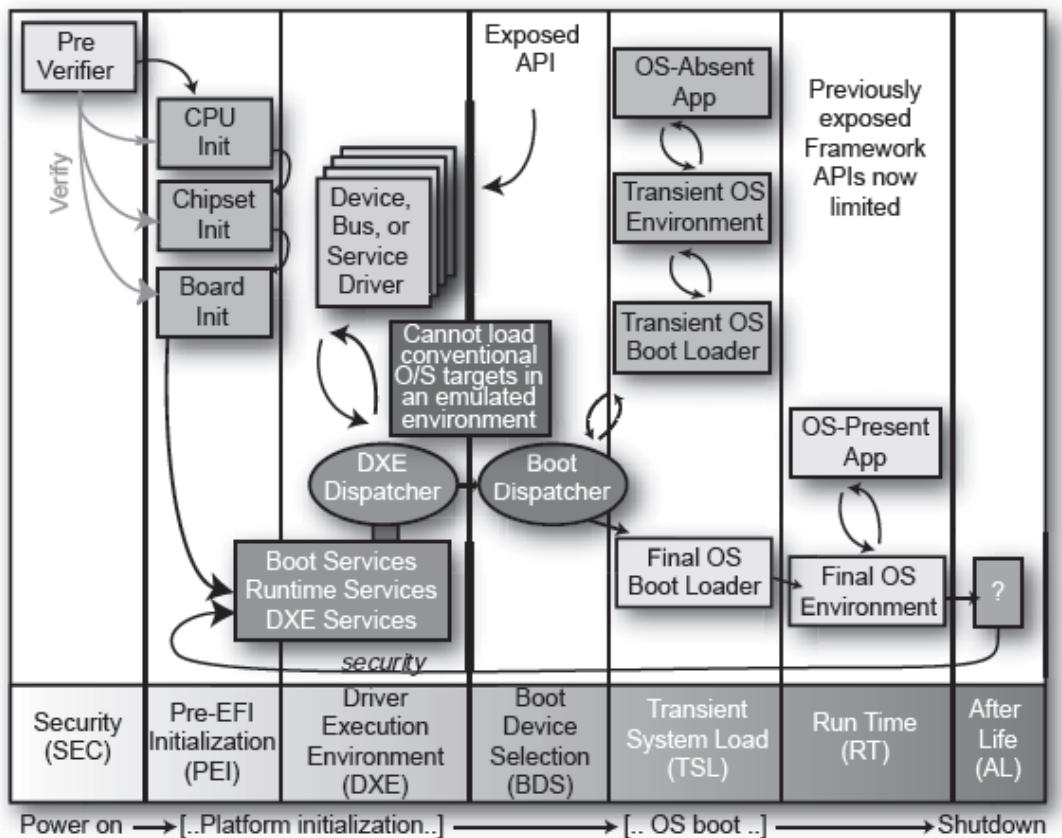


图 16.3 The Firmware Emulation Environment Itself

图 16.3 所示的时间表显示了所仿真的固件环境的运行时序，它可以包括固件环境执行的所有阶段。当然在仿真环境下不需要进行硬件初始化，某些操作只是被仿真而已。比如内存的初始化在真实的平台上将比仿真环境下需要更多的操作。

仿真固件的各个阶段

仿真环境有以下几个不同的阶段

为仿真环境建立 WinNT thunk。

这个阶段产生一种可能，即通过固件部分能涉及到一些“硬件”部分。这是通过联合和，操作系统底层 api 调用相联系的可见的固件部分，来实现的。

图 16.4 提供了若干个固件的构件与操作系统本地调用相关联的例子比如要创建一个文件，我们可以创建一个固件调用接口(例如 WinNTCreateFile)去调用操作系统的名为"CreateFile"的系统调用。下面的例子代码解释了如何将固件调用接口连接到 Windows 的系统调用上，类似的机制可以类似地应用到任何操作系统上。

```
typedef struct {
    UINT64 Signature;
```

```
//  
// Win32 Process APIs  
  
//  
WinNtGetProcAddress GetProcAddress;  
WinNtGetTickCount GetTickCount;  
WinNtLoadLibraryEx LoadLibraryEx;  
WinNtFreeLibrary FreeLibrary;  
WinNtSetPriorityClass SetPriorityClass;  
WinNtSetThreadPriority SetThreadPriority;  
WinNtSleep Sleep;  
WinNtSuspendThread SuspendThread;  
WinNtGetCurrentThread GetCurrentThread;  
WinNtGetCurrentThreadId GetCurrentThreadId;  
WinNtGetCurrentProcess GetCurrentProcess;  
WinNtCreateThread CreateThread;  
WinNtTerminateThread TerminateThread;  
WinNtSendMessage SendMessage;  
WinNtExitThread ExitThread;  
WinNtResumeThread ResumeThread;  
WinNtDuplicateHandle DuplicateHandle;  
  
//  
// Win32 Mutex primitive  
  
//  
WinNtInitializeCriticalSection InitializeCriticalSection;  
WinNtEnterCriticalSection EnterCriticalSection;  
WinNtLeaveCriticalSection LeaveCriticalSection;  
WinNtDeleteCriticalSection DeleteCriticalSection;  
WinNtTlsAlloc TlsAlloc;  
  
WinNtTlsFree TlsFree;  
WinNtTlsSetValue TlsSetValue;  
WinNtTlsGetValue TlsGetValue;  
WinNtCreateSemaphore CreateSemaphore;  
WinNtWaitForSingleObject WaitForSingleObject;  
WinNtReleaseSemaphore ReleaseSemaphore;  
  
//  
// Win32 Console APIs  
  
//  
WinNtCreateConsoleScreenBuffer CreateConsoleScreenBuffer;  
WinNtFillConsoleOutputAttribute FillConsoleOutputAttribute;  
WinNtFillConsoleOutputCharacter FillConsoleOutputCharacter;  
WinNtGetConsoleCursorInfo GetConsoleCursorInfo;
```

```
WinNtGetNumberOfConsoleInputEvents GetNumberOfConsoleInputEvents;
WinNtPeekConsoleInput PeekConsoleInput;
WinNtScrollConsoleScreenBuffer ScrollConsoleScreenBuffer;
WinNtReadConsoleInput ReadConsoleInput;
WinNtSetConsoleActiveScreenBuffer SetConsoleActiveScreenBuffer;
WinNtSetConsoleCursorInfo SetConsoleCursorInfo;
WinNtSetConsoleCursorPosition SetConsoleCursorPosition;
WinNtSetConsoleScreenBufferSize SetConsoleScreenBufferSize;
WinNtSetConsoleTitleW SetConsoleTitleW;
WinNtWriteConsoleInput WriteConsoleInput;
WinNtWriteConsoleOutput WriteConsoleOutput;

//  
// Win32 File APIs  
//
WinNtCreateFile CreateFile;
WinNtDeviceIoControl DeviceIoControl;
WinNtCreateDirectory.CreateDirectory;
WinNtRemoveDirectory.RemoveDirectory;
WinNtGetFileAttributes.GetFileAttributes;
WinNtSetFileAttributes.SetFileAttributes;
WinNtCreateFileMapping.CreateFileMapping;
WinNtCloseHandle.CloseHandle;
WinNtDeleteFile.DeleteFile;
WinNtFindFirstFile.FindFirstFile;
WinNtFindNextFile.FindNextFile;
WinNtFindClose.FindClose;
WinNtFlushFileBuffers.FlushFileBuffers;
WinNtGetEnvironmentVariable.GetEnvironmentVariable;
WinNtGetLastError.GetLastError;
WinNtSetErrorMode.SetErrorMode;
WinNtGetStdHandle.GetStdHandle;
WinNtMapViewOfFileEx.MapViewOfFileEx;
WinNtReadFile.ReadFile;
WinNtSetEndOfFile.SetEndOfFile;
WinNtSetFilePointer.SetFilePointer;
WinNtWriteFile.WriteFile;
WinNtGetFileInformationByHandle.GetFileInformationByHandle;
WinNtGetDiskFreeSpace.GetDiskFreeSpace;
WinNtGetDiskFreeSpaceEx.GetDiskFreeSpaceEx;
WinNtMoveFile.MoveFile;
WinNtSetFileTime.SetFileTime;
WinNtSystemTimeToFileTime.SystemTimeToFileTime;
```

```
//  
// Win32 Time APIs  
  
//  
WinNtFileTimeToLocalFileTime    FileTimeToLocalFileTime;  
WinNtFileTimeToSystemTime     FileTimeToSystemTime;  
WinNtGetSystemTime      GetSystemTime;  
WinNtSetSystemTimeSetSystemTime;  
WinNtGetLocalTime      GetLocalTime;  
WinNtSetLocalTime      SetLocalTime;  
WinNtGetTimeZoneInformation   GetTimeZoneInformation;  
WinNtSetTimeZoneInformation  SetTimeZoneInformation;  
WinNttimeSetEvent      timeSetEvent;  
WinNttimeKillEvent      timeKillEvent;  
  
//  
// Win32 Serial APIs  
  
//  
WinNtClearCommError  ClearCommError;  
WinNtEscapeCommFunctionEscapeCommFunction;  
WinNtGetCommModemStatus  GetCommModemStatus;  
WinNtGetCommState      GetCommState;  
WinNtSetCommState      SetCommState;  
WinNtPurgeComm        PurgeComm;  
WinNtSetCommTimeoutsSetCommTimeouts;  
  
WinNtExitProcess      ExitProcess;  
WinNtSprintf SPrintf;  
WinNtGetDesktopWindow  GetDesktopWindow;  
WinNtGetForegroundWindow GetForegroundWindow;  
WinNtCreateWindowEx CreateWindowEx;  
WinNtShowWindow        ShowWindow;  
WinNtUpdateWindow      UpdateWindow;  
WinNtDestroyWindow     DestroyWindow;  
WinNtInvalidateRectInvalidateRect;  
WinNtGetWindowDC       GetWindowDC;  
WinNtGetClientRect     GetClientRect;  
WinNtAdjustWindowRectAdjustWindowRect;  
WinNtSetDIBitsToDevice SetDIBitsToDevice;  
WinNtBitBlt          BitBlt;  
WinNtGetDC            GetDC;  
WinNtReleaseDC        ReleaseDC;  
WinNtRegisterClassEx RegisterClassEx;  
WinNtUnregisterClass UnregisterClass;
```

```
WinNtBeginPaintBeginPaint;
WinNtEndPaint EndPaint;
WinNtPostQuitMessage PostQuitMessage;
WinNtDefWindowProc DefWindowProc;
WinNtLoadIcon LoadIcon;
WinNtLoadCursor LoadCursor;
WinNtGetStockObject GetStockObject;
WinNtSetViewportOrgEx SetViewportOrgEx;
WinNtSetWindowOrgEx SetWindowOrgEx;
WinNtMoveWindow MoveWindow;
WinNtGetWindowRect GetWindowRect;
WinNtGetMessage GetMessage;
WinNtTranslateMessage TranslateMessage;
WinNtDispatchMessage DispatchMessage;
WinNtGetProcessHeap GetProcessHeap;
WinNtHeapAlloc HeapAlloc;
WinNtHeapFree HeapFree;
} EFI_WIN_NT_THUNK_PROTOCOL;
```

图 16.4 Thunk Protocol that Associates Some Firmware Names with Operating System APIs

(译者注：以上API即为图16.4，在此将其转换为文字)

为仿真平台创建特定的 EFI 硬件接口模块

在图 16.5 中，我们将 EFI_SERIAL_IO_PROTOCOL 接口与操作系统所提供的调用联系起来。在这个例子中，这个具体的平台函数将被指向模拟环境。

```
SerialIo.Revision      = SERIAL_IO_INTERFACE_REVISION;
SerialIo.Reset          = WinNtSerialIoReset;
SerialIo.SetAttributes  = WinNtSerialIoSetAttributes;
SerialIo.SetControl     = WinNtSerialIoSetControl;
SerialIo.GetControl     = WinNtSerialIoGetControl;
SerialIo.Write           = WinNtSerialIoWrite;
SerialIo.Read            = WinNtSerialIoRead;
SerialIo.Mode            = SerialIoMode;
```

图 16.5 Establishing an EFI API to Call Platform Specific Operations

平台相关的函数（例如仿真平台）负责处理上层对 EFI 接口的调用，然后通过前面创建的 WinNTThunk 接口调用相应的操作系统接口。

在图 16.6 中给出了在一个 EFI 接口处理函数中可能出现的一些 WinNTThunk 调用。

```
//  
// Example of reading from a file  
//  
Result = WinNtThunk->ReadFile (  
    NtHandle,  
    Buffer,  
    (DWORD)*BufferSize,  
    &BytesRead,  
    NULL  
);  
  
//  
// Example of resetting a serial device  
//  
WinNtThunk->PurgeComm (  
    NtHandle,  
    PURGE_TXCLEAR | PURGE_RXCLEAR  
);  
  
//  
// Example of getting local time components  
//  
WinNtThunk->GetLocalTime (&SystemTime);  
WinNtThunk->GetTimeZoneInformation (&TimeZone);
```

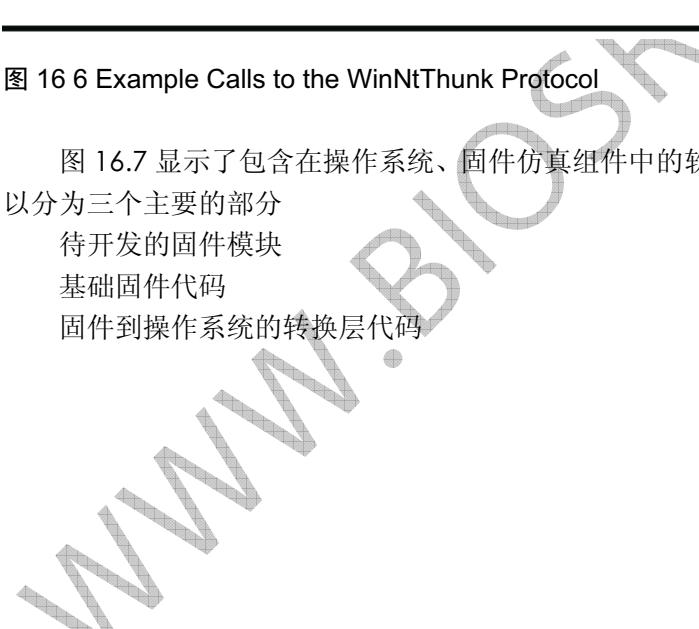
图 16.6 Example Calls to the WinNtThunk Protocol

图 16.7 显示了包含在操作系统、固件仿真组件中的软件逻辑以及它们之间的交互，这些可以分为三个主要的部分

待开发的固件模块

基础固件代码

固件到操作系统的转换层代码



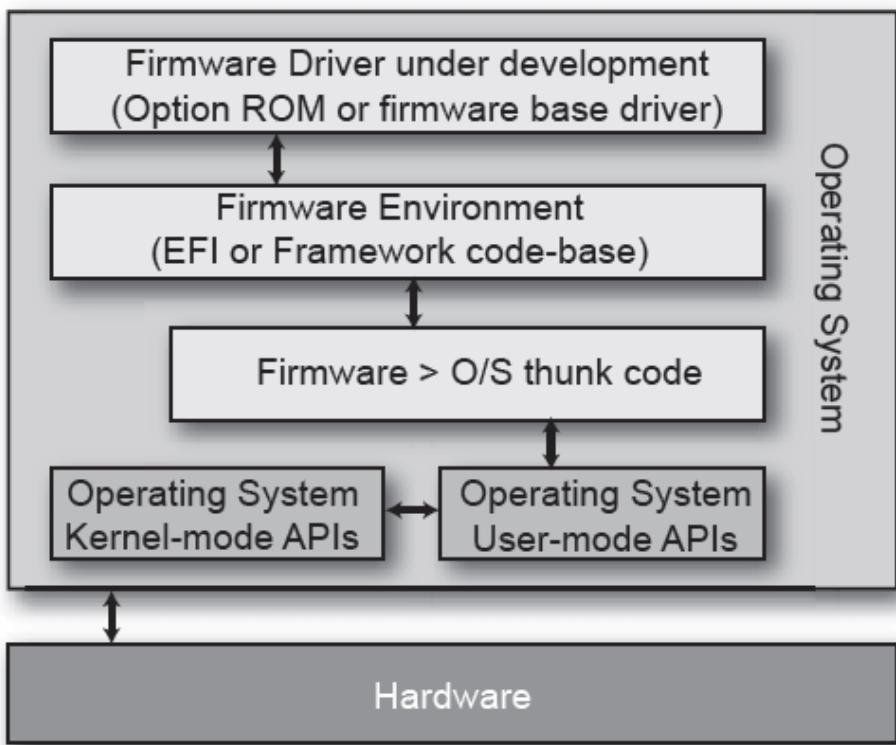


图 16.7 Firmware Emulation Software Logic Flow

直接访问硬件

通过前面的例子，我们已经了解到底层的固件可以利用操作系统的调用。不过仿真环境本质上来说只是操作系统的一个应用程序而已，它无法使用操作系统的所有调用。大多数操作系统为防止应用程序的错误导致整个操作系统崩溃，大都采用了将用户空间和内核空间分隔的设计，这种设计使得操作系统在检测到错误时，可以简单地把出问题的用户进程杀掉而不影响操作系统的其余部分。

我们可以在样例实现中引入一些扩展来为仿真环境提供更强的能力。比如可以构造一个操作系统内核驱动来访问某些不能使用的系统函数，这样做会绕开一些操作系统内建的安全机制，如果不小心的话可能会造成系统的崩溃。使用内核驱动的仿真环境可以通过驱动来获取运行平台的某些硬件资源，再通过接口方式提供给上层调用，在仿真环境内给上层提供更多的硬件资源。

图 16.8 显示了若干个模块的相互关系

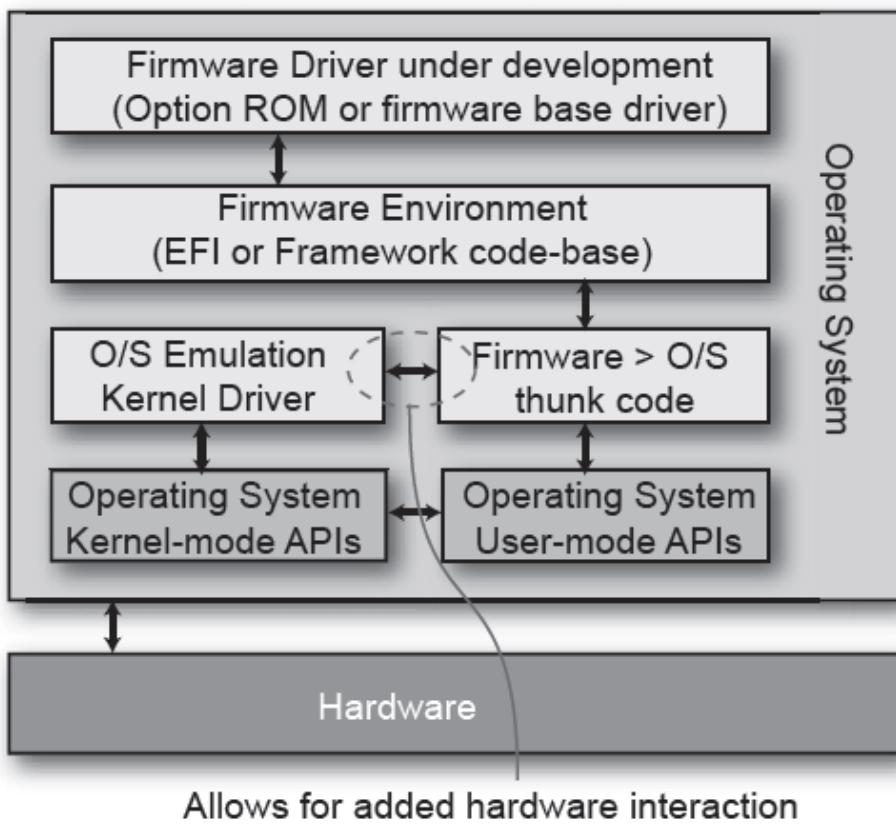


图 16.8 Software Flow for Hardware Enhanced Firmware Emulation

第十七章

Compatibility Support Module (CSM)

古制迁易，让渡於新
—Alfred Tennyson

Framework 定义了新的固件(firmware)模型，并引进了非常多的优点，如支持模块化、多团队并行开发，极大地缩短了产品推向市场的时间。但是，Framework 没有定义传统 OS 需要的接口(传统的中断调用)。由于这个原因，Framework 要保证大量 legacy 软件 Option ROM 仍然能正常工作具有极大挑战。

Framework 的设计需要确保大量 legacy 程序能平滑移植和运行。legacy 软件运行需要调用传统 BIOS 提供的中断服务，为了使 Framework 能提供类似的中断服务，CSM 应运而生。

CSM 完成将 EFI 下面的相关信息转换成传统 BIOS 下的信息，并提供在操作系统引导和运行过程中所需的传统 BIOS 中断服务。此外，CSM 还使传统的 Option ROM 可以在 EFI 环境下运行，简言之，CSM 使 EFI 具备了兼容传统 BIOS 的能力。

CSM:传统和革新的桥梁

计算机产业中一项新技术的引进和推广往往都需要一个过渡周期。Framework 的引入也需要一定时间，或许很短，或许很长。CSM 模块作为 EFI BIOS 和传统 BIOS 之间的桥梁，显著缩短了这个过渡时间。CSM 模块使得 EFI 下可以调用传统 BIOS 的各种中断服务，因此可以引导各种传统操作系统，如 DOS, Windows2000, Windows XP 等。同样，CSM 也支持从传统 Option ROM 控制下的设备引导 EFI 原生操作系统(EFI-aware OS)。CSM 的出现使得 IBV 可以将传统 BIOS 和 EFI BIOS 无缝对接，而不需要丢弃传统 BIOS 下的很多功能和成果。因此，直接或间接地，CSM 在传统 BIOS 向 EFI BIOS 过渡过程中扮演着至关重要的角色。

CSM 利用 Framework 完成硬件平台初始化、设备枚举及系统启动路径的选择。EFI 中添加了 CSM 的组件——EFI 兼容模块(EfiCompatibility)，以支持传统操作系统和传统的 Option ROMs。EfiCompatibility 模块、IBV 提供的 Compatibility16BIOS 模块、Compatibility16SMM 模块一起构成了整个 CSM。在传统 BIOS 向 EFI BIOS 转变过程中，支持传统 Option ROMs 的时间会比传统 OS 的时间长(原因是 EFI 原生的操作系统有可能会从运行传统 Option ROM 的设备中引导)。由于 CSM 是模块化组件，如果不需要，可以轻松从整个 EFI 中移除。图 17.1 展示了在 Framework 下 CSM 模块的工作原理：

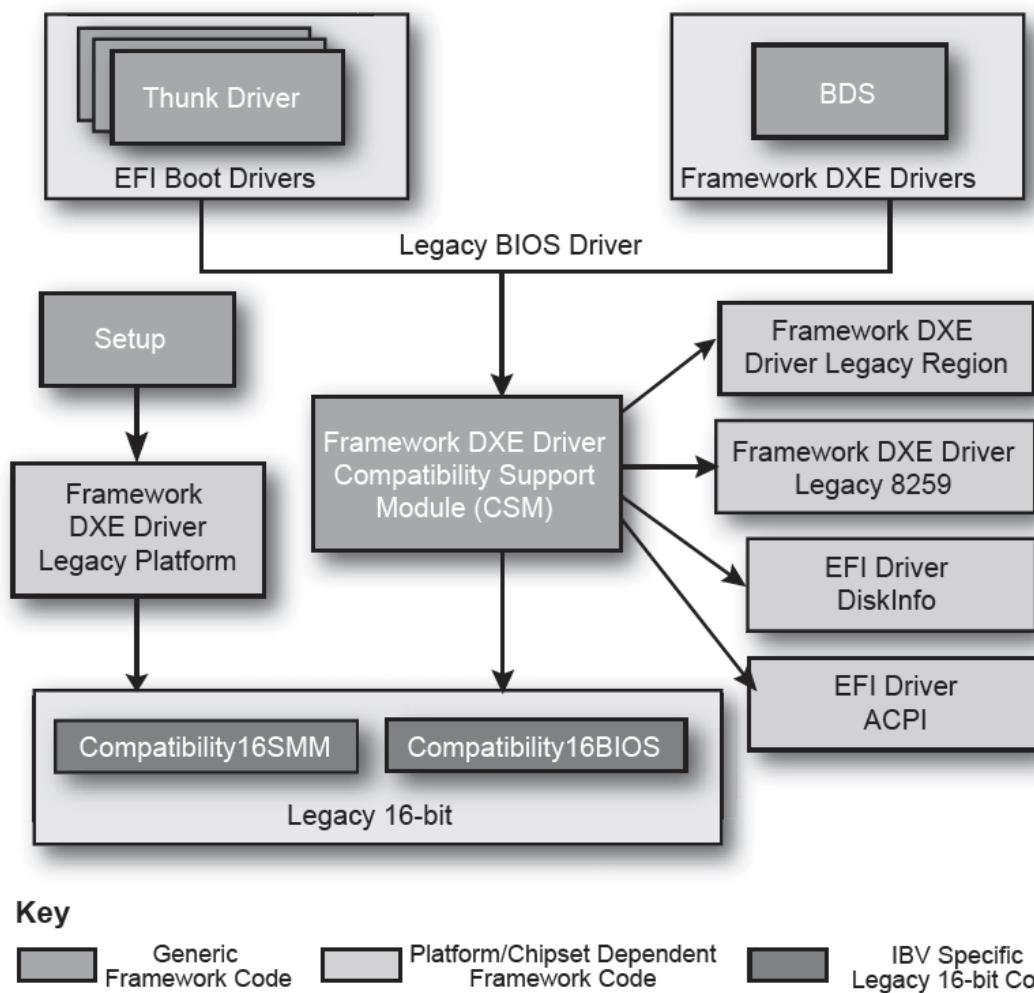


Figure 17.1 Legacy System under the Framework

EFI 执行过程中,在 BDS(Boot Device Select)阶段之前只完成少部分的系统初始化和配置,不显示和执行 Option ROM。只有到了 BDS 阶段,根据以下三种信息确定是否需要调用 CSM 模块:

- 目标操作系统
- 选择启动的设备
- 设备的初始化策略

由于传统操作系统(non-EFI-aware)需要调用传统 BIOS 提供的中断服务。在 EFI BIOS 中必须调用 CSM 模块以支持传统操作系统以及在引导传统操作系统过程中需要执行的传统 Option ROM。

如果一个 OS 启动设备本身依赖于传统 Option ROM, 不管目标操作系统是传统操作系统还是 EFI 原生的操作系统,都需要调用 CSM 模块(这也是上文为什么说“在传统 BIOS 向 EFI BIOS 转变过程中, 支持传统 Option ROMs 的时间会比传统 OS 的时间长”的原因)。

如果系统的启动策略是初始化所有的设备, 只要其中有一个非引导设备(non-boot device)涉及到传统 Option ROM, 都必须执行 CSM 模块以支持该设备。

CSM 架构

CSM 模块使得 EFI BIOS 具备了加载引导传统操作系统和执行传统 Option ROM 的能力。它主要包括以下五个组件：

- EFI 兼容模块(EfiCompatibility, 简称 CSM32)
- 传统 BIOS 兼容模块 (Compatibility16BIOS, 简称 CSM16)
- 传统 SMM 驱动兼容模块 (Compatibility16SMM)
- 保护模式/实模式切换模块 (Thunk and ReverseThunk)
- 传统 Option ROM

上述五个组件中，除传统 Option ROM 可能独立于 EFI firmware 之外，其余组件都被包含在 EFI firmware 里面。

CSM 主要有两种应用场景：第一种，引导传统 OS；第二种，从运行传统 Option ROM 的设备中引导 EFI 原生的 OS。引导传统操作系统是传统的应用，业界的可能趋势是当传统 OS 被 EFI 原生的 OS 替代后，仍需要支持传统 Option ROM。更明确地说，通过传统 Option ROM 引导 EFI 原生的 OS 功能是整个传统 OS 引导功能的子集。EFI 架构充分考虑了这个过渡功能，并且如果需要，允许在未来将其移除。

EFI 兼容模块(CSM32)

EFI 兼容模块运行在 32 位保护模式下，和 EFI 本身进行交互，调用执行由 CSM16 提供的 16 位实模式下的函数接口及传统 BIOS 中断服务。CSM32 模块由 Framework 提供，主要包括以下四种类型的 EFI 驱动：

- 硬件平台无关的驱动
- 平台无关的传统组件驱动，如传统的 8259 可编程中断控制器驱动
- 芯片组相关的驱动，如 ICH7
- 硬件平台相关驱动，如 PCI 中断路由驱动

传统 BIOS 兼容模块(CSM16)

CSM16 模块运行在 16 位实模式下，提供传统 BIOS 中断服务调用。主要由 INT13、INT19、INT15 等中断服务程序组成。除中断服务程序外，CSM16 还包括和 CSM32 交互的接口程序。CSM16 一般由 IBV 提供。

传统 SMM 兼容模块(Compatibility16SMM)

Compatibility16SMM 提供传统的 SMM 服务，作为 EFI SMM 驱动的补充，是可选的，由 IBV 或 OEM 厂家提供。

保护模式/实模式切换模块 (Thunk and ReverseThunk)

该模块提供了保护模式/实模式相互转换的驱动。CSM32 通过 Thunk 程序从 EFI 32 位保护模式切换到传统的 16 位实模式，CSM16 通过 ReverseThunk 从 16 位实模式切换回 32 位保护模式。该模块由 Framework 提供。

传统 Option ROM (Legacy Option ROM)

传统 Option ROM 虽然不被包含在 CSM 模块之中，但从整个 BIOS 系统层面来看，它是一个非常重要的组件，它和 CSM 模块联合工作，共同完成了传统 OS 的加载引导。

Option ROM 一般由设备厂家提供，有两种方式：一种是集成打包到 EFI firmware 里面，另一种是放在对应设备的 ROM 里。

EFI 兼容模块(CSM32)

CSM32 主要由以下几大协议组成：

- Legacy BIOS Protocol
- Legacy BIOS Platform Protocol
- Legacy Region Protocol
- Legacy 8259 Protocol
- Legacy Interrupt Protocol

Legacy BIOS Protocol 是 CSM 中最主要的协议，与硬件平台无关。Legacy BIOS Platform Protocol 是平台相关的，Chipset 相同但硬件平台不同，该协议内部驱动实现会有差异。Legacy Region Protocol 是 chipset 相关的，用于对内存区域 0xc0000-0xFFFFF 的读写控制。Legacy Region Protocol 还可以选择对 0xc0000-0xFFFFF 某一指定地址段进行控制，以防止对其他“混杂”地址段(alias address)造成误写。Legacy 8259 Protocol 和硬件平台、chipset 都无关，可以在 32 位保护模式或者 16 位实模式下对 8259 可编程中断控制器进行控制、实现中断屏蔽、电平触发或边沿触发工作模式编程。Legacy interrupt Protocol 则依赖于具体的 chipset，用来给 PCI 设备分配中断路由。

除上面介绍的几个协议提供的功能外，EFI 兼容模块还包括许多模拟传统中断服务的驱动，如 UGA 设备中断 INT10，键盘中断 INT16 以及块设备的 I/O 访问中断服务。

声明：本资料仅供学习参考用，未经授权，不得进行任何复制、转载、传播、出版等非法之用途，请在学习后 24 小时内删除，否则后果自负。

一些 Option ROM 运行时需要使用 INT10，因此 UGA 驱动需要运行在 VGA 模式以支持 INT10。该驱动将 EFI 下的控制台输出数据转化成对应的 VGA 格式。该驱动基于以下假设：所有的 INT10 中断服务功能都已实现，也就是说 Option ROM 可以直接访问 VGA 所有寄存器及显存。UGA 设备支持 VGA 模式并且根据要求 UGA 和 VGA 两种模式可以相互切换。

在进入传统启动路径阶段之前，由于 CSM16 没有调用，不能处理 USB 键盘的中断服务请求，因此需要在 EFI 下模拟 INT16 中断服务。将 INT16 中断服务转换成 EFI 服务，接收数据后以 INT16 的格式返回。

在 EFI 阶段还需要 block I/O 驱动，以便对软盘、硬盘等 BAID(BIOS Aware IPL Device)设备进行 I/O 读写访问。该驱动将 EFI 下的块设备的 I/O 读写请求转换成传统 BIOS 下 INT13 请求，并将结果返回到 EFI 环境。此外，对于由传统 Option ROM 控制的设备，如 SCSI Option ROM 控制的 SCSI 设备，需要一个独立的 Block I/O 读写驱动。

Legacy Bios Driver

Legacy Bios Driver 是整个 CSM 的主模块，其余都是配套支持模块。Legacy Bios Driver 的主要功能包括：

- 初始化它本身及加载 CSM16 模块
- 确定启动设备
- 加载 CSM32 中的其他驱动程序
- 加载保护模式/实模式切换模块
- 初始化和 CSM16 交互的接口
- 寻找、加载及调用板载/非板载的 Option ROM
- 加载引导 OS

Legacy BIOS Driver 分析当前系统的数据，调用 EFI 提供的 APIs 收集 CSM16 所需的系统信息，并将这些信息转化成 CSM16 的数据结构。CSM16 利用这些数据结构初始化位于系统内存 0x400-0x500 的传统 BIOS 数据区(BDA)、扩展 BIOS 数据区(EBDA)和 CMOS。Legacy BIOS Protocol 还会使用这些数据来重设那些传统设备的资源。.

EFI 对硬件设备的枚举、配置是比较简单、低级的，不对诸如串行口或并行口等传统设备分配中断分配。但 CSM16 要求这些传统设备具有正确的中断，因此 Legacy BIOS Driver 需要对这些传统设备进行重新配置以满足这个要求。Legacy BIOS Driver 有两种实现选择：简单版和完全版。对于那些通过传统 Option ROM 引导 EFI 原生操作系统的应用，可以选择简单版。而对于需要引导传统操作系统的应用，则选择完全版。

Legacy BIOS Protocol

Legacy BIOS Protocol 和 Legacy BIOS Driver 的初始化程序构成了整个 CSM 的基础。Legacy BIOS Protocol 由以下一系列函数组成：

BootUnconventionalDevice() 函数用于从非传统设备引导系统，如 PARTIES。

CheckPciRom() 函数用于检查 PCI 设备中是否有传统 Option ROM，以及检查该设备是

否只有传统 Option ROM 而没有 EFI Option ROM。此外，还会检查传统操作系统启动设备有效性。

CopyLegacyRegion()函数用于 EFI 下拷贝数据到函数 **GetLegacyRegion()**指定的区域。

FarCall86()使 32 位保护模式下的程序使用长调来调用 16 位实模式下的程序。该函数通过调用 Thunk 程序切换到 16 位实模式，将输入参数指定的数据区载入 CPU 寄存器，然后使用长调。长调用返回后，使用 CPU 寄存器的值更新数据区，并切换回 32 位保护模式。

GetBbsInfo()允许外部驱动访问 CSM32 中的 BBS 数据结构。该函数主要被 BIOS setup 程序调用。

GetLeagcyRegion()允许在内存区域 0xe0000-0xFFFFF 地址段分配内存。

Int86()使 32 位保护模式程序调用 16 位软中断程序，比如 INT 16。该函数通过调用 Thunk 程序切换到 16 位实模式，将输入参数指定的数据区载入 CPU 寄存器，然后执行相应的软中断程序。中断程序返回后，使用 CPU 寄存器的值更新数据区，并切换回 32 位保护模式。

InstallPciRom()函数的功能是将传统 Option ROM 装载到内存区域 0xc0000-0xFFFFF。

LegacyBoot()用来引导传统 OS，实现了 CSM 的大部分主要功能，由于引导传统 OS 的一些要求，执行该函数可能会关闭 EFI 相关功能。该函数最终调用 CSM16 模块以执行传统 INT19 中断服务，如果从选定的设备启动操作系统失败，CSM16 会将控制权返回给该函数，这种情况下，如果原来的 EFI 环境是保留的，或者在 EFI 启动和传统启动方式相互切换时 EFI 环境可以被恢复，则函数 **LegacyBoot()**会将控制权返回给调用者。

PrepareToBootEfi()函数主要完成 EFI 原生操作系统的引导环境准备。执行包括给传统引导设备分配驱动号在内的 **LegacyBoot()**函数的部分功能。

ShadowAllLegacyOproms()函数加载运行所有传统 Option ROMs，运行 Option Rom 之前会卸载设备原有的 EFI 驱动，因此，如果需要，执行完该函数后必须重新连接 EFI 驱动到设备上。

UpdateKeyboardLedStatus()函数用来把 EFI 下的键盘 LED 状态信息同步到传统 BIOS 数据区。该函数不对键盘进行操作，而调用 CSM16 中涉及键盘 LED 状态的函数来维护键盘灯的状态信息。

Legacy BIOS Platform Protocol

Legacy BIOS Platform Protocol可以根据硬件平台的具体配置及 OEM 要求对 CSM 进行客户化定制，其中包含了多个函数以支持对 CSM 进行客户定制。

GetPlatformHandle()搜索指定实体的所有句柄(handles)，并返回按照优先级分类的特定类型的句柄队列。不同类型的句柄包括 VGA 设备句柄、IDE 控制器句柄、ISA 总线控制器句柄以及 USB 设备句柄等。

GetPlatfromInfo()函数用来获取平台特定的二进制目标文件或数据，包括 MP table，OEM 特定的 16 位或 32 位的映像文件、TPM 二进制文件等。

GetRoutingTable()函数为中断路由表获取平台相关的 PCI 中断路由信息。

PlatformHooks()函数执行平台相关的一些动作，如完成扫描 Option ROM 之前的一些特殊处理，映射物理设备无关的 Option ROM(PXE 驱动及 BIS)，以及 Option ROM 初始化后的一些处理等。

PrepareToBoot()函数完成传统操作系统的引导。

`SmmInit()`函数检查 EFI 固件卷(firmware volumes)中是否有 16 位模式下的 SMM 驱动模块(Compatibility16SMM)。如果找到，使用 EFI SMM 驱动注册 `CSM16 SMM` 模块。在 EFI 固件卷中可能存在多个 16 位 SMM 驱动模块，或完全没有。

`TranslatePirq()`函数将中断线信息写到对应的芯片组中断路由寄存器中，同时返回指定设备可用的中断号。

Legacy Region Protocol

Legacy Region Protocol 是芯片组相关的，用来对内存区域 0xc0000-0xFFFFF 进行读写控制。

`Decode()`函数完成对芯片组的初始化，使能或禁止对该内存区域的解码。

`Lock()`函数将指定区域设为只读(写保护)

`BootLock()`函数在引导传统 OS 之前调用，该函数将指定区域设为写保护以防止误操作其他“混杂”地址段(aliasated address)

`Unlock()`函数将指定内存区设为可读可写，同时可以防止误操作其他“混杂”地址段(aliasated address)

Legacy 8259 Protocol

该协议和具体 `chipset` 无关，在 32 位保护模式及 16 位实模式下完成对 8259 可编程控制器的控制。由以下函数组成：

`SetVectorBase()`函数初始化主、从 8259 可编程中断控制器的中断向量基址。在 EFI 环境下，主 8259 控制器的中断向量基址设为 0x1A0(INT68)，从 8259 控制器的中断向量基址设为 0x1C0(INT70)。在 16 位实模式下，主 8259 控制器的中断向量基址设为 0x20(INT08)，从 8259 控制器的中断向量基址设为 0x1C0(INT70)。32 位保护模式下主 8259 控制器的中断向量基址设为不同可以防止中断和 CPU 异常被覆盖。

`GetMask()`函数用来获取 EFI 环境下或 16 位实模式下当前主/从 8259 控制器的中断屏蔽信息及中断触发方式(边沿触发还是电平触发)

`SetMask()`函数完成在 32 位 EFI 环境下或 16 位传统环境下设置主/从 8259 控制器的中断屏蔽信息及中断触发方式。

`SetMode()`函数完成在 32 位 EFI 环境下或 16 位传统环境下设置主/从 8259 控制器的中断屏蔽信息及中断触发方式。`GetVector()`函数完成将某一硬中断号转换成对应的软中断号，如，在 EFI 下将 IRQ0 转换成 INT68。

`EnableIrq()`函数在 EFI 下使能某一个硬中断，非 CSM 驱动可以调用。

`DisableIrq()`函数在 EFI 下屏蔽某一个硬中断，非 CSM 驱动可以调用。

`GetInterruptLine()`函数读取指定 PCI 设备的配置空间并返回分配给该设备的中断号。

`EndOfInterrupt()`函数下发中断结束(EOI)命令给中断控制器。

Legacy Interrup Protocol

The Legacy Interrupt Protocol 协议和具体 `chipset` 相关，用以对 PCI 中断的编程控制。

`GetNumberPirqs()`函数返回 `chipset` 支持的所有 PCI 中断线号个数。

`GetLocation()`函数返回支持该协议的 PCI 设备的位置。

`ReadPirq()`函数读取指定 PIRQ (PCI 中断线) 寄存器并返回该寄存器的值。

`WritePirq()`函数填写数据到指定的 PIRQ 寄存器。

CSM16

CSM16 模块可以看成是一个没有开机自检 (POST: Power On Self Test) 和 Setup 配置界面的传统 BIOS。CSM16 模块设计目的是希望使 CSM16 模块通用与所有平台，也就是同样的 CSM16 模块可以用于桌面电脑、服务器和移动电脑。因此，CSM16 是独立于桥片和平台的。CSM16 模块不会配置任何底层硬件，然而它可以通过传统 BIOS 的接口来控制一些传统硬件，比如并口、串口、键盘、PS2 鼠标等。桥片的配置是由 EFI 或 EFI 兼容模块 (EfiCompatibility module) 完成。与桥片和平台无关的设计使 CSM16 模块得以尽可能多的重用，尽可能少的维护。

CSM16 模块包含全部的 runtime 服务和传统 BIOS 需要的软中断和硬中断服务程序。模块还必须支持与传统 BIOS 兼容的 BIOS 数据区 (BDA: BIOS Data Area)、扩展数据区 (EBDA: Extend BDA) 以及一些中断服务在 F000 内存段的固定的入口地址。此外，还提供一些用于 EFI 和 CSM16 之间互相通讯的 16 位程序。表 17.1 列举了 CSM16 模块提供的传统中断服务程序。

表 17.1 CSM16 模块提供的传统中断服务程序

中断	描述
0x02	NMI 不可屏蔽中断
0x05	屏幕打印
0x08	系统时钟中断 (IRQ0)
0x09	键盘中断 (IRQ1)
0x10	显卡 (显示) 服务
0x11	设备检测
0x12	基本内存大小检测
0x13	磁盘中断服务 (软盘、硬盘、光驱、U 盘等)
0x14	串口通讯服务
0x15	系统服务
0x16	键盘服务
0x17	并口打印机服务
0x18	系统启动管理服务
0x19	系统引导程序服务

0x1A	系统时间服务
0x1B	程序中断服务，等效于按下 Ctrl+Break
0x1C	定时器服务
0x1D	显示参数
0x1E	软驱控制参数
0x1F	字符矩阵
0x40	软盘服务
0x41	主硬盘控制器参数
0x46	从硬盘控制器参数
0x70	RTC 中断 (IRQ8)
0x74	PS2 鼠标中断 (IRQ12)
0x75	数字协处理器中断 (IRQ13)
0x76	主 IDE 硬盘控制器中断 (IRQ14)
0x77	从 IDE 硬盘控制器中断 (IRQ15)

CSM32 与 CSM16 之间的交互

CSM32 与 CSM16 的交互通过三个机制来完成：Compatibility16Table、Compatibility16 函数接口、Compatibility16 数据结构。图 17.2 描述了 CSM32 与 CSM16 之间的交互机制。

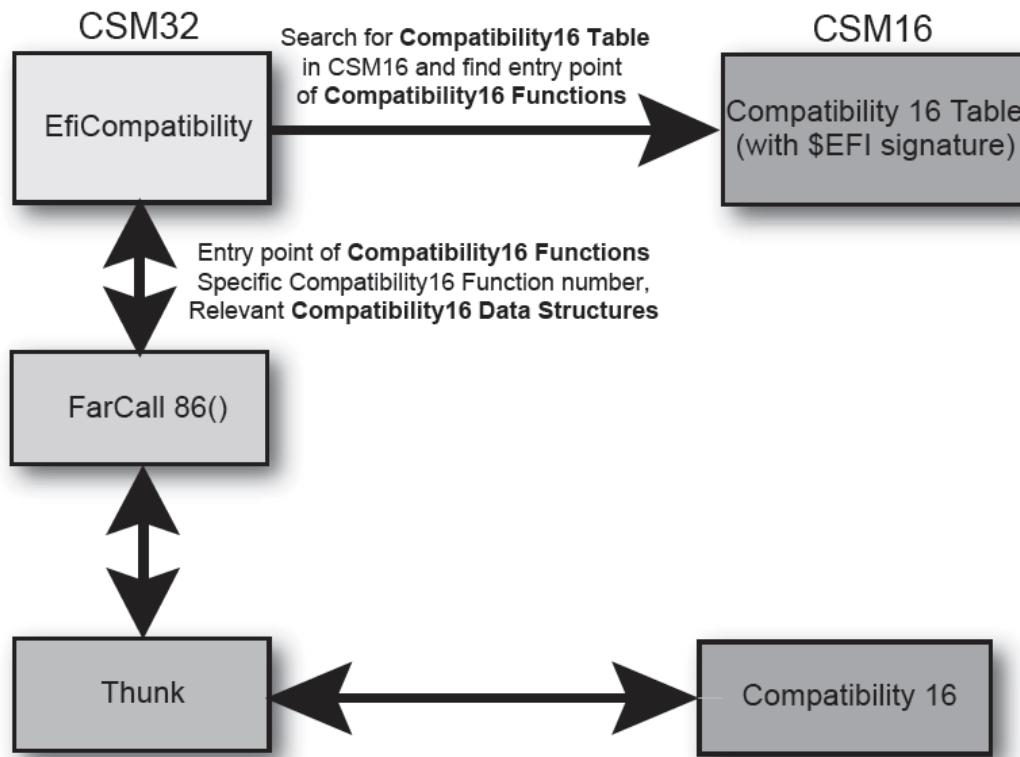


Figure 17.2 Communication between CSM32 and CSM16

CSM16Table

Compatibility16 Table 是 CSM16 模块中非常重要的部分，驻留在 0xE0000~0xFFFFF 这段内存中 16 字节对齐的某个位置。这个表中包含一些 CSM16 模块在创建时生成的静态信息。也包含一些动态数据，由 EFI 和 CSM16 模块在运行时填写。在 CSM16 创建时，会建立一个缺省的表。表头是一个标识字符串“\$EFI”，故又称此表为“\$EFI”表。表中最重要的两个数据是 Compatibility16CallSegment 和 Compatibility16CallOffset，这两个数据分别是 CSM16 模块中 Compatibility16 功能函数入口地址的段地址和偏移地址。EfiCompatibility 通过 Legacy BIOS 协议中的 FarCall86() 函数通过 thunk 远调这个入口来达到调用 Compatibility16 功能。在一个具体的 CSM 设计实现中 Compatibility16 表的格式称为 EFI_COMPATIBILITY16_TABLE。CSM16 的设计允许增加表项以满足未来的需求，但是不允许删除任何现有的表项。表 17.2 描述了 EFI_COMPATIBILITY16_TABLE 中各表项的含义。

表 17.1 CSM16 模块提供的传统中断服务程序

表项	偏移	描述
Signature	0x00	“\$EFI” 表明是表的开始。 注意字符的存放顺序，第 0 个字节是

		“!”，第 3 个字节是 “\$”
TableChecksum	0x04	表的检验值。整个表的每个字节的数据累加和是 0，表的长度在 TableLength 中指定。
TableLength	0x5	表的长度，以字节单位。
EFIMajorRevision	0x06	EFI 主版本号
EFIMinorRevision	0x07	EFI 次版本号
TableMajorRevision	0x08	本表主版本号
TableMinorRevision	0x09	本表次版本号
Reserved	0x0A	保留位
Compatibility16CallSegment	0x0C	CSM16 功能接口的入口地址的段地址
Compatibility16CallOffset	0x0E	CSM16 功能接口的入口地址的偏移地址
PnPInstallationCheckSegment	0x10	Pnp 结构的段地址
PnPInstallationCheckOffset	0x12	Pnp 结构的偏移地址
EFISystemTable	0x14	EFI 系统资源表的 32 位物理地址
OemIdStringPointer	0x18	OEM ID 字符串的 32 位物理地址
AcpiRsdPtrPointer	0x1C	ACPI RSD PTR 的 32 位地址。ACPI 表的实际数据依旧在 EFI 中原有位置。
OemRevision	0x20	用于 OEM 自定义
E820Pointer	0x22	INT15 E820 数据的存放地址， EFICompatibility 负责填写具体的数据。
E820Length	0x26	EFICompatibility 填写的 E820 数据长度。
IrqRoutingTablePointer	0x2A	PCI 中断路由表的 32 位地址， EFICompatibility 负责填写具体的数据。

IrqRoutingTableLength	0x2E	PCI 中断路由表长度， EFICompatibility 负责填写。
MpTablePtr	0x32	支持多处理器的 MP table 表 32 位地址，EFICompatibility 负责填写具体的数据。
MpTableLength	0x36	MP table 表长度，EFICompatibility 负责填写。
OemIntSegment	0x3A	OEM 自定义的中断服务或表入口的段地址
OemIntOffset	0x3C	OEM 自定义的中断服务或表入口的偏移地址。
Oem32Segment	0x3E	OEM 自定义的 32 位服务或表入口的段地址
Oem32Offset	0x40	OEM 自定义的 32 位服务或表入口的偏移地址。
Oem16Segment	0x42	OEM 自定义的 16 位服务或表入口的段地址
Oem16Offset	0x44	OEM 自定义的 16 位服务或表入口的偏移地址。
TpmSegment	0x46	TPM 模块的段地址
TpmOffset	0x48	TPM 模块的偏移地址
IbvPointer	0x4A	BIOS 厂商 ID 字符串地址。
PciExpressBase	0x4E	PCIe 配置空间映射到内存空间的基地址。对于不支持 PCIe 的系统此值为空 NULL。EFICompatibility 在调用 Compatibility16InitializeYourself() 前需要先初始化此值。
LastPciBus	0x52	系统中最大的 PCI 总线号

CSM16 函数接口

EFICompatibility 模块使用 Compatibility16 函数接口与 CSM16 模块交互。Compatibility16 函数接口是 CSM16 模块中除了提供传统 BIOS 中的服务之外的功能接口。EFICompatibility 使用这个函数接口来创建一个传统 BIOS 的运行环境来运行软件。此外，BIOS 厂商可以使用这个函数接口来加入任何私有的功能。Compatibility16 表中的 Compatibility16CallSegment 和 Compatibility16CallOffset 是 Compatibility16 函数接口的入口地址。EFICompatibility 使用 Legacy BIOS 协议中的 FarCall86() 函数通过 Thunk 远调这个入口来调用 Compatibility16 函数功能。CSM16 允许根据未来的需求增加新的函数接口，但是不允许删除任何已经存在的接口。一个具体的 CSM16 设计中，需要以下的函数接口。

```
typedef enum{
    Compatibility16InitializeYourself          0x0000,
    Compatibility16UpdateBbs                  0x0001,
    Compatibility16PrepareToBoot             0x0002,
    Compatibility16Boot                      0x0003,
    Compatibility16RetrieveLastBootDevice     0x0004,
    Compatibility16DispatchOprom            0x0005,
    Compatibility16GetTableAddress           0x0006,
    Compatibility16SetKeyboardLeds          0x0007,
    Compatibility16InstallPciHandler        0x0008
} EFI_COMPATIBILITY16_FUNCTIONS;
```

Compatibility16InitializeYourself()

这是 EFICompatibility 第一个调用的函数。EFICompatibility 将一些必要的信息作为参数传递进来。Compatibility16InitializeYourself() 负责初始化 16 位的运行环境。

这个函数需要完成的功能包括（但不限于）：安装内存管理服务，初始化传统设备（比如软驱、硬盘）需要的数据区，初始化启动设备选择表（BBS）数据区，初始化中断向量，安装 0xE000~0xFFFF 区域的内存管理服务。

Compatibility16UpdateBbs()

这个函数允许 CSM16 模块更新 BBS 表数据结构。EFICompatibility 会提供足够的信息作为入参给此函数。IBV 主要使用此函数将 USB 存储设备增加到 BBS 表中，将 USB 存储设备模拟成一个 BBS 兼容的设备。

此外，函数还可以将一个 BBS 不兼容的 Option ROM 模拟称一个 BBS 兼容的 Option

ROM。比如：一个 Option ROM 在初始化时勾住 INT19 入口向量，本函数会将这个新的入口向量作为一个 BEV 设备添加到 BBS 中，然后恢复 INT19 的入口向量。

Compatibility16PrepareToBoot()

这个函数让 CSM16 模块在引导操作系统前准备好系统环境。EFICompatibility 会提供足够的信息作为入参给此函数。这个函数需要完成的功能包括（但不限于）：更新 INT15 E820 表，初始化 CMOS 中的基本内存大小和扩展内存大小，准备 ACPI RSD PTR 头，初始化 SMBIOS 数据结构，初始化软驱、串口、并口、键盘、鼠标等的数据区，根据 BBS 表中不同的启动设备建立内部的数据表，卸载 PMM 和 0xE000~0xFFFFF 段的内存管理服务。

Compatibility16Boot()

这是 EFICompatibility 最后一个调用的函数，用来引导操作系统。0xE000~0xFFFFF 段的内存是写保护的。EFICompatibility 会提供足够的信息作为入参给此函数。此功能必须保证 CPU 是 16 位实模式，使用 INT19 进行引导操作系统。

Compatibility16RetrieveLastBootDevice()

这个函数返回最后一个启动设备的优先级。EFICompatibility 会提供足够的信息作为入参给此函数。此函数可以让 EFICompatibility 在多启动设备的情况下检查是否是最后一个启动设备。

Compatibility16DispatchOprom()

这个函数用来让 CSM16 模块加载 Option ROM 进行初始化。EFICompatibility 会提供足够的信息作为入参给此函数。此函数可以让 IBV 继续使用其在 Option ROM 处理上的已有技术成果。

此功能还被用来将 BBS 不支持的 Option ROM 模拟成 BBS 支持的 Option ROM。比如，如果一个 BBS 不支持的 Option ROM 通过勾住 INT13 入口向量安装了一个存储设备，本函数会把这个新的入口向量以一个 BCV 设备添加到 BBS 表中，保存这个新向量的相关信息，然后恢复 INT13 向量入口。通过这个新的向量可以使用这个模拟的 BCV 设备。

Compatibility16GetTableAddress()

EFICompatibility 模块使用这个函数在 0xE000~0xFFFFF 的内存区域中生成各种数据表。EFICompatibility 模块首先调用 Compatibility16GetTableAddress() 在 0xE000~0xFFFFF 段申请一段内存空间，然后复制相应的数据到这段空间。CSM16 模块需要有 0xE000~0xFFFFF 段内存的内存管理功能。

Compatibility16SetKeyboardLeds()

EFICompatibility 模块使用这个函数来同步键盘的 LED 灯状态。此函数根据 BDA 数据区中的数据来更新键盘 LED 灯状态。

Compatibility16InstallPciHandler()

EFICompatibility 模块使用这个函数来为那些没有 Option ROM 的存储设备安装中断处理程序。这类存储设备有：并口硬盘（PATA）和串口硬盘（SATA）。

Compatibility16 Data Structures

EFICompatibility 模块使用 Compatibility16 数据结构作为 Compatibility16 功能接口的入参和出参，使用不同的函数接口来维护和更新这些数据结构。这些数据结构是与 CSM 的具体实现相关的，不同 CSM 实现间的移植需要进行一些修改（更多信息参见 Intel 的 CSM 规范）。CSM 具体实现中的一些 Compatibility16 数据结构包括：

EFI_TO_COMPATIBILITY16_INIT_TABLE: EFICompatibility 模 块 调 用 Compatibility16InitializeYourself() 函数时使用这个数据结构作为入参、出参。这个数据结构包含了创建 CSM16 运行环境时所必须要的信息。这些信息包括，但不限于：PMM 使用的低端内存和高端内存的分配表，Thunk 和 ReverseThunk 模块的地址和大小（需要被排除在 PMM 可分配内存之内），扩展内存大小（用于设定 CMOS 中的值）。

EFI_TO_COMPATIBILITY16_BOOT_TABLE: EFICompatibility 模 块 调 用 Compatibility16UpdateBbs() 和 Compatibility16PrepareToBoot() 函数时使用这个数据结构作为入参、出参。这个数据结构包含了与引导环境有关的信息。这些信息包括，但不限于：串口、并口、软驱、键盘、鼠标、硬盘/光驱等的细节信息，包含启动设备细节的 BBS 表，非传统的设备细节（比如 PARTIES）。

EFI_DISPATCH_OROM_TABLE: EFICompatibility 模 块 调 用 Compatibility16DispatchOrom() 函数时使用这个数据结构作为入参、出参。这个数据结构包含了加载 Option ROM 和处理 Option ROM 初始化程序返回数据有关的信息。

EFI_LEGACY_INSTALL_PCI_HANDLER: EFICompatibility 模 块 调 用 Compatibility16InstallPciHandler() 函数时使用这个数据结构作为入参。这个数据结构包含了 Compatibility16 功能函数安装非 Option ROM 类的 PCI 存储设备中断服务程序需要的有关信息。

Compatibility16SMM

Compatibility16SMM 模块是一个可选择支持的模块。用来提供一些 EFI 没有提供的 SMM 功能。一些平台需求，传统的特性，一些私有的特性等可能需要这个模块。Compatibility16SMM 模块是与桥片相关的。这个模块允许 IBV 通过 SMI 来继续使用它们之前在传统环境中的已有的技术成果。一些事例如：INT15 D042 功能用来更新 CPU 微码，支持 USB 键盘、数据，支持 USB 存储设备。

Thunk and Reverse Thunk

Thunk 模块将 32 位保护模式运行环境切换到 16 位实模式运行环境。ReverseThunk 模块则从 16 位实模式运行环境切换到 32 位保护模式运行环境。Thunk 和 ReverseThunk 均会根据相应的运行环境合理设置 8259 可编程中断控制器。图 17.3 展示了 Thunk 和 ReverseThunk 在 16 位和 32 位运行环境下是如何运行的。

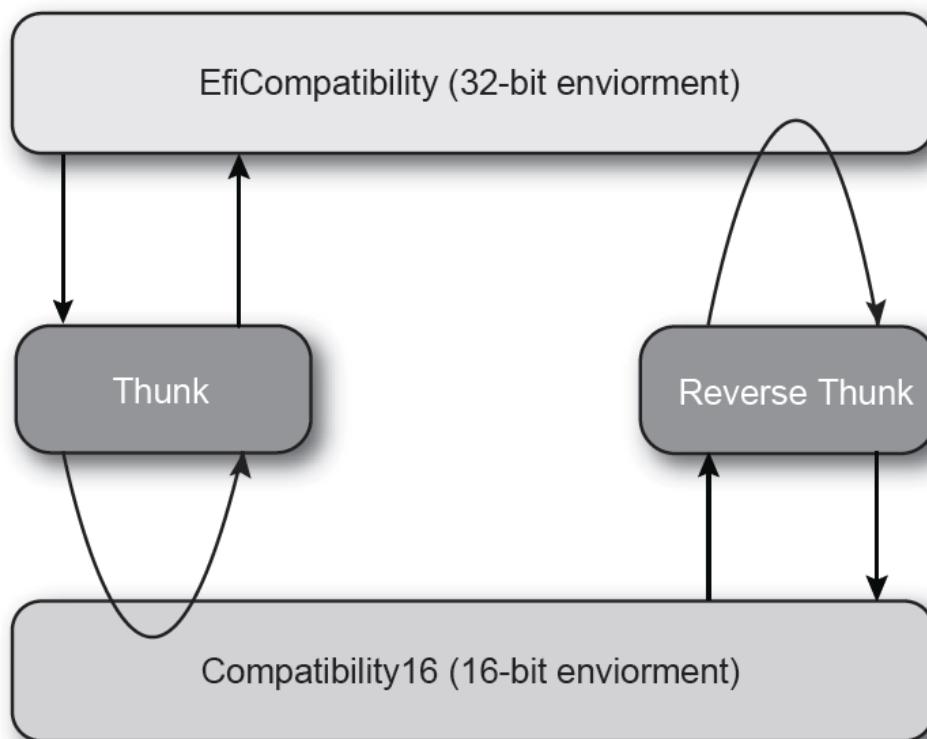


Figure 17.3 Thunk and ReverseThunk in the Framework

EFI 的运行环境（32 位模式）使用轮询模式而非中断模式来推动，它仅支持系统时钟中断。因此当系统运行模式从 32 位变为 16 位时，必须为传统运行环境（16 位模式）所需要硬件和软件中断提供支持。EfiCompatibility 模块在从 EFI 进入 Compatibility16BIOS 或 Option ROM 时会调用 Thunk 模块。Thunk 模块执行下列动作：

- 对中断控制器进行重新设置
- 导入正确的 GDT 和 IDT 表
- 切换到 16 位模式
- 使用提供的输入数据初始化 IA32 寄存器
- 执行必须的动作 FAR CALL 或软中断
- 使用 IA32 寄存器的值设置输出数据
- 恢复 32 位的中断运行环境
- 返回 EFI

16 位的 FAR 函数通过执行（或模拟）RETF 指令返回 Thunk。16 位的软中断处理函数通过执行（或模拟）IRET 指令返回 Thunk。

ReverseThunk 模块和 Thunk 模块很相似，但它是被 16 位的代码所调用，用于从 16 位运行模式切换到 32 位运行模式。当 Compatibility16BIOS 模块需要执行 32 位模式下的代码时，则会调用 ReverseThunk。

Functional Visualization of CSM

当所有必须的 DXE 驱动被执行并且 DXE 框架产生 EFI 启动服务和 EFI 运行时服务，DXE 分发器把控制权转给 BDS 架构协议。BDS 架构协议建立控制台驱动然后尝试引导操作系统。

BDS 架构协议在预启动环境执行各种应用程序。这些应用程序包括 setup，系统配置显示，系统诊断，OEM 增值应用程序和操作系统 boot loader。

图 17.4 显示了 BDS 的 CSM 的架构图。如果传统 Option ROM 需要，或者发现传统的启动选项，或者引导传统的操作系统，BDS 将会调用 CSM。当一个设备没有 EFI 驱动，只有传统 Option ROM 时，EFI 环境依旧需要传统 Option ROM 的支持。在这种情况下，设备和正常的 EFI 驱动间的绑定失败了。通过 CSM 中的 Legacy BIOS Protocol，BDS 会将设备与它的传统 Option ROM 绑定起来。BDS 使用 Legacy BIOS 驱动来使用和初始化设备的 Option ROM。

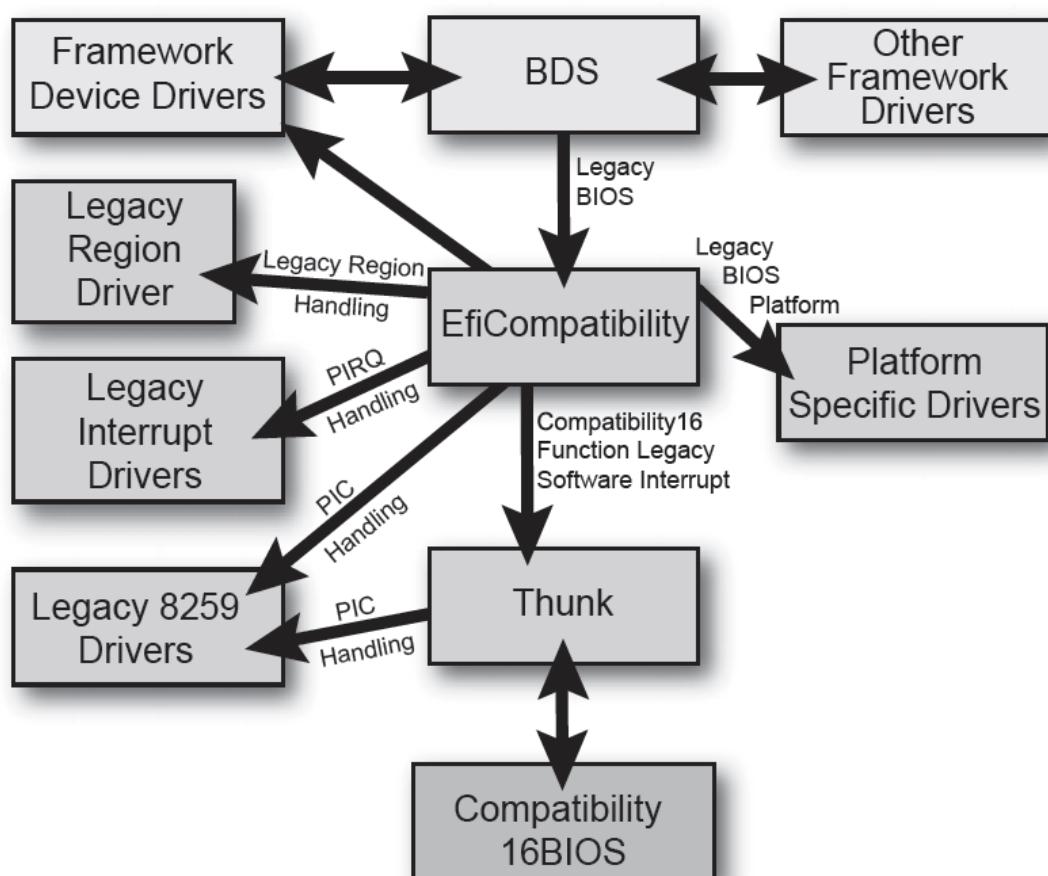


Figure 17.4 Architectural View of CSM under BDS

EFI 环境中的传统启动路径，即使一个设备有 EFI 的驱动和传统 Option ROM，对 Option ROM 的支持也是必须的，因为引导传统的操作系统需要使用 Option ROM 的支持，BDS 会调

用 CSM。

图 17.5 显示了 BDS 阶段的 CSM 的功能视图。通过 Legacy BIOS Protocol 协议，BDS 确定启动设备并且完成操作系统相关的各种动作。一个设备由 EFI 驱动和传统 Option ROM 驱动可能会有不同的现象。EFI 驱动产生一个启动设备列表。ShadowAllLegacyOproms()函数关闭所有的 EFI 设备。GetBbsInfo()函数产生传统启动设备列表。在 Option ROM 初始化期间，CSM 会更新内部的 BBS 表。BDS 产生完整的启动设备列表，所以 Setup 能全面的给用户显示启动选项以便用户选择。用户从启动设备列表中选择启动设备。CSM16 模块把传统的盘号分配给可用的设备。

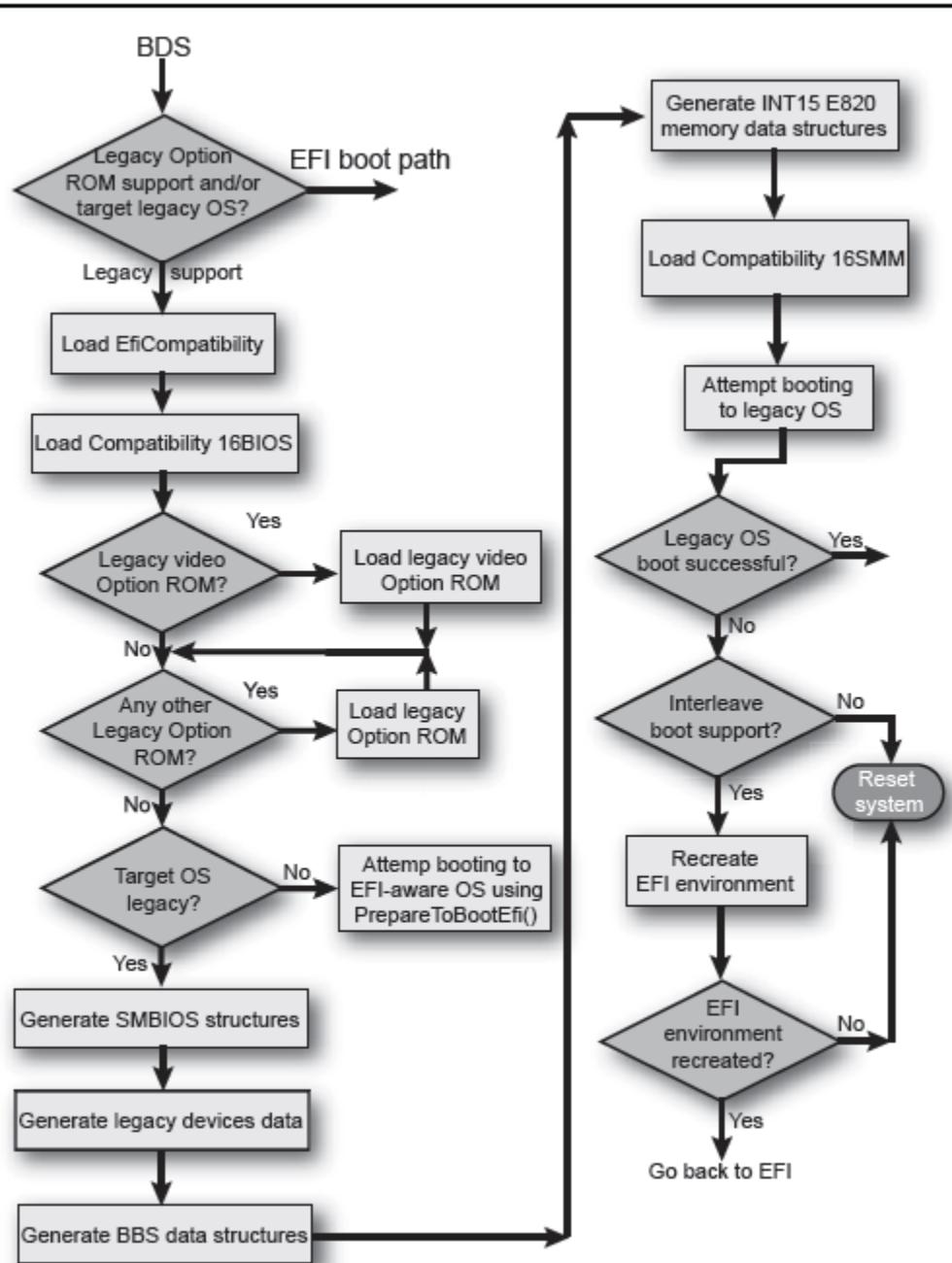


Figure 17.5 Functional View of CSM under BDS

EFI 的设备路径能区分出 EFI 原生操作系统(不管设备是 EFI Option ROM 还是传统 Option

ROM 控制的)还是传统 OS, LegacyBoot()函数就是用来引导传统 OS 的。一个好的实现应该在设置启动设备之前, 检查一个可移动的媒介是否存在。这个检查可以加快启动的速度, 并且可以避免引起系统复位。当引导坏的传统启动选项时, 是复位系统还是返回到 EFI 取决于具体的实现。一个比较好的实现是能支持引导传统和 EFI 的启动选项, 当传统启动失败时, 能返回控制权给 EFI。如果引导至 EFI 原生操作系统和传统 Option ROM 被初始化, 接下来就会调用 PrepareToBootEfi() 函数。

安装传统 BIOS 环境

当安装传统环境, BDS 调用 Legacy BIOS Driver 执行各种操作来初始化 CSM32 和 CSM16 的环境。这些操作包括但不限于下面的内容:

- 查找 Legacy Region Protocol, Legacy Interrupt Protocol, Legacy BIOS Platform Protocol, Legacy 8259 Protocol。
- 从传统内存区域 0x00000 – 0x00500 内存区域, 分配中断向量和 BDA 需要的内存。
- 为 EBDA 和 EfiCompatibility 分配内存。一个好的实现应该分配 0x80000 – 0x9FBFF 区间给 EfiCompatibility, 分配 0x9FC00 – 0xFFFFF 区间给 EBDA。
- 为 Thunk 和 ReverseThunk 分配内存
- 初始化 Thunk 包括更新任何可重定位地址。
- 分配低端内存(小于 1M)和高端内存(大于 1M)给 CSM16 的 PMM 使用。
- 产生传统的内存映射表
- 从 Firmware Volume 中查找 CSM16 模块
- 计算 CSM16 模块的起始地址和大小
- 使 0xE0000 – 0xFFFFF 区域可读写, 然后把 CSM16 模块拷贝到该目标区域
- 查找并使能 Compatibility16 Table, 保存 Compatibility16 函数的入口地址并且更新内部数据。
 - 产生初步的 E820 表, 初始化 BDA、EBDA 和传统 CMOS。
 - 填写 Compatibility16 Table 的必要字段
 - 通过 Thunk 和 Compatibility16 函数调用 Compatibility16InitializeYourself() 函数
 - 从 Compatibility16 表获取即插即用设备的安装信息, 初始化内部的数据结构。
 - 安装传统 BIOS Protocol

启动一个传统操作系统

当启动一个传统操作系统, 通过 LegacyBoot() 函数, BDS 执行各种操作。这些操作包括但不限于下面的内容:

- 确保传统 BIOS 区域可读可写。
- 生成 SMBIOS 数据结构
- 确保所有的 PCI 中断线被正确配置使用 8259 中断控制器。
- 建立串口、并口和软驱的信息
- 建立并确定 IDE ATA/ATAPI 设备的信息。
- 初始化 BDA 中的定时器信息区域
- 重建 E820 表, 然后通过 Compatibility16GetTableAddress() 函数拷贝到传统 BIOS 区域

- 建立 BBS 表并且确定启动设备的优先顺序
- 使用 Compatibility16GetTableAddress() 函数，拷贝 MP 表，OEM 自定义的 16 位和 32 位数据到传统 BIOS 区域。
- 调整 8259 PIC 的传统中断向量基址
- 调用 Compatibility16BIOS 的 PrepareToBoot() 函数
- 使传统 BIOS 区域只读(写保护)
- 通过 Compatibility16Boot() 函数调用 INT19 来引导传统 OS。

www.BIOSREN.COM