

UEFI

Unified Extensible Firmware Interface (UEFI) 规范

V2.9

UEFI 规范 - 中文



UEFI

2022-10

修订记录

序号	修订人	修订时间	修订记录
V0.1	张栋栋	2022 年 10 年 09 日	创建文档
V0.2	GeorgyKwe	2022 年 12 年 29 日	添加 Chapt 6,7,8

目录

1 引言	1
1.1 UEFI 驱动模型扩展	1
1.2 章节安排	2
1.3 目标	5
1.4 目标受众	6
1.5 UEFI 设计概述	7
1.6 UEFI 驱动模型	8
1.6.1 UEFI 驱动程序模型目标	8
1.6.2 传统 Option ROM 问题	9
1.7 迁移要求	9
1.7.1 旧版操作系统支持	9
1.7.2 在旧平台上支持 UEFI 规范	9
1.8 本文档中使用的约定	10
1.8.1 数据结构描述	10
1.8.2 协议描述	10
1.8.3 过程描述	10
1.8.4 指令描述	11
1.8.5 伪代码约定	11
1.8.6 排版约定	11
1.8.7 数字格式	12
1.8.8 二进制前缀	12
1.8.9 修订号	13
2 概述	13
2.1 启动管理器	14
2.1.1 UEFI 镜像	15
2.1.2 UEFI 应用程序	16
2.1.3 UEFI 操作系统加载器	16
2.1.4 UEFI 驱动	17
2.2 固件核心	17
2.2.1 UEFI 服务	17
2.2.2 运行时服务	18
2.3 调用约定	19
2.3.1 数据类型	20
2.3.2 IA32 平台	22

2.3.3 基于 Intel® Itanium® 的平台	24
2.3.4 x64 平台	27
2.3.5 AArch32 平台	30
2.3.6 AArch64 平台	33
2.3.7 RISC-V 平台	39
2.4 协议	44
2.5 UEFI 驱动模型	49
2.5.1 传统 Option ROM 问题	51
2.5.2 驱动程序初始化	53
2.5.3 主机总线控制器	54
2.5.4 总线驱动	57
2.5.5 平台组件	58
2.5.6 热插拔事件	59
2.5.7 EFI 服务绑定	59
2.6 要求	61
2.6.1 必要元素	61
2.6.2 平台特定的元素	61
2.6.3 驱动程序的特定要素	64
2.6.4 在其他地方发布的对本规范的扩展	66
3 启动管理器	67
3.1 固件启动管理器	67
3.1.1 启动管理器编程	68
3.1.2 加载选项处理	69
3.1.3 加载选项	71
3.1.4 启动管理器的功能	72
3.1.5 启动 Boot#### 应用程序	73
3.1.6 使用热键启动 Boot#### 加载选项	73
3.1.7 必要的系统准备应用	74
3.2 启动管理器策略协议	75
3.2.1 EFI_BOOT_MANAGER_POLICY_PROTOCOL	75
3.2.2 EFI_BOOT_MANAGER_POLICY_PROTOCOL.ConnectDevicePath()	76
3.2.3 EFI_BOOT_MANAGER_POLICY_PROTOCOL.ConnectDeviceClass()	77
3.3 全局定义的变量	79
3.4 引导选项恢复	84
3.4.1 操作系统定义的引导选项恢复	84
3.4.2 平台定义的引导选项恢复	85
3.4.3 引导选项变量默认引导行为	85

3.5 引导机制	86
3.5.1 通过简单文件协议启动	86
3.5.2 通过加载文件协议启动	87
3.6 UEFI Image Entry Point	88
3.7 EFI 表头	89
3.8 EFI 系统表	90
3.9 EFI Boot Services Table	92
3.10 EFI Runtime Services Table	95
3.11 EFI Configuration Table & Properties Table	97
3.12 镜像入口点示例	104
3.12.1 镜像入口点示例	104
3.12.2 UEFI Driver Model Example	106
3.12.3 UEFI Driver Model Example (Unloadable)	107
3.12.4 EFI Driver Model Example (Multiple Instances)	108
4 GUID Partition Table(GPT) Disk Layout	109
4.1 GPT 和 MBR 磁盘布局对比	109
4.2 LBA 0 格式	109
4.2.1 传统主引导记录 (MBR)	110
4.2.2 操作系统类型	112
4.2.3 Protective MBR	112
4.2.4 分区信息	114
4.3 GUID 分区表 (GPT) 磁盘布局	114
4.3.1 GPT 概述	114
4.3.2 GPT Header	116
4.3.3 GPT 分区条目数组	118
5 区块转换表 (BTT) 布局	122
5.1 区块转换表 (BTT) 背景	122
5.2 区块转换表 (BTT) 数据结构	124
5.2.1 BTT 信息块	124
5.2.2 BTT Map Entry	127
5.2.3 BTT Flog	128
5.2.4 BTT 数据区	130
5.2.5 NVDIMM 标签协议地址抽象指南	130
5.3 BTT 操作理论	130
5.3.1 BTT 区域	131
5.3.2 区域中数据块的原子性	132

5.3.3	BTT 数据结构的原子性	132
5.3.4	编写初始 BTT 布局	133
5.3.5	启动时验证 BTT 区域	133
5.3.6	启动时验证 Flog Entry	134
5.3.7	读取路径	135
5.3.8	写入路径	137
6	引导服务	138
6.1	事件、计时器和任务优先级服务	139
6.2	内存分配服务	159
6.3	协议处理程序服务 (Protocol Handler Services)	173
6.4	镜像服务 (Image Services)	215
6.5	杂项启动服务 (Miscellaneous Boot Services)	228
7	运行时服务	234
7.1	运行时服务规则和限制 (Runtime Services Rules and Restrictions)	235
7.1.1	Machine Check, INIT 和 NMI 异常 (Exception for Machine Check, INIT and NMI)	236
7.2	可变服务 (Variable Services)	237
7.2.1	使用 EFI_VARIABLE_AUTHENTICATION_3 描述符(Using the EFI_VARIABLE_AUTHENTICATION_3 descriptor)	250
7.2.2	使用 EFI_VARIABLE_AUTHENTICATION_2 描述符 (Using the EFI_VARIABLE_AUTHENTICATION_2 descriptor)	254
7.2.3	使用 EFI_VARIABLE_AUTHENTICATION 描述符(Using the EFI_VARIABLE_AUTHENTICATION descriptor)	256
7.2.4	硬件错误记录持久化 (Hardware Error Record Persistence)	256
7.3	时间服务 (Time Services)	257
7.4	虚拟内存服务 (Virtual Memory Services)	264
7.5	杂项运行时服务 (Miscellaneous Runtime Services)	268
7.5.1	重置系统 (Reset System)	268
7.5.2	获取下一个高单调计数 (Get Next High Monotonic Count)	269
7.5.3	更新胶囊 (Update Capsule)	271

List of Figures

1	UEFI 概念概述	7
2	启动顺序	14
3	调用 AddressOfEntryPoint 后的堆栈, IA-32	24
4	软件服务关系	60
5	UEFI 镜像类型	87
6	内存属性定义的使用	104
7	GUID 分区表 (GPT) 示例	115
8	BTT 区域中的 BTT 布局	123
9	大命名空间里包含多个区域的 BTT	124
10	循环序列号的 Flog entrys	129
11	BTT Read Path Overview	136
12	BTT Write Path Overview	138
13	table7-1 Event, Timer, and Task Priority Functions	140
14	table7-2-1TPL Usage	141
15	table7-2-2TPL Usage	141
16	table7-3-1TPL Restrictions	142
17	table7-3-2TPL Restrictions	143
18	table7-3-3TPL Restrictions	144
19	table7-3-4TPL Restrictions	145
20	table7-3_0	148
21	table7-3_1	152
22	table7-3_2	153
23	table7-3_3	154
24	table7-3_4	155
25	table7-3_5	156
26	table7-3_6	157
27	table7-4 Memory Allocation Functions	159
28	table7-5	161
29	table7-6-1	162
30	table7-6-2	163
31	table7-6_0	166
32	table7-6_1	167
33	table7-6_2	171
34	table7-6_3	172
35	table7-6_4	172
36	table7-7	174

37	Figure 7-1 Device Handle to Protocol Handler Mapping	175
38	Figure 7-2 Handle Database	177
39	table7-7_0	179
40	table7-7_1	181
41	table7-7_2	182
42	table7-7_3	184
43	table7-7_4	185
44	table7-7_5	187
45	table7-7_6	188
46	table7-7_7	192
47	table7-7_8	197
48	table7-7_9	199
49	table7-7_10	202
50	table7-7_11	206
51	table7-7_12	208
52	table7-7_13	209
53	table7-7_14	213
54	table7-7_15	214
55	table7-7_16	215
56	Table7-8-1	216
57	Table7-8-1	217
58	Table7-9	217
59	table7-9_1.jpg	220
60	table7-9_1	222
61	table7-9_3	223
62	table7-7_4	226
63	table7-7_5	227
64	table7-10	228
65	table7-10_1	229
66	table7-10_2	230
67	table7-10_5	232
68	table7-10_6	233
69	table7-10_7	234
70	table8-1	236
71	table8-2	237
72	table8-3	238
73	table8-3_1	241
74	table8-3_2	243

75	table8-3_3	249
76	table8-3_4	250
77	table8-4 Hardware Error Record Persistence Variables	257
78	table8-5 Time Services Functions	258
79	table8-5_1	261
80	table8-5_2	262
81	table8-5_3	263
82	table8-5_4	264
83	table8-6 Virtual Memory Functions	265
84	table8-6_1	266
85	table8-6_2	267
86	table8-7 Miscellaneous Runtime Services	268
87	table8-7_1	270
88	table8-8 Flag Firmware Behavior	274
89	table8-8_1	275
90	Figure 8-1 Scatter-Gather List of EFI_CAPSULE_BLOCK_DESCRIPTOR Structures	277
91	table8-8_2	280
92	Table 8-9 Variables Using EFI_CAPSULE_REPORT_GUID	283
93	table8-9_1	286

1 引言

统一可扩展固件接口 (UEFI) 规范描述了操作系统和平台固件之间的接口。UEFI 之前是可扩展固件接口规范 1.10 (EFI)。因此，一些代码和某些协议名称保留了 EFI 名称。除非另有说明，本规范中的 EFI 名称可假定为 UEFI 的一部分。

该接口采用数据表的形式，其中包含与平台相关的信息，以及可供 OS 加载程序和 OS 使用的启动和运行时服务调用。它们共同提供了一个启动操作系统的标准环境。本规范是作为一个纯粹的接口规范设计的。因此，**该规范定义了平台固件必须实现的接口和结构集**。类似地，该规范定义了操作系统在启动时可能使用的一组接口和结构。无论是固件开发者选择如何实现所需的元素，还是操作系统开发者选择如何利用这些接口和结构，都由开发者自己决定。

该规范的目的是定义一种方法，使操作系统和平台固件仅通信支持操作系统启动过程所必需的信息。这是通过平台和固件提供给操作系统的软件可见接口的正式和完整的抽象规范来实现的。

本规范的目的是为操作系统和平台固件定义一种方式，以仅传递支持操作系统启动过程所需的信息。这是通过平台和固件呈现给操作系统的软件可见接口的抽象规范来实现的。

使用这一正式定义，旨在运行在与受支持的处理器规范兼容的平台上的收缩包装操作系统将能够在各种系统设计上启动，而无需进一步的平台或操作系统定制。该定义还允许平台创新引入新特性和功能，以增强平台的能力，而不需要按照操作系统的启动顺序编写新代码。

此外，抽象规范开辟了一条替代遗留设备和固件代码的路径。新的设备类型和相关代码可以通过相同定义的抽象接口提供同等的功能，同样不会影响 OS 启动支持代码。

该规范适用于从移动系统到服务器的所有硬件平台。该规范提供了一组核心服务以及一组协议接口。协议接口的选择可以随着时间的推移而发展，并针对不同的平台市场细分进行优化。与此同时，该规范允许 oem 提供最大限度的可扩展性和定制能力，以实现差异化。在这方面，UEFI 的目的是定义一个从传统的“PC-AT”风格的启动世界到一个没有遗留 API 的环境的进化路径。

1.1 UEFI 驱动模型扩展

对启动设备的访问是通过一系列的协议接口提供的。UEFI 驱动模型的一个目的是为“PC-AT”式的 Option ROM (TODO) 提供一个替代品。需要指出的是，写在 UEFI 驱动模型上的驱动，被设计为在预启动环境中访问启动设备。它们并不是为了取代高性能的、针对操作系统的驱动程序。

UEFI 驱动模型被设计为支持执行模块化的代码，也被称为驱动，在预启动环境中运行。这些驱动程序可以管理或控制平台上的硬件总线和设备，也可以提供一些软件衍生的、平台特定的服务。

UEFI 驱动模型还包含了 UEFI 驱动编写者所需的信息，以设计和实现平台启动 UEFI 兼容的操作系统可能需要的任何总线驱动和设备驱动的组合。

UEFI 驱动模型被设计为通用的，可以适应任何类型的总线或设备。UEFI 规范描述了如何编写 PCI 总线驱动

程序、PCI 设备驱动程序、USB 总线驱动程序、USB 设备驱动程序和 SCSI 驱动程序。提供了允许将 UEFI 驱动程序存储在 PCI Option ROM 中的其他详细信息，同时保持了与旧 Option ROM 镜像的兼容性。

UEFI 规范的一个设计目标是使驱动镜像尽可能的小。然而，如果一个驱动程序需要支持多个处理器架构，那么也需要为每个支持的处理器架构提供一个驱动程序对象文件。为了解决这个空间问题，本规范还定义了 EFI 字节代码虚拟机（EFI Byte Code Virtual Machine）。一个 UEFI 驱动可以被编译成一个 EFI 字节代码对象文件。UEFI Specification-complaint (TODO) 的固件必须包含一个 EFI 字节代码解释器。这使得支持多种处理器架构的单一 EFI 字节代码对象文件可以被运出。另一种节省空间的技术是使用压缩。该规范定义了压缩和解压算法，可用于减少 UEFI 驱动程序的大小，从而减少 UEFI 驱动程序存储在 ROM 设备中时的开销。

OSV、IHV、OEM 和固件供应商可以使用 UEFI 规范中包含的信息来设计和实现符合本规范的固件、生成标准协议接口的驱动程序以及可用于启动 UEFI 兼容的操作系统加载程序操作系统。

1.2 章节安排

本规范的章节组织如下：

章节名	内容
引言/概述	介绍 UEFI 规范，并描述 UEFI 的主要组件。
启动管理器	管理器用于加载写入此规范的驱动程序和应用程序。
EFI 系统表和分区	描述了一个 EFI 系统表，它被传递给每个兼容的驱动程序和应用程序，并定义了一个基于 GUID 的分区方案。
块转换表	用于执行块 I/O 的布局和规则集，可提供单个块的断电写入原子性。
启动服务	包含在启动操作系统之前存在于 UEFI 兼容系统中的基本服务的定义。
运行时服务	包含在操作系统启动之前和之后存在于兼容系统中的基本服务的定义。
协议	<p>EFI 加载图像协议描述已加载到内存的 UEFI 镜像。</p> <p>设备路径协议提供了在 UEFI 环境中构建和管理设备路径所需的信息。</p>
	<p>UEFI 驱动模型描述了一组服务和协议，适用于每个总线和设备类型。</p> <p>控制台支持协议定义了 I/O 协议，处理系统用户在启动服务环境中执行的基于文本的信息的输入和输出。</p> <p>媒介访问协议定义了加载文件协议，文件系统格式和媒介格式处理可移动媒介。</p>

章节名	内容
	PCI 总线支持协议定义 PCI 总线驱动程序, PCI 设备驱动程序和 PCI Option ROM 布局。所描述的协议包括 PCI 根桥 I/O 协议和 PCI I/O 协议。
	SCSI 驱动程序模型和总线支持定义了 SCSI I/O 协议和扩展 SCSI Pass Thru 协议, 用于抽象访问由 SCSI 主机控制器产生的 SCSI 通道。
	iSCSI 协议定义了通过 TCP/IP 传输 SCSI 数据。
	USB 支持协议定义了 USB 总线驱动程序和 USB 设备驱动程序。
	调试器支持协议描述了一组可选的协议, 提供所需的服务, 以实现一个源级调试器的 UEFI 环境。
	压缩算法规范详细描述了压缩/解压缩算法, 外加一个标准的 EFI 解压缩接口, 用于启动时使用。
	ACPI 协议可用于从平台上安装或删除 ACPI 表。
	字符串服务: Unicode 排序协议允许在启动服务环境中运行的代码对给定语言的 Unicode 字符串执行词法比较函数; 正则表达式协议用于根据正则表达式模式匹配 Unicode 字符串。
EFI 字节码虚拟机	定义 EFI 字节码虚拟处理器及其指令集。它还定义了如何将 EBC 对象文件加载到内存中, 以及从本机代码到 EBC 代码再转换到本机代码的机制。
固件更新和报告	为设备提供一个抽象, 以提供固件管理支持。
网络协议	SNP、PXE、BIS 和 HTTP 启动协议定义了在 UEFI 启动服务环境中执行时提供对网络设备访问的协议。
	受管网络协议定义了 EFI 受管网络协议, 它提供原始(未格式化)异步网络数据包 I/O 服务和托管网络服务绑定协议, 用于定位 MNP 驱动支持的通信设备。
	VLAN、EAP、Wi-Fi 和 Suplicant 协议定义了一个协议, 为 VLAN 配置提供可管理性接口。
	蓝牙协议定义。
	TCP、IP、PIPsec、FTP、GTLS 和 Configurations 协议定义了 EFI TCPv4 (Transmission Control Protocol version 4) 协议和 EFI IPv4 (Internet Protocol version 4) 协议。
	ARP、DHCP、DNS、HTTP 和 REST 协议定义了 ARP (Address Resolution Protocol) 协议接口和 EFI DHCPv4 协议。

章节名	内容
	UDP 和 MTFTP 协议定义了 EFI UDPv4 (User Datagram Protocol version 4) 协议，该协议在 EFI IPv4 协议上接口，并定义了 EFI MTFTPV4 协议接口，该接口建立在 EFI UDPv4 协议之上。
安全启动和驱动程序签名	介绍 Secure Boot 和生成 UEFI 数字签名的方法。
人机界面基础设施 (HII)	定义实现人机接口基础设施 (HII) 所需的核心代码和服务，包括管理用户输入和相关协议的代码定义的基本机制。
	描述用于管理系统配置的数据和 api: 描述旋钮和设置的实际数据。
用户标识	描述描述平台当前用户的服务。
安全技术	描述用于利用安全技术的协议，包括加密散列和密钥管理。
杂项协议	Timestamp 协议提供了一个独立于平台的接口来检索高分辨率的时间戳计数器。当调用 ResetSystem 时，重置通知协议提供注册通知的服务。
附录	<p>GUID 和时间格式。</p> <p>基于基本文本的控制台要求，符合 efi 系统需要提供通信能力。</p> <p>设备路径使用数据结构的例子，定义各种硬件设备的启动服务。</p> <p>状态代码列出了 UEFI 接口返回的成功、错误和警告代码。</p> <p>通用网络驱动程序接口定义了 32/64 位硬件和软件通用网络驱动程序接口 (UNDIs)。</p> <p>使用简单指针协议。</p> <p>使用 EFI 扩展 SCISI 直通协议。</p> <p>压缩源代码的一个压缩算法的实现。</p> <p>一个 EFI 解压缩算法的实现的解压源代码。</p> <p>EFI 字节码虚拟机操作码列表提供了相应指令集的摘要。</p> <p>字母功能列表按字母顺序标识所有 UEFI 接口功能。</p> <p>EFI 1.10 协议变更和折旧清单标识了协议、GUID、修订标识符名称变更以及与 EFI 1.10 规范相比已弃用的协议。</p> <p>平台错误记录描述了用于表示平台硬件错误的常见平台错误记录格式。</p> <p>UEFI ACPI Data Table 定义了 UEFI ACPI 表格式。</p> <p>硬件错误记录持久性使用。</p>

章节名	内容
引用	
术语表	
索引	提供规范中关键术语和概念的索引。

1.3 目标

“PC-AT”启动环境对行业内的创新提出了重大挑战。每一个新的平台功能或硬件创新都要求固件开发人员设计越来越复杂的解决方案，并且通常要求操作系统开发人员修改启动代码，然后客户才能从创新中受益。这可能是一个耗时的过程，需要大量的资源投资。

UEFI 规范的主要目标是定义一个替代启动环境，可以减轻这些考虑。在这个目标中，该规范类似于其他现有的启动规范。本规范的主要属性可以概括为以下属性：

- **一致的、可扩展的平台环境。**该规范为固件定义了一个完整的解决方案，以描述所有平台特性和 OS 的 surface platform(TODO) 功能在启动过程中。这些定义非常丰富，足以涵盖一系列当代处理器设计。
- **从固件中抽象操作系统。**该规范定义了平台功能的接口。通过使用抽象接口，该规范允许在构建 OS 加载器时，而无需了解作为这些接口基础的平台和固件。这些接口代表了底层平台和固件实现与操作系统加载程序之间定义良好的稳定边界。这样的边界允许底层固件和操作系统加载程序更改，前提是两者都将交互限制在定义的接口上。
- **合理的设备抽象，不需要遗留接口。**“PC-AT”BIOS 接口要求操作系统加载程序对某些硬件设备的工作有特定的了解。该规范为 OS 加载器开发人员提供了一些不同的东西：抽象接口使得可以构建在一系列底层硬件设备上工作的代码，而无需明确了解该范围内每个设备的细节。
- **从固件中提取 Option ROM。**该规范定义了平台功能的接口，包括 PCI、USB 和 SCSI 等标准总线类型。支持的总线类型可能会随着时间的推移而增加，因此包括了一种扩展到未来总线类型的能力。这些定义的接口以及扩展到未来总线类型的能力是 UEFI 驱动程序模型的组件。UEFI 驱动模型的一个目的是解决现有“PC-AT”Option ROM 中存在的广泛问题。与 OS 加载程序一样，驱动程序使用抽象接口，因此可以构建设备驱动程序和总线驱动程序，而无需了解作为这些接口基础的平台和固件。
- **架构上可共享的系统分区。**扩展平台功能和添加新设备的计划通常需要软件支持。在许多情况下，当这些平台创新 (TODO) 在操作系统控制平台之前被激活时，它们必须由特定于平台而不是客户选择的操作系统的代码支持。解决这个问题的传统方法是在制造过程中将代码嵌入平台中（例如，在闪存设备中）。对这种持久存储的需求正在快速增长。该规范定义了大型海量存储媒介类型上的持久存储，以供平台支持代码扩展使用，以补充传统方法。规范中明确了其工作原理的定义，以确保固件开发商、OEM、操作系统供应商甚至第三方可以安全地共享空间，同时增加平台功能。

可以通过多种方式定义提供这些属性的启动环境。实际上，在编写本规范时，已经存在几种替代方案，从学术角度来看可能是可行的。然而，考虑到当前围绕支持的处理器平台的基础设施能力，这些替代方案通常会带来很高的门槛。本规范旨在提供上面列出的属性，同时也认识到行业的独特需求，该行业在兼容性方面进行了大量投资，并且拥有大量无法立即放弃的系统安装基础。这些需求推动了对本规范中体现的附加属性的要求：

- **进化性的，而不是革命性的。** 规范中的接口和结构旨在尽可能地减少初始实现的负担。虽然已经小注意保在接口本身中维护适当的抽象，但该设计还确保可以重用 BIOS 代码来实现接口，而只需要最少的额外编码工作。换句话说，在 PC-AT 平台上，规范最初可以作为基于现有代码的底层实现之上的薄接口（thin Interface TODO）层来实现。同时，抽象接口的引入提供了将来从遗留代码的迁移。一旦抽象被确立为固件和操作系统加载程序在启动期间交互的手段，开发人员就可以随意替换抽象接口下的遗留代码。类似的硬件遗留迁移也是可能的。由于抽象隐藏了设备的细节，因此可以移除底层硬件，并用提供改进功能、降低成本或两者兼而有之的新硬件替换它。显然，这需要编写新的平台固件来支持设备并通过抽象接口将其呈现给 OS 加载器。但是，如果没有接口抽象，则可能根本无法移除旧设备。
- **设计上的兼容性。** 系统分区结构的设计还保留了当前在“PC-AT”启动环境中使用的所有结构。因此，构建一个能够从同一磁盘启动传统操作系统或 EFI-aware 操作系统的单一系统是一件简单的事情。
- **简化了操作系统中立的平台增值的添加。** 该规范定义了一个开放的、可扩展的接口，它有助于创建平台“驱动程序”。这些可能类似于操作系统驱动程序，在启动过程中为新设备类型提供支持，或者它们可能用于实现增强的平台功能，例如容错或安全性。此外，这种扩展平台能力的能力从一开始就被设计到规范中。这旨在帮助开发人员避免在尝试将新代码挤入传统 BIOS 环境时所固有的许多挫败感。由于包含用于添加新协议的接口，OEM 或固件开发人员拥有以模块化方式向平台添加功能的基础设施。由于规范中定义的调用约定和环境，此类驱动程序可能会使用高级编码语言来实现。这反过来可能有助于降低创新的难度和成本。系统分区选项为此类扩展提供了非易失性存储器存储的替代方案。
- **建立在现有投入的基础上。** 在可能的情况下，规范避免在现有行业规范提供足够覆盖的领域重新定义接口和结构。例如，ACPI 规范为操作系统提供了发现和配置平台资源所需的所有信息。同样，规范设计的这种哲学选择旨在尽可能降低采用该规范的障碍。

1.4 目标受众

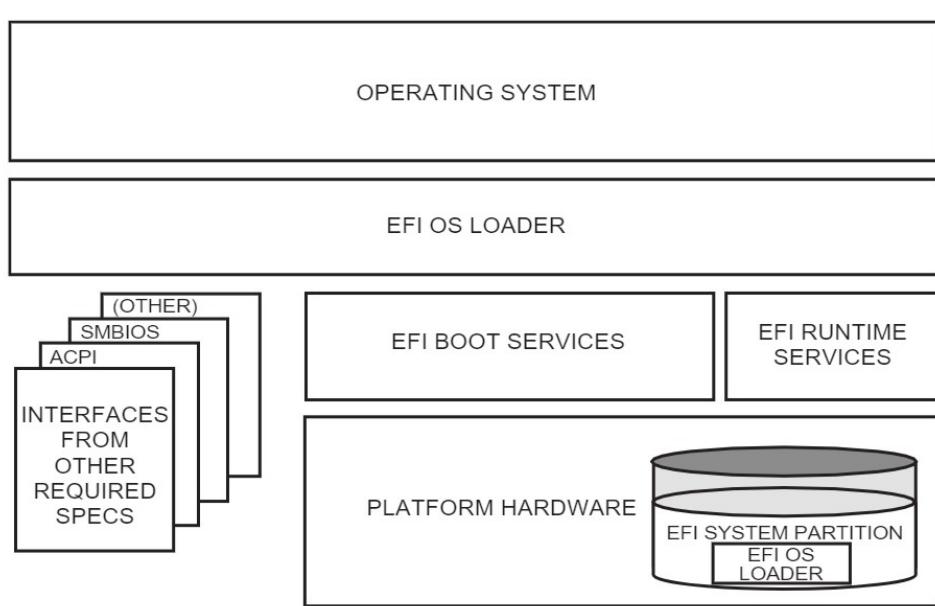
本文档主要适用于以下读者：

- 将实现 UEFI 驱动程序的 IHV 和 OEM。
- 将创建支持的处理器平台的 OEM 厂商，旨在启动 shrink-wrap（TODO）的操作系统。
- BIOS 开发人员，无论是创建通用 BIOS 和其他固件产品的人员，还是修改这些产品的支持人员。
- 操作系统开发人员将调整他们的 shrink-wrap（TODO）操作系统产品，用来在支持的基于处理器的平台上运行。

1.5 UEFI 设计概述

UEFI 的设计基于以下基本要素：

- **重用现有的基于表格的接口。**为了保持对现有基础支持代码（包括操作系统和固件）的投资，必须在希望符合 UEFI 规范的平台上，实现通常在与支持的处理器规范兼容的平台上，实现的许多现有规范。（有关更多信息，请参阅附录 Q：参考资料。）
- **系统分区。**系统分区定义了一个分区和文件系统，可允许多个供应商之间安全共享，并用于不同目的。包含单独的、可共享的系统分区的能力提供了增加平台附加值的机会，而不会显著增加对非易失性平台存储器的需求。
- **启动服务。**启动服务为可在启动期间使用的设备和系统功能提供接口。设备访问是通过“句柄”(handles) 和“协议”(protocols) 抽象出来的。这有利于重用现有 BIOS 代码，将基本实现要求保持在规范之外，而不会给访问设备的消费者带来负担。
- **运行时服务。**提供了一组最小的运行时服务，以确保对基础平台的硬件资源进行适当的抽象，这些资源可能是操作系统在正常运行时需要的。



OM13141

图 1. UEFI 概念概述

图 1-1 描述了用于完成平台和操作系统启动的符合 UEFI 规范的系统的各种组件的交互。

平台固件能够从系统分区中检索操作系统加载器镜像。该规范提供了各种大容量存储设备类型，包括磁盘、CD-ROM 和 DVD，以及通过网络的远程启动。通过可扩展的协议接口，有可能增加其他的启动媒介类型，尽管如果这些媒介需要使用本文件中定义的协议以外的协议，可能需要修改操作系统加载器。

一旦启动，操作系统加载程序将继续启动整个操作系统。为此，它可以使用本规范或其他所需规范定义的 EFI 启动服务和接口来探测、解析和初始化各种平台组件和管理它们的操作系统软件。在启动阶段，EFI 运行时服务也可供 OS 加载器使用。

1.6 UEFI 驱动模型

本节描述了符合本规范的固件的驱动模型的目标。目标是让这个驱动模型为所有类型的总线和设备提供一个实现总线驱动和设备驱动的机制。在撰写本文时，支持的总线类型包括 PCI、USB 等。

随着硬件架构的不断发展，平台中存在的总线数量和类型也在不断增加。这种趋势在高端服务器中尤为明显。然而，更多样化的总线类型被设计到桌面和移动系统，甚至一些嵌入式系统中。这种日益增长的复杂性，意味着在预启动环境中，需要一种简单的方法来描述和管理平台中的所有总线和设备。UEFI 驱动模型以协议、服务和启动服务的形式提供了这种简单的方法。

1.6.1 UEFI 驱动程序模型目标

UEFI 驱动模型有以下目标：

- **兼容** - 符合此规范的驱动程序必须保持与 EFI 1.10 规范和 UEFI 规范的兼容性。这意味着 UEFI 驱动程序模型利用 UEFI 2.0 规范中的可扩展性机制来添加所需的功能。
- **简单** - 符合本规范的驱动程序必须易于实现，易于维护。UEFI 驱动模型必须允许驱动编写者专注于正在开发的特定设备的驱动。驱动程序不应关注平台策略或平台管理问题。这些考虑应该留给系统固件。
- **可扩展性** - UEFI 驱动模型必须能够适应所有类型的平台。这些平台包括嵌入式系统、移动和桌面系统，以及工作站和服务器。
- **灵活** - UEFI 驱动模型必须支持枚举所有设备的能力，或者只枚举启动所需操作系统的那些设备。最小的设备枚举提供了对更快速的启动能力的支持，而完整的设备媒体提供了在系统中存在的任何启动设备上执行操作系统安装、系统维护或系统诊断的能力。
- **可扩展性** - UEFI 驱动模型必须能够扩展到未来定义的总线类型。
- **可移植性** - 根据 UEFI 驱动模型编写的驱动，必须在不同平台和支持的处理器架构之间可移植。
- **可互操作性** - 驱动程序必须与其他驱动程序和系统固件共存，并且必须在不产生资源冲突的情况下进行操作。
- **描述复杂的总线层次结构** - UEFI 驱动模型必须能够描述各种总线拓扑结构，从非常简单的单总线平台到包含许多不同类型总线的非常复杂的平台。

- **驱动占用空间小** - 由 UEFI 驱动程序模型产生的可执行文件的大小必须最小化，以减少整体平台成本。虽然灵活性和可扩展性是目标，但支持这些所需的额外开销必须保持在最低水平，以防止固件组件的大小变得无法管理。
- **解决遗留 Option ROM 的问题** - UEFI 驱动模型必须直接解决遗留 Option ROM 的约束和限制。具体来说，必须能够建立同时支持 UEFI 驱动和传统 Option ROM 的插件卡，这种卡可以在传统 BIOS 系统和符合 UEFI 的平台上执行，而无需修改卡上的代码。该解决方案必须提供一个从传统 Option ROM 驱动程序迁移到 UEFI 驱动程序的进化路径。

1.6.2 传统 Option ROM 问题

这个支持驱动模型的想法来自于对 UEFI 规范的反馈，它提供了一个明确的、由市场驱动的对传统选项 ROM（有时也被称为扩展 ROM）的替代要求。人们认为，UEFI 规范的出现代表了一个机会，通过用一种在 UEFI 规范框架内工作的替代机制来取代传统选项 ROM 镜像的构建和操作，从而摆脱隐含的限制。

1.7 迁移要求

迁移要求涵盖了从最初实施本规范到未来所有平台和操作系统都实施本规范的过渡时期。在这一时期，有两个主要的兼容性考虑是很重要的。

- 能够继续启动传统的操作系统；
- 能够在现有的平台上实现 UEFI，尽可能多地复用现有的固件代码，将开发资源和时间要求降到最低。

1.7.1 旧版操作系统支持

UEFI 规范代表了收缩式操作系统和固件在启动过程中进行通信的首选方式。然而，选择制作一个符合该规范的平台，并不排除该平台，也支持不了解 UEFI 规范的，现有传统操作系统二进制文件。

UEFI 规范并不限制平台设计者，选择同时支持 UEFI 规范和更传统的“PC-AT”启动基础架构。如果要实现这样的传统基础架构，应该按照现有的行业惯例来开发，这些惯例是在本规范范围之外定义的。在任何给定的平台上，支持的传统操作系统的选项是由该平台的制造商决定的。

1.7.2 在旧平台上支持 UEFI 规范

UEFI 规范经过精心设计，允许以最少的开发工作扩展现有系统以支持它。特别是 UEFI 规范中定义的抽象结构和服务，都可以在遗留平台上得到支持。

例如，要在现有且受支持的基于 32 位的平台上实现此类支持，该平台使用传统 BIOS 来支持操作系统启动，需要提供额外的固件代码层。需要这些额外的代码来将服务和设备的现有接口转换为对本规范中定义的抽象的支持。

1.8 本文档中使用的约定

1.8.1 数据结构描述

支持的处理器是“小端”机器。这种区别意味着内存中多字节数据项的低位字节位于最低地址，而高位字节位于最高地址。一些受支持的 64 位处理器可以配置为“小端”和“大端”操作。所有旨在符合本规范的实现都使用“小端”操作。

在某些内存布局描述中，某些字段被标记为保留。软件必须将这些字段初始化为零并在读取时忽略它们。在更新操作中，软件必须保留任何保留字段。

1.8.2 协议描述

协议描述一般有以下格式：

- **协议名称**：协议接口的正式名称。
- **摘要**：协议接口的简要描述。
- **GUID**：协议接口的 128 位 GUID (Globally Unique Identifier)。
- **协议接口结构**：一种“c 风格”的数据结构定义，包含由该协议接口产生的过程和数据字段。
- **参数**：协议接口结构中各字段的简要说明。
- **描述**：对接口提供的功能的描述，包括调用者应该知道的任何限制和警告。
- **相关定义**：协议接口结构或其任何过程中使用的类型声明和常量。

1.8.3 过程描述

过程描述通常具有以下格式：

- **过程名称**：过程的正式名称。
- **摘要**：过程的简要说明。
- **原型**：定义调用序列的“C 风格”过程标头。
- **参数**：对程序原型中每个字段的简要描述。
- **描述**：对接口所提供的功能的描述，包括调用者应该注意的任何限制和注意事项。
- **相关定义**：仅由该过程使用的类型声明和常量。
- **返回的状态代码**：对接口所返回的任何代码的描述。该过程需要实现本表中列出的任何状态代码。可以返回更多的错误代码，但是它们不会被标准的符合性测试所测试，而且任何使用该程序的软件，都不能依赖于实现可能提供的任何扩展错误代码。

1.8.4 指令描述

EBC 指令的指令描述一般有以下格式：

- **指令名称**: 指令的正式名称。
- **语法**: 指令的简要描述。
- **描述**: 对指令所提供的功能的描述，并附有指令编码的详细表格。
- **操作**: 详细说明对操作数进行的操作。
- **行为和限制**: 逐项描述指令中涉及的每个操作数的行为，以及适用于操作数或指令的任何限制。

1.8.5 伪代码约定

提出伪代码是为了以更简洁的形式描述算法。本文件中的所有算法都不打算直接进行编译。代码是在与周围文本相对应的水平上呈现的。

在描述变量时，列表是一个无序的同质对象的集合。一个队列是一个同质对象的有序列表。除非另有说明，否则假设排序为先进先出。

伪代码以类似于 C 的格式呈现，在适当的地方使用 C 约定。编码风格，特别是缩进风格，是为了可读性，不一定符合 UEFI 规范的实现。

1.8.6 排版约定

本文件采用了以下描述的排版和说明性惯例。

- **纯文本**: 规范中的绝大部分描述性文本都使用普通文本字体。
- **纯文本（蓝色）**: 任何有下划线和蓝色的纯文本都表示与交叉参考资料的活动链接。点击该词，就可以跟踪超链接。
- **加粗**: 在文本中，粗体字标识了一个处理器寄存器的名称。在其他情况下，黑体字可以作为段落中的标题。
- **斜体**: 在文本中，斜体字可以用作强调，以引入一个新的术语或表示手册或规范的名称。
- **加粗等宽（暗红色）**: 计算机代码、示例代码段和所有原型代码段使用 **BOLD Monospace** 字体，颜色为暗红色。这些代码列表通常出现在一个或多个独立的段落中，尽管单词或片段也可以嵌入到一个正常的文本段落中。
- **加粗等宽（蓝色）**: 用粗体单色字体的字，下划线和蓝色的字，表示该功能或类型定义的代码定义的活动超链接。点击该词，即可进入超链接。

注意：出于管理和文件大小的考虑，每一页上只有第一次出现的参考文献是一个主动链接。同一页上的后续参考文献不会被主动链接到定义上，而是使用标准的、无下划线的 **BOLD Monospace** 字体。在页面上找到该名称的第一个实例（使用下划线的 **BOLD Monospace** 字体），点击该词即可跳转到该功能或类型的定义。

- 斜体等宽：在代码或文本中，斜体字表示必须提供的变量信息的占位符名称（即参数）。

1.8.7 数字格式

在本标准中，二进制数字是由仅由西方阿拉伯数字 0 和 1 组成的任何数字序列表示的，后面紧跟一个小写的 b（例如，0101b）。在二进制数字表示中的字符之间可以包含下划线或空格，以增加可读性或划分领域边界（例如，0 0101 1010b 或 0_0101_1010b）。

1.8.7.1 十六进制

十六进制数字在本标准中用 0x 表示，前面是仅由西阿拉伯数字 0 至 9 和/或大写英文字母 A 至 F 组成的任何数字序列（例如，0xFA23）。十六进制数字表示中的字符之间可以包含下划线或空格，以增加可读性或划定字段边界（例如，0xB FD8C FA23 或 0xB_FD8C_FA23）。

1.8.7.2 十进制

在本标准中，小数是由仅由阿拉伯数字 0 到 9 组成的任何数字序列来表示的，后面不紧跟小写的 b 或小写的 h（例如，25）。本标准使用以下惯例来表示小数：

- 小数点分隔符（即分隔数字的整数部分和小数部分）是一个句号；
- 千位数分隔符（即分隔数字部分的三位数组）是一个逗号；
- 千位数分隔符用于数字的整数部分，不用于数字的小数部分。

1.8.8 二进制前缀

本标准使用国际单位制（SI）中定义的前缀来表示 10 的幂值。见“SI 二进制前缀”标题下的“UEFI 相关文件链接”（<http://uefi.org/uefi>）。

Table 1-1 SI prefixes

10^3	1,000	kilo	K
10^6	1,000,000	mega	M
10^9	1,000,000,000	giga	G

本标准使用 ISO/IEC 80000-13《数量和单位-第 13 部分：信息科学和技术》和 IEEE 1514《二进制倍数前缀标准》中定义的二进制前缀，用于表示 2 的幂值。

Table 1-2 Binary prefixes

Factor	Factor	Name	Symbol
2^{10}	1,024	kibi	Ki
2^{20}	1,048,576	mebi	Mi
2^{30}	1,073,741,824	gibi	Gi

例如，4 KB 意味着 4000 个字节，4 KiB 意味着 4096 个字节。

1.8.9 修订号

对 UEFI 规范的更新被认为是新的修订或勘误表，如下所述：

- 当有实质性的新内容或可能修改现有行为的变化时，就会产生一个新的修订。新的修订版由一个主要的次要的版本号来指定（例如：xx.yy）。在变化特别小的情况下，我们可能有一个 major.minor.minor 的命名惯例（例如 xx.yy.zz）。
- 当批准的规范更新不包括任何重要的新材料或修改现有行为时，就会产生勘误的版本。勘误的指定方法是在版本号后面加上一个大写字母，如 xx.yy 勘误 A。

2 概述

UEFI 允许通过加载 UEFI 驱动程序和 UEFI 应用程序镜像来扩展平台固件。加载 UEFI 驱动程序和 UEFI 应用程序后，它们可以访问所有 UEFI 定义的运行时和启动服务。见图 2-1

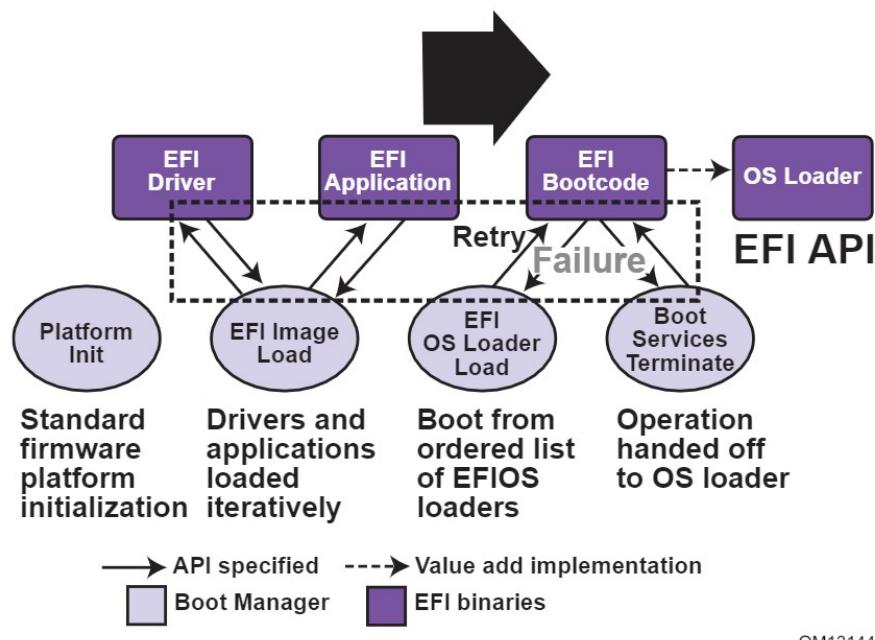


图 2. 启动顺序

UEFI 允许将来自 OS 加载程序和平台固件的启动菜单合并到单个平台固件菜单中。这些平台固件菜单，将允许从 UEFI 启动服务支持的任何启动介质上的任何分区中选择任何 UEFI OS 加载程序。UEFI OS 加载程序可以支持用户界面上的多个选项。还可以包括传统启动选项，例如从平台固件启动菜单中的 A: 或 C: 驱动器启动。

UEFI 支持从包含 UEFI 操作系统加载程序或 UEFI 定义的系统分区的媒介启动。UEFI 需要 UEFI 定义的系统分区才能从块设备启动。UEFI 不需要对分区的第一个扇区进行任何更改，因此可以构建媒介并在旧架构和 UEFI 平台上启动。

2.1 启动管理器

UEFI 包含一个启动管理器，它允许加载符合本规范的应用程序或者驱动程序，这些程序可以放在任何符合 UEFI 规范的文件系统上。它也允许通过使用 UEFI 定义的镜像加载服务来加载符合该规范编写的应用程序（包括操作系统第一阶段加载器）或 UEFI 驱动程序。UEFI 定义了 NVRAM 变量，用来指向要加载的文件。这些变量还包含直接传递给 UEFI 应用程序的应用程序特定数据。这些变量还包含一个可读的字符串，可以在菜单中显示给用户。

UEFI 定义的变量允许系统固件包含一个启动菜单，可以指向所有的操作系统，甚至是同一操作系统的多个

版本。UEFI 的设计目标是要有一套可以在平台固件中存在的启动菜单。UEFI 只指定了用于选择启动选项的 NVRAM 变量。UEFI 将菜单系统的实现作为增值的实现空间。

UEFI 大大扩展了系统的启动灵活性，超过了目前 PC-AT 级系统的技术状态。今天的 PC-AT 级系统被限制在从第一个软盘、硬盘、CD-ROM、USB 键或连接到系统的网卡启动。从一个普通的硬盘驱动器启动会导致操作系统之间的许多互操作性问题，以及同一供应商的不同版本的操作系统。

2.1.1 UEFI 镜像

UEFI 镜像是由 UEFI 定义的一类文件，包含可执行代码。UEFI 镜像最突出的特点是，UEFI 镜像文件的第一组字节包含一个镜像头，定义了可执行镜像的编码。

UEFI 使用 PE32+ 镜像格式的一个子集，并修改了头签名。对 PE32+ 镜像中签名值的修改是为了将 UEFI 镜像与正常的 PE32 可执行文件区分开来。PE32 的“+”添加提供了标准 PE32 格式的 64 位重定位修复扩展。

对于具有 UEFI 镜像签名的镜像，PE 镜像头中的 *Subsystem* 值定义如下。镜像类型之间的主要区别是固件将镜像加载到的内存类型，以及镜像的入口点退出或返回时采取的行动。当控制权从镜像入口点返回时，UEFI 应用程序镜像总是被卸载。一个 UEFI 驱动镜像只有在控制权被传回，并有 UEFI 错误代码时才被卸载。

```
1 // PE32+ Subsystem type for EFI images
2 #define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION 10
3 #define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER 11
4 #define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER 12
5 // PE32+ Machine type for EFI images
6 #define EFI_IMAGE_MACHINE_IA32 0x014c
7 #define EFI_IMAGE_MACHINE_IA64 0x0200
8 #define EFI_IMAGE_MACHINE_EBC 0x0EBC
9 #define EFI_IMAGE_MACHINE_X64 0x8664
10 #define EFI_IMAGE_MACHINE_ARMTHUMB_MIXED 0x01C2
11 #define EFI_IMAGE_MACHINE_AARCH64 0xAA64
12 #define EFI_IMAGE_MACHINE_RISCV32 0x5032
13 #define EFI_IMAGE_MACHINE_RISCV64 0x5064
14 #define EFI_IMAGE_MACHINE_RISCV128 0x5128
```

注意：选择这种镜像类型是为了使 UEFI 镜像包含 Thumb 和 Thumb2 指令，同时将 EFI 接口本身定义为 ARM 模式。

Table 2-1 UEFI Image Memory Types

Subsystem Type	Code Memory Type	Data Memory Type
EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION	EfiLoaderCode	EfiLoaderData
EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER	EfiBootServicesCode	EfiBootServicesData
EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER	EfiRuntimeServicesCode	EfiRuntimeServicesData

在 PE 镜像文件头中的 `Machine` 值是用来指示镜像的机器码类型。带有 UEFI 镜像签名的镜像的机器码类型定义如下。一个给定的平台必须实现该平台的本地镜像类型和 EFI 字节码 (EBC) 的镜像类型。对其他机器码类型的支持对平台来说是可选的。

UEFI 镜像是通过 `EFI_BOOT_SERVICES.LoadImage()` 启动服务加载到内存中。这个服务将一个 PE32+ 格式的镜像加载到内存中。这个 PE32+ 加载器需要将 PE32+ 镜像的所有部分加载到内存中。一旦镜像被加载到内存中，并进行了适当的修复，根据处理器（32 位，64 位或 128 位）的应用程序的正常间接调用惯例，控制权被转到到 `AddressOfEntryPoint` 处的镜像。所有其他与 UEFI 镜像的链接都是通过编程完成的。

2.1.2 UEFI 应用程序

根据该规范编写的应用程序，由启动管理器或其他 UEFI 应用程序加载。为了加载 UEFI 应用程序，固件会分配足够的内存来容纳镜像，将 UEFI 应用程序镜像中的部分复制到分配的内存中，并应用所需的重定位修复。重定位完成后，分配的内存将被设置为镜像的代码和数据。然后控制权被转移到 UEFI 应用程序的入口点。当应用程序从其入口点返回时，或者当它调用启动服务 `EFI_BOOT_SERVICES.Exit()` 时，UEFI 应用程序被从内存中卸载，控制权被返回到加载 UEFI 应用程序的 UEFI 组件。

当 Boot Manager 加载一个 UEFI 应用程序时，镜像句柄可被用来定位 UEFI 应用程序的“加载选项”。加载选项存储在非 volatile 性存储器中，与正在加载的 UEFI 应用程序相关，并由 Boot Manager 执行。

2.1.3 UEFI 操作系统加载器

UEFI 操作系统加载器 (UEFI OS Loader) 是一种特殊类型的 UEFI 应用程序，通常从符合本规范的固件中接管对系统的控制。当加载时，UEFI 操作系统加载器的行为与其他 UEFI 应用程序一样，它只能使用它从固件中分配的内存，并且只能使用 UEFI 服务和协议来访问固件所暴露的设备。如果 UEFI 操作系统加载器包含任何启动服务类型的驱动函数，它必须使用适当的 UEFI 接口来获得对总线特定资源的访问。也就是说，I/O 和内存映射的设备寄存器，必须通过总线特定的 I/O 调用来访问，就像 UEFI 驱动程序所执行的那样。

如果 UEFI 操作系统加载器遇到问题，不能正确加载其操作系统，它可以释放所有分配的资源，并通过启动服务 `Exit()` 调用将控制权返回给固件。`Exit()` 调用允许返回一个错误代码和 `ExitData`。`ExitData` 包含一个字

字符串和操作系统加载器特定的数据。

如果 UEFI 操作系统加载器成功加载其操作系统, 它可以通过使用启动服务 `EFI_BOOT_SERVICES.ExitBootServices()` 来控制系统。在成功调用 `ExitBootServices()` 后, 系统中所有的启动服务被终止, 包括内存管理, UEFI 操作系统加载器负责系统的继续运行。

2.1.4 UEFI 驱动

UEFI 驱动程序是由启动管理器、符合本规范的固件或其他 UEFI 应用程序加载的。为了加载 UEFI 驱动程序, 固件会分配足够的内存来容纳镜像, 将 UEFI 驱动程序镜像中的部分复制到分配的内存中, 并应用需要的重定位修复。一旦完成, 分配的内存就会被保存代码和数据类型, 以用于镜像。然后控制权被转移到 UEFI 驱动的入口点。当 UEFI 驱动从其入口点返回时, 或者当它调用启动服务 `EFI_BOOT_SERVICES.Exit()` 时, UEFI 驱动被选择性地从内存中卸载, 控制被返回到加载 UEFI 驱动的组件。如果 UEFI 驱动程序返回的状态码是 `EFI_SUCCESS`, 它就不会从内存中卸载。如果 UEFI 驱动的返回代码是一个错误的状态代码, 那么该驱动将从内存中卸载。

有两种类型的 UEFI 驱动: 启动服务驱动 (boot service drivers) 和运行时驱动 (runtime drivers)。这两种驱动类型的唯一区别是, UEFI 运行时驱动是在 UEFI 操作系统加载器通过启动服务 `EFI_BOOT_SERVICES.ExitBootServices()` 控制了平台之后才可用。

当 `ExitBootServices()` 被调用时, UEFI 启动服务驱动被终止, UEFI 启动服务驱动所占用的所有内存资源被释放, 以便在操作系统环境中使用。

当操作系统调用 `SetVirtualAddressMap()` 时, `EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER` 类型的运行时驱动程序运行在虚拟地址上。

2.2 固件核心

本节概述了 UEFI 定义的服务。这些服务包括启动服务和运行时服务。

2.2.1 UEFI 服务

UEFI 接口的目的是定义一个通用的启动环境抽象, 供加载的 UEFI 镜像使用, 其中包括 UEFI 驱动程序、UEFI 应用程序和 UEFI 操作系统加载器。这些调用被定义为一个完整的 64 位接口, 这样就为未来的发展留出了空间。这套抽象的平台调用的目标是允许平台和操作系统独立发展和创新。同时, 一套标准的原始运行时服务 (TODO) 可以被操作系统使用。

本节中定义的平台接口允许使用标准的即插即用 Option ROM, 作为启动服务的基本实现方法。这些接口被设计成可以映射到传统的接口。这些接口并没有受到传统 Option ROM 的任何限制。

UEFI 平台接口的目的是在平台和平台上启动的操作系统之间提供一个抽象。UEFI 规范还提供了诊断程序或

实用程序与平台之间的抽象性。但是，它并不试图实现一个完整的诊断操作系统环境。我们设想在 UEFI 系统上可以很容易地建立一个类似诊断操作系统的环境。本规范没有描述这样的诊断环境。

本规范增加的接口分为以下几类，并在本文件后面详细介绍：

- 运行时服务
- 启动服务接口，有以下子类：
 - 全局启动服务接口
 - 基于设备句柄的启动服务接口
 - 设备协议
 - 协议服务

2.2.2 运行时服务

本节介绍 UEFI 运行时服务功能。运行时服务的主要目的是将平台的硬件实现的次要部分从操作系统中抽象出来。运行时服务功能在启动过程中可用，在运行时也可用，前提是操作系统切换到平面物理寻址模式来进行运行时调用。然而，如果操作系统加载器或操作系统使用运行时服务 `SetVirtualAddressMap()` 服务，操作系统将只能在虚拟寻址模式下调用运行时服务。所有的运行时接口都是非阻塞接口，如果需要的话，可以在禁用中断的情况下调用。为了确保与现有平台的最大兼容性，建议将所有构成运行时服务的 UEFI 模块在 `MemoryMap` 中表示为类型为 `EfiRuntimeServicesCode` 的单一 `EFI_MEMORY_DESCRIPTOR`。

在所有情况下，运行时服务使用的内存必须被保留，不被操作系统使用。运行时服务的内存总是对 UEFI 函数可用，绝不会被操作系统或其组件直接操纵。UEFI 负责定义运行时服务使用的硬件资源，因此当运行时服务调用时，操作系统可以与这些资源同步，或者保证操作系统永远不使用这些资源。

表 2-2 列出了运行时服务包含的函数。

函数名	功能描述
<code>GetTime()</code>	返回当前硬件时间、时间上下文和时钟性能。
<code>SetTime()</code>	设置当前硬件时间和时间上下文 (TODO)。
<code>GetWakeupTime()</code>	返回当前的唤醒定时器的状态。
<code>SetWakeupTime()</code>	设置当前唤醒定时器 (启用或禁用)。
<code>GetVariable()</code>	返回系统变量的值。
<code>GetNextVariableName()</code>	枚举所有系统变量。
<code>SetVariable()</code>	设置并在需要时创建一个系统变量。
<code>SetVirtualAddressMap()</code>	将所有运行时函数从物理寻址切换到虚拟寻址。

函数名	功能描述
ConvertPointer()	用于将一个指针从物理寻址转换为虚拟寻址。
GetNextHighMonotonicCount()	包含平台的单调计数器功能。
ResetSystem()	重置所有处理器和设备并重新启动系统。
UpdateCapsule()	用虚拟和物理映射将胶囊（TODO）传递给固件。
QueryCapsuleCapabilities()	返回是否可以通过 UpdateCapsule() 支持胶囊。（TODO）
QueryVariableInfo()	返回有关 EFI 变量存储的信息。

译者注：时钟性能，用 `EFI_TIME_CAPABILITIES` 表示，包括分辨率、精度等指标。

2.3 调用约定

除非另有说明，UEFI 规范中定义的所有函数都是通过 C 语言编译器中常见的、架构上定义的调用惯例中的指针来调用。各种全局 UEFI 函数的指针在 `EFI_RUNTIME_SERVICES` 和 `EFI_BOOT_SERVICES` 表中找到，它们通过系统表（System Table）定位。对本规范中定义的其他函数的指针通过设备句柄动态地定位。在所有情况下，指向 UEFI 函数的指针都用 `EFI API` 这个词来转换。这允许每个架构的编译器提供适当的编译器关键字以实现所需的调用约定。当向启动服务、运行时服务和协议接口传递指针参数时，调用者有以下责任：

- 传递引用物理内存位置的指针参数是调用者的责任。如果传递的指针没有指向物理内存位置（即内存映射的 I/O 区域），结果是不可预测的，系统可能会停止运行。
- 调用者有责任以正确的对齐方式传递指针参数。如果一个未对齐的指针被传递给一个函数，其结果是不可预测的，系统可能会停止运行。
- 除非明确允许，否则调用者不要向函数传递一个 `NULL` 参数。如果一个 `NULL` 指针被传递给一个函数，其结果是不可预测的，系统可能会挂起。
- 除非另有说明，如果函数以错误方式返回，调用者不应该对指针参数的状态做任何假设。
- 调用者不得按值传递大于本机寻址宽度的结构，这些结构必须由调用者通过引用（通过指针）传递。在堆栈中传递大于本机宽度（在支持的 32 位处理器上为 4 字节；在支持的 64 位处理器指令上为 8 字节）的结构将产生未定义的结果。

支持的 32 位和支持的 64 位应用程序的调用惯例在下文有更详细的描述。任何函数或协议都应该返回有效的返回代码。

一个 UEFI 模块的所有公共接口必须遵循 UEFI 的调用惯例。公共接口包括镜像入口点、UEFI 事件处理程序和协议成员函数。`EFI API` 类型被用来表示符合本节中定义的调用惯例。非公共接口，如私有函数和静态库调用，不需要遵循 UEFI 的调用约定，并可能被编译器优化。

2.3.1 数据类型

表 2-3 列出了接口定义中使用的常见数据类型，表 2-4 列出了它们的修饰语。除非另有规定，所有数据类型都是自然对齐的。结构在等于结构最大内部基准的边界上对齐，内部数据被隐含地填充以实现自然对齐。
传入或由 UEFI 接口返回的指针值必须为底层类型提供自然对齐。

记号	描述
BOOLEAN	逻辑布尔值。1 字节的值，0 代表 FALSE 或 1 代表 TRUE。其他值是未定义的。
INTN	本地宽度的有符号的值。(在支持的 32 位处理器指令上为 4 字节，在支持的 64 位处理器指令上为 8 字节，在支持的 128 位处理器指令上为 16 字节)。
UINTN	本地宽度的无符号值。(在支持的 32 位处理器指令上为 4 字节，在支持的 64 位处理器指令上为 8 字节，在支持的 128 位处理器指令上为 16 字节)。
INT8	1 字节有符号值。
UINT8	1 字节无符号值。
INT16	2 字节有符号值。
UINT16	2 字节无符号值。
INT32	4 字节有符号值。
UINT32	4 字节无符号值。
INT64	8 字节有符号值。
UINT64	8 字节无符号值。
INT128	16 字节有符号值。
UINT128	16 字节无符号值。
CHAR8	1 字节的字符。除非另有规定，所有 1 字节或 ASCII 字符和字符串都以 8 位 ASCII 编码格式存储，使用 ISO-拉丁文 1 字符集。
CHAR16	2 字节字符。除非另有规定，所有字符和字符串都以 UCS-2 编码格式存储，该格式由 Unicode 2.1 和 ISO/IEC 10646 标准定义。
VOID	未声明的类型。

记号	描述
EFI_GUID	128 位缓冲区，包含一个唯一的标识符值。除非另有规定，否则在 64 位边界对齐。
EFI_STATUS	状态代码。类型为 <code>UINTN</code> 。
EFI_HANDLE	一个相关接口的集合。类型 <code>VOID *</code> 。
EFI_EVENT	一个事件结构的指针。类型 <code>VOID *</code> 。
EFI_LBA	逻辑块地址。类型 <code>UINT64</code> 。
EFI_TPL	任务优先级别。类型为 <code>UINTN</code> 。
EFI_MAC_ADDRESS	32 字节的缓冲区，包含一个网络媒体访问控制地址。
EFI_IPv4_ADDRESS	4 字节的缓冲区。一个 IPv4 互联网协议地址。
EFI_IPv6_ADDRESS	16 字节的缓冲区。一个 IPv6 互联网协议地址。
EFI_IP_ADDRESS	16 字节的缓冲区，以 4 字节为边界对齐。一个 IPv4 或 IPv6 互联网协议地址。
sizeof (VOID *)	标准 ANSI C 枚举类型声明的元素。类型 <code>INT32</code> . 或 <code>UINT32</code> 。ANSI C 没有定义枚举的符号大小，所以它们不应该在结构中使用。ANSI C 的整数推广规则使 <code>INT32</code> 或 <code>UINT32</code> 在作为参数传递给函数时可以互换。
比特域	在支持的 32 位处理器指令上是 4 字节。在支持的 64 位处理器指令上为 8 字节。在支持的 128 位处理器上为 16 字节。 比特域的排序：第 0 位是最不重要的位。

记号	描述
IN	Datum 被传递给函数。
OUT	Datum 从函数中返回。
OPTIONAL	向函数传递数据点是可选的，如果不提供数据点的值，可以传递 <code>NULL</code> 。
CONST	Datum 是只读的。
EFIAPI	定义了 UEFI 接口的调用惯例。

2.3.2 IA32 平台

所有的函数都是按照 C 语言的调用惯例来调用的。在整个函数调用过程中都是 **volatile** 的通用寄存器是 **eax**, **ecx**, 和 **edx**。所有其他的通用寄存器都是**nonvolatile**的，并且被目标函数保留下来。此外，除非函数定义另有规定，所有其他的寄存器都被保留。

在操作系统调用 `ExitBootServices()` 之前，固件启动服务和运行时服务以下列处理器执行模式运行：

- 单处理器，在下列参考资料的第 8 章详细描述：
 - Intel 64 and IA-32 Architectures Software Developer's Manual
 - Volume 3, System Programming Guide, Part 1
 - Order Number: 253668-033US, December 2009
 - See [Links to UEFI-Related Documents](#) under the heading Intel Processor Manuals
- 保护模式
- 可以启用分页模式。如果启用了分页模式，推荐使用 PAE（Physical Address Extensions，物理地址扩展）模式。如果启用了分页模式，则 UEFI 内存映射定义的任何内存空间都是标识映射的（虚拟地址等于物理地址）。对其他区域的映射是未定义的，可能因实现情况不同而不同。
- 选择器（Selectors）被设置为平坦，除此之外不使用。
- 中断被启用—尽管除了 UEFI 启动服务的定时器功能外，不支持任何中断服务（所有加载的设备驱动都是通过“轮询”同步服务的）。
- EFLAG 中的方向标志是明确的。
- 其他通用标志寄存器未定义。
- 128 KiB 或更多可用堆栈空间。
- 堆栈必须是 16 字节对齐的。堆栈可以在身份（TODO）映射的页面表中被标记为不可执行。
- 浮点控制字必须被初始化为 **0x027F**（所有例外都被屏蔽，双精度，四舍五入）。
- 多媒体扩展控制字（如果支持的话）必须被初始化为 **0x1F80**（所有例外都被屏蔽了，四舍五入，屏蔽下溢时冲到零）。
- CR0.EM 必须为零。
- CR0.TS 必须为零。

根据本规范编写的应用程序可以改变处理器的执行模式，但 UEFI 镜像必须确保固件启动服务和运行时服务以规定的执行环境执行。

在操作系统调用 `ExitBootServices()` 后，固件启动服务不再可用，调用任何启动服务都是非法的。在 `ExitBootServices` 之后，固件运行服务仍然可用，如果 `SetVirtualAddressMap()` 被调用描述了固件运行服务使用的所有虚拟地址范围，则可以在启用分页和虚拟地址指针的情况下调用。

一个操作系统要使用任何 UEFI 运行时服务，必须：

- 保留内存映射中所有标记为运行时代码和运行时数据的内存；
- 调用运行时服务函数，条件如下：

- 在保护模式下;
 - 分页可能会也可能不会被启用，但是如果分页被启用并且 `SetVirtualAddressMap()` 没有被调用，任何由 UEFI 内存映射定义的内存空间都是身份 (TODO) 映射的（虚拟地址等于物理地址），尽管某些区域的属性可能没有所有的读、写和执行属性，或者为了平台保护的目的没有标记。对其他区域的映射是未定义的，可能因实现不同而不同。调用此函数后，有关内存映射的详细信息，请参见 `SetVirtualAddressMap()` 的描述。
 - EFLAGS 中的方向标志清除;
 - 4 KiB 或更多的可用堆栈空间;
 - 堆栈必须是 16 字节对齐的;
 - 浮点控制字必须被初始化为 `0x027F` (所有例外都被屏蔽，双精度，四舍五入);
 - 多媒体扩展控制字 (如果支持) 必须初始化为 `0x1F80` (所有异常都被屏蔽、四舍五入、为屏蔽下溢刷新为零);
 - CRO.EM 必须为零;
 - CRO.TS 必须为零;
 - 由调用者决定禁用或启用中断功能;
- 在启动时加载的 ACPI 表可以包含在 `EfiACPIReclaimMemory` (推荐) 或 `EfiACPIMemoryNVS` 类型的内存中。ACPI FACS 必须包含在 `EfiACPIMemoryNVS` 类型的内存中;
 - 系统固件不得为任何 `EfiACPIReclaimMemory` 或 `EfiACPIMemoryNVS` 类型的内存描述符请求虚拟映射;
 - `EfiACPIReclaimMemory` 和 `EfiACPIMemoryNVS` 类型的 EFI 内存描述符必须在 4KiB 边界上对齐，并且必须是 4KiB 的倍数;
 - 任何通过 `EFI_MEMORY_DESCRIPTOR` 请求虚拟映射的 UEFI 内存描述符,如果设置了 `EFI_MEMORY_RUNTIME` 位，必须在 4KiB 边界上对齐，并且必须是 4KiB 的倍数;
 - ACPI 内存操作区域必须从 UEFI 内存映射中继承可缓存属性。如果系统内存映射不包含可缓存属性，ACPI 内存操作区域必须从 ACPI 命名空间中继承其可缓存属性。如果在系统内存映射或 ACPI 命名空间中不存在可缓存属性，那么该区域必须被认为是不可缓存的。
 - 在运行时加载的 ACPI 表必须包含在 `EfiACPIMemoryNVS` 类型的内存中。运行时加载的 ACPI 表的可缓存属性应在 UEFI 内存映射中定义。如果 UEFI 内存映射中不存在关于表位置的信息，可从 ACPI 内存描述符中获得缓存属性。如果 UEFI 内存映射或 ACPI 内存描述符中没有关于表位置的信息，则假定该表为非缓存的。
 - 一般来说，在启动时加载的 UEFI 配置表（例如，SMBIOS 表）可以包含在 `EfiRuntimeServicesData` (推荐)、`EfiBootServicesData`、`EfiACPIReclaimMemory` 或 `EfiACPIMemoryNVS` 类型的内存中。在运行时加载的表必须包含在 `EfiRuntimeServicesData` (推荐) 或 `EfiACPIMemoryNVS` 类型的内存中。

注意：以前的 EFI 规范允许在运行时加载的 ACPI 表在 `EfiReservedMemoryType` 中，对于其他 EFI 配置表没有提供指导。`EfiReservedMemoryType` 不打算用于存储任何 EFI 配置表。另外，只有符合 UEFI 规范的操作系统才能保证处理 `EfiBootServicesData` 类型内存中的 SMBIOS 表。

2.3.2.1 切换状态

加载 32 位 UEFI 操作系统时，系统固件将控制权移交给平面 32 位模式下的操作系统。所有描述符都设置为它们的 4GiB 限制，以便可以从所有段访问所有内存。

图 2-2 显示在支持的 32 位系统上，镜像的 PE32+ 头中的 `AddressOfEntryPoint` 被调用后的堆栈。所有的 UEFI 镜像入口点都需要两个参数。这两个参数是 UEFI 镜像的句柄，以及一个指向 EFI 系统表的指针。

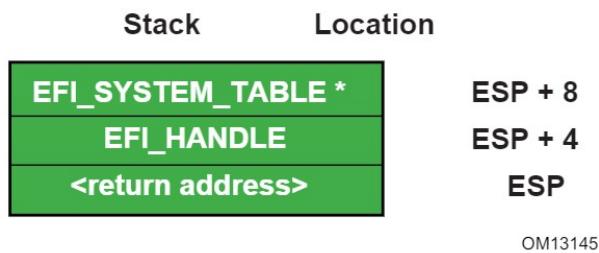


图 3. 调用 `AddressOfEntryPoint` 后的堆栈，IA-32

2.3.2.2 调用约定

所有的函数都是按照 C 语言的调用惯例来调用的。在整个函数调用过程中的 `volatile` 寄存器是 `eax`, `ecx`, 和 `edx`。所有其他的通用寄存器都是 `nonvolatile` 的，并由目标函数保留。

此外，除非函数定义另有规定，所有其他 CPU 寄存器（包括 MMX 和 XMM）都被保留。

浮点状态寄存器不被目标函数保留。浮点控制寄存器和 MMX 控制寄存器由目标函数保存。

如果返回值是浮点数或双精度浮点类型，则以 `ST(0)` 返回。

2.3.3 基于 Intel® Itanium® 的平台

UEFI 作为 SAL 执行环境的扩展来执行，其规则与 SAL 规范所规定的相同。

在启动服务时间内，处理器处于以下执行模式：

- 单处理器，详见以下参考资料的第 13.1.2 章：
 - Intel Itanium Architecture Software Developer's Manual
 - Volume 2: System Architecture
 - Revision 2.2
 - January 2006
 - See [Links to UEFI-Related Documents](#) under the heading “Intel Itanium Documentation”
 - Document Number: 245318-005
- 物理模式

- 128 KiB 或更多可用堆栈空间;
- 16 KiB 或更多的可用备份存储空间;
 - FPSR.traps: 设置为全 1 (禁用所有异常);
 - FPSR.sf0:
 - .pc:Precision Control - 11b (extended precision)
 - .rc:Rounding Control - 0 (round to nearest)
 - .wre:Widest Range Exponent - 0 (IEEE mode)
 - .ftz:Flush-To-Zero mode - 0 (off)
 - FPSR.sf1:
 - .td:Traps Disable = 1 (traps disabled)
 - .pc:Precision Control - 11b (extended precision)
 - .rc:Rounding Control - 0 (round to nearest)
 - wre:Widest Range Exponent - 1 (full register exponent range)
 - ftz:Flush-To-Zero mode - 0 (off)
 - FPSR.sf2,3:
 - .td:Traps Disable = 1 (traps disabled)
 - pc:Precision Control - 11b (extended precision)
 - rc:Rounding Control - 0 (round to nearest)
 - wre:Widest Range Exponent - 0 (IEEE mode)
 - ftz:Flush-To-Zero mode - 0 (off)

根据本规范编写的应用程序可以改变处理器的执行模式，但 UEFI 镜像必须确保固件启动服务和运行时服务以规定的执行环境执行。

在操作系统调用 `ExitBootServices()` 后，固件启动服务不再可用，调用任何启动服务都是非法的。在 `ExitBootServices` 之后，固件运行时服务仍然可用，当调用运行时服务时，分页可能被启用，也可能不被启用，但是如果分页被启用并且 `SetVirtualAddressMap()` 没有被调用，任何由 UEFI 内存映射定义的内存空间都是身份映射 (TODO) 的（虚拟地址等于物理地址）。对其他区域的映射是未定义的，可能因实现不同而不同。参见 `SetVirtualAddressMap()` 的描述，了解该函数被调用后内存映射的细节。在 `ExitBootServices()` 之后，运行时服务函数可以在禁用或启用中断的情况下被调用，由调用者决定。

- 在启动时加载的 ACPI 表可以包含在 `EfiACPIReclaimMemory` (推荐) 或 `EfiACPIMemoryNVS` 类型的内存中。ACPI FACS 必须包含在 `EfiACPIMemoryNVS` 类型的内存中。
- 系统固件不得为任何 `EfiACPIReclaimMemory` 或 `EfiACPIMemoryNVS` 类型的内存描述符请求虚拟映射。
- EFI 内存描述符的类型 `EfiACPIReclaimMemory` 和 `EfiACPIMemoryNVS`，必须在 8KiB 边界上对齐，并且大小必须是 8KiB 的倍数。
- 任何通过 `EFI_MEMORY_DESCRIPTOR` 请求虚拟映射的 UEFI 内存描述符，如果设置了 `EFI_MEMORY_RUNTIME` 位，必须在 8KiB 边界上对齐，并且大小必须是 8KiB 的倍数。
- ACPI 内存操作区域必须从 UEFI 内存映射中继承可缓存属性。如果系统内存映射不包含缓存属性，

ACPI 内存操作区域必须从 ACPI 命名空间中继承其缓存属性。

- 如果没有可缓存性属性在系统内存映射或 ACPI 命名空间中存在，那么该区域必须被认为是不可缓存的。
- 在运行时加载的 ACPI 表必须包含在 `EfiACPIMemoryNVS` 类型的内存中。运行时加载的 ACPI 表的可缓存属性应在 UEFI 内存映射中定义。如果 UEFI 内存映射中不存在关于表位置的信息，可从 ACPI 内存描述符中获得缓存属性。如果 UEFI 内存映射或 ACPI 内存描述符中没有关于表位置的信息，则假定该表为非缓存的（non-cacheable）。
- 一般来说，启动时加载的配置表（如 SMBIOS 表）可以包含在 `EfiRuntimeServicesData`（推荐）、`EfiBootServicesData`、`EfiACPIReclaimMemory` 或 `EfiACPIMemoryNVS` 类型的内存中。在运行时加载的表必须包含在 `EfiRuntimeServicesData`（推荐）或 `EfiACPIMemoryNVS` 类型的内存中。

注意：以前的 EFI 规范允许在运行时加载的 ACPI 表在 `EfiReservedMemoryType` 中，对其他 EFI 配置表没有提供指导。`EfiReservedMemoryType` 不打算被固件使用。另外，只有符合 UEFI 规范的操作系统才能保证处理 `EfiBootServicesData` 类型内存中的 SMBIOS 表。

详情请参考 IA-64 系统抽象层规范（见附录 Q）。

UEFI 程序以 P64 C 调用约定被调用，这些调用约定为基于 Intel® Itanium® 的应用程序定义的。更多信息请参考 *64 Bit Runtime Architecture and Software Conventions for IA-64*（见附录 Q）。

2.3.3.1 切换状态

UEFI 使用 P64 C 调用约定，这些调用约定是为基于 Itanium 的操作系统定义的。图 2-3 显示了 `ImageEntryPoint` 在基于 Itanium 的系统上被调用后的堆栈。参数也被存储在寄存器中：`out0` 包含 `EFI_HANDLE`，`out1` 包含 `EFI_SYSTEM_TABLE` 的地址。UEFI 镜像的 `gp` 将从镜像的 PE32+ 头中 `AddressOfEntryPoint` 所指向的 `pLabel` 加载。所有的 UEFI 镜像入口点都有两个参数。这两个参数是镜像的句柄和一个指向系统表的指针。

Stack	Location	Register
<code>EFI_SYSTEM_TABLE *</code>	<code>SP + 8</code>	<code>out1</code>
<code>EFI_HANDLE</code>	<code>SP</code>	<code>out0</code>

OM13146

SAL 规范（见附录 Q）定义了系统寄存器在启动交换（handoff）时的状态。SAL 规范还定义了哪些系统寄存器只能在 UEFI 启动服务被正确终止后使用。

2.3.3.2 调用约定

UEFI 作为 SAL 执行环境的一个扩展来执行，其规则与 SAL 的规定相同。UEFI 程序是使用为基于 Intel® Itanium® 的应用程序定义的 P64 C 调用惯例来调用的。请参考文件 *64 Bit Runtime Architecture and Software Conventions for IA-64*（更多信息请参见术语表）。

对于浮点数，函数只能使用较低的 32 个浮点寄存器返回值出现在 f8-f15 寄存器中。单精度值、双精度值和扩展值都使用适当的格式返回。寄存器 f6-f7 是本地寄存器，不为调用者保留。所有其他的浮点寄存器都被保留下来。注意，在编译 UEFI 程序时，可能需要指定一个特殊的开关，以保证编译器不使用 f32-f127，这些寄存器在 Itanium 的常规调用约定中通常不被保留。使用保留的浮点寄存器之一的程序必须保存和恢复调用者的原始内容，而不产生 NaT 消耗故障（NaT consumption fault）。

浮点参数尽可能在 f8-f15 寄存器中传递。超出寄存器的参数出现在内存中，这在 Itanium 软件约定和运行时架构指南的第 8.5 节有解释。在被调用的函数中，这些是本地寄存器，不会为调用者保留。寄存器 f6-f7 是本地寄存器，不为调用者保留。所有其他的浮点寄存器都被保留了。注意，在编译 UEFI 程序时，可能需要指定一个特殊的开关，以保证编译器不使用 f32-f127，这些寄存器在 Itanium 的常规调用约定中通常不被保留。使用保留的浮点寄存器之一的程序必须保存和恢复调用者的原始内容，而不产生 NaT 消耗故障。

浮点状态寄存器必须在对目标函数的调用中被保留。SF1,2,3 中的标志字段不会为调用者保留。返回时 SFO 中的标志字段将反映传入的值，并且位设置为 1，表示在作为被调用者的一部分执行的非推测浮点操作上检测到的任何 IEEE 异常。

被调用者执行的浮点操作可能需要软件仿真。调用者必须准备好处理 FP 软件辅助（FPSWA）的中断。被调用者不应该通过将 FPSR.trap 位改为 0，然后执行引起这种 trap 的浮点操作来引发 IEEE traps。

2.3.4 x64 平台

所有的函数都是以 C 语言的调用惯例来调用的。更多细节见 2.3.4.2 节。

在启动服务时间内，处理器处于以下执行模式。

- 单处理器，如第 8.4 章所述：
 - Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, System Programming Guide, Part 1, Order Number: 253668-033US, December 2009
 - 见“Intel Processor Manuals”标题下 UEFI [相关文件链接](#)。
- 长模式，64 位模式下；
- 启用分页模式，并且 UEFI 内存映射定义的任何内存空间都是身份映射的（虚拟地址等于物理地址），尽管某些区域的属性可能不具有所有读、写和执行属性，或者出于平台保护的目的未标记。到其他区域的映射，例如未接受内存的映射，是未定义的，并且可能因实现而异；
- 选择器被设置为平坦，除此之外不使用；
- 中断被启用—尽管除了 UEFI 启动服务的定时器功能外，不支持任何中断服务（所有加载的设备驱动都是通过“轮询”同步服务的）；
- EFLAG 中的方向标志是明确的；

- 其他通用标志寄存器未定义；
- 128 KiB 或更多可用堆栈空间；
- 堆栈必须是 16 字节对齐的。堆栈可以在身份映射（TODO）的页面表中被标记为不可执行；
- 浮点控制字必须初始化为 0x037F（所有例外都被屏蔽，双倍扩展精度，四舍五入）；
- 多媒体扩展控制字（如果支持的话）必须被初始化为 0x1F80（所有的例外都被屏蔽了，四舍五入，对于屏蔽的下溢，冲到零。）；
- CRO.EM 必须为零；
- CRO.TS 必须为零；

一个操作系统要使用任何 UEFI 运行时服务，必须：

- 保留内存映射中所有标记为运行时代码和运行时数据的内存；
- 调用运行时服务函数，条件如下：
 - 在长模式下，在 64 位模式下；
 - 开启分页；
- 所有的选择器设置为平坦的，虚拟等于物理地址。如果 UEFI 操作系统加载器或操作系统使用 [SetVirtualAddressMap\(\)](#) 来重新定位虚拟地址空间中的运行时服务，那么这个条件就不必满足。参见 [SetVirtualAddressMap\(\)](#) 的描述，以了解该函数被调用后的内存映射的细节；
- EFLAG 中的方向标志清除；
- 4 KiB 或更多可用堆栈空间；
- 堆栈必须是 16 字节对齐的；
- 浮点控制字必须初始化为 0x037F（所有例外都被屏蔽，双倍扩展精度，四舍五入）；
- 多媒体扩展控制字（如果支持的话）必须被初始化为 0x1F80（屏蔽所有异常，四舍五入，屏蔽下溢时刷新为 0）
- CRO.EM 必须为零；
- CRO.TS 必须为零；
- 中断可由调用者决定是否禁用或启用；
- 在启动时加载的 ACPI 表可以包含在 [EfiACPIReclaimMemory](#)（推荐）或 [EfiACPIMemoryNVS](#) 类型的内存中。ACPI FACS 必须包含在 [EfiACPIMemoryNVS](#) 类型的内存中；
- 系统固件不得为任何 [EfiACPIReclaimMemory](#) 或 [EfiACPIMemoryNVS](#) 类型的内存描述符请求虚拟映射；
- [EfiACPIReclaimMemory](#) 和 [EfiACPIMemoryNVS](#) 类型的 EFI 内存描述符必须在 4KiB 边界上对齐，并且大小必须是 4KiB 的倍数；
- 任何通过 [EFI_MEMORY_DESCRIPTOR](#) 请求虚拟映射的 UEFI 内存描述符，如果设置了 [EFI_MEMORY_RUNTIME](#) 位，必须在 4KiB 边界上对齐，并且大小必须是 4KiB 的倍数；
- ACPI 内存操作区域必须从 UEFI 内存映射中继承可缓存属性。如果系统内存映射不包含可缓存属性，ACPI 内存操作区域必须从 ACPI 命名空间中继承其可缓存属性。如果在系统内存映射或 ACPI 命名空间中不存在可缓存属性，那么该区域必须被认为是不可缓存的；
- 在运行时加载的 ACPI 表必须包含在 [EfiACPIMemoryNVS](#) 类型的内存中。运行时加载的 ACPI 表的可缓存属性应在 UEFI 内存映射中定义。如果 UEFI 内存映射中不存在关于表位置的信息，可从 ACPI 内存

描述符中获得缓存属性。如果 UEFI 内存映射或 ACPI 内存描述符中没有关于表位置的信息，则假定该表为非缓存的；

- 一般来说，在启动时加载的 UEFI 配置表（例如，SMBIOS 表）可以包含在 `EfiRuntimeServicesData`（推荐）、`EfiBootServicesData`、`EfiACPIReclaimMemory` 或 `EfiACPIMemoryNVS` 类型的内存中。在运行时加载的表必须包含在 `EfiRuntimeServicesData`（推荐）或 `EfiACPIMemoryNVS` 类型的内存中；

注意：以前的 EFI 规范允许在运行时加载的 ACPI 表在 `EfiReservedMemoryType` 中，对其他 EFI 配置表没有提供指导。`EfiReservedMemoryType` 不打算被固件使用。另外，只有符合 UEFI 规范的操作系统才能保证处理 `EfiBootServicesData` 类型内存中的 SMBIOS 表。

2.3.4.1 切换状态

- 1 `Rcx` – `EFI_HANDLE`
- 2 `Rdx` – `EFI_SYSTEM_TABLE`
- 3 `*RSP` – <return address>

2.3.4.2 详细的调用约定

调用者在寄存器中传递前四个整数参数。整数值在 `Rcx`、`Rdx`、`R8` 和 `R9` 寄存器中从左到右传递。调用者将第五个及以上的参数传入堆栈。所有的参数在传递的寄存器中必须是右对齐的。这确保被调用者只能处理寄存器中需要的位。

调用者通过一个指向调用者分配的内存的指针传递数组和字符串。调用者传递大小为 8、16、32 或 64 位的结构和联合体，就像它们是相同大小的整数一样。调用者不允许传递这些大小以外的结构和联合体，必须通过指针传递这些联合体和结构。

如果需要，被调用者必须将寄存器参数转储到其影子空间。最常见的要求是把一个参数的地址。

如果参数是通过 `varargs` 传递的，那么基本上适用于典型的参数传递，包括将第五个和随后的参数放到堆栈。被调用者必须转存那些地址被占用的参数。

固定在 64 位的返回值会在 `Rax` 寄存器中返回。如果返回值不在 64 位之内，那么调用者必须分配并传递一个返回值的指针作为第一个参数，`Rcx`。随后的参数会向右移动一个参数，例如，参数一会在 `Rdx` 中传递。要返回的用户定义类型必须是 1、2、4、8、16、32 或 64 位的长度。

寄存器 `Rax`, `Rcx` `Rdx` `R8`, `R9`, `R10`, `R11`, 和 `XMM0-XMM5` 是 `volatile` 的，因此，在函数调用时被销毁。

寄存器 `RBX`、`RBP`、`RDI`、`RSI`、`R12`、`R13`、`R14`、`R15` 和 `XMM6-XMM15` 被认为是 `non-volatile` 的，必须由使用调用的函数来保存和恢复。

函数指针是指向相应函数标签的指针，不需要特殊处理。

调用者必须总是在堆栈 16 字节对齐的情况下调用。

对于 `MMX`、`XMM` 和浮点值，可以存入 64 位的返回值通过 `RAX` 返回（包括 `MMX` 类型）。然而，`XMM` 的 128 位类

型、浮点和双数则在 `XMM0` 中返回。浮点状态寄存器不被目标函数保存。浮点和双精度参数在 `XMM0-XMM3` (最多 4 个) 中传递，通常用于该卡槽的整数卡槽 (`RCX`、`RDX`、`R8` 和 `R9`) 被忽略 (见例子)，反之亦然。`XMM` 类型从不通过即时值传递，而是通过一个指针传递给调用者分配的内存。`MMX` 类型将被传递，就像它们是相同大小的整数一样。在没有提供正确的异常处理程序的情况下，被调用者不得解除异常的屏蔽。

2.3.4.3 在应用程序中启用分页或替代翻译

启动服务定义了一个执行环境，其中不启用分页（支持 32 位）但启用翻译（translations, TODO）并且映射的虚拟地址等于物理地址 (x64)。本节将描述如何编写一个具有替代翻译或启用分页的应用程序。一些操作系统要求操作系统加载器能够在启动服务时启用操作系统要求的翻译。

如果 UEFI 应用程序使用自己的分页表、GDT 或 IDT，应用程序必须确保固件执行每个被取代的数据结构。当应用程序启用分页时，符合本规范的固件有两种方式可以执行。

- 显式固件调用
- 固件通过定时器事件抢占应用程序

启用翻译的应用程序可以在每次 UEFI 调用之前恢复固件所需的映射。然而，抢占的可能性可能要求启用转换的应用程序在启用备用转换时禁用中断。如果应用程序在调用 UEFI 中断 ISR 之前捕捉到中断并恢复 EFI 固件环境，那么翻译启用的应用程序启用中断是合法的。在 UEFI ISR 上下文被执行后，它将返回到翻译启用的应用程序上下文，并恢复应用程序所需的任何映射关系。

2.3.5 AArch32 平台

所有的函数都是按照第 2.3.5.3 节中规定的 C 语言调用惯例来调用的。此外，如果进程中存在未对齐访问支持，则调用操作系统可以假定启用了未对齐访问支持。

在启动服务时间内，处理器处于以下执行模式：

- 如果支持，应启用未对齐访问；否则会启用对齐错误
- 单处理器
- 特权模式
- MMU 被启用 (CP15 c1 系统控制寄存器 (SCTL) SCTL.M=1)，任何由 UEFI 内存映射定义的 RAM 都被身份映射 (虚拟地址等于物理地址)。对其他区域的映射是未定义的，可能因实现情况不同而不同。
- 核心将被配置如下 (在所有处理器架构的版本中都是通用的)：
 - 启用 MMU
 - 启用指令和数据缓存
 - 禁用访问标识
 - 禁用翻译重映射

- 小端模式
- 域访问控制机制（如果支持的话）将被配置为检查页面描述符中的访问许可位
- 必须禁用快速上下文切换扩展 (FCSE) 这将通过以下方式实现：
 - 配置 CP15 c1 系统控制寄存器 (SCTRL) 如：I=1, C=1, B=0, TRE=0, AFE=0, M=1;
 - 将 CP15 c3 域访问控制寄存器 (DACR) 配置为 0x33333333;
 - 配置 CP15 c1 系统控制寄存器 (SCTRL)，在 ARMv4 和 ARMv5 上 A=1，在 ARMv6 和 ARMv7 上 A=0, U=1；其他系统控制寄存器位的状态不是由本规范决定的。
- 启动服务的实现将启用架构上可管理的缓存和 TLB，如那些可以使用《ARM 架构参考手册》中定义的机制和程序直接使用 CP15 操作进行管理的缓存。它们不应启用需要平台信息来管理或调用非架构缓存/TLB 锁定机制的缓存。
- MMU 配置-实现时必须只使用 4k 页和一个翻译基础寄存器。在支持多个翻译基础寄存器的设备上，必须只使用 TTBR0。绑定并没有规定页表是缓存的还是不缓存的。
 - 在实现 ARMv4 到 ARMv6K 架构定义的处理器上，如果存在扩展页表支持，该内核将被额外配置为禁用。这将通过对 CP15 c1 系统控制寄存器 (SCTRL) 进行如下配置来实现。XP=0
 - 在实现 ARMv7 及以后的架构定义的处理器上，内核将被配置为启用扩展页表格式并禁用 TEX 重映射机制。这将通过对 CP15 c1 系统控制寄存器 (SCTRL) 进行如下配置来实现。XP=1, TRE=0
- 中断被启用-尽管除了 UEFI 启动服务的定时器功能外，不支持任何中断服务（所有加载的设备驱动都是通过“轮询”同步服务的）。
- 128 KiB 或更多的可用堆栈空间

一个操作系统要使用任何运行时服务，它必须：

- 保留内存图中所有标记为运行时代码和运行时数据的内存
- 调用运行时服务函数，条件如下：
 - 特权模式
 - EFI 内存映射中所有设置了 `EFI_MEMORY_RUNTIME` 位的条目所描述的系统地址区域必须与 EFI 启动环境的身份映射一致。如果操作系统加载器或操作系统使用 `SetVirtualAddressMap()` 来重新定位虚拟地址空间中的运行时服务，那么这个条件就不需要满足。参见 `SetVirtualAddressMap()` 的描述，以了解该函数被调用后内存映射的细节。
 - 处理器必须处于一种模式中，在 `EFI_MEMORY_RUNTIME` 位设置的情况下，它可以访问 EFI 内存映射中指定的系统地址区域。
 - 4 KiB 或更多可用堆栈空间
 - 中断可由调用者决定是否禁用或启用

根据本规范编写的应用程序可以改变处理器的执行模式，但调用的操作系统必须确保固件启动服务和运行时服务是在规定的执行环境下执行。

如果支持 ACPI：

- 在启动时加载的 ACPI 表可以包含在 `EfiACPIReclaimMemory`（推荐）或 `EfiACPIMemoryNVS` 类型的内存中。ACPI FACS 必须包含在 `EfiACPIMemoryNVS` 类型的内存中；
- 系统固件不得为任何 `EfiACPIReclaimMemory` 或 `EfiACPIMemoryNVS` 类型的内存描述符请求虚拟映射；
- `EfiACPIReclaimMemory` 和 `EfiACPIMemoryNVS` 类型的 EFI 内存描述符必须在 4KiB 边界上对齐，并且大小必须是 4KiB 的倍数；
- 任何通过 `EFI_MEMORY_DESCRIPTOR` 请求虚拟映射的 UEFI 内存描述符，如果设置了 `EFI_MEMORY_RUNTIME` 位，必须在 4KiB 边界上对齐，并且大小必须是 4KiB 的倍数；
- ACPI 内存操作区域必须从 UEFI 内存映射中继承可缓存属性。如果系统内存映射不包含可缓存属性，ACPI 内存操作区域必须从 ACPI 命名空间中继承其可缓存属性。如果在系统内存映射或 ACPI 命名空间中不存在可缓存属性，那么该区域必须被认为是不可缓存的。
- 在运行时加载的 ACPI 表必须包含在 `EfiACPIMemoryNVS` 类型的内存中。运行时加载的 ACPI 表的可缓存属性应在 UEFI 内存映射中定义。如果 UEFI 内存映射中不存在关于表位置的信息，可从 ACPI 内存描述符中获得缓存属性。如果 UEFI 内存映射或 ACPI 内存描述符中没有关于表位置的信息，则假定该表为非缓存的。
- 一般来说，在启动时加载的 UEFI 配置表（例如，SMBIOS 表）可以包含在 `EfiRuntimeServicesData`（推荐）、`EfiBootServicesData`、`EfiACPIReclaimMemory` 或 `EfiACPIMemoryNVS` 类型的内存中。在运行时加载的表必须包含在 `EfiRuntimeServicesData`（推荐）或 `EfiACPIMemoryNVS` 类型的内存中。

注意：以前的 EFI 规范允许在运行时加载的 ACPI 表在 `EfiReservedMemoryType` 中，对其他 EFI 配置表没有提供指导。`EfiReservedMemoryType` 不打算被固件使用。另外，只有符合 UEFI 规范的操作系统才能保证处理 `EfiBootServicesData` 类型内存中的 SMBIOS 表。

2.3.5.1 切换状态

```
1 R0 - EFI_HANDLE  
2 R1 - EFI_SYSTEM_TABLE *  
3 R14 - Return Address
```

2.3.5.2 在应用程序中启用分页或备用翻译

启动服务定义了一个特定的执行环境。本节将描述如何编写一个创建替代执行环境的应用程序。一些操作系统要求操作系统加载器能够在启动服务时启用操作系统所需的翻译，并对 UEFI 定义的执行环境进行其他修改。如果 UEFI 应用程序使用自己的页表，或其他处理器状态，应用程序必须确保固件执行每个被取代的功能。符合本规范的固件有两种方式可以在这种替代的执行环境中执行：

- 显示的固件调用
- 固件通过定时器事件抢占应用程序

具有备用执行环境的应用程序可以在每次 UEFI 调用之前恢复固件环境。然而，抢占的可能性可能要求启用替代执行环境的应用程序在替代执行环境处于活动状态时禁用中断。如果应用程序在调用 UEFI 中断 ISR 之前捕捉到中断并恢复 EFI 固件环境，那么启用备用执行环境的应用程序启用中断是合法的。在 UEFI ISR 上下文被执行后，它将返回到备用执行环境启用的应用程序上下文。

由 UEFI 应用程序创建的替代执行环境不得改变 UEFI 固件在调用 `ExitBootServices()` 之前创建的 MMU 配置的语义或行为，包括页表项的位布局。

在操作系统加载程序调用 `ExitBootServices()` 后，它应该立即配置异常向量以指向适当的代码。

2.3.5.3 调用约定

这里定义了 ARM 绑定的基本调用约定。

ARM 架构的程序调用标准 V2.06（或更高）。参见“Arm Architecture Base Calling Convention”标题下 [UEFI 相关文件链接](#)。

这种绑定方式在以下这些方面进一步约束了调用惯例：

- 对 UEFI 定义的接口的调用必须假设目标代码需要 ARM 指令集状态。除了调用 UEFI 接口时，镜像可以自由使用其他指令集状态；
- 不得使用浮点、SIMD、向量操作和其他指令集扩展；
- 只支持小端操作；
- 堆栈将按照 AAPCS 中描述的公共接口保持 8 字节对齐；
- 不得使用协处理器寄存器来传递调用参数；
- 结构（或大于 64 位的其他类型）必须通过引用而不是通过值传递；
- EFI ARM 平台绑定将寄存器 r9 定义为一个额外的被调用保存的变量寄存器；

2.3.6 AArch64 平台

AArch64 UEFI 将只执行 64 位 ARM 代码，因为 ARMv8 架构不允许在同一权限级别混合使用 32 位和 64 位代码。

所有的函数都是按照下面详细的调用约定中规定的 C 语言调用约定来调用的。在启动服务期间，只有一个处理器被用于执行。所有的二级处理器必须关闭电源或保持在静止状态。

主处理器处于以下执行模式：

- 必须启用非对齐访问；
- 使用最高的 64 位非安全权限级别；非安全 EL2（Hyp）或非安全 EL1（Kernel）；
- MMU 被启用，任何由 UEFI 内存映射定义的 RAM 都是身份映射的（虚拟地址等于物理地址）。对其他区域的映射是未定义的，可能因实现情况不同而不同。
- 该核心将被配置如下。

- 启用 MMU
- 启用指令和数据缓存
- 小端模式
- 强制执行堆栈对齐
- 不是最高字节被忽略 (NOT Top Byte Ignored)
- 有效的物理地址空间
- 4K 翻译颗粒

以下方式实现

1. 配置系统控制寄存器 SCTRLR_EL2 或 SCTRLR_EL1:

- EE=0, I=1, SA=1, C=1, A=0, M=1

2. 配置适当的翻译控制寄存器:

- TCR_EL2
 - TBI=0
 - PS 必须包含有效的物理地址空间大小。
 - TG0=00
- TCR_EL1
 - TBI0=0
 - IPS 必须包含有效的中间物理地址空间大小。
 - TG0=00

注意：其他系统控制寄存器位的状态不是由本规范决定的。

- 所有的浮点陷阱和异常将在相关的异常级别被禁用(FPCR=0,CPACR_EL1.FPEN=11,CPTR_EL2.TFP=0)。这意味着 FP 单元在默认情况下将被启用。
- 启动服务的实现将启用架构上可管理的缓存和 TLB，即那些可以使用 ARM 架构参考手册中定义的机制和程序直接使用与实现无关的寄存器进行管理的缓存。它们不应启用需要平台信息来管理或调用非架构缓存/TLB 锁定机制的缓存。MMU 配置。实现必须只使用 4k 页和一个翻译基础寄存器。在支持多个翻译基础寄存器的设备上，必须只使用 TTBR0。绑定并没有规定页表是缓存的还是不缓存的。
- 中断被启用，尽管除了 UEFI 启动服务的定时器功能外，不支持任何中断服务（所有加载的设备驱动都通过“轮询”同步服务）。所有的 UEFI 中断必须只被路由到 IRQ 向量。
- 架构通用定时器必须被初始化和启用。计数器频率寄存器 (CNTFRQ) 必须用定时器频率进行编程。必须通过设置寄存器 CNTHCTL_EL2 中的位 EL1PCTEN 和 EL1PCEN 来为非安全的 EL1 和 E0 提供定时器访问。
- 系统固件不应初始化没有架构复位值的 EL2 寄存器，除非固件本身在 EL2 运行并需要这样做。
- 128 KiB 或更多的可用堆栈空间

- ARM 架构允许以各种粒度映射页面，包括 4KiB 和 64KiB。如果一个 64KiB 的物理页包任何具有以下类型的 4KiB 页，那么 64KiB 页中的所有 4KiB 页必须使用相同的 ARM 内存页属性（如表 2-5 中所述）。
 - EfiRuntimeServicesCode
 - EfiRuntimeServicesData
 - EfiReserved
 - EfiACPIMemoryNVS 不允许在一个大的页面内进行混合属性的映射。

注意：这个约束允许一个基于 64K 分页的操作系统安全地映射运行时服务内存。

对于一个操作系统来说，要使用任何运行时服务，运行时服务必须：

- 支持来自 EL1 或 EL2 例外级别的调用；
- 一旦调用，不允许从 EL1 和 EL2 同时或嵌套调用；

注意：允许从 EL1 和 EL2 进行连续的、不重叠的调用。

运行时服务被允许向更高的异常级别进行同步的 SMC 和 HVC 调用。

注意：这些规则允许启动服务在 EL2 启动，而运行时服务被分配到 EL1 操作系统。在这种情况下，对 SetVirtualAddressMap() 的调用预计将提供一套适合 EL1 的映射。

一个操作系统要使用任何运行时服务，它必须：

- 启用非对齐访问支持；
- 保留内存图中所有标记为运行时代码和运行时数据的内存；
- 调用运行时服务函数，条件如下：
 - 来自 EL1 或 EL2 例外级别；
 - 从同一个异常级别一致地调用运行时服务。在不同的异常级别之间共享运行时服务是不允许的；
 - 运行时服务必须只分配给一个操作系统或管理程序。它们不能在多个客户操作系统之间共享；
 - EFI 内存映射中所有设置了 EFI_MEMORY_RUNTIME 位的条目所描述的系统地址区域必须与 EFI 启动环境的身份映射一致。如果操作系统加载器或操作系统使用 SetVirtualAddressMap() 来重新定位虚拟地址空间中的运行时服务，那么这个条件就不需要满足。参见 SetVirtualAddressMap() 的描述，以了解该函数被调用后内存映射的细节；
 - 处理器必须处于一种模式中，它可以访问 EFI 内存映射中指定的系统地址区域，并设置 EFI_MEMORY_RUNTIME 位；
 - 8 KiB，或更多的可用堆栈空间；
 - 堆栈必须是 16 字节对齐的（128 位）；
 - 中断可由调用者决定是否禁用或启用；

根据本规范编写的应用程序可以改变处理器的执行模式，但调用的操作系统必须确保固件启动服务和运行时

服务是在规定的执行环境下执行。

如果支持 ACPI:

- 在启动时加载的 ACPI 表可以包含在以下类型的内存中 [EfiACPIReclaimMemory](#)(推荐)或 [EfiACPIMemoryNVS](#)。
 -
- ACPI FACS 必须包含在 [EfiACPIMemoryNVS](#) 类型的内存中。系统固件不得为任何 [EfiACPIReclaimMemory](#) 或 [EfiACPIMemoryNVS](#) 类型的内存描述符请求虚拟映射。
- [EfiACPIReclaimMemory](#) 和 [EfiACPIMemoryNVS](#) 类型的 EFI 内存描述符必须在 4KiB 边上对齐，并且必须是 4KiB 大小的倍数。
- 任何通过 [EFI_MEMORY_DESCRIPTOR](#) 请求虚拟映射的 UEFI 内存描述符,如果设置了 [EFI_MEMORY_RUNTIME](#) 位，必须在 4KiB 边界上对齐，并且大小必须是 4KiB 的倍数。
- ACPI 内存操作区域必须从 UEFI 内存映射中继承可缓存属性。如果系统内存映射不包含可缓存属性，ACPI 内存操作区域必须从 ACPI 命名空间中继承其可缓存属性。如果在系统内存映射或 ACPI 命名空间中不存在可缓存属性，那么该区域必须被认为是不可缓存的。
- 在运行时加载的 ACPI 表必须包含在 [EfiACPIMemoryNVS](#) 类型的内存中。运行时加载的 ACPI 表的可缓存属性应在 UEFI 内存映射中定义。如果 UEFI 内存映射中不存在关于表位置的信息，可从 ACPI 内存描述符中获得缓存属性。如果 UEFI 内存映射或 ACPI 内存描述符中没有关于表位置的信息，则假定该表为非缓存的。
- 一般来说，在启动时加载的 UEFI 配置表（例如，SMBIOS 表）可以包含在 [EfiRuntimeServicesData](#) (推荐)、[EfiBootServicesdata](#)、[EfiACPIReclaimMemory](#) 或 [EfiACPIMemoryNVS](#) 类型的内存中。在运行时加载的表必须包含在 [EfiRuntimeServicesData](#) (推荐) 或 [EfiACPIMemoryNVS](#) 类型的内存中。

注意：以前的 EFI 规范允许在运行时加载的 ACPI 表在 [EfiReservedMemoryType](#) 中，对其 EFI 配置表没有提供指导。[EfiReservedMemoryType](#) 不打算被固件使用。UEFI 2.0 阐明了向前发展的情况。另外，只有符合 UEFI 规范的操作系统才能保证处理内存中 [fiBootServiceData](#) 类型的 SMBIOS 表。

2.3.6.1 存储器类型

Table 2-5 Map: EFI Cacheability Attributes to AArch64 Memory Types

EFI Memory Type	ARM Memory Type: MAIR attribute encoding Attr<n> [7:4] [3:0]	ARM Memory Type: Meaning
EFI_MEMORY_UC (Not cacheable)	0000 0000	Device-nGnRnE (Device non-Gathering, non-Reordering, no Early Write Acknowledgement)
EFI_MEMORY_WC (Write combine)	0100 0100	Normal Memory Outer non-cacheable Inner non-cacheable
EFI_MEMORY_WT (Write through)	1011 1011	Normal Memory Outer Write-through non-transient Inner Write-through non-transient
EFI_MEMORY_WB (Write back)	1111 1111	Normal Memory Outer Write-back non-transient Inner Write-back non-transient
EFI_MEMORY_UCE		Not used or defined

Table 2-6 Map: UEFI Permission Attributes to ARM Paging Attributes

EFI Memory Type	ARM Paging Attributes
EFI_MEMORY_XP	EL2 translation regime: XN Execute never EL1/0 translation regime: UXN Unprivileged execute never PWN Privileged execute never
EFI_MEMORY_RO	Read only access AP[2]=1
EFI_MEMORY_RP	Not used or defined
EFI_MEMORY_WP	

2.3.6.2 切换状态

- 1 X0 – EFI_HANDLE
- 2 X1 – EFI_SYSTEM_TABLE *
- 3 X30 – Return Address

2.3.6.3 在应用程序中启用分页或备用翻译

启动服务定义了一个特定的执行环境。本节将描述如何编写一个应用程序来创建一个替代的执行环境。一些操作系统要求操作系统加载器能够在启动服务时启用操作系统所需的翻译，并对 UEFI 定义的执行环境进行其他更改。

如果 UEFI 应用程序使用自己的页表，或其他处理器状态，应用程序必须确保固件执行每个被取代的功能。符合本规范的固件有两种方式可以在这种替代的执行环境中执行。

- 明确的固件调用
- 固件通过定时器事件抢占应用程序

具有备用执行环境的应用程序可以在每次 UEFI 调用之前恢复固件环境。然而，抢占的可能性可能要求启用替代执行环境的应用程序在替代执行环境处于活动状态时禁用中断。如果应用程序在调用 UEFI 中断 ISR 之前捕捉到中断并恢复 EFI 固件环境，那么启用备用执行环境的应用程序启用中断是合法的。在 UEFI ISR 上下文被执行后，它将返回到备用执行环境启用的应用程序上下文。

由 UEFI 应用程序创建的替代执行环境不得改变 UEFI 固件在调用 `ExitBootServices()` 之前创建的 MMU 配置的语义或行为，包括页表项的位布局（bit layout）。

在操作系统加载器调用 `ExitBootServices()` 后，它应该立即配置异常向量以指向适当的代码。

2.3.6.4 调用约定

AArch64 绑定的基本调用惯例在文件 *Procedure Call Standard for the ARM 64-bit Architecture Version A-0.06*（或更高版本）中定义。

请参阅“ARM 64 位基础调用公约”标题下的 UEFI [相关文件链接](#)。

此绑定通过以下方式进一步限制调用约定：

- AArch64 的执行状态不能被被调用者修改。
- 所有的代码退出，无论是正常的还是特殊的，都必须来自 A64 指令集。
- 可以使用浮点和 SIMD 指令。
- 可选的向量操作和其他指令集扩展只能被使用。
 - 在动态地检查它们的存在之后。
 - 保存并随后恢复任何额外的执行状态背景。
 - 额外的功能启用或控制，如电源，必须明确地管理。
- 只支持小端操作。
- 堆栈将保持 16 字节对齐。
- 结构（或大于 64 位的其他类型）必须通过引用而不是通过值传递。
- EFI AArch64 平台绑定将平台寄存器（r18）定义为“不使用”。避免在固件中使用 r18 使得代码与操作系统平台 ABI 定义的 r18 的固定作用以及操作系统及其应用程序将 r18 作为临时寄存器的使用兼容。

2.3.7 RISC-V 平台

所有的函数都是以 C 语言的调用惯例来调用的。更多细节见 2.3.7.3。

在 RISC-V 平台上，目前在 RISC-V 架构中引入了三个特权级别。除了用户权限，监管者权限和机器权限涵盖了 RISC-V 系统的所有方面。特权指令也被定义在每个权限级别中。

级别	编码	名称	缩写
0	0	用户/应用模式	U
1	1	监管者模式	S
2	10	保留	
3	11	机器模式	M

一个 RISC-V 平台可以包含一个或多个 RISC-V 内核和其他组件，如物理内存、固定功能加速器和 I/O 设备。术语 RISC-V 核心是指一个包含独立指令获取单元的组件。一个 RISC-V 核心可以有多个 RISC-V 兼容的硬件线程，或称 HART。在整个 POST 过程中，RISC-V UEFI 固件可以在机器模式或监管者模式下执行，这取决于 HART 的能力和平台设计。然而，如果平台被设计为启动监管者模式的操作系统或操作系统加载器，RISC-V UEFI 固件必须在 POST 初期或后期将启动程序切换到监管者模式。

机器模式具有最高的权限，该模式是 RISC-V 平台唯一必须的权限级别；所有其他权限级别是可选的，取决于平台的要求。机器模式是在上电复位时进入的初始特权模式。此级别在 UEFI 中用于对硬件平台的低级别访问。

UEFI 固件实现可以提供监督者二进制接口 (SBI)，以允许监督者模式执行环境调用特权功能或访问特权硬件。在启动服务期间，处理器处于以下执行模式：

- 共有 32 个通用寄存器 x1-x31。寄存器 x0 被硬连线为 0。每个寄存器都有其 ABI（应用二进制接口）名称。更多细节见 2.3.7.3
- 本机基数整数的宽度取决于 RISC-V 特权模式的实现
- XLEN 是一个通用术语，用来指基数整数的宽度，单位为比特
 - 对于 32 位宽度的基础整数 ISA，XLEN = 32
 - 对于 64 位宽度的基础整数 ISA，XLEN = 64
 - 对于 128 位宽度的基础整数 ISA，XLEN = 128
- 处理器寄存器的宽度可以通过将立即数 4 放在寄存器中然后一次将寄存器左移 31 位来确定。如果一次移位后为零，那么机器就是 RV32。如果两次移位后为零，那么机器就是 RV64，否则就是 RV128
- 处理器复位向量是平台指定的。在 UEFI 中，它被配置为平台实现定义的复位向量。复位向量地址是 RISC-V 处理器在开机复位时获取的第一条指令

- 复位后的 mcause 值有特定的实现解释，在不区分不同复位条件的实现中，应该返回 0 值。区分不同复位条件的实现应该只使用 0 来表示最完整的复位（例如，硬复位）。复位的原因可能是开机复位、外部硬复位、检测到断电、看门狗定时器过期、睡眠模式唤醒等，机器模式的 UEFI 系统固件必须区分这些原因
- mstatus.xIE 表示当前特权模式下的处理器中断激活情况
 - mstatus.MIE 被设置为 1，而 mstatus.SIE 和 mstatus.UIE 在 UEFI POST 早期阶段被设置为 0
- 在 UEFI 的启动服务中，机器模式中断被启用。有两种中断被启用，一种是定时器中断，另一种是软件中断
- mie.MSIE = 1
- mie.MTIE = 1
- 该内存处于物理寻址模式。在 UEFI 启动服务期间，页面在 RISC-V 机器模式下被禁用
- I/O 访问是通过内存映射 I/O
- 在 UEFI 中只支持机器级的控制和状态寄存器（CSR）
- 机器 ISA(misa) 寄存器包含有关 CPU 实现能力的信息。misa.MXL 字段编码了机器模式下的本地基数整数 ISA 宽度。MXLEN（机器 XLEN）是由 misa.MXL 的设置决定的
 - misa.MXL = 1, MXLEN 为 32 位
 - misa.MXL = 2, MXLEN 为 64 位
 - misa.MXL = 3, MXLEN 为 128 位
- RISC-V 处理器支持广泛的定制和专业化指令集。RISC-V 的变化提供了各种目的的处理器实现，处理器的能力被报告在 misa 寄存器的扩展位中。在执行指定的 RISC-V 扩展指令之前，UEFI 驱动程序需要知道处理器的能力。扩展字段编码标准扩展的存在，每个字母有一个位。Bit 0 编码为扩展 “A” 存在，Bit 1 编码扩展 “B” 存在，以此类推。目前，单字母扩展记忆法如下：
 - A - 原子扩展
 - B - 暂时保留用于位操作扩展
 - C - 压缩扩展
 - D - 双精度浮点扩展
 - E - 减少的寄存器集指示器 RV32E (16 个寄存器)。
 - F - 单精度浮点扩展
 - G - 存在额外的标准扩展
 - H - 管理程序扩展
 - I - RV32I/64I/128I 基础 ISA
 - J - 暂时保留给动态翻译语言扩展
 - K - 保留
 - L - 暂时为十进制浮点扩展保留
 - M - 整数乘法和除法扩展
 - N - 支持用户级中断

- O - 保留
- P - 暂时为 Packed-SIMD 扩展保留
- Q - 四精度浮点扩展
- S - 实现监督员模式
- T - 暂时为事务性内存扩展保留
- U - 实现用户模式
- V - 暂时保留给向量扩展
- W - 保留
- X - 存在非标准扩展
- Y - 保留
- Z - 保留
- Zifenci - 指令 - 取物栅栏
- Zicsr - 控制和状态寄存器访问

- 机器供应商 ID 注册

- mvendorid 是一个 32 位的只读寄存器，编码零件的制造。值为 0 表示这个字段没有实现，或这是一个非商业的实现。

- 机器结构 ID 寄存器

- marchid 是一个 MXLEN 位只读寄存器，编码 hart 的基本微架构。mvendorid 和 marchid 的组合应能唯一地识别所实现的 hart 微架构的类型。

- 机器实现 ID 寄存器

- 这为处理器实现的版本提供了一个独特的编码。

根据本规范编写的应用程序可以改变处理器的执行模式，但 UEFI 镜像必须确保固件启动服务和运行时服务以规定的执行环境执行。

在操作系统调用 `ExitBootServices()` 后，固件启动服务不再可用，调用任何启动服务都是非法的。在 `ExitBootServices` 之后，固件运行服务仍然可用，如果 `SetVirtualAddressMap()` 已经被调用，描述了固件运行服务使用的所有虚拟地址范围，则可以在启用分页和虚拟地址指针的情况下调用。

如果支持 ACPI：

- 在启动时加载的 ACPI 表可以包含在 `EfiACPIReclaimMemory`（推荐）或 `EfiACPIMemoryNVS` 类型的内存中。`ACPI FACS` 必须包含在 `EfiACPIMemoryNVS` 类型的内存中
- 系统固件不得为任何类型的内存描述符请求一个虚拟映射。`EfiACPIReclaimMemory` 或 `EfiACPIMemoryNVS`。
- `EfiACPIReclaimMemory` 和 `EfiACPIMemoryNVS` 类型的 EFI 内存描述符必须在 4KiB 边界上对齐，并且必须是 4KiB 大小的倍数。

- 任何 UEFI 内存描述符通过以下方式请求一个虚拟映射设置了 `EFI_MEMORY_RUNTIME` 位的 `EFI_MEMORY_DESCRIPTOR` 必须在 4KiB 边界上对齐，并且必须是 4KiB 大小的倍数。
- ACPI 内存操作区域必须从 UEFI 内存映射中继承可缓存属性。如果系统内存映射不包含缓存属性，ACPI 内存操作区域必须从 ACPI 命名空间中继承其缓存属性。如果在 UEFI 内存映射中不存在可缓存属性，那么系统内存映射或 ACPI 命名空间，那么该区域必须被认为是不可缓存的。
- 在运行时加载的 ACPI 表必须包含在 `EfiACPIMemoryNVS` 类型的内存中。
 - 运行时加载的 ACPI 表的可缓存属性应该在 UEFI 内存映射中定义。如果 UEFI 内存映射中不存在关于表位置的信息，可以从 ACPI 内存描述符中获得可缓存属性。如果 UEFI 内存映射或 ACPI 内存描述符中没有关于表位置的信息，则假定该表为非缓存的。
- 一般来说，在启动时加载的 UEFI 配置表（例如，SMBIOS 表）可以包含在 `EfiRuntimeServicesData`（推荐）、`EfiBootServicesData`、`EfiACPIReclaimMemory` 或 `EfiACPIMemoryNVS` 类型的内存中。在运行时加载的表必须包含在 `EfiRuntimeServicesData`（推荐）或 `EfiACPIMemoryNVS` 类型的内存中。

注意：以前的 EFI 规范允许在运行时加载的 ACPI 表在 `EfiReservedMemoryType` 中，对于其他 EFI 配置表没有提供指导。`EfiReservedMemoryType` 不打算被固件使用。UEFI 规范打算澄清未来的情况。另外，只有符合 UEFI 规范的操作系统才能保证处理内存中 `EfiBootServicesData` 类型的 SMBIOS 表。

2.3.7.1 交接状态

当 UEFI 固件将控制权交给监管者模式 OS 时，RISC-V boot hart 必须在监管者模式下运行，内存寻址必须在 Bare 模式下运行，即没有内存地址转换或通过虚拟页表条目进行保护。

为了描述 POST 后下一个引导阶段的异构 RISC-V 内核和 HART，如果目标可引导镜像需要这些信息，UEFI 固件必须在固件数据结构中构建内核和 hart 硬件能力信息。（例如，如果平台支持 SMBIOS 结构，SMBIOS 记录类型 44 记录，请参阅“RISC-V 处理器 SMBIOS 规范”标题下的“链接到 UEFI 规范相关文档”）

UEFI 固件必须向操作系统公开固件数据结构中的 RISC-V Boot hart ID：

- 如果平台支持 SMBIOS，那么在“RISC-V 处理器特定数据”结构中，SMBIOS 类型 44 的记录必须将“Boot Hart”设置为 1。
- 如果平台支持设备树，设备树必须在/`chosen` 节点下包含一个无符号整数（32 位）属性“boot-hartid”，它将向主管操作系统指出启动的 hart ID。

如果平台支持设备树结构来描述系统配置，必须在 EFI 配置表中安装扁平化设备 Blob (DTB)（详见 4.6 节）。所有的 UEFI 镜像都需要两个参数：UEFI 镜像句柄和 EFI 系统表的指针。根据 RISC-V 的调用惯例，`EFI_HANDLE` 通过 a0 寄存器传递，`EFI_SYSTEM_TABLE` 则通过 a1 寄存器传递。

- x10 -`EFI_HANDLE`(ABI 名称：a0)
- x11 -`EFI_SYSTEM_TABLE *`(ABI 名称：a1)
- x1 - 返回地址(ABI 名称：ra)

2.3.7.2 数据对齐

在 RV32I 和 RV64I 中，数据类型存储在内存中时必须按其自然大小对齐。下表描述了 UEFI 中 RV32I 和 RV64I 的数据类型及其对齐情况。

Table 2-7 RV32 datatype alignment

Datatype	Description	Alignment
BOOLEAN	Logical Boolean	1
INTN	Signed value in native width.	4
UINTN	Unsigned value in native width.	4
INT8	1-byte signed value	1
UINT8	1-byte unsigned value	1
INT16	2-byte signed value	2
UINT16	2-byte unsigned value	2
INT32	4-byte signed value	4
UINT32	4-byte unsigned value	4
INT64	8-byte signed value	8
UINT64	8-byte unsigned value	8
CHAR8	1-byte character	1
CHAR16	2-byte character	2
VOID	Undeclared type	4

Table 2-8 RV64 datatype alignment

Datatype	Description	Alignment
BOOLEAN	Logical Boolean	1
INTN	Signed value in native width.	8
UINTN	Unsigned value in native width.	8
INT8	1-byte signed value	1
UINT8	1-byte unsigned value	1
INT16	2-byte signed value	2
UINT16	2-byte unsigned value	2
INT32	4-byte signed value	4
UINT32	4-byte unsigned value	4
INT64	8-byte signed value	8
UINT64	8-byte unsigned value	8
CHAR8	1-byte character	1
CHAR16	2-byte character	2
VOID	Undeclared type	8

2.3.7.3 调用约定

RISC-V 的调用惯例在必要时用寄存器传递参数。在 RISC-V 中，总共声明了 32 个通用寄存器，每个寄存器

都有其相应的 ABI 名称。

Table 2-9 Register name and ABI name.

Register	ABI Name	Description
x0	zero	Hardwired to zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-7	t0-2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10-11	a0-1	Function arguments/Return values
x12-17	a2-7	Function arguments
x18-27	s2-11	Saved registers
x28-31	t3-6	Temporaries

在 RISC-V 的调用惯例中，最多有 8 个整数寄存器用于传递参数，a0-a7。a0-a7 是 ABI 的名称，对应的寄存器是 x10-x17。数值从整数寄存器 a0 和 a1 中返回，这些是寄存器 x10 和 x11。在标准的 RISC-V 调用惯例中，堆栈向下增长，堆栈点总是保持 16 字节对齐。五个整数寄存器 t0-t6 是临时寄存器，在整个调用过程中是不稳定的，如果以后使用，必须由调用者保存。12 个整数寄存器 s0-s11 在不同的调用中保留，如果使用的话必须由被调用者保存。

鉴于以下声明：

“在标准 ABI 中，程序不应该修改整数寄存器 tp 和 gp，因为信号处理程序可能依赖于它们的值”在 RISC-V EFL psABI 规范中提到，RISC-V 的调用惯例是 gp 和 tp 寄存器没有被分配一个特定的所有者来保存和恢复它们的值（见下面的链接），UEFI 固件在任何情况下都不能相信 tp 和 gp 的值，也不能假设拥有对这些寄存器的写入权限。（如在 EFI 启动服务、EFI 运行时服务、EFI 管理模式服务和任何可能被 EFI 驱动程序、操作系统或外部固件有效载荷调用的 UEFI 固件接口）。

如果 UEFI 固件需要改变 gp 或 tp 寄存器的值，请保留这些值，并且在 `ExitBootServices()` 之后不要再碰它们。是否以及如何在 UEFI 固件环境中保存 gp 和 tp 是具体实现的。

参见“RISC-V EFL psABI 规范”标题下的 UEFI 规范 [相关文件链接](#)，以及 RISC-V Unprivileged ISA 规范中的 RISC-V 汇编程序员手册部分。

2.4 协议

设备句柄支持的协议是通过 `EFI_BOOT_SERVICES.HandleProtocol()` 启动服务或 `EFI_BOOT_SERVICES.OpenProtocol()` 启动服务发现的。每个协议都有一个规范，包括以下内容：

- 该协议的全局唯一 ID (GUID)。

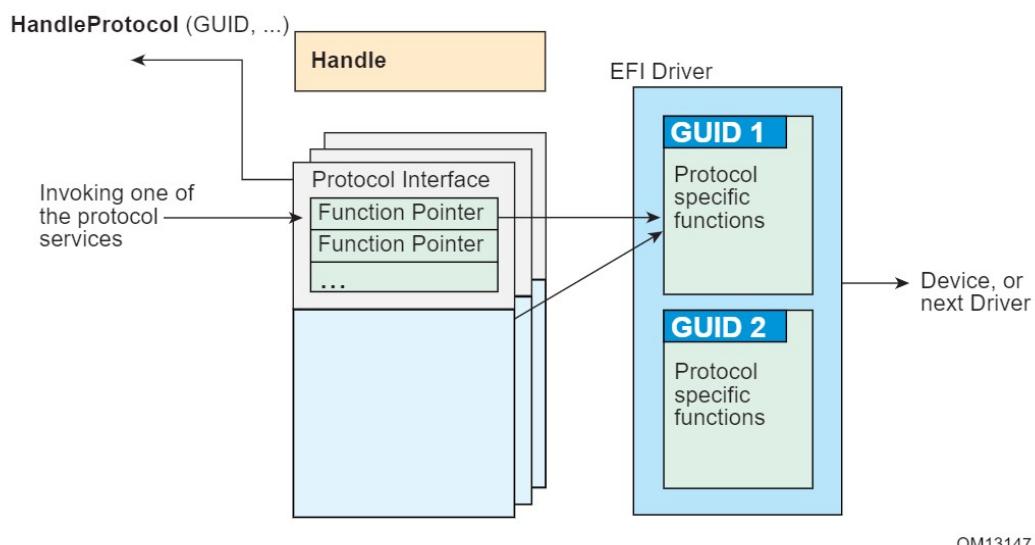
- 协议接口结构
- 协议服务

除非另有规定，否则协议的接口结构不会从运行时在内存中分配，协议成员函数也不应在运行时被调用。如果没有明确规定，一个协议成员函数可以在小于或等于 `TPL_NOTIFY` 的 TPL 级别下被调用（见第 7.1 节）。除非另有规定，否则一个协议的成员函数不具有可重入性或 MP 安全。（TODO）

协议成员函数定义定义的任何状态码都需要实现，可能会返回附加错误码，但不会通过标准符合性测试，任何使用该程序的软件不能依赖任何扩展错误码一个实现可以提供的。

为了确定句柄是否支持任何给定的协议，协议的 GUID 被传递给 `HandleProtocol()` 或 `OpenProtocol()`。如果设备支持所请求的协议，将返回一个指向定义的协议接口结构的指针。协议接口结构将调用者与该设备要使用的特定协议服务联系起来。

图 2-4 显示了一个协议的构造。UEFI 驱动程序包含一个或多个协议实现的特定功能，并将它们注册到启动服务 `EFI_BOOT_SERVICES.InstallProtocolInterface()`。固件返回协议的接口，然后用于调用协议的特定服务。UEFI 驱动程序保持私有的、特定于设备的协议接口的上下文。



OM13147

下面的 C 语言代码片段说明了协议的使用：

```

1 // There is a global "EffectsDevice" structure. This
2 // structure contains information pertinent to the device.
3 // Connect to the ILLUSTRATION_PROTOCOL on the EffectsDevice,
4 // by calling HandleProtocol with the device's EFI device handle
5 // and the ILLUSTRATION_PROTOCOL GUID.
6

```

```

7 EffectsDevice.Handle = DeviceHandle;
8 Status = HandleProtocol (
9     EffectsDevice.EFIHandle,
10    &IllustrationProtocolGuid,
11    &EffectsDevice.IllustrationProtocol
12 );
13 // Use the EffectsDevice illustration protocol's "MakeEffects"
14 // service to make flashy and noisy effects.
15
16 Status = EffectsDevice.IllustrationProtocol->MakeEffects (
17     EffectsDevice.IllustrationProtocol,
18     TheFlashyAndNoisyEffect
19 );

```

表 2-10 列出本规范所定义的 UEFI 协议。

协议	描述
EFI_LOADED_IMAGE_PROTOCOL	提供关于镜像的信息。
EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL	指定通过 EFI Boot Service LoadImage() 加载 PE/COFF 镜像时使用的设备路径。
EFI_DEVICE_PATH_PROTOCOL	提供设备的位置。
EFI_DRIVER_BINDING_PROTOCOL	提供服务以确定 UEFI 驱动程序是否支持给定的控制器，以及提供服务以启动和停止给定的控制器。
EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL	提供驱动家族 Override 机制，为给定的控制器选择最佳驱动。
EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL	提供一个平台特定的 Override 机制，用于为给定的控制器选择最佳驱动程序。
EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL	提供一个针对总线的 Override 机制，以便为给定的控制器选择最佳驱动程序。
EFI_DRIVER_DIAGNOSTICS2_PROTOCOL	为 UEFI 驱动所管理的控制器提供诊断服务。
EFI_COMPONENT_NAME2_PROTOCOL	为 UEFI 驱动和驱动所管理的控制器提供人类可读的名称。
EFI_SIMPLE_TEXT_INPUT_PROTOCOL	支持简单控制台风格文本输入的设备的协议接口。
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL	用于支持控制台风格文本显示的设备的协议接口。
EFI_SIMPLE_POINTER_PROTOCOL	用于鼠标和轨迹球等设备的协议接口。
EFI_SERIAL_IO_PROTOCOL	用于支持串行字符传输的设备的协议接口。

协议	描述
EFI_LOAD_FILE_PROTOCOL	用于从任意设备读取文件的协议接口。
EFI_LOAD_FILE2_PROTOCOL	用于从任意设备读取非启动选项文件协议接口。
EFI_SIMPLE_FILE_SYSTEM_PROTOCOL	用于打开包含 UEFI 文件系统的磁盘卷。
EFI_FILE_PROTOCOL	提供对支持的文件系统的访问。
EFI_DISK_IO_PROTOCOL	可分层到任何 BLOCK_IO 或 BLOCK_IO_EX 接口。
EFI_BLOCK_IO_PROTOCOL	用于支持块 I/O 风格访问的设备的协议接口。
EFI_BLOCK_IO2_PROTOCOL	用于支持块状 I/O 风格访问的设备的协议接口。该接口能够进行非阻塞式交易。
EFI_UNICODE_COLLATION_PROTOCOL	用于字符串比较操作的协议接口。
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL	用于将内存、I/O、PCI 配置和 DMA 访问抽象到 PCI 根桥控制器。
EFI_PCI_IO_PROTOCOL	用于抽象内存、I/O、PCI 配置和 DMA 访问 PCI 总线上的一个 PCI 控制器。
EFI_USB_IO_PROTOCOL	用于抽象访问一个 USB 控制器。
EFI_SIMPLE_NETWORK_PROTOCOL	为支持基于数据包传输的设备提供接口。
EFI_PXE_BASE_CODE_PROTOCOL	为支持网络启动的设备提供协议接口。
EFI_BIS_PROTOCOL	在加载和调用启动镜像之前验证它们的协议接口。
EFI_DEBUG_SUPPORT_PROTOCOL	协议接口，用于保存和恢复处理器上下文，并钩住处理器异常。
EFI_DEBUGPORT_PROTOCOL	协议接口，抽象出调试主机和调试目标系统之间的字节流连接。
EFI_DECOMPRESS_PROTOCOL	用于解压使用 EFI 压缩算法压缩的镜像。
EFI_EBC_PROTOCOL	支持 EFI Byte Code 解释器所需的协议接口。
EFI_GRAPHICS_OUTPUT_PROTOCOL	用于支持图形输出的设备的协议接口。
EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL	允许向 NVM Express 控制器发布 NVM Express 命令的协议接口。
EFI_EXT_SCSI_PASS_THRU_PROTOCOL	用于 SCSI 通道的协议接口，允许向 SCSI 设备发送 SCSI 请求包。
EFI_USB2_HC_PROTOCOL	用于抽象访问 USB 主机控制器的协议接口。

协议	描述
EFI_AUTHENTICATION_INFO_PROTOCOL	提供与特定设备路径相关的通用认证信息的访问。
EFI_DEVICE_PATH_UTILITIES_PROTOCOL	帮助创建和操作设备路径。
EFI_DEVICE_PATH_TO_TEXT_PROTOCOL	将设备节点和路径转换为文本。
EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL	将文本转换为设备路径和设备节点。
EFI_EDID_DISCOVERED_PROTOCOL	包含从视频输出设备检索的 EDID 信息。
EFI_EDID_ACTIVE_PROTOCOL	包含一个活动视频输出设备的 EDID 信息。
EFI_EDID_OVERRIDE_PROTOCOL	由平台产生，允许平台向图形输出协议的生产者提供 EDID 信息。
EFI_ISCSI_INITIATOR_NAME_PROTOCOL	设置和获取 iSCSI 启动器名称。
EFI_TAPE_IO_PROTOCOL	提供控制和访问磁带机的服务。
EFI_MANAGED_NETWORK_PROTOCOL	用于定位 MNP 驱动程序支持的通信设备，并创建和销毁可以使用底层通信设备的 MNP 子协议驱动程序的实例。
EFI_ARP_SERVICE_BINDING_PROTOCOL	用于定位 ARP 驱动程序支持的通信设备，并创建和销毁 ARP 子协议驱动程序的实例。
EFI_ARP_PROTOCOL	用于将本地网络协议地址解析为网络硬件地址。
EFI_DHCP4_SERVICE_BINDING_PROTOCOL	用于定位 EFI DHCPv4 协议驱动程序支持的通信设备，并创建和销毁能够使用底层通信设备的 EFI DHCPv4 协议子驱动程序实例。
EFI_DHCP4_PROTOCOL	用于收集 EFI IPv4 协议驱动程序的配置信息，并提供 DHCPv4 服务器和 PXE 启动服务器发现服务。
EFI_TCP4_SERVICE_BINDING_PROTOCOL	用于定位 EFI TCPv4Protocol 驱动，以创建和销毁驱动的子代，与其他使用 TCP 协议的主机通信。
EFI_TCP4_PROTOCOL	提供发送和接收数据流的服务。
EFI_IP4_SERVICE_BINDING_PROTOCOL	用于定位 EFI IPv4 协议驱动程序支持的通信设备，并创建和销毁可以使用底层通信设备的 EFI IPv4 协议子协议驱动程序的实例。
EFI_IP4_PROTOCOL	提供基本的网络 IPv4 数据包 I/O 服务。
EFI_IP4_CONFIG_PROTOCOL	EFI IPv4 配置协议驱动程序执行与平台和策略相关的 EFI IPv4 协议驱动程序的配置。

协议	描述
EFI_IP4_CONFIG2_PROTOCOL	EFI IPv4 配置 II 协议驱动程序执行与平台和策略相关的 EFI IPv4 协议驱动程序的配置。
EFI_UDP4_SERVICE_BINDING_PROTOCOL	用于定位 EFI UDPv4 协议驱动程序支持的通信设备，并创建和销毁可以使用底层通信设备的 EFI UDPv4 协议子协议驱动程序的实例。
EFI_UDP4_PROTOCOL	提供简单的面向数据包的服务，以传输和接收 UDP 数据包。
EFI_MTFTP4_SERVICE_BINDING_PROTOCOL	用于定位 EFI MTFTPv4 协议驱动程序所支持的通信设备，并创建和销毁可以使用底层通信设备的 EFI MTFTPv4 协议子协议驱动程序的实例。
EFI_MTFTP4_PROTOCOL	为客户端的单播或多播 TFTP 操作提供基本服务。
EFI_HASH_PROTOCOL	允许使用一种或多种哈希算法创建任意消息摘要的哈希值。
EFI_HASH_SERVICE_BINDING_PROTOCOL	用于定位驱动程序提供的哈希服务支持，并创建和销毁 EFI 哈希协议的实例，以便多个驱动程序能够使用底层哈希服务。
EFI_SD_MMCC_PASS_THRU_PROTOCOL	允许 SD/eMMC 命令被发送到 SD/eMMC 控制器的协议接口。

2.5 UEFI 驱动模型

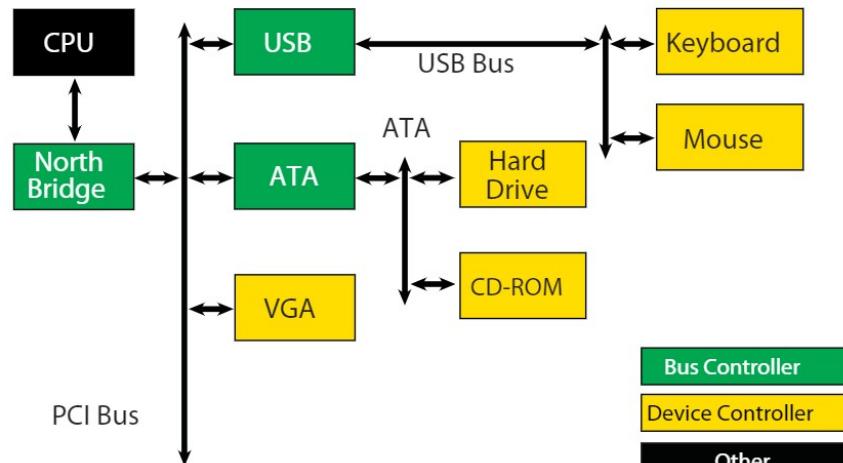
UEFI 驱动模型旨在简化设备驱动的设计和实现，并产生较小的可执行镜像大小。因此，一些复杂性已被转移到总线驱动器中，并在更大程度上转移到通用固件服务中。

设备驱动程序需要在加载驱动程序的同一镜像句柄上产生一个驱动程序绑定协议。然后，它等待系统固件将驱动程序连接到一个控制器。当这种情况发生时，设备驱动程序负责在控制器的设备句柄上产生一个协议，抽象出控制器所支持的 I/O 操作。总线驱动器执行这些完全相同的任务。此外，总线驱动程序还负责发现总线上的任何子控制器，并为发现的每个子控制器创建一个设备句柄。

一个假设是，系统的结构可以被看作是一组连接到一个或多个核心芯片组的一个或多个处理器。核心芯片组负责产生一个或多个 I/O 总线。UEFI 驱动模型并不试图描述处理器或核心芯片组。相反，UEFI 驱动模型描述了由核心芯片组产生的一组 I/O 总线，以及这些 I/O 总线的任何子总线。这些总线可以是设备，也可以是额外的 I/O 总线。这可以被看作是一棵总线和设备的树，核心芯片组在这棵树的根部。

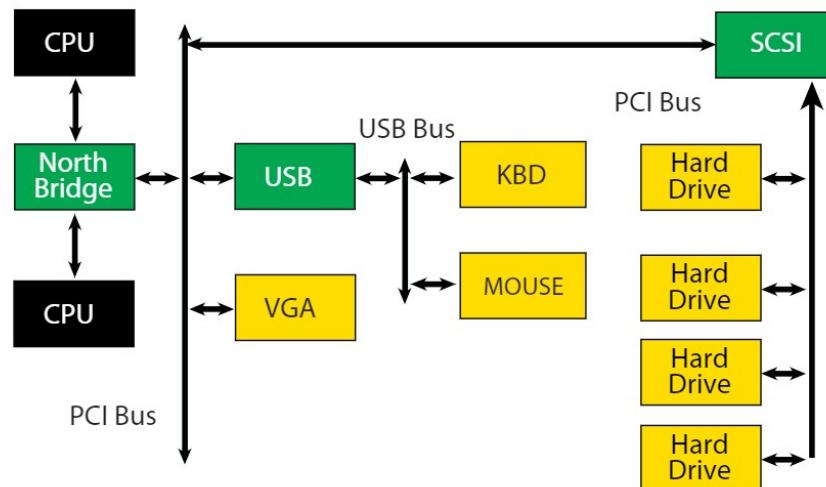
这个树状结构中的叶子节点是执行某种类型的 I/O 的外围设备。这可能包括键盘、显示器、磁盘、网络，等

等。非叶子节点是在设备和总线之间或不同总线类型之间移动数据的总线。图 2-5 显示了一个有四条总线和六个设备的桌面系统例子。



OM13142

图 2-6 是一个更复杂的服务器系统的例子。我们的想法是使 UEFI 驱动模型变得简单和可扩展，因此像下面这样的更复杂的系统可以在预启动环境中被描述和管理。这个系统包含六条总线和八个设备。



OM13143

在任何给定的平台中，固件服务、总线驱动器和设备驱动器的组合可能是由各种各样的供应商生产的，包括 OEM、IBV 和 IHV。这些来自不同供应商的不同组件需要一起工作，以产生一个 I/O 设备的协议，可以用来

启动一个符合 UEFI 的操作系统。因此，为了提高这些组件的互操作性，UEFI 驱动模型被描述得非常详细。本节的其余部分是对 UEFI 驱动模型的简要概述。它描述了 UEFI 驱动模型旨在解决的传统 Option ROM 问题、驱动的入口点、主机总线控制器、设备驱动的属性、总线驱动的属性，以及 UEFI 驱动模型如何适应热插拔事件。

2.5.1 传统 Option ROM 问题

传统 Option ROM 有很多限制和约束，限制了平台设计者和适配器供应商的创新。在撰写本文时，ISA 和 PCI 适配器都使用传统的 Option ROM。在本讨论中，只考虑 PCI Option ROM；传统的 ISA Option ROM 在 UEFI 规范中不被支持。

下面列出了传统 Option ROM 的主要约束和限制。对于每一个问题，都列出了设计 UEFI 驱动模型时的考虑。因此，UEFI 驱动模型的设计直接解决了克服 PC-AT 风格的传统 Option ROM 所隐含的限制的解决方案的要求。

2.5.1.1 32 位/16 位的真实模式二进制文件

传统 Option ROM 通常包含 IA-32 处理器的 16 位实模式代码。这意味着 PCI 卡上的传统 Option ROM 在不支持执行 IA-32 实模式二进制文件的平台上无法运行。另外，16 位实模式只允许驱动程序直接访问系统内存的低 1 MiB。驱动程序有可能将处理器切换到实模式以外的模式，以便访问 1 MiB 以上的资源，但这需要大量的额外代码，并导致与其他 Option ROM 和系统 BIOS 的互操作性问题。另外，将处理器切换到其他执行模式的 Option ROM 与 Itanium 处理器不兼容。

UEFI 驱动模型的设计考虑：

- 驱动程序需要平坦的内存模式，可以完全访问系统组件。
- 驱动程序需要用 C 语言编写，以便它们在不同的处理器架构之间可以移植。
- 驱动程序可以被编译成虚拟机可执行文件，允许单一二进制驱动程序在使用不同处理器架构的机器上工作。

2.5.1.2 使用 Option ROM 的固定资源

由于传统 Option ROM 只能直接寻址系统内存的低 1 MiB，这意味着传统 Option ROM 的代码必须存在 1 MiB 以下。在 PC-AT 平台上，从 `0x00000-0x9FFFF` 的内存是系统内存。从 `0xA0000-0xBFFFF` 的内存是 VGA 内存，而从 `0xF0000-0xFFFFF` 的内存是为系统 BIOS 保留的。另外，由于多年来系统 BIOS 变得更加复杂，许多平台也将 `0xE0000-0xEFFFF` 用于系统 BIOS。这就为传统 Option ROM 留下了从 `0xC0000-0xDFFFF` 的 128KB 的内存。这限制了在 BIOS POST 期间可以运行多少个传统 Option ROM。

另外，对于传统的 Option ROM 来说，分配系统内存不容易。它们的选择是：从扩展 BIOS 数据区（EBDA）分配内存，通过后期内存管理器（PMM）分配内存，或者根据启发式搜索空闲内存。其中，只有 EBDA 是标

准的，其他的在适配器之间或 BIOS 供应商之间使用不一致，这增加了复杂性和潜在的冲突。

UEFI 驱动模型的设计考虑：

- 驱动程序需要平坦的内存模式，可以完全访问系统组件。
- 驱动程序需要能够被重新定位，以便它们能够被加载到内存的任何地方（PE/COFF 镜像）。
- 驱动程序应该通过启动服务来分配内存。这些都是规范的接口，可以保证在各种平台的实现中都能发挥预期的作用。

2.5.1.3 将 Option ROM 与它们的设备相匹配

目前还不清楚哪个控制器可能被某个特定的传统 Option ROM 所管理。一些传统的 Option ROM 会在整个系统中搜索要管理的控制器。这可能是一个漫长的过程，取决于平台的大小和复杂性。另外，由于 BIOS 设计的限制，所有的传统 Option ROM 都必须被执行，而且在操作系统启动之前，它们必须扫描所有的外围设备。这也可能是一个漫长的过程，特别是当 SCSI 总线必须扫描 SCSI 设备时。这意味着传统的 Option ROM 正在对平台的初始化方式进行策略决策，哪些控制器由哪些传统的 Option ROM 管理。这使得系统设计者很难预测传统 Option ROM 将如何相互作用。这也会导致板载控制器的问题，因为一个传统的 Option ROM 可能会错误地选择管理板载控制器。

UEFI 驱动模型的设计考虑。

- 驱动器与控制器的匹配必须是确定性的。
- 通过平台驱动覆盖协议和驱动配置协议给 OEM 厂商更多控制权。
- 必须能够只启动启动操作系统所需的驱动程序和控制器。

2.5.1.4 与 PC-AT 系统设计的联系

传统的 Option ROM 假定了一个类似 PC-AT 的系统结构。它们中的许多包括直接触及硬件寄存器的代码。这可能使它们在无传统问题和无头平台上不兼容。传统的 Option ROM 也可能包含设置程序，这些程序假定有一个类似 PC-AT 的系统结构来与键盘或视频显示器互动。这使得设置程序在无传统系统和无头（TODO）平台上不兼容。

UEFI 驱动模型的设计考虑：

- 驱动程序应使用定义明确的协议与系统硬件、系统输入设备和系统输出设备进行交互。

2.5.1.5 规范中的歧义和经验中的变通方法

由于传统 Option ROM 和系统 BIOS 之间的不兼容，许多传统 Option ROM 和 BIOS 代码包含了变通方法。这些不兼容的情况之所以存在，部分原因是在如何编写传统 Option ROM 或编写系统 BIOS 方面没有明确的规范另外，中断链和启动设备的选择在传统的 Option ROM 中是非常复杂的。并不总是清楚哪个设备会成为操作系统的启动设备。

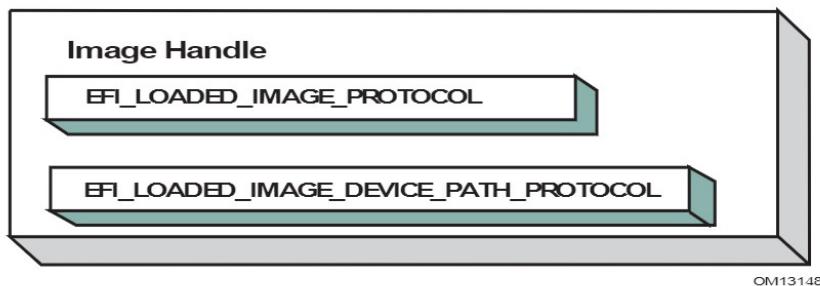
UEFI 驱动模型的设计考虑：

- 驱动程序和固件的编写要遵循这一规范。由于这两个组件都有明确定义的规范，因此可以开发合规性测试，以证明驱动程序和系统固件是合规的。这将消除在驱动程序或系统固件中建立变通方法的需要（除了那些可能需要解决特定硬件问题的方法）。
- 通过《平台驱动程序覆盖协议》和《驱动程序配置协议》以及其他 OEM 增值组件，让 OEM 拥有更多控制权，以管理启动设备的选择过程。

2.5.2 驱动程序初始化

驱动程序镜像的文件必须从某种类型的媒体上加载。这可能包括 ROM、FLASH、硬盘、软盘、CDROM，甚至是网络连接。一旦找到了驱动镜像，就可以通过启动服务 `EFI_BOOT_SERVICES.LoadImage()` 将其加载到系统内存。`LoadImage()` 将一个 PE/COFF 格式的镜像加载到系统内存。

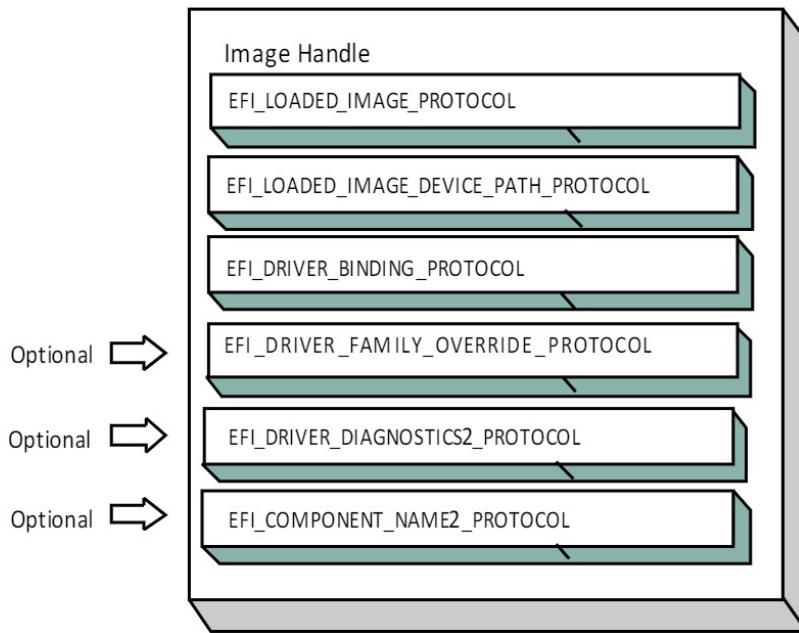
为驱动程序创建一个句柄，并在该句柄上放置一个加载的镜像协议实例。一个包含加载镜像协议实例的句柄被称为镜像句柄。在这一点上，驱动程序还没有被启动。它只是坐在内存中等待被启动。图 2-7 显示了在调用 `LoadImage()` 后驱动程序的镜像句柄的状态。



在用启动服务 `LoadImage()` 加载了一个驱动程序后，必须用启动服务 `EFI_BOOT_SERVICES.StartImage()` 来启动它。所有类型的 UEFI 应用程序和 UEFI 驱动程序都是如此，它们可以在符合 UEFI 的系统上加载和启动。遵循 UEFI 驱动模型的驱动的入口点必须遵循一些严格的规则。首先，它不允许接触任何硬件。相反，驱动程序只允许在它自己的镜像句柄上安装协议实例。遵循 UEFI 驱动模型的驱动必须在它自己的镜像句柄上安装一个驱动绑定协议的实例。

它可以选择安装驱动配置协议、驱动诊断协议或组件名称协议。此外，如果一个驱动程序希望是可卸载的，它可以选择性地更新加载的镜像协议（见第 9 节），以提供它自己的 `Unload()` 函数。

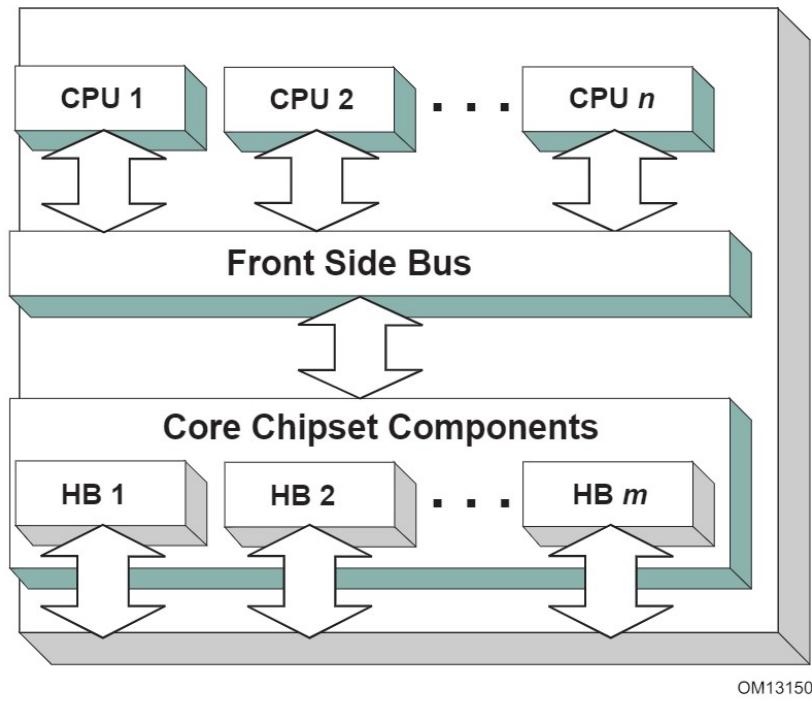
最后，如果一个驱动程序需要在启动服务 `EFI_BOOT_SERVICES.ExitBootServices()` 被调用时执行任何特殊操作，它可以选择性地创建一个带有通知函数的事件，在启动服务 `ExitBootServices()` 被调用时被触发。一个包含驱动绑定协议实例的镜像句柄被称为驱动镜像句柄。图 2-8 显示了图 2-7 中的镜像处理在启动服务后的可能配置 `StartImage()` 已被调用。



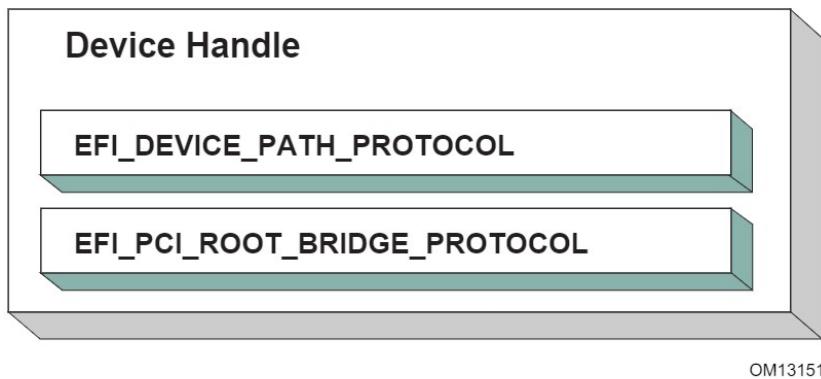
2.5.3 主机总线控制器

驱动程序不允许接触任何硬件，在驱动程序的入口点。因此，驱动程序将被加载和启动，但它们都将等待被告知管理系统中的一个或多个控制器。一个平台组件，如启动管理器，负责管理驱动程序与控制器的连接。然而，在进行第一次连接之前，必须要有一些初始的控制器集合供驱动程序管理。这个控制器的初始集合被称为主机总线控制器。主机总线控制器提供的 I/O 抽象是由 UEFI 驱动模型范围之外的固件组件产生的。主机总线控制器的设备句柄和每个控制器的 I/O 抽象必须由平台上的核心固件产生，或者由可能不遵循 UEFI 驱动模型的驱动程序产生。参见 PCI 根桥 I/O 协议规范，了解 PCI 总线的 I/O 抽象的例子。

一个平台可以被看作是一组处理器和一组核心芯片组部件，它们可能产生一个或多个主机总线。图 2-9 显示了一个有 n 个处理器（CPU）的平台，以及一组产生 m 个主机桥的核心芯片组组件。



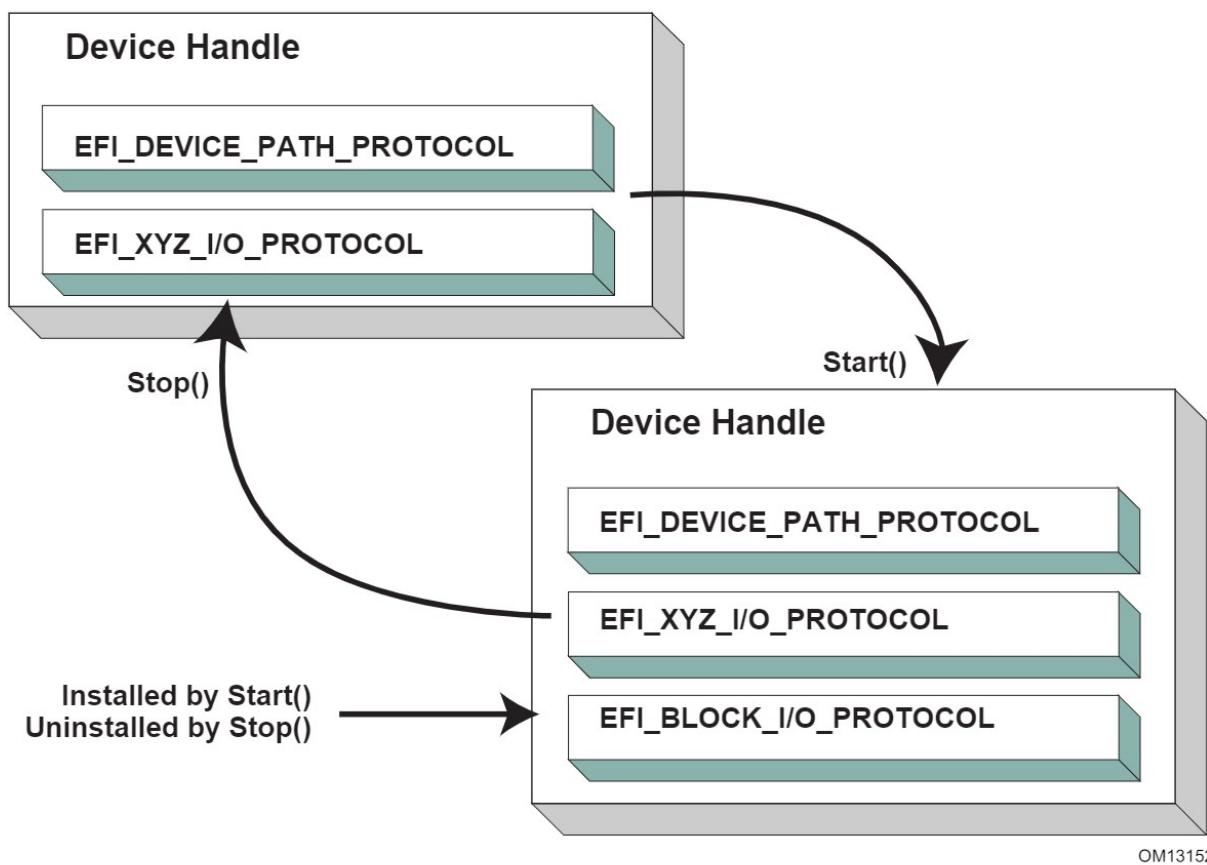
每个主机桥在 UEFI 中被表示为一个设备句柄，它包含一个设备路径协议实例，以及一个抽象了主机总线可以执行的 I/O 操作的协议实例。例如，一个 PCI 主机总线控制器支持一个或多个 PCI 根桥，这些根桥是由 PCI 根桥 I/O 协议抽象出来的。图 2-10 显示了一个 PCI 根桥的设备处理实例。



一个 PCI 总线驱动程序可以连接到这个 PCI 根桥，并为系统中的每个 PCI 设备创建子句柄。然后，PCI 设备驱动程序应连接到这些子句柄，并产生 I/O 抽象，可用于启动 UEFI 兼容的操作系统。下面的章节描述了在 UEFI 驱动模型中可以实现的不同类型的驱动。UEFI 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。相反，主要的类型将被涵盖，可以作为设计和实现其他驱动类型的起点。

设备驱动程序不允许创建任何新的设备句柄。相反，它在现有的设备句柄上安装了额外的协议接口。最常见的设备驱动程序类型是将一个 I/O 抽象附加到一个由总线驱动程序创建的设备句柄上。这个 I/O 抽象可以用来启动一个 UEFI 兼容的操作系统。一些 I/O 抽象的例子包括简单文本输出、简单输入、块 I/O 和简单网络协议。图 2-11 显示了在设备驱动程序连接到它之前和之后的设备句柄。在这个例子中，设备句柄是 XYZ 总线的一个子节点，所以它包含一个 XYZ 总线支持的 I/O 协议。它还包含一个设备路径协议，该协议是由 XYZ 总线驱动程序放在那里的。设备路径协议不是所有设备句柄都需要的。它只对代表系统中物理设备的设备句柄有要求。虚拟设备的句柄将不包含设备路径协议。



连接到图 2-11 中设备句柄的设备驱动程序必须在自己的镜像句柄上安装了一个驱动程序绑定协议。驱动程序绑定协议（见第 11.1 节）包含三个函数，称为 `Supported()`、`Start()` 和 `Stop()`。`Supported()` 函数测试驱动程序是否支持一个给定的控制器。在这个例子中，驱动程序将检查设备句柄是否支持设备路径协议和 XYZ I/O 协议。如果一个驱动程序的 `Supported()` 函数通过了，那么该驱动程序就可以通过调用驱动程序的 `Start()` 函数连接到控制器。`Start()` 函数实际上是将额外的 I/O 协议添加到设备句柄中。在这个例子中，块 I/O 协议正在被安装。为了提供对称性，驱动程序绑定协议也有一个 `Stop()` 函数，迫使驱动程序停止管理一个设备句柄。

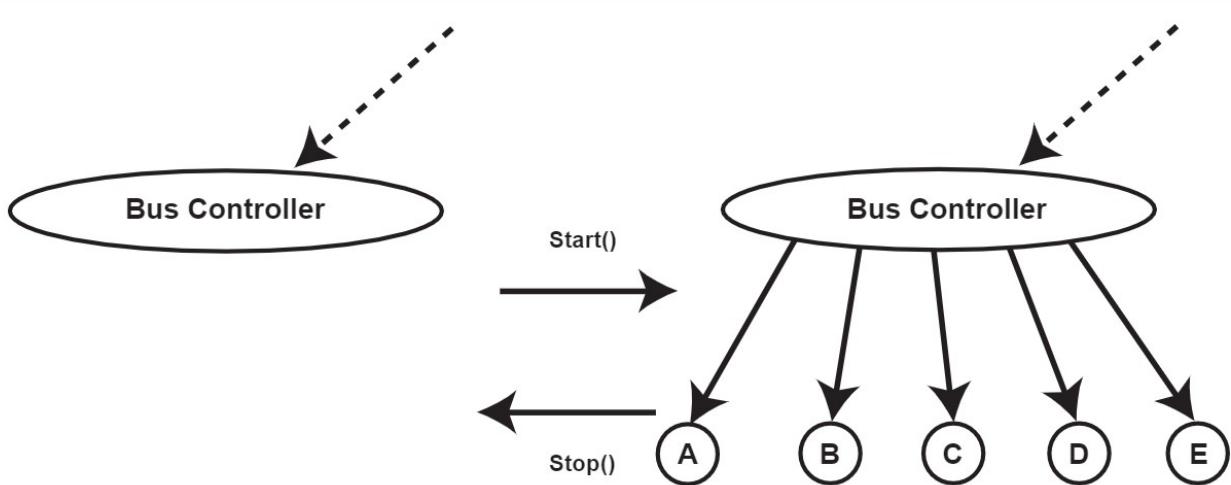
柄。这将导致设备驱动卸载任何在 `Start()` 中安装的协议接口。

EFI 驱动绑定协议的 `Supported()`、`Start()` 和 `Stop()` 函数需要利用启动服务 `EFI_BOOT_SERVICES.OpenProtocol()` 来获取协议接口，启动服务 `EFI_BOOT_SERVICES.CloseProtocol()` 来释放一个协议接口。

`OpenProtocol()` 和 `CloseProtocol()` 更新由系统固件维护的句柄数据库，以跟踪哪些驱动程序正在消费协议接口。把柄数据库中的信息可以用来检索有关驱动程序和控制器的信息。新的启动服务 `EFI_BOOT_SERVICES.OpenProtocolInformation()` 可以用来获取当前正在消费特定协议接口的组件列表。

2.5.4 总线驱动

从 UEFI 驱动模型的角度来看，总线驱动和设备驱动几乎是相同的。唯一的区别是，总线驱动为总线驱动在其总线上发现的子控制器创建新的设备句柄。因此，总线驱动比设备驱动稍微复杂一些，但这反过来又简化了设备驱动的设计和实现。总线驱动器有两种主要类型。第一种是在第一次调用 `Start()` 时为所有子控制器创建句柄。另一种类型允许在多次调用 `Start()` 时为子控制器创建句柄。第二种类型的总线驱动器在支持快速启动能力方面非常有用。它允许创建几个子句柄或甚至一个子句柄。在那些需要很长时间来列举所有子节点的总线上（比如 SCSI），这可以为平台的启动节省很大的时间。图 2-12 显示了在调用 `Start()` 之前和之后的总线控制器的树状结构。进入总线控制器节点的虚线代表与总线控制器的父控制器的链接。如果该总线控制器是一个主机总线控制器，那么它就没有一个父控制器。节点 A、B、C、D 和 E 代表总线控制器的子控制器。

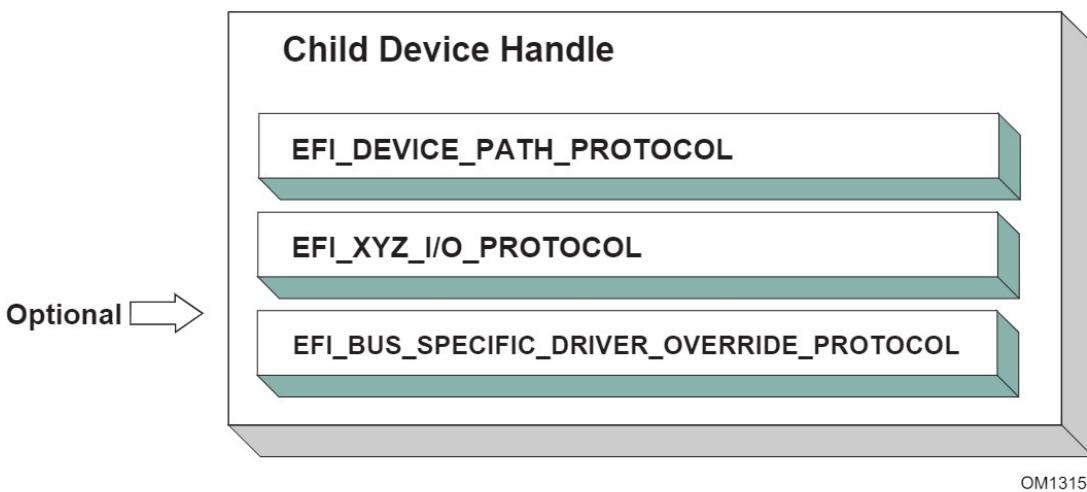


OM13153

一个支持在每次调用 `Start()` 时创建一个孩子的总线驱动器可能会选择先创建孩子 C，然后是孩子 E，然后是剩下的孩子 A、B 和 D。

总线驱动器必须在创建的每个子控制器上安装协议接口。至少，它必须安装一个协议接口，为子控制器提供一个总线服务的 I/O 抽象。如果总线驱动程序创建的子句柄代表一个物理设备，那么总线驱动程序也必须在子句

柄上安装一个设备路径协议实例。总线驱动器可以选择在每个子句柄上安装一个总线特定的驱动程序覆盖协议。这个协议是在驱动程序连接到子控制器时使用的。启动服务 `EFI_BOOT_SERVICES.ConnectController()` 使用架构上定义的优先级规则来为给定的控制器选择最佳的驱动程序集。总线特定驱动程序覆盖协议的优先级高于一般的驱动程序搜索算法，而低于平台覆盖协议的优先级。一个总线特定驱动程序选择的例子发生在 PCI 上。PCI 总线驱动程序存储在 PCI 控制器的 Option ROM 中的驱动程序比存储在平台其他地方的驱动程序具有更高的优先权。图 2-13 显示了一个子设备句柄的例子，它是由支持总线特定驱动程序覆盖机制的 XYZ 总线驱动程序创建的



2.5.5 平台组件

在 UEFI 驱动模型下，连接和断开驱动与平台中的控制器的行为是在平台固件的控制下进行的。这通常是作为 UEFI 启动管理器的一部分来实现的，但其他实现也是可能的。平台固件可以使用启动服务 `EFI_BOOT_SERVICES.ConnectController()` 和 `EFI_BOOT_SERVICES.DisconnectController()` 来决定哪些控制器被启动，哪些没有。如果平台希望执行系统诊断或安装操作系统，那么它可以选择将驱动程序连接到所有可能的启动设备。如果一个平台希望启动一个预装的操作系统，它可以选择只将驱动程序连接到启动所选操作系统所需的设备上。UEFI 驱动模型通过启动服务 `ConnectController()` 和 `DisconnectController()` 支持这两种操作模式。此外，由于负责启动的平台组件是平台必须与控制台设备和启动选项的设备路径一起工作，UEFI 驱动模型中涉及的所有服务和协议都在考虑设备路径的情况下进行了优化。

由于平台固件可以选择只连接生产控制台所需的设备，并获得对启动设备的访问，所以操作系统现在的设备驱动程序不能假设设备的 UEFI 驱动程序已经被执行。在系统固件或 Option ROM 中出现的 UEFI 驱动并不能保证 UEFI 驱动会被加载、执行，或被允许管理平台中的任何设备。所有操作系统中的设备驱动必须能够处理已经被 UEFI 驱动管理的设备和没有被 UEFI 驱动管理的设备。

平台也可以选择产生一个名为平台驱动程序覆盖协议的协议。这与总线专用驱动程序覆盖协议类似，但它的优先级更高。这使平台固件在决定哪些驱动程序连接到哪些控制器时具有最高优先权。平台驱动程序覆盖协议被连接到系统中的一个句柄。如果系统中存在该协议，启动服务 `ConnectController()` 将使用该协议。

2.5.6 热插拔事件

在过去，系统固件不需要处理预启动环境中的热插拔事件。然而，随着像 USB 这样的总线的出现，终端用户可以在任何时候添加和移除设备，重要的是要确保在 UEFI 驱动模型中可以描述这些类型的总线。这取决于支持热添加和删除设备的总线的总线驱动程序是否为这类事件提供支持。对于这些类型的总线，一些平台管理将不得不转移到总线驱动器中。例如，当键盘被热添加到平台上的 USB 总线上时，终端用户会希望键盘处于活动状态。USB 总线驱动程序可以检测到热添加事件并为键盘设备创建一个子句柄。然而，因为除非 `EFI_BOOT_SERVICES.ConnectController()` 被调用，否则驱动程序不会被连接到控制器，键盘不会成为一个活动的输入设备。让键盘驱动成为活动设备需要 USB 总线驱动在发生热添加事件时调用 `ConnectController()`。此外，当发生热移除事件时，USB 总线驱动程序将不得不调用 `EFI_BOOT_SERVICES.DisconnectController()`。如果 `EFI_BOOT_SERVICES.DisconnectController()` 返回错误，USB 总线驱动程序需要从一个定时器事件中重试 `EFI_BOOT_SERVICES.DisconnectController()`，直到它成功。

设备驱动程序也会受到这些热插拔事件的影响。在 USB 的情况下，一个设备可以在没有任何通知的情况下被移除。这意味着，USB 设备驱动程序的 `Stop()` 函数将不得不处理关闭一个不再存在于系统中的设备的驱动程序的问题。因此，任何未完成的 I/O 请求将不得不被刷新，而实际上无法接触到设备硬件。

一般来说，增加对热插拔事件的支持会大大增加总线驱动和设备驱动的复杂性。添加这种支持取决于驱动程序的编写者，因此需要将驱动程序的额外复杂性和大小与预启动环境中对该功能的需求进行权衡。

2.5.7 EFI 服务绑定

UEFI 驱动模型可以很好地映射到硬件设备、硬件总线控制器，以及在硬件设备之上的软件服务的简单组合。然而，UEFI 驱动模型并不能很好地映射到复杂的软件服务组合上。因此，对于更复杂的软件服务组合，需要一套额外的补充协议。

图 2-14 包含了三个例子，显示了软件服务相互之间的不同方式。在前两种情况下，每个服务消耗一个或多个其它服务，并且最多一个其它的服务消耗所有的服务。

`EFI_DRIVER_BINDING_PROTOCOL` 可以用来模拟案例 #1 和 #2，但它不能用来模拟案例 #3，因为 UEFI 启动服务 `OpenProtocol()` 的行为方式。当与 `BY_DRIVER` 开放模式一起使用时，`OpenProtocol()` 允许每个协议最多只有一个消费者。这个功能非常有用，可以防止多个驱动程序试图管理同一个控制器。然而，这使得产生像案例 3 那样的软件服务集变得困难。

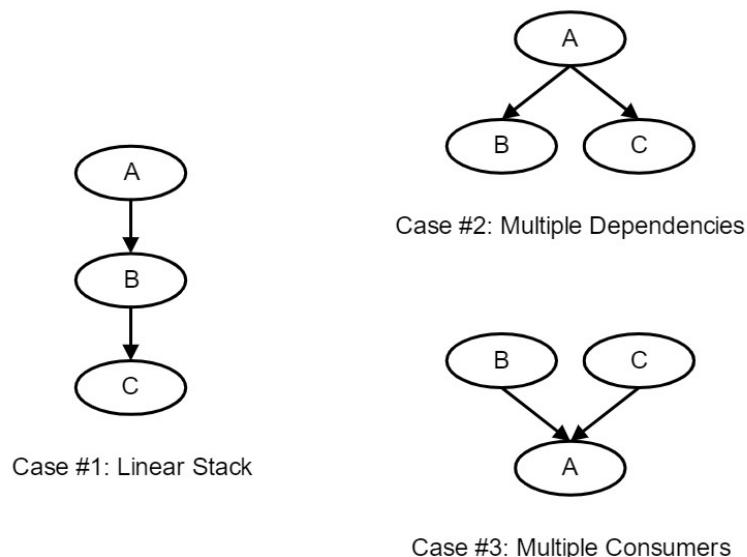


图 4. 软件服务关系

`EFI_SERVICE_BINDING_PROTOCOL` 提供的机制允许协议有一个以上的消费者。`EFI_SERVICE_BINDING_PROTOCOL` 与 `EFI_DRIVER_BINDING_PROTOCOL` 一起使用。一个 UEFI 驱动程序产生的协议需要同时提供给一个以上的消费者，将同时产生 `EFI_DRIVER_BINDING_PROTOCOL` 和 `EFI_SERVICE_BINDING_PROTOCOL`。这种类型的驱动程序是一种混合驱动程序，它将在其驱动程序入口点中产生 `EFI_DRIVER_BINDING_PROTOCOL`。当驱动程序收到开始管理一个控制器的请求时，它将在被启动的控制器的句柄上产生 `EFI_SERVICE_BINDING_PROTOCOL`。`EFI_SERVICE_BINDING_PROTOCOL` 与 UEFI 规范中定义的其他协议略有不同。它没有一个与之相关的 GUID。相反，这个协议实例结构实际上代表了一个协议系列。每个需要 `EFI_SERVICE_BINDING_PROTOCOL` 实例的软件服务驱动程序都需要为自己的 `EFI_SERVICE_BINDING_PROTOCOL` 类型生成一个新的 GUID。这一要求就是本规范中的各种网络协议包含两个 GUID 的原因。一个是该网络协议的 `EFI_SERVICE_BINDING_PROTOCOL` GUID，另一个 GUID 是包含网络驱动程序产生的特定成员服务的协议。这里定义的机制不限于网络协议驱动程序。它可以应用于 `EFI_DRIVER_BINDING_PROTOCOL` 不能直接映射的任何协议集，因为这些协议包含一个或多个关系，如图 2-14 中的案例 #3。无论是 `EFI_DRIVER_BINDING_PROTOCOL` 还是组合 `EFI_DRIVER_BINDING_PROTOCOL` 和 `EFI_SERVICE_BINDING_PROTOCOL` 可以处理圆形的依赖关系。有一些方法可以允许循环引用，但它们要求循环链接在短时间内存在。当使用跨越循环链接的协议时，这些方法还要求协议必须以 `EXCLUSIVE` 的开放模式打开，这样，任何试图通过调用 `DisconnectController()` 来解构协议集的行为都会失败。一旦驱动完成了跨循环链路的协议，该协议就应该被关闭。

2.6 要求

本文件是一个架构规范。因此，在实现中保留了最大的灵活度。但是，有一些要求，这个规范的元素必须被实现，以确保操作系统加载器和其他设计成与 UEFI 启动服务一起运行的代码可以依赖一个一致的环境。为了描述这些要求，这个规范被分成了必要的和可选的元素。一般来说，一个可选的元素在与该元素名称相匹配的章节中被完全定义。然而，对于必需的元素，在少数情况下，定义可能不完全包含在为特定元素命名的部分中。在实现必要元素时，应该注意涵盖本规范中定义的与特定元素相关的所有语义。

2.6.1 必要元素

表 2-11 列出了所需必要元素。任何符合本规范的系统，都必须实现这些元素。这意味着所有必要的服务功能和协议都必须存在，并且实现必须为所有的调用和参数组合提供规范中定义的全部语义。应用程序、驱动程序或操作系统加载器，他们可以假设所有这些系统都实现了所有的必要要素

系统供应商可能会选择不实现所有要求的元素，例如，在专门的系统配置上，不支持要求的元素所隐含的所有服务和功能。然而，由于大多数应用程序、驱动程序和操作系统加载器的编写是假设所有必要的元素都存在于实现 UEFI 规范的系统上；任何这样的代码都可能需要明确的定制，以运行在对该规范中所要求的元素的不完全实现。

2.6.2 平台特定的元素

根据平台所需的特定功能，可以添加或删除许多元素。平台固件开发者需要根据所包含的功能来实现 UEFI 元素。以下是可能的平台特征和每种特征类型所需的要素清单：

1. 如果一个平台包括控制台设备,必须实现 `EFI_SIMPLE_TEXT_INPUT_PROTOCOL`、`EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` 和 `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL`。
2. 如果一个平台包括一个配置基础设施,那么 `EFI_HII_DATABASE_PROTOCOL`、`EFI_HII_STRING_PROTOCOL`、`EFI_HII_CONFIG_ROUTING_PROTOCOL`, `EFI_HII_CONFIG_ACCESS_PROTOCOL` 是必须的。如果你支持位图字体, 你必须支持 `EFI_HII_FONT_PROTOCOL`。
3. 如果一个平台包括图形控制台设备,那么必须实现 `EFI_GRAPHICS_OUTPUT_PROTOCOL`、`EFI_EDID_DISCOVERED_PROTOCOL` 和 `EFI_EDID_ACTIVE_PROTOCOL`。为了支持 `EFI_GRAPHICS_OUTPUT_PROTOCOL`, 一个平台必须包含一个驱动程序来消费 `EFI_GRAPHICS_OUTPUT_PROTOCOL` 并产生 `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL`, 即使 `EFI_GRAPHICS_OUTPUT_PROTOCOL` 是由一个外部驱动程序产生的。
4. 如果一个平台包括一个指针设备作为其控制台支持的一部分, `EFI_SIMPLE_POINTER_PROTOCOL` 必须被实现。
5. 如果一个平台包括从磁盘设备启动的能力,那么就需要 `EFI_BLOCK_IO_PROTOCOL`、`EFI_DISK_IO_PROTOCOL`、`EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 以及 `EFI_UNICODE_COLLATION_PROTOCOL`。此外, 必须实现对 MBR、GPT 和 El Torito 的分区支持。对于支持 SPC-4 或 ATA8-ACS 命令集安全命令的磁盘设备, 还需要 `EFI_STORAGE_SECURITY_COMMAND_PROTOCOL`。（TODO 原文多了下划线）外部驱动程序可以产生

块 I/O 协议和 `EFI_STORAGE_SECURITY_COMMAND_PROTOCOL`。所有其他从磁盘设备启动所需的协议必须作为平台的一部分进行。

6. 如果一个平台可以从网络设备 TFTP 启动, 那么就需要 `EFI_PXE_BASE_CODE_PROTOCOL`。平台必须准备好在 `EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL(UNDI)`、`EFI_SIMPLE_NETWORK_PROTOCOL` 或 `EFI_MANAGED_NETWORK_PROTOCOL` 中的任何一种上产生这个协议。如果平台包括验证通过网络设备收到的启动镜像的能力。也需要支持镜像验证, 包括 `SetupMode` 等于 0, 启动镜像的哈希值或镜像对应的验证证书存在于'db' 变量而不是'dbx' 变量中。一个外部驱动可以产生 UNDI 接口。从网络设备启动所需的所有其他协议必须由平台来执行。
7. 如果一个平台支持 UEFI 通用网络应用,那么 `EFI_MANAGED_NETWORK_PROTOCOL`, `EFI_MANAGED_NETWORK_SERVICE`, `EFI_ARP_PROTOCOL`, `EFI_ARP_SERVICE_BINDING_PROTOCOL`, `EFI_DHCP4_PROTOCOL`, `EFI_DHCP4_SERVICE_BINDING_PROTOCOL`, `EFI_TCP4_PROTOCOL`, `EFI_TCP4_SERVICE_BINDING_PROTOCOL`, `EFI_IP4_PROTOCOL`, `EFI_IP4_SERVICE_BINDING_PROTOCOL`, `EFI_IP4_CONFIG2_PROTOCOL`, `EFI_UDP4_PROTOCOL` 和 `EFI_UDP4_SERVICE_BINDING_PROTOCOL` 是必需的。如果该平台需要额外的 IPv6 支持,那么需要 `EFI_DHCP6_PROTOCOL`、`EFI_DHCP6_SERVICE_BINDING_PROTOCOL`、`EFI_TCP6_PROTOCOL`、`EFI_TCP6_SERVICE_BINDING_PROTOCOL`。`EFI_IP6_PROTOCOL`、`EFI_IP6_SERVICE_BINDING_PROTOCOL`、`EFI_IP6_CONFIG_PROTOCOL`、`EFI_UDP6_PROTOCOL` 和 `EFI_UDP6_SERVICE_BINDING_PROTOCOL` 是额外要求的。如果网络应用需要 DNS 功能, `EFI_DNS4_SERVICE_BINDING_PROTOCOL` 和 `EFI_DNS4_PROTOCOL` 是 IPv4 协议栈的必备条件。IPv6 协议栈需要 `EFI_DNS6_SERVICE_BINDING_PROTOCOL` 和 `EFI_DNS6_PROTOCOL`。如果网络环境需要 TLS 功能, 需要 `EFI_TLS_SERVICE_BINDING_PROTOCOL`、`EFI_TLS_PROTOCOL` 和 `EFI_TLS_CONFIGURATION_PROTOCOL`。如果网络环境需要 IPSEC 功能,需要 `EFI_IPSEC_CONFIG_PROTOCOL` 和 `EFI_IPSEC2_PROTOCOL`。如果网络环境需要 VLAN 功能, 需要 `EFI_VLAN_CONFIG_PROTOCOL`。
8. 如果一个平台包括一个字节流设备, 如 UART, 那么 `EFI_SERIAL_IO_PROTOCOL` 必须被实现。
9. 如果一个平台包括 PCI 总线支持, 那么 `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`, `EFI_PCI_IO_PROTOCOL`, 必须被实现。
10. 如果一个平台包括 USB 总线支持, 那么必须实现 `EFI_USB2_HC_PROTOCOL` 和 `EFI_USB_IO_PROTOCOL`。一个外部设备可以通过产生一个 USB 主机控制器协议来支持 USB。
11. 如果一个平台包括一个 NVM Express 控制器,那么必须实现 `EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL`。
12. 如果一个平台支持从面向块的 NVM Express 控制器启动, 那么必须实现 `EFI_BLOCK_IO_PROTOCOL`。一个外部驱动程序可以产生 `EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL`。从 NVM Express 子系统启动所需的所有其他协议必须由平台携带。
13. 如果一个平台包括一个利用 SCSI 命令包的 I/O 子系统, 那么 `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` 必须被实现。
14. 如果一个平台支持从面向块的 SCSI 外设启动,那么必须实现 `EFI_SCSI_IO_PROTOCOL` 和 `EFI_BLOCK_IO_PROTOCOL`。一个外部驱动程序可以产生 `EFI_EXT_SCSI_PASS_THRU_PROTOCOL`。从 SCSI I/O 子系统启动所需的所有其他协议必须由平台携带。
15. 如果一个平台支持从 iSCSI 外围启动, 那么必须实现 `EFI_ISCSI_INITIATOR_NAME_PROTOCOL` 和 `EFI_AUTHENTICATION_INFO_PROTOCOL`。
16. 如果一个平台包括调试功能, 那么 `EFI_DEBUG_SUPPORT_PROTOCOL`、`EFI_DEBUGPORT_PROTOCOL` 和 `EFI_MIRROR_INFORMATION_TABLE` 必须被实现。

17. 如果一个平台包括将默认驱动程序覆盖到 UEFI 驱动程序模型提供的控制器匹配算法的能力, 那么必须实现 [EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL](#)。
18. 如果一个平台包括一个利用 ATA 命令包的 I/O 子系统, 那么必须实现 [EFI_ATA_PASS_THRU_PROTOCOL](#)。
19. 如果一个平台支持来自非永久连接到平台的设备的 Option ROM, 并且支持验证这些 Option ROM 的能力, 那么它必须支持《网络协议-UDP 和 MTFTP》中描述的 Option ROM 验证方法和第 8.1.1 节中描述的验证 UEFI 变量。
20. 如果一个平台包括验证 UEFI 镜像的能力, 并且该平台可能支持一个以上的操作系统加载器, 它必须支持网络协议-UDP 和 MTFTP 中描述的方法以及第 8.1.1 节中描述的验证 UEFI 变量。
21. 从 UEFI 规范 2.8 版开始, 不再需要 EBC 支持。如果一个 EBC 解释器被实现, 那么它必须产生 [EFI_EBC_PROTOCOL](#) 接口。
22. 如果一个平台包括从网络设备执行基于 HTTP 的启动的能力, 那么就需要 [EFI_HTTP_SERVICE_BINDING_PROTOCOL](#)、[EFI_HTTP_PROTOCOL](#) 和 [EFI_HTTP_UTILITIES_PROTOCOL](#)。如果它包括从网络设备执行基于 HTTPS 的启动的能力, 除了上述协议, 还需要 [EFI_TLS_SERVICE_BINDING_PROTOCOL](#)、[EFI_TLS_PROTOCOL](#) 和 [EFI_TLS_CONFIGURATION_PROTOCOL](#)。如果它包括执行基于 HTTP(S) 的启动和 DNS 功能的能力, 那么 IPv4 堆栈需要 [EFI_DNS4_SERVICE_BINDING_PROTOCOL](#)、[EFI_DNS4_PROTOCOL](#); IPv6 堆栈需要 [EFI_DNS6_SERVICE_BINDING_PROTOCOL](#) 和 [EFI_DNS6_PROTOCOL](#)。
23. 如果一个平台包括从具有 EAP 功能的网络设备执行无线启动的能力, 并且如果该平台提供独立的无线 EAP 驱动程序, 则需要 [EFI_EAP_PROTOCOL](#)、[EFI_EAP_CONFIGURATION_PROTOCOL](#) 和 [EFI_EAP_MANAGEMENT2_PROTOCOL](#); 如果该平台提供独立的无线请求器, 则需要 [EFI_SUPPLICANT_PROTOCOL](#) 和 [EFI_EAP_CONFIGURATION_PROTOCOL](#)。如果它包括使用 TLS 功能进行无线启动的能力, 那么需要 [EFI_TLS_SERVICE_BINDING_PROTOCOL](#)、[EFI_TLS_PROTOCOL](#) 和 [EFI_TLS_CONFIGURATION_PROTOCOL](#)。
24. 如果一个平台支持经典蓝牙, 那么必须实现 [EFI_BLUETOOTH_HC_PROTOCOL](#)、[EFI_BLUETOOTH_IO_PROTOCOL](#) 和 [EFI_BLUETOOTH_CONFIG_PROTOCOL](#), 并且可以实现 [EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL](#)。如果一个平台支持 Bluetooth Smart (Bluetooth Low Energy), 那么必须实现 [EFI_BLUETOOTH_HC_PROTOCOL](#)、[EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL](#) 和 [EFI_BLUETOOTH_LE_CONFIG_PROTOCOL](#)。如果一个平台同时支持蓝牙经典和蓝牙 LE, 那么上述两个要求都应该得到满足。
25. 如果一个平台支持通过 HTTP 或通过带内路径与 BMC 进行 RESTful 通信, 那么必须实现 [EFI_REST_PROTOCOL](#) 或 [EFI_REST_EX_PROTOCOL](#)。如果 [EFI_REST_EX_PROTOCOL](#) 被实现, [EFI_REST_EX_SERVICE_BINDING_PROTOCOL](#) 也必须被实现。如果一个平台支持通过 HTTP 或通过带内路径与 BMC 进行 Redfish 通信, 可以实现 [EFI_REDFISH_DISCOVER_PROTOCOL](#) 和 [EFI_REST_JSON_STRUCTURE_PROTOCOL](#)。
26. 如果一个平台包括使用硬件功能来创建高质量的随机数的能力, 这种能力应该通过 [EFI_RNG_PROTOCOL](#) 的实例暴露出来, 至少有一种 EFI RNG 算法被支持。
27. 如果一个平台允许安装加载选项变量 (Boot####, 或 Driver####, 或 SysPrep####), 该平台必须支持和识别变量内所有定义的属性值, 并在 [BootOptionSupport](#) 中报告这些能力。如果一个平台支持安装 Driver#### 类型的加载选项变量, 所有安装的 Driver#### 变量必须被处理, 并在每次启动时加载和初始化指定的驱动程序。而且所有安装的 SysPrep#### 选项必须在处理 Boot#### 选项之前被处理。
28. 如果平台支持 UEFI 安全启动, 如安全启动和驱动程序签名中所述, 平台必须提供第 37.4 节中描述

的 PKCS 验证功能。

29. 如果一个平台包括一个利用 SD 或 eMMC 命令包的 I/O 子系统,那么必须实现 [EFI_SD_MMC_PASS_THRU_PROTOCOL](#)。
。
30. 如果一个平台包括创建/销毁指定的 RAM 磁盘的能力, [EFI_RAM_DISK_PROTOCOL](#) 必须被实现, 并且这个协议只存在一个实例。
31. 如果一个平台包括一个支持在指定范围内基于硬件擦除的大容量存储设备, 那么必须实现 [EFI_ERASE_BLOCK_PROTOCOL](#)。
32. 如果一个平台包括在调用 [ResetSystem](#) 时注册通知的功能,那么必须实现 [EFI_RESET_NOTIFICATION_PROTOCOL](#)。
。
33. 如果一个平台包括 UFS 设备, 必须实现 [EFI_UFS_DEVICE_CONFIG_PROTOCOL](#)。
34. 如果一个平台在调用 [ExitBootServices\(\)](#) 后不能支持 [EFI_RUNTIME_SERVICES](#) 中定义的调用, 该平台允许提供这些运行时服务的实现, 在运行时调用时返回 [EFI_UNSUPPORTED](#)。在这样的系统上, 应该发布一个 [EFI_RT_PROPERTIES_TABLE](#) 配置表, 描述哪些运行时服务在运行时被支持。
35. 如果平台包括对具有相干内存的 CXL 设备的支持, 那么平台必须支持从设备中提取相干设备属性表 (CDAT), 使用 CXL 数据对象交换服务 (如 CXL 2.0 规范中定义的) 或安装在该设备上的 [EFI_ADAPTER_INFORMATION_PROTOCOL](#) 实例 (具有 [EFI_ADAPTER_INFO_CDAT_TYPE_GUID](#) 类型)。

注意:一些所需的协议实例是由相应的服务绑定协议创建的。例如, [EFI_IP4_PROTOCOL](#) 是由 [EFI_IP4_SERVICE_BINDING](#) 创建。详细情况请参考服务绑定协议的相应章节。

2.6.3 驱动程序的特定要素

有一些 UEFI 元素可以被添加或删除, 这取决于特定驱动程序所需的功能。驱动程序可以由平台固件开发者实现, 以支持特定平台的总线和设备。驱动程序也可以由附加卡供应商实现, 用于可能被集成到平台硬件中的设备或通过扩展槽添加到平台中的设备。

下面的列表包括可能的驱动程序特性, 以及每种特性类型所需的 UEFI 元素。

1. 如果一个驱动程序遵循本规范的驱动程序模型, 就必须实现 [EFI_DRIVER_BINDING_PROTOCOL](#)。强烈建议所有遵循本规范的驱动程序模型的驱动程序也实现 [EFI_COMPONENT_NAME2_PROTOCOL](#)。
2. 如果一个驱动程序需要配置信息, 该驱动程序必须使用 [EFI_HII_DATABASE_PROTOCOL](#)。驱动程序不应该以其他方式向用户显示信息或向用户请求信息。
3. 如果一个驱动程序需要诊断, 必须实现 [EFI_DRIVER_DIAGNOSTICS2_PROTOCOL](#)。为了支持低启动时间, 在正常启动期间限制诊断。耗时的诊断应该推迟到调用 [EFI_DRIVER_DIAGNOSTICS2_PROTOCOL](#) 时进行。
4. 如果一个总线支持能够为驱动程序提供容器的设备 (例如, Option ROM), 那么该总线类型的总线驱动程序必须实现 [EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL](#)。
5. 如果为控制台输出设备编写驱动程序, 那么必须实现 [EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL](#)。

6. 如果为图形控制台输出设备编写驱动程序,则必须实现 `EFI_GRAPHICS_OUTPUT_PROTOCOL`、`EFI_EDID_DISCOVERED_PROTOCOL` 和 `EFI_EDID_ACTIVE_PROTOCOL`。
7. 如果为控制台输入设备编写驱动程序,那么必须实现 `EFI_SIMPLE_TEXT_INPUT_PROTOCOL` 和 `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL`。
8. 如果为一个指针设备编写驱动程序, 那么必须实现 `EFI_SIMPLE_POINTER_PROTOCOL`。
9. 如果为网络设备编写驱动程序, 那么必须实现 `EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL`、`EFI_SIMPLE_NETWORK_PROTOCOL` 或 `EFI_MANAGED_NETWORK_PROTOCOL`。如果硬件中支持 VLAN, 那么网络设备的驱动程序可以实现 `EFI_VLAN_CONFIG_PROTOCOL`。如果网络设备选择只产生 `EFI_MANAGED_NETWORK_PROTOCOL`, 那么网络设备的驱动程序必须实现 `EFI_VLAN_CONFIG_PROTOCOL`。如果为网络设备编写驱动, 除了上述协议外, 还提供无线功能, `EFI_ADAPTER_INFORMATION_PROTOCOL` 必须实现。如果无线驱动程序不提供用户配置功能, 必须实现 `EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL`。如果无线驱动程序是为提供独立的无线 EAP 驱动程序的平台编写的, 则必须实现 `EFI_EAP_PROTOCOL`。
10. 如果为磁盘设备编写驱动程序, 那么必须实现 `EFI_BLOCK_IO_PROTOCOL` 和 `EFI_BLOCK_IO2_PROTOCOL`。此外,对于支持 SPC-4 或 ATA8-ACS 命令集安全命令的磁盘设备,必须实现 `EFI_STORAGE_SECURITY_COMMAND_PROTOCOL`。此外, 对于在主机存储控制器中支持倾斜加密的设备, 必须支持 `EFI_BLOCK_IO_CRYPTO_PROTOCOL`。
11. 如果为磁盘设备编写驱动程序, 那么必须实现 `EFI_BLOCK_IO_PROTOCOL` 和 `EFI_BLOCK_IO2_PROTOCOL`。此外, `EFI_STORAGE_SECURITY_COMMAND_PROTOCOL` 必须用于支持 SPC-4 或 ATA8-ACS 命令集安全命令的磁盘设备。
12. 如果为一个不是面向块的设备编写的驱动程序, 但它可以提供一个类似文件系统的接口, 那么必须实现 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`。
13. 如果为 PCI 根桥编写驱动程序, 那么 `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` 和 `EFI_PCI_IO_PROTOCOL` 必须被实现。
14. 如果为 NVM Express 控制器编写驱动程序, 那么必须实现 `EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL`。
15. 如果为 USB 主机控制器编写驱动程序,那么必须实现 `EFI_USB2_HC_PROTOCOL` 和 `EFI_USB_IO_PROTOCOL`。如果为 USB 主机控制器编写驱动程序, 那么必须实现该。
16. 如果为 SCSI 控制器编写驱动程序, 那么必须实现 `EFI_EXT_SCSI_PASS_THRU_PROTOCOL`。
17. 如果一个驱动程序是数字签名的, 它必须在 PE/COFF 镜像中嵌入数字签名, 如第 1691 页的“嵌入式签名”所述。
18. 如果为一个不是面向块的设备、基于文件系统的设备或控制台设备的启动设备编写驱动程序, 那么必须实现 `EFI_LOAD_FILE2_PROTOCOL`。
19. 如果一个驱动遵循本规范的驱动模型, 并且该驱动想为用户产生警告或错误信息, 那么必须使用 `EFI_DRIVER_HEALTH_PROTOCOL` 来产生这些信息。启动管理器可以选择向用户显示这些信息。
20. 如果一个驱动程序遵循本规范的驱动程序模型, 并且该驱动程序需要执行不属于正常初始化序列的修复操作, 并且该修复操作需要很长一段时间, 那么必须使用 `EFI_DRIVER_HEALTH_PROTOCOL` 来提供修复功能。如果 Boot Manager 检测到一个需要修复操作的启动设备, 那么 Boot Manager 必须使用 `EFI_DRIVER_HEALTH_PROTOCOL` 来执行修复操作。在驱动器执行修复操作时, Boot Manager 可以选择性地显示进度指示器。
21. 如果一个驱动程序遵循本规范的驱动程序模型, 并且该驱动程序要求用户在使用该驱动程序所管理

的启动设备之前进行软件和/或硬件配置的改变，那么必须产生 `EFI_DRIVER_HEALTH_PROTOCOL`。如果 Boot Manager 检测到一个启动设备需要软件和/或硬件配置的改变以使该启动设备可用，那么 Boot Manager 可以选择允许用户进行这些配置的改变。

22. 如果为一个 ATA 控制器编写驱动程序，那么必须实现 `EFI_ATA_PASS_THRU_PROTOCOL`。
23. 如果一个驱动程序遵循本规范的驱动程序模型，并且在为控制器选择最佳驱动程序时，该驱动程序希望以高于总线特定驱动程序覆盖协议的优先级使用，那么 `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL` 必须与 `EFI_DRIVER_BINDING_PROTOCOL` 产生在同一把手上。
24. 如果一个驱动程序支持外部代理或应用程序的固件管理，那么必须使用 `EFI_FIRMWARE_MANAGEMENT_PROTOCOL` 来支持固件管理。
25. 如果一个驱动程序遵循本规范的驱动程序模型，并且一个驱动程序是第 2.5 节中定义的设备驱动程序，它必须通过父级总线驱动程序产生的总线抽象协议执行总线事务。因此，符合 PCI 规范的设备的驱动程序必须使用 `EFI_PCI_IO_PROTOCOL` 进行所有的 PCI 内存空间、PCI I/O、PCI 配置空间和 DMA 操作。
26. 如果为经典蓝牙控制器编写驱动程序，那么必须实现 `EFI_BLUETOOTH_HC_PROTOCOL`、`EFI_BLUETOOTH_IO_PROTOCOL` 和 `EFI_BLUETOOTH_CONFIG_PROTOCOL`，并且可以实现 `EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL`。如果是为 Bluetooth Smart(Bluetooth Low Energy)控制器编写的驱动程序，则必须实现 `EFI_BLUETOOTH_HC_PROTOCOL`、`EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL` 和 `EFI_BLUETOOTH_LE_CONFIG_PROTOCOL`。如果一个驱动程序同时支持蓝牙经典和蓝牙 LE，那么上述两个要求都应该得到满足。
27. 如果为 SD 控制器或 eMMC 控制器编写驱动程序，那么必须实现 `EFI_SD_MMC_PASS_THRU_PROTOCOL`。
28. 如果为 UFS 设备编写驱动程序，那么必须实现 `EFI_UFS_DEVICE_CONFIG_PROTOCOL`。

2.6.4 在其他地方发布的对本规范的扩展

随着时间的推移，本规范已被扩展，包括对新设备和技术的支持。正如该规范的名称所暗示的那样，其定义的初衷是为固件接口创建一个可扩展的基线，而不需要在本规范的主体中包含扩展。

本规范的读者可能会发现，本规范没有处理某个功能或设备类型。这并不一定意味着在声称符合本规范的实现中，没有约定的“标准”方式来支持该特征或设备。有时，其他标准组织发布他们自己的扩展可能更合适，这些扩展旨在与这里的定义协同使用。例如，与等待本规范的修订相比，这样做可以更及时地支持新的功能，或者说，这种支持是由一个在该主题领域具有特殊专长的团体来定义的。因此，建议读者在创建自己的扩展之前，向适当的标准小组询问，以确定是否已经存在适当的扩展出版物。

举例来说，在撰写本规范时，UEFI 论坛知道有一些扩展出版物与本规范兼容，并为其设计。这些扩展包括：

基于 Itanium® 架构的服务器的开发者接口指南：由 DIG64 小组发布和主持（见“基于 Itanium® 架构的服务器的开发者接口指南”标题下的”与 UEFI 相关文件链接”。该文件是一套技术指南，定义了基于 Itanium™ 的服务器的硬件、固件和操作系统的兼容性。

TCG EFI 平台规范：由 Trusted Computing Group 发布和主持（见“TCG EFI 平台规范”标题下的”UEFI 相关文件链接“。这份文件是关于启动 EFI 平台和在该平台上启动操作系统的过程。具体来说，本规范包含了将启动

事件测量到 TPM> PCR 和将启动事件条目添加到事件日志的要求。

TCG EFI 协议规范：由 Trusted Computing Group 发布和主持（参见”[UEFI 相关文件链接](#)”。本文件定义了 EFI 平台上 TPM 的标准接口。

其他的扩展文件可能存在于 UEFI 论坛的视野之外，也可能是在本文件最后一次修订之后创建的。

3 启动管理器

UEFI 启动管理器是一个固件策略引擎，可以通过修改架构上定义的全局 NVRAM 变量来进行配置。启动管理器将尝试按照全局 NVRAM 变量定义的顺序加载 UEFI 驱动程序和 UEFI 应用程序（包括 UEFI 操作系统启动加载器）。平台固件必须使用全局 NVRAM 变量中指定的启动顺序进行正常启动。平台固件可以添加额外的启动选项或从启动顺序列表中删除无效的启动选项。

如果在固件启动过程中发现异常情况，平台固件也可以在启动管理器中实现增值功能 (TODO)。一个增值功能 (TODO) 的例子是，如果第一次加载 UEFI 驱动程序时启动失败，则不加载该驱动程序。另一个例子是，如果在启动过程中发现关键错误，则启动到 OEM 定义的诊断环境。

UEFI 的启动顺序包括以下内容：

- 启动顺序列表是从全局定义的 NVRAM 变量中读取的。对这个变量的修改只保证在下次平台重置后生效。启动顺序列表定义了一个 NVRAM 变量的列表，其中包含了要启动的内容的信息。每个 NVRAM 变量定义了一个可以显示给用户的启动选项的名称。
- 该变量还包含一个指向硬件设备和该硬件设备上包含要加载的 UEFI 镜像的文件的指针。
- 该变量还可能包含操作系统分区和目录的路径，以及其他配置特定的目录。

NVRAM 也可以包含直接传递给 UEFI 镜像的加载选项。平台固件不知道加载选项中包含什么。当更高级别的软件写到全局 NVRAM 变量来设置平台固件的启动策略时，加载选项就被设置了。如果操作系统内核的位置与 UEFI 操作系统加载器的位置不同，该信息可以用来定义操作系统内核的位置。

3.1 固件启动管理器

启动管理器是符合本规范的固件中的一个组件，它决定哪些驱动程序和应用程序应该被显式加载以及何时加载。一旦符合规范的固件被初始化，它就会把控制权交给启动管理器。然后启动管理器负责决定加载什么，以及在做出这样的决定时可能需要与用户进行的任何互动。

启动管理器采取的行动取决于系统类型和系统设计者设置的策略。对于允许安装新启动变量（第 3.4 节）的系统，启动管理器必须自动或根据加载项的请求，初始化至少一个系统控制台，并执行主设备中指示的所有必需的初始化启动目标。对于此类系统，启动管理器还需要遵守在 `BootOrder` 变量中设置的优先级。

特别是，可能的实施方案可能包括任何有关启动的控制台界面，启动选择的综合平台管理，以及可能的对其他内部应用或恢复驱动的了解，这些都可能通过启动管理器集成到系统中。

3.1.1 启动管理器编程

与启动管理器的编程交互是通过全局定义的变量完成的。初始化时，启动管理器读取包含 UEFI 环境变量中所有已发布加载选项的值。通过使用 `SetVariable()` 函数，可以修改包含这些环境变量的数据。此类修改保证在下一次系统启动后生效。但是，启动管理器实现可以选择改进此保证，并让更改对所有后续访问影响启动管理器行为的变量立即生效，而无需任何形式的系统重置。

每个加载选项条目都驻留在 `Boot####`、`Driver####`、`SysPrep####`、`OsRecovery####` 或 `PlatformRecovery####` 变量中，其中 `####` 被一个唯一的选项号码所取代，该号码为可打印的十六进制表示，使用数字 0-9 和字符 A-F 的大写版本 (0000-FFFF)。

`####`，必须始终是四位数，所以小数字必须使用前导零。然后，加载的选项由一个以所需顺序列出的选项号码阵列进行逻辑排序。正常启动时有两个这样的选项排序列表。第一个是 `DriverOrder`，它将 `Driver####` 的加载选项变量排序到它们的加载顺序。第二个是 `BootOrder`，它将 `Boot####` 加载的选项变量排序到它们的加载顺序中。

例如，要增加一个新的启动选项，就要增加一个新的 `Boot####` 变量。然后，新的 `Boot####` 变量的选项号将被添加到 `BootOrder` 列表中，`BootOrder` 变量将被重写。要改变现有的 `Boot####` 的启动选项，只需要重写 `Boot####` 变量。类似的操作也可以用来增加、删除或修改驱动加载列表。

如果通过 `Boot####` 返回的状态是 `EFI_SUCCESS`，平台固件支持启动管理器菜单，如果固件被配置为以交互模式启动，那么启动管理器将停止处理 `BootOrder` 变量并向用户展示启动管理器菜单。如果不满足上述任何一个条件，`BootOrder` 变量中的下一个 `Boot####`，直到所有的可能性都被用完。在这种情况下，必须恢复启动选项（见 3.4 节）。

启动管理器可以对数据库变量进行自动维护。例如，它可以删除未被引用的加载选项变量或任何不能被解析的加载选项变量，它可以重写任何有序的列表，以删除任何没有相应加载选项变量的加载选项。启动管理器也可以根据自己的判断，为管理员提供调用手动维护操作的能力。例子包括选择任何或所有加载选项的顺序，激活或停用加载选项，启动操作系统定义的或平台定义的恢复等。此外，如果平台打算创建 `PlatformRecovery####`，在尝试加载和执行任何 `DriverOrder` 或 `BootOrder` 条目之前，固件必须创建任何和所有 `PlatformRecovery####` 变量（见 3.4.2 节）。在正常操作下，固件不应自动删除当前由 `BootOrder` 或 `BootNext` 变量引用的任何正确形成的 `Boot####` 变量。这种删除应仅限于固件由用户直接交互式操作的情况。

启动管理器可以对数据库变量进行自动维护。例如，它可以删除未被引用的加载选项变量或任何不能被解析的加载选项变量，它可以重写任何有序的列表，以删除任何没有相应加载选项变量的加载选项。启动管理器也可以根据自己的判断，为管理员提供调用手动维护操作的能力。例子包括选择任何或所有加载选项的顺序，激活或停用加载选项，启动操作系统定义的或平台定义的恢复等。此外，如果平台打算创建 `PlatformRecovery####`，在尝试加载和执行任何 `DriverOrder` 或 `BootOrder` 条目之前，固件必须创建任何和所有 `PlatformRecovery####` 变量（见 3.4.2 节）。在正常操作下，固件不应自动删除当前由 `BootOrder` 或 `BootNext` 变量引用的任何正确形成的 `Boot####` 变量。这种删除应限于固件由用户直接互动指导的情况。

`PlatformRecovery####` 的内容表示如果在当前启动期间启动恢复，固件将尝试的最终恢复选项，并且不需要

包含反映重大硬件重新配置等突发事件的条目，或与固件的特定硬件相对应的条目目前还不知道。

当安全启动被启用时，UEFI 启动管理器的行为会受到影响，见第 32.4 节。

3.1.2 加载选项处理

启动管理器需要在启动加载选项条目之前处理驱动加载选项条目。如果 `EFI_OS_INDICATIONS_START_OS_RECOVERY` 位在 `OsIndications` 中被设置，固件应尝试操作系统定义的恢复（见 3.4.1 节）而不是正常的启动处理。如果 `EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY` 位在 `OsIndications` 中被设置，固件应尝试平台定义的恢复（见 3.4.2 节），而不是正常启动处理或处理 `EFI_OS_INDICATIONS_START_OS_RECOVERY` 位。在任何一种情况下，这两个位都应该被清空。

否则，启动管理器还需要启动 `BootNext` 变量所指定的启动选项作为下一次启动的第一个启动选项，而且只在下一次启动时启动。在把控制权转移到 `BootNext` 启动选项之前，启动管理器会删除 `BootNext` 变量。在尝试了 `BootNext` 启动选项之后，会使用正常的 `BootOrder` 列表。为了防止循环，启动管理器在把控制权转移到预选的启动选项之前删除了 `BootNext`。

如果 `BootNext` 和 `BootOrder` 的所有条目都被用尽而没有成功，或者如果固件被指示尝试启动顺序恢复，那么固件必须尝试启动选项恢复（见 3.4 节）。

启动管理器必须调用 `EFI_BOOT_SERVICES.LoadImage()`，它至少支持 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 和 `EFI_LOAD_FILE_PROTOCOL` 来解析加载选项。如果 `LoadImage()` 成功，在调用 `EFI_BOOT_SERVICES.SetWatchdogTimer()` 启动服务之前，启动管理器必须通过使用 `EFI_BOOT_SERVICES.StartImage()` 启用看门狗计时器 5 分钟。如果一个启动选项将控制权返回给启动管理器，启动管理器必须通过额外调用 `SetWatchdogTimer()` 启动服务来禁用看门狗计时器。

如果启动镜像没有通过 `EFI_BOOT_SERVICES.LoadImage()` 加载，那么启动管理器需要检查一个默认的应用程序来启动。在可移动和固定的媒体类型上都会发生搜索默认的应用程序来启动。当任何启动选项中列出的启动镜像的设备路径直接指向 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 设备，并且没有指定要加载的确切文件时，就会发生这种搜索。文件发现方法在第 3.4 节中有解释。`EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 以外的协议的默认媒体启动情况由目标设备路径的 `EFI_LOAD_FILE_PROTOCOL` 处理，不需要由启动管理器处理。

UEFI 启动管理器必须支持从一个简短的设备路径启动，该路径的第一个元素是 `USB WWID`（见表 10-23）或 `USB Class`（见表 10-25）设备路径。对于 `USB WWID`，启动管理器必须使用设备供应商 ID、设备产品 ID 和序列号，并且必须匹配系统中任何包含这些信息的 USB 设备。如果有一个以上的设备与 `USB WWID` 设备路径相匹配，启动管理器将任意选择一个。对于 `USB` 类，启动管理器必须使用供应商 ID、产品 ID、设备类、设备子类和设备协议，并且必须与系统中任何包含这些信息的 USB 设备相匹配。如果任何一个 ID、产品 ID、设备类、设备子类、设备协议中包含所有的 F（`0xFFFF` 或 `0xFF`），这个元素就会被跳过，以进行匹配。如果有一个以上的设备与 `USB` 类设备路径相匹配，启动管理器将任意选择一个。

如果 `Boot Image` 没有通过 `EFI_BOOT_SERVICES.LoadImage()` 加载，那么启动管理器需要检查一个默认的应用程序来启动。在可移动和固定的媒体类型上都会发生搜索默认的应用程序来启动。当任何启动选项中列出

的启动镜像的设备路径直接指向 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 设备，并且没有指定要加载的确切文件时，就会发生这种搜索。文件发现方法在第 3.4 节中有解释。`EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 以外的协议的默认媒体启动情况由目标设备路径的 `EFI_LOAD_FILE_PROTOCOL` 处理，不需要由启动管理器处理。

UEFI 启动管理器必须支持从一个简短的设备路径启动，该路径的第一个元素是 `USB WWID`（见表 10-23）或 `USB Class`（见表 10-25）设备路径。对于 `USB WWID`，启动管理器必须使用设备供应商 ID、设备产品 ID 和序列号，并且必须匹配系统中任何包含这些信息的 USB 设备。如果有一个以上的设备与 `USB WWID` 设备路径相匹配，启动管理器将任意选择一个。对于 `USB` 类，启动管理器必须使用供应商 ID、产品 ID、设备类、设备子类和设备协议，并且必须与系统中任何包含这些信息的 USB 设备相匹配。如果任何一个 ID、产品 ID、设备类、设备子类、设备协议中包含所有的 F (0xFFFF 或 0xFF)，这个元素就会被跳过，以进行匹配。如果有一个以上的设备与 `USB` 类设备路径相匹配，启动管理器将任意选择一个。

启动管理器还必须支持从简短的设备路径启动，该路径的第一个元素是硬盘媒体设备路径（见表 10-49）。启动管理器必须使用硬盘设备路径中的 GUID 或签名和分区号来将其与系统中的设备相匹配。如果硬盘支持 GPT 分区方案，那么硬盘介质设备路径中的 GUID 将与 GUID 分区条目的 `UniquePartitionGuid` 字段进行比较（见表 5-6）。如果硬盘支持 PC-AT MBR 方案，则将硬盘介质设备路径中的签名与传统主启动记录中的 `UniqueMBRSignature` 进行比较（见表 5-1）。如果签名匹配，那么分区号也必须匹配。硬盘设备路径可以被附加到匹配的硬件设备路径上，然后可以使用正常的启动行为。如果有一个以上的设备与硬盘设备路径相匹配，启动管理器将任意选择一个。因此操作系统必须保证硬盘上签名的唯一性，以保证确定的启动行为。

启动管理器还必须支持从以第一个元素为硬盘驱动器媒体设备路径开始的短格式设备路径启动（参见表 10-49）。启动管理器必须使用硬盘驱动器设备路径中的 GUID 或签名和分区号，以将其与系统中的设备相匹配。如果驱动器支持 GPT 分区方案，则将硬盘驱动器媒体设备路径中的 GUID 与 GUID 分区条目的 `UniquePartitionGuid` 字段进行比较（参见表 5-6）。如果驱动器支持 PC-AT MBR 方案，则将硬盘驱动器媒体设备路径中的签名与 `Legacy Master Boot Record` 中的 `UniqueMBRSignature` 进行比较（参见表 5-1）。如果进行了签名匹配，则分区号也必须匹配。可以将硬盘驱动器设备路径附加到匹配的硬件设备路径，然后可以使用正常的启动行为。如果多个设备与硬盘驱动器设备路径匹配，则启动管理器将任意选择一个。因此，操作系统必须确保硬盘驱动器上签名的唯一性，以保证确定的启动行为。

启动管理器还必须支持从一个简短的设备路径启动，这个路径的第一个元素是文件路径媒体设备路径（见表 10-52）。当启动管理器试图启动一个短格式的文件路径媒体设备路径时，它将列举所有可移动媒体设备，然后是所有固定媒体设备，为每个设备创建启动选项。启动选项 `FilePathList[0]` 是通过将短格式的 `File Path Media Device Path` 附加到一个媒体的设备路径上而构建的。每组内的顺序是未定义的。这些新的启动选项不能被保存到非易失性存储中，也不能被添加到 `BootOrder` 中。然后启动管理器将尝试从每个启动选项启动。如果一个设备不支持 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`，但是支持 `EFI_BLOCK_IO_PROTOCOL` 协议，那么 `EFI Boot Service ConnectController` 必须在 `DriverImageHandle` 和 `RemainingDevicePath` 设置为 `NULL`，并且递归标志设置为 `TRUE` 的情况下为这个设备调用。然后，固件将尝试使用上述算法从产生的任何子手柄启动。

启动管理器还必须支持从一个短形式的设备路径启动，该路径的第一个元素是 URI 设备路径（见表 10-40）。当启动管理器试图启动一个短形式的 URI 设备路径时，它可以尝试连接任何设备，这些设备会产生一个包括 URI 设备路径节点的设备路径协议，直到它匹配到一个设备，或者无法匹配任何设备。启动管理器将枚举所

有的 `LoadFile` 协议实例，并在匹配过程中调用 `LoadFile` 协议，并将 `FilePath` 设置为短格式设备路径

3.1.3 加载选项

每个加载选项变量包含一个 `EFI_LOAD_OPTION` 描述符，它是一个由可变长度的字段组成的字节打包的缓冲区。

```

1 typedef struct _EFI_LOAD_OPTION {
2     UINT32                                     Attributes; UINT16
3     FilePathListLength;                      Description[];
4     // CHAR16                                FilePathList[];
5     // EFI_DEVICE_PATH_PROTOCOL                OptionalData[];
6 } EFI_LOAD_OPTION;

```

参数

- **Attributes**: 此加载选项条目的属性。所有未使用的位必须为零，并由 UEFI 规范为未来的发展保留。参见“相关定义”。
- **FilePathListLength**: 以字节为单位的 `FilePathList` 的长度。`OptionalData` 从 `EFI_LOAD_OPTION` 描述符的偏移量 `sizeof(UINT32)+ sizeof(UINT16)+ StrSize(Description)+ FilePathListLength` 开始。
- **Description**: 加载选项的用户可读描述。这个字段以一个空字符结束。
- **FilePathList**: 一个 UEFI 设备路径的打包数组。数组的第一个元素是一个设备路径，描述了这个加载选项的设备和镜像的位置。`FilePathList[0]` 是特定于设备类型的。其他设备路径可以选择存在于 `FilePathList` 中，但它们的使用是 OSV 特定的。数组中的每个元素都是可变长度的，并在设备路径的末端结构处结束。因为 `Description` 的大小是任意的，这个数据结构不能保证在自然边界上对齐。这个数据结构在使用前可能要被复制到一个对齐的自然边界上。
- **OptionalData**: 加载选项描述符中的剩余字节是一个二进制数据缓冲区，它被传递给加载的图像。如果该字段的长度为 0 字节，则将传递一个 `NULL` 指针给加载的图像。`OptionalData` 中的字节数可以通过从 `EFI_LOAD_OPTION` 的总大小（字节）中减去 `OptionalData` 的起始偏移来计算。

相关定义

```

1 //*****
2 // Attributes
3 //*****
4 #define LOAD_OPTION_ACTIVE          0x00000001
5 #define LOAD_OPTION_FORCE_RECONNECT 0x00000002
6 #define LOAD_OPTION_HIDDEN          0x00000008
7 #define LOAD_OPTION_CATEGORY        0x00001F00
8 #define LOAD_OPTION_CATEGORY_BOOT   0x00000000
9 #define LOAD_OPTION_CATEGORY_APP    0x00000100// All values 0x00000200-0x00001F00 are
      reserved

```

调用 `SetVariable()` 会创建一个加载选项。加载选项的大小与创建该变量的 `SetVariable()` 调用的 `DataSize` 参数的大小相同。当创建一个新的加载选项时，所有未定义的属性位必须写成 0。当更新一个加载选项时，所有未定义的属性位必须被保留下来。

如果一个加载选项被标记为 `LOAD_OPTION_ACTIVE`，那么启动管理器将尝试使用加载选项中的设备路径信息自动启动。这提供了一个简单的方法来禁用或启用加载选项，而不需要删除和重新添加它们。

如果任何 `Driver####` 加载选项被标记为 `LOAD_OPTION_FORCE_RECONNECT`，那么系统中所有的 UEFI 驱动将被断开连接，并在最后一个 `Driver####` 加载选项被处理后重新连接上。这允许用 `Driver####` 加载选项加载的 UEFI 驱动覆盖在执行 UEFI 启动管理器之前加载的 UEFI 驱动。

在 `Driver####` 加载选项中，`FilePathList[0]` 指示的可执行文件必须是 `EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER` 或 `EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER` 类型，否则指示的可执行文件将不会被输入初始化。

在 `SysPrep###`、`Boot####` 或 `OsRecovery####` 加载选项中，`FilePathList[0]` 指示的可执行文件必须是 `EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION` 类型，否则指示的可执行文件将不会被输入。

`LOAD_OPTION_CATEGORY` 是 `Attributes` 的一个子字段，它为启动管理器提供细节，描述它应该如何分组 `Boot####` 加载选项。这个字段对于 `Driver####`、`SysPrep####`，或者 `OsRecovery####` 形式的变量是被忽略的。

将 `LOAD_OPTION_CATEGORY` 设置为 `LOAD_OPTION_CATEGORY_BOOT` 的 `Boot####` 加载选项是正常启动处理的一部分。

将 `LOAD_OPTION_CATEGORY` 设置为 `LOAD_OPTION_CATEGORY_APP` 的 `Boot####` 加载选项是可执行文件，它不是正常启动处理的一部分，但是如果提供了启动菜单，可以选择执行，或者通过热键。详见 3.1.6 节。

带有保留类别值的启动选项，将被启动管理器忽略。

如果任何 `Boot####` 的加载选项被标记为 `LOAD_OPTION_HIDDEN`，那么这个加载选项就不会出现在启动管理器提供的用于加载选项选择的菜单中（如果有的话）。

3.1.4 启动管理器的功能

启动管理器可以通过全局变量 `BootOptionSupport` 来报告它的能力。如果全局变量不存在，那么安装程序或应用程序必须像返回值为 0 一样行事。

```
1 #define EFI_BOOT_OPTION_SUPPORT_KEY      0x00000001
2 #define EFI_BOOT_OPTION_SUPPORT_APP       0x00000002
3 #define EFI_BOOT_OPTION_SUPPORT_SYSPREP   0x00000010
4 #define EFI_BOOT_OPTION_SUPPORT_COUNT     0x000000300
```

如果 `EFI_BOOT_OPTION_SUPPORT_KEY` 被设置，那么启动管理器支持使用按键启动 `Boot####` 加载选项。如果 `EFI_BOOT_OPTION_SUPPORT_APP` 被设置，那么启动管理器支持使用 `LOAD_OPTION_CATEGORY_APP` 启动选项。如

果 `EFI_BOOT_OPTION_SUPPORT_SYSPREP` 被设置，那么引导管理器支持 `SysPrep####` 形式的引导选项。

`EFI_BOOT_OPTION_SUPPORT_COUNT` 中指定的值描述了启动管理器在 `EFI_KEY_OPTION.KeyData.InputKeyCount` 中支持的最大按键数。这个值只有在 `EFI_BOOT_OPTION_SUPPORT_KEY` 被设置时才有效。指定了更多按键的按键序列会被忽略。

3.1.5 启动 Boot##### 应用程序

引导管理器可以为应用程序支持一个单独的 `Boot#####` 加载选项类别。启动管理器通过在 `BootOptionSupport` 全局变量中设置 `EFI_BOOT_OPTION_SUPPORT_APP` 来表明它支持这个单独的类别。

当一个应用程序的 `Boot#####` 选项被添加到 `BootOrder` 中时，安装者应该清除 `LOAD_OPTION_ACTIVE`，这样引导管理器就不会试图自动“启动”这个应用程序。如果启动管理器指出它支持一个单独的应用程序类别，如上所述，安装者应该设置 `LOAD_OPTION_CATEGORY_APP`。如果不是，它应该设置 `LOAD_OPTION_CATEGORY_BOOT`。

3.1.6 使用热键启动 Boot##### 加载选项

启动管理器可能支持使用一个特殊的按键来启动 `Boot#####` 加载选项。如果是这样，引导管理器通过在 `BootOptionSupport` 全局变量中设置 `EFI_BOOT_OPTION_SUPPORT_KEY` 来报告这种能力。

一个支持按键启动的启动管理器从控制台读取当前的按键信息。然后，如果有一个按键被按下，它将返回的按键与零个或多个 `Key####` 全局变量进行比较。如果找到了匹配，它就会验证指定的 `Boot#####` 加载选项是否有效，如果有效，就会尝试立即启动它。`Key####` 中的 `####` 是一个可打印的十六进制数字（‘0’-‘9’，‘A’-‘F’），前面是零。检查 `Key####` 变量的顺序是根据具体实施情况而定的。

当指定的热键与内部引导管理器功能重叠时，引导管理器可以忽略 `Key####` 变量。建议启动管理器删除这些键。`Key####` 变量有以下格式。

结构体原型

```
1 typedef struct _EFI_KEY_OPTION {
2     EFI_BOOT_KEY_DATA    KeyData;
3     UINT32              BootOptionCrc;
4     UINT16              BootOption;
5     // EFI_INPUT_KEY     Keys[];
6 } EFI_KEY_OPTION;
```

参数

- **KeyData**: 指定关于如何处理钥匙的选项。`EFI_BOOT_KEY_DATA` 类型在下面“相关定义”中定义。
- **BootOptionCrc**: 应该与 `BootOption` 所指的整个 `EFI_LOAD_OPTION` 的 CRC-32 匹配。如果 CRC-32 与此值不匹配，那么这个关键选项将被忽略。

- **BootOption:** `Boot####`, 如果按下这个键, 并且启动选项处于激活状态 (`LOAD_OPTION_ACTIVE` 被设置), 将被调用。
- **Keys:** 与 `EFI_SIMPLE_TEXT_INPUT` 和 `EFI_SIMPLE_TEXT_INPUT_EX` 协议返回的密钥代码进行比较。键码的数量 (0-3) 由 `KeyOptions` 中的 `EFI_KEY_CODE_COUNT` 字段指定。

相关定义

```

1 typedef union {
2   struct {
3     UINT32 Revision : 8;
4     UINT32 ShiftPressed : 1;
5     UINT32 ControlPressed : 1;
6     UINT32 AltPressed : 1;
7     UINT32 LogoPressed : 1;
8     UINT32 MenuPressed : 1;
9     UINT32 SysReqPressed : 1;
10    UINT32 Reserved : 16;
11    UINT32 InputKeyCount : 2;
12  }Options;
13  UINT32 PackedValue;
14 } EFI_BOOT_KEY_DATA;

```

- **Revision:** 表示 `EFI_KEY_OPTION` 结构的修订。这个修订级别应该是 0。
- **ShiftPressed:** 必须按下左或右的 Shift 键 (1) 或不按下 (0)。
- **ControlPressed:** 左边或右边的 Control 键必须被按下 (1) 或不能被按下 (0)。
- **AltPressed:** 左边或右边的 Alt 键必须被按下 (1) 或不能被按下 (0)。
- **LogoPressed:** 左边或右边的徽标键必须被按下 (1) 或不能被按下 (0)。
- **MenuPressed:** 菜单键必须按下 (1) 或不按下 (0)。
- **SysReqPressed:** SysReq 键必须被按下 (1) 或不被按下 (0)。
- **InputKeyCount:** 指定 `EFI_KEY_OPTION.Keys` 中的实际条目数, 从 0-3。如果为零, 那么只考虑移位状态。如果多于一个, 那么只有当所有指定的键都以相同的移位状态按下时, 启动选项才会被启动。例 #1: ALT 是热键。`KeyData.PackedValue = 0x00000400`。例 #2: CTRL-ALT-P-R。`KeyData.PackedValue = 0x80000600`。例 #3: CTRL-F1。`KeyData.PackedValue = 0x40000200`。

3.1.7 必要的系统准备应用

`SysPrep####` 形式的加载选项旨在指定一个 UEFI 应用程序, 该应用程序需要执行, 以便在处理任何 `Boot####` 变量之前完成系统准备。`SysPrep####` 应用程序的执行顺序由变量 `SysPrepOrder` 的内容决定, 其方式直接类似于 `BootOrder` 对 `Boot####` 选项的排序。

平台需要检查在 `SysPrepOrder` 中引用的所有 `SysPrep####` 变量。如果属性位 `LOAD_OPTION_ACTIVE` 被设置,

并且 `FilePathList[0]` 所引用的应用程序存在, 那么由此确定的 UEFI 应用程序必须按照它们在 `SysPrepOrder` 中出现的顺序加载和启动, 并且在启动任何 `Boot####` 类型的加载选项之前。

当启动时, 平台需要为由 `SysPrep####` 加载的应用程序提供相同的服务, 如控制台和网络, 就像通常在启动时提供给由 `Boot####` 变量引用的应用程序一样。`SysPrep####` 应用程序必须退出, 不得调用 `ExitBootServices()`。对退出时返回的任何 `Error Code` 的处理是根据系统策略进行的, 不一定会改变对以下启动选项的处理。任何由 `SysPrep####` 启动选项支持的、需要保持驻留的功能的驱动部分都应该通过使用 `Driver####` 变量加载。

`LOAD_OPTION_FORCE_RECONNECT` 属性选项对于 `SysPrep####` 变量来说是被忽略的, 如果这样启动的应用程序执行了一些增加可用硬件或驱动程序的操作, 系统准备应用程序本身应利用对 `ConnectController()` 或 `DisconnectController()` 的适当调用来修改驱动程序和硬件之间的连接。

在所有 `SysPrep####` 变量启动和退出后, 平台应通知 `EFI_EVENT_GROUP_READY_TO_BOOT` 和 `EFI_EVENT_GROUP_AFTER_READY_TO_BOOT` 事件组, 并开始按照 `BootOrder` 定义的顺序评估属性设置为 `LOAD_OPTION_CATEGORY_BOOT` 的 `Boot####` 变量。在 `EFI_EVENT_GROUP_AFTER_READY_TO_BOOT` 事件组处理完成之前, 不应评估标记为 `LOAD_OPTION_CATEGORY_BOOT` 的变量的 `FilePathList`。

3.2 启动管理器策略协议

3.2.1 EFI_BOOT_MANAGER_POLICY_PROTOCOL

3.2.1.1 摘要

这个协议被 EFI 应用程序用来请求 UEFI 启动管理器使用平台策略连接设备。

3.2.1.2 GUID

```
1 #define EFI_BOOT_MANAGER_POLICY_PROTOCOL_GUID \
2 { 0xFEDF8E0C, 0xE147, 0x11E3, \
3 { 0x99, 0x03, 0xB8, 0xE8, 0x56, 0x2C, 0xBA, 0xFA } }
```

3.2.1.3 协议接口结构

```
1 typedef struct _EFI_BOOT_MANAGER_POLICY_PROTOCOL EFI_BOOT_MANAGER_POLICY_PROTOCOL; struct \
2     _EFI_BOOT_MANAGER_POLICY_PROTOCOL { UINT64 \
3         Revision; \
4     EFI_BOOT_MANAGER_POLICY_CONNECT_DEVICE_PATH ConnectDevicePath; \
5     EFI_BOOT_MANAGER_POLICY_CONNECT_DEVICE_CLASS ConnectDeviceClass; \
6 }; \
7 \
8 ConnectDevicePath Connect a Device Path following the platforms EFI Boot Manager policy.
```

6 ConnectDeviceClassConnect a class of devices, named by EFI_GUID, following the platforms UEFI Boot Manager policy.

3.2.1.4 描述

EFI_BOOT_MANAGER_POLICY_PROTOCOL 是由平台固件产生的，以暴露启动管理器策略和平台特定的 EFI_BOOT_SERVICES.ConnectController() 行为。

3.2.1.5 相关定义

```
1 #define EFI_BOOT_MANAGER_POLICY_PROTOCOL_REVISION 0x00010000
```

3.2.2 EFI_BOOT_MANAGER_POLICY_PROTOCOL.ConnectDevicePath()

3.2.2.1 摘要

按照平台的 EFI 启动管理器策略，连接一个设备路径。

3.2.2.2 原型

```
1 typedef EFI_STATUS(EFIAPI *EFI_BOOT_MANAGER_POLICY_CONNECT_DEVICE_PATH)(  
2     IN EFI_BOOT_MANAGER_POLICY_PROTOCOL    *This,  
3     IN EFI_DEVICE_PATH                    *DevicePath,  
4     IN BOOLEAN                           Recursive  
5 );
```

3.2.2.3 参数

- This:指向 EFI_BOOT_MANAGER_POLICY_PROTOCOL 实例的指针。类型为上面定义的 EFI_BOOT_MANAGER_POLICY_PROTOCOL。
- DevicePath: 指向要连接的 EFI 设备路径的起点。如果 DevicePath 为 NULL, 那么系统中的所有控制器都将使用平台的 EFI 启动管理器策略进行连接。
- Recursive: 如果是 TRUE, 那么 ConnectController() 将被递归调用，直到 DevicePath 指定的控制器下面的整个控制器树都被创建。如果是 FALSE, 那么控制器树只扩展一级。如果 DevicePath 是 NULL, 那么递归将被忽略。

3.2.2.4 描述

`ConnectDevicePath()` 函数允许调用者使用与 EFI Boot Manager 相同的策略连接 DevicePath。如果递归为 true，那么 `ConnectController()` 将被递归调用，直到 DevicePath 指定的控制器下面的整个控制器树都被创建。如果递归为 FALSE，那么控制器树只扩展一级。如果 DevicePath 是 NULL，那么递归将被忽略。

3.2.2.5 返回的状态代码

状态码	描述
<code>EFI_SUCCESS</code>	设备路径已被连接
<code>EFI_NOT_FOUND</code>	未找到设备路径
<code>EFI_NOT_FOUND</code>	没有驱动程序被连接到 DevicePath
<code>EFI_SECURITY_VIOLATION</code>	用户没有权限启动 UEFI 设备驱动程序设备路径
<code>EFI_UNSUPPORTED</code>	当前的 TPL 不是 TPL_APPLICATION

3.2.3 EFI_BOOT_MANAGER_POLICY_PROTOCOL.ConnectDeviceClass()

3.2.3.1 摘要

使用平台启动管理器策略连接一类设备。

3.2.3.2 原型

```

1 typedef EFI_STATUS(EFIAPI *EFI_BOOT_MANAGER_POLICY_CONNECT_DEVICE_CLASS)(
2     IN EFI_BOOT_MANAGER_POLICY_PROTOCOL *This,
3     IN EFI_GUID                      *Class
4 );

```

3.2.3.3 参数

- This: 指向 EFI_BOOT_MANAGER_POLICY_PROTOCOL 实例的一个指针。上面定义了 EFI_BOOT_MANAGER_类型
- Class: 一个指向 EFI_GUID 的指针，代表将使用 Boot Manager 的平台策略连接的设备类别

3.2.3.4 描述

`ConnectDeviceClass()` 函数允许调用者请求引导管理器连接一个设备类别。

如果 Class 是 EFI_BOOT_MANAGER_POLICY_CONSOLE_GUID, 那么 Boot Manager 将使用平台策略来连接控制台。一些平台在尝试快速启动时可能会限制连接的控制台数量, 调用 Class 值为 EFI_BOOT_MANAGER_POLICY_CONSOLE 的 ConnectDeviceClass() 必须连接遵循引导管理器平台策略的控制台集合。并且 EFI_SIMPLE_TEXT_INPUT_PROTOCOL、EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 和 EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL 为在连接的手柄上产生。开机管理器可以根据平台策略限制哪些控制台可以被连接, 例如, 安全策略可能要求不连接某个特定的控制台。如果 Class 是 EFI_BOOT_MANAGER_POLICY_NETWORK_GUID, 那么启动管理器将在一个或多个手柄上连接平台支持的 UEFI 通用网络应用的协议。与 UEFI 通用网络应用相关的协议在第 2.6.2 节中定义, 列表中的第 7 项。如果有一个以上的网络控制器, 平台将根据平台策略连接一个、多个或所有的网络。连接 UEFI 网络协议, 如 EFI_DHCP4_PROTOCOL, 并不在网络上建立连接。调用 ConnectDeviceClass() 的 UEFI 通用网络应用程序可能需要使用发布的协议来建立网络连接。启动管理器可以选择有一个策略来建立网络连接。如果 Class 是 EFI_BOOT_MANAGER_POLICY_CONNECT_ALL_GUID, 那么 Boot Manager 将使用 UEFI Boot Service EFI_BOOT_SERVICES.ConnectController() 连接所有 UEFI 驱动程序。如果 Boot Manager 有与连接所有 UEFI 驱动相关的策略, 将使用这个策略

一个平台也可以定义平台特定的 Class 值, 因为正确生成的 EFI_GUID 绝不会与本规范冲突。

3.2.3.5 相关定义

```

1 #define EFI_BOOT_MANAGER_POLICY_CONSOLE_GUID \
2 { 0xCAB0E94C, 0xE15F, 0x11E3, \
3 { 0x91, 0x8D, 0xB8, 0xE8, 0x56, 0x2C, 0xBA, 0xFA } } \
4 #define EFI_BOOT_MANAGER_POLICY_NETWORK_GUID \
5 { 0xD04159DC, 0xE15F, 0x11E3, \
6 { 0xB2, 0x61, 0xB8, 0xE8, 0x56, 0x2C, 0xBA, 0xFA } } \
7 #define EFI_BOOT_MANAGER_POLICY_CONNECT_ALL_GUID \
8 { 0x113B2126, 0xFC8A, 0x11E3, \
9 { 0xBD, 0x6C, 0xB8, 0xE8, 0x56, 0x2C, 0xBA, 0xFA } }

```

3.2.3.6 返回的状态代码

状态码	描述
EFI_SUCCESS	该类至少有一个设备被连接
EFI_DEVICE_ERROR	由于一个错误, 设备没有被连接
EFI_NOT_FOUND	该类平台不支持
EFI_UNSUPPORTED	当前的 TPL 不是 TPL_APPLICATION

3.3 全局定义的变量

本节定义了一组具有架构定义含义的变量。除了定义的数据内容外，每个这样的变量都有一个架构上定义的属性，表明当数据变量可以被访问。属性为 NV 的变量是非易失性的。这意味着它们的值在复位和电源循环中是持久的。任何没有这个属性的环境变量的值都会在系统断电后丢失，而且固件保留内存的状态也不会被保留下。具有 BS 属性的变量仅在 `EFI_BOOT_SERVICES.ExitBootServices()` 被调用之前可用。这意味着这些环境变量只能在预启动环境中被检索或修改。它们对操作系统是不可见的。属性为 RT 的环境变量在 `ExitBootServices()` 被调用之前和之后都可用。这种类型的环境变量可以在预启动环境和操作系统中被检索和修改。属性为 AT 的变量是具有第 8.2.1 节中定义的基于时间的验证写入权限的变量。所有架构定义的变量都使用 `EFI_GLOBAL_VARIABLE`。

```

1 EFI_GLOBAL_VARIABLE VendorGuid;
2 # define EFI_GLOBAL_VARIABLE \{0x8BE4DF61,0x93CA,0x11d2,\ {0xAA,0x0D,0x00,0xE0,0x98,0x03
,0x2B,0x8C}\}

```

为了防止与未来可能的全局定义的变量发生名称冲突，这里没有定义的其他内部固件数据变量必须用一个唯一的 `VendorGuid` 来保存，而不是 `EFI_GLOBAL_VARIABLE` 或 UEFI 规范定义的任何其他 GUID。只有当 UEFI 规范中记录了这些变量时，实施才必须允许用 UEFI 规范定义的 `VendorGuid` 创建这些变量。

变量名	属性	描述
AuditMode	BS, RT	系统是否在审核模式下运行 (1) 或 (0)。所有其他值均保留。应被视为只读，除非 <code>DeployedMode</code> 为 0。调用 <code>ExitBootServices()</code> 后始终变为只读。
Boot####	NV, BS, RT	引导加载选项。#### 是打印的十六进制值。十六进制值中不包含 0x 或 h
BootCurrent	BS, RT	引导加载选项。#### 是打印的十六进制值。十六进制值中不包含 0x 或 h
BootNext	NV, BS, RT	仅用于下次启动的启动选项。
BootOrder	NV, BS, RT	有序引导选项加载列表。
BootOptionSupport	BS, RT	引导管理器支持的引导选项类型。应视为只读。
ConIn	NV, BS, RT	默认输入控制台的设备路径。

变量名	属性	描述
ConInDev	BS, RT	所有可能的控制台输入设备的设备路径。
ConOut	NV, BS, RT	默认输出控制台的设备路径。
ConOutDev	BS, RT	所有可能的控制台输出设备的设备路径。
dbDefault	BS, RT	OEM 的默认安全启动签名存储。应被视为只读。
dbrDefault	BS, RT	OEM 的默认操作系统恢复签名存储。应视为只读。
dbtDefault	BS, RT	OEM 的默认安全启动时间戳签名存储。应被视为只读。
dbxDefault	BS, RT	OEM 的默认安全启动黑名单签名存储。应视为只读。
DeployedMode	BS, RT	系统是否在部署模式下运行 (1) 或 (0)。所有其他值均保留。当其值为 1 时应被视为只读。在调用 ExitBootServices() 后始终变为只读。
Driver####	NV, BS, RT	驱动程序加载选项。#### 是打印的十六进制值。
DriverOrder	NV, BS, RT	bbbbbb
ErrOut	NV, BS, RT	默认错误输出设备的设备路径。
ErrOutDev	BS, RT	所有可能的错误输出设备的设备路径。
HwErrRecSupport	NV, BS, RT	标识平台实现的硬件错误记录持久性支持级别。此变量仅由固件修改并且对操作系统是只读的。
KEK	NV, BS, RT, AT	密钥交换密钥签名数据库。
KEKDefault	BS, RT	OEM 的默认密钥交换密钥签名数据库。应被视为只读。
Key####	NV, BS, RT	描述热键与 Boot#### 加载选项的关系。

变量名	属性	描述
Lang	NV, BS, RT	系统配置的语言代码。此值已弃用。
LangCodes	BS, RT	固件所支持的语言代码。此值已被废弃。
OsIndications	NV, BS, RT	允许操作系统请求固件启用某些功能并执行某些操作。
OsIndicationsSupported	BS, RT	允许固件向操作系统指示支持的功能和操作
OsRecoveryOrder	BS, RT, NV, AT	操作系统指定的恢复选项
PK	NV, BS, RT, AT	公共平台密钥。
PKDefault	BS, RT	OEM 的默认公共平台密钥。应被视为只读
PlatformLangCodes	BS, RT	固件支持的语言代码。
PlatformLang	NV, BS, RT	系统配置的语言代码。
PlatformRecovery####	BS, RT	平台指定的恢复选项。这些变量仅由固件修改并且对操作系统是只读的。
SignatureSupport	BS, RT	表示平台固件支持的签名类型的 GUID 数组。应被视为只读
SecureBoot	BS, RT	平台固件是否在安全启动模式下运行 (1) 或 (0)。所有其他值均保留。应被视为只读。
SetupMode	BS, RT	系统是否应要求对安全启动策略变量的 SetVariable() 请求进行身份验证 (0) 或 (1)。应视为只读。 当 SetupMode==1, AuditMode==0, DeployedMode==0 时, 系统处于“设置模式”

变量名	属性	描述
SysPrep####	NV, BS, RT	包含 EFI_LOAD_OPTION 描述符的 System Prep 应用程序加载选项。#### 是打印的十六进制值。
SysPrepOrder	NV, BS, RT	有序的系统准备应用程序加载选项列表。
Timeout	NV, BS, RT	在启动默认引导选择之前，固件的引导管理器超时（以秒为单位）
VendorKeys	BS, RT	系统是否配置为仅使用供应商提供的密钥。应视为只读。

`PlatformLangCodes` 变量包含一个以 null 结尾的 ASCII 字符串，表示固件可以支持的语言代码。在初始化时，固体会计算支持的语言并创建此数据变量。由于固件在每次初始化时都会创建此值，因此其内容不会存储在非易失性存储器中。该值被认为是只读的。`PlatformLangCodes` 以 Native RFC 4646 格式指定。请参阅附录 M。`LangCodes` 已弃用，可能会提供以实现向后兼容性。

`PlatformLang` 变量包含机器已配置的以 null 结尾的 ASCII 字符串语言代码。该值可以更改为 `PlatformLangCodes` 支持的任何值。如果在预引导环境中进行此更改，则更改将立即生效。如果此更改是在操作系统运行时进行的，则更改要到下次启动时才会生效。如果语言代码设置为不受支持的值，固件将在初始化时选择受支持的默认值并将 `PlatformLang` 设置为受支持的值。`PlatformLang` 以 Native RFC 4646 数组格式指定。请参阅附录 M。`Lang` 已弃用，可能会提供以实现向后兼容性。

`Lang` 已被弃用。如果平台支持这个变量，它必须以适当的形式将 `Lang` 变量中的任何更改映射到 `PlatformLang`。

`Langcodes` 已被弃用。如果平台支持此变量，则它必须以适当的格式将 `Langcodes` 变量中的任何更改映射到 `PlatformLang`。

`Timeout` 变量包含一个二进制 `UINT16`，它提供固件在启动原始默认引导选择之前将等待的秒数。值 0 表示默认引导选择将在引导时立即启动。如果该值不存在，或包含 `0xFFFF` 的值，则固件将在引导前等待用户输入。这意味着固件不会自动启动默认启动选择。

`ConIn`、`ConOut` 和 `ErrOut` 变量分别包含一个 `EFI_DEVICE_PATH_PROTOCOL` 描述符，定义了启动时使用的默认设备。在预启动环境中对这些值的改变会立即生效。在操作系统运行时对这些值的改变要到下一次启动时才会生效。如果固件不能解决设备路径，允许它根据需要自动替换这些值，为系统提供一个控制台。如果设备路径以 USB 类设备路径开始（见表 10-25），那么任何符合设备路径的输入或输出设备都必须作为控制台使用，如果它被固件支持的话。

`ConInDev`、`ConOutDev` 和 `ErrOutDev` 变量均包含一个 `EFI_DEVICE_PATH_PROTOCOL` 描述符，该描述符定义了所

有可能在引导时使用的默认设备。这些变量是易变的，并且在每次启动时动态设置。`ConIn`、`ConOut` 和 `ErrOut` 始终是 `ConInDev`、`ConOutDev` 和 `ErrOutDev` 的真子集。

每个 `Boot####` 变量都包含一个 `EFI_LOAD_OPTION`。每个 `Boot####` 变量都是名称“Boot”附加一个唯一的四位十六进制数字。例如，`Boot0001`、`Boot0002`、`Boot0A02` 等。

`OsRecoveryOrder` 变量包含一个 `EFI_GUID` 结构的数组。每个 `EFI_GUID` 结构为包含操作系统定义的恢复条目的变量指定一个命名空间（见第 3.4.1 节）。对这个变量的写访问由安全密钥数据库 `dbr` 控制（见第 8.2.1 节）。

`PlatformRecovery####` 变量与 `Boot####` 变量共享相同的结构。这些变量是在系统执行引导选项恢复时处理的。

`BootOrder` 变量包含一个 `UINT16` 的数组，构成了 `Boot####` 选项的有序列表。数组中的第一个元素是第一个逻辑启动选项的值，第二个元素是第二个逻辑启动选项的值，等等。`BootOrder` 顺序列表被固件的引导管理器作为默认的引导顺序。

`BootNext` 变量是单个 `UINT16`，它定义了 `Boot####` 选项，该选项将在下次启动时首先尝试。在尝试使用 `BootNext` 引导选项后，将使用正常的 `BootOrder` 列表。为防止循环，引导管理器在将控制转移到预选引导选项之前删除此变量。

`BootCurrent` 变量是一个单一的 `UINT16`，它定义了当前启动时选择的 `Boot####` 选项。

`BootOptionSupport` 变量是一个 `UINT32`，它定义了引导管理器支持的引导选项类型。

每个 `Driver####` 变量都包含一个 `EFI_LOAD_OPTION`。每个加载选项变量都附加一个唯一的编号，例如 `Driver0001`、`Driver0002` 等。

`DriverOrder` 变量包含一个 `UINT16` 的数组，构成了 `Driver####` 变量的有序列表。数组中的第一个元素是第一个逻辑驱动加载选项的值，第二个元素是第二个逻辑驱动加载选项的值，等等。该 `DriverOrder` 列表被固件的启动管理器用作 UEFI 驱动程序的默认加载顺序，它应该明确地加载这些驱动程序。

`Key####` 变量将按键与单个引导选项相关联。每个 `Key####` 变量都是名称“Key”附加一个唯一的四位十六进制数字。例如，`Key0001`、`Key0002`、`Key00A0` 等。

`HwErrRecSupport` 变量包含一个二进制 `UINT16`，它提供对由平台实现的硬件错误记录持久性（请参阅第 8.2.4 节）的支持级别。如果该值不存在，则平台不实现对硬件错误记录持久性的支持。零值表示平台不支持硬件错误记录持久性。值为 1 表示平台实现了第 8.2.4 节中定义的硬件错误记录持久性。固件初始化这个变量。所有其他值保留供将来使用。

`SetupMode` 变量是一个 8 位无符号整数，它定义系统是否应该在 `SetVariable()` 请求安全引导策略变量时要求身份验证 (0) 或不需要 (1)。安全启动策略变量包括：

- 全局变量 `PK`、`KEK` 和 `OsRecoveryOrder`
- 所有 `VendorGuid` 下名为 `OsRecovery####` 的所有变量
- 具有 `VendorGuid` `EFI_IMAGE_SECURITY_DATABASE_GUID` 的所有变量。

必须使用 `EFI_VARIABLE_AUTHENTICATION_2` 结构创建安全启动策略变量。

`AuditMode` 变量是一个 8 位无符号整数，它定义系统当前是否在审计模式下运行。

`DeployedMode` 变量是一个 8 位无符号整数，定义系统当前是否在部署模式下运行。

`KEK` 变量包含当前密钥交换密钥数据库。

`PK` 变量包含当前的平台密钥。

`VendorKeys` 变量是一个 8 位无符号整数，用于定义安全引导策略变量是否已被平台供应商或供应商提供的密钥持有者以外的任何人修改。值为 0 表示平台供应商或供应商提供的密钥的持有者以外的其他人修改了安全启动策略变量，否则该值为 1。

`KEKDefault` 变量（如果存在）包含平台定义的密钥交换密钥数据库。这在运行时不使用，但提供是为了允许操作系统恢复 OEM 的默认密钥设置。此变量的内容不包括 `EFI_VARIABLE_AUTHENTICATION` 或 `EFI_VARIABLE_AUTHENTICATION2` 结构。

`PKDefault` 变量（如果存在）包含平台定义的平台密钥。这在运行时不使用，但提供是为了允许操作系统恢复 OEM 的默认密钥设置。此变量的内容不包括 `EFI_VARIABLE_AUTHENTICATION2` 结构。

`dbDefault` 变量（如果存在）包含平台定义的安全启动签名数据库。这在运行时不使用，但提供是为了允许操作系统恢复 OEM 的默认密钥设置。此变量的内容不包括 `EFI_VARIABLE_AUTHENTICATION2` 结构。

`dbrDefault` 变量（如果存在）包含平台定义的安全启动授权恢复签名数据库。这在运行时不使用，但提供是为了允许操作系统恢复 OEM 的默认密钥设置。此变量的内容不包括 `EFI_VARIABLE_AUTHENTICATION2` 结构。

`dtbDefault` 变量（如果存在）包含平台定义的安全启动时间戳签名数据库。这在运行时不使用，但提供是为了允许操作系统恢复 OEM 的默认密钥设置。此变量的内容不包括 `EFI_VARIABLE_AUTHENTICATION2` 结构。

3.4 引导选项恢复

引导选项恢复由两个独立的部分组成，操作系统定义的恢复和平台定义的恢复。操作系统定义的恢复是一种尝试，允许已安装的操作系统恢复任何需要的引导选项，或启动完整的操作系统恢复。平台定义的恢复包括平台在未找到操作系统时作为最后手段执行的任何补救措施，例如默认引导行为（请参阅第 3.4.3 节）。这可能包括保修服务重新配置或诊断选项等行为。

如果必须执行启动选项恢复，启动管理器必须首先尝试操作系统定义的恢复，然后通过 `Boot####` 和 `BootOrder` 变量重新尝试正常启动，如果没有选项成功，最后尝试平台定义的恢复。

3.4.1 操作系统定义的引导选项恢复

如果在 `OsIndications` 中设置了 `EFI_OS_INDICATIONS_START_OS_RECOVERY` 位，或者如果 `BootOrder` 的处理没有成功，则平台必须处理操作系统定义的恢复选项。如果由于 `OsIndications` 而进入操作系统定义的恢复，则不应处理 `SysPrepOrder` 和 `SysPrep####` 变量。请注意，为了避免意图歧义，如果设置了 `EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY`，则在 `OsIndications` 中忽略此位。

操作系统定义的恢复使用 `OsRecoveryOrder` 变量，以及使用供应商特定的 `VendorGuid` 值创建的变量和遵循模式 `OsRecovery####` 的名称。这些变量中的每一个都必须是具有 `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCE` 属性集的经过身份验证的变量。

为了处理这些变量，引导管理器遍历 `OsRecoveryOrder` 变量中的 `EFI_GUID` 结构数组，并且每个指定的 `GUID` 都被视为与一系列变量名称相关联的 `VendorGuid`。对于每个 `GUID`，固件会尝试以十六进制排序顺序加载和执行每个具有该 `GUID` 和名称遵循模式 `OsRecovery####` 的变量。这些变量与 `Boot####` 变量具有相同的格式，并且启动管理器必须验证它尝试加载的每个变量都是使用与证书相关联的公钥创建的，该证书链接到授权恢复签名数据库 `dbr` 中列出的证书并且不在禁止的签名数据库中，或者由密钥交换密钥数据库 `KEK` 或当前平台密钥 `PK` 中的密钥创建。

如果引导管理器在没有调用 `EFI_BOOT_SERVICES.ExitBootServices()` 或 `ResetSystem()` 的情况下完成 `OsRecovery####` 选项的处理，则它必须尝试第二次处理 `BootOrder`。如果在该过程中引导不成功，操作系统定义的恢复失败，引导管理器必须尝试基于平台的恢复。

如果在处理 `OsRecovery####` 变量时，引导管理器遇到由于违反安全策略而无法加载或执行的条目，则它必须忽略该变量。

3.4.2 平台定义的引导选项恢复

如果在 `OsIndications` 中设置了 `EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY` 位，或者如果操作系统定义的恢复失败，则系统固件必须通过以与 `OsRecovery####` 相同的方式迭代其 `PlatformRecovery####` 变量来开始特定于平台的恢复，但必须如果任何条目成功，则停止处理。如果由于 `OsIndications` 而输入特定于平台的恢复，则不应处理 `SysPrepOrder` 和 `SysPrep####` 变量。

3.4.3 引导选项变量默认引导行为

全局定义变量的默认状态是特定于固件供应商的。但是，在平台上不存在有效引导选项的例外情况下，引导选项需要标准默认行为。任何时候 `BootOrder` 变量不存在或仅指向不存在的引导选项，或者如果 `BootOrder` 中的条目都无法成功执行，则必须调用默认行为。

如果系统固件支持启动选项恢复，如第 3.4 节所述，系统固件必须包含一个 `PlatformRecovery####` 变量，指定一个简短格式的文件路径媒体设备路径（请参阅第 3.1.2 节），其中包含可移动媒体的平台默认文件路径（见表 3-2）。为了最大程度地兼容本规范的先前版本，建议将此条目作为第一个此类变量，尽管它可能位于列表中的任何位置。

预计此默认引导将加载操作系统或维护实用程序。如果这是操作系统设置程序，则它负责为后续引导设置必要的环境变量。平台固件还可以决定恢复或设置为一组已知的启动选项。

3.5 引导机制

EFI 可以使用 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 或 `EFI_LOAD_FILE_PROTOCOL` 从设备引导。支持 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 的设备必须实现该设备可引导的文件系统协议。如果设备不希望支持完整的文件系统，它可能会生成一个 `EFI_LOAD_FILE_PROTOCOL`，允许它直接实现图像。引导管理器将首先尝试使用 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 进行引导。如果失败，则将使用 `EFI_LOAD_FILE_PROTOCOL`。

3.5.1 通过简单文件协议启动

当通过 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 引导时，`FilePath` 将以指向实现 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 或 `EFI_BLOCK_IO_PROTOCOL` 的设备的设备路径开始。`FilePath` 的下一部分可能指向文件名，包括包含可引导映像的子目录。如果文件名是空设备路径，则文件名必须根据下面定义的规则生成。

如果 `FilePathList[0]` 设备不支持 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`，但支持 `EFI_BLOCK_IO_PROTOCOL` 协议，则必须为 `FilePathList[0]` 调用 EFI 引导服务 `EFI_BOOT_SERVICES.ConnectController()`, `DriverImageHandle` 和 `RemainingDevicePath` 设置为 `NULL`，递归标志设置为 `TRUE`。然后固件将尝试从使用下面概述的算法生成的任何子句柄启动。

指定文件系统的格式包含在第 13.3 节中。虽然固件必须生成一个理解 UEFI 文件系统的 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`，但是任何文件系统都可以使用 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 接口进行抽象。

3.5.1.1 移动媒介引导行为

要在 `FilePath` 中不存在文件名时生成文件名，固件必须以 `\EFI\BOOT\BOOT{machine type short-name}.EFI` 形式附加默认文件名，其中 `machine type short-name` 定义 PE32+ 图像格式建筑学。每个文件只包含一种 UEFI 映像类型，系统可能支持从一种或多种映像类型启动。表 3-2 列出了 UEFI 图像类型。

Table 3-2 UEFI Image Types

	File Name Convention	PE Executable Machine Type *
32-bit	BOOTIA32.EFI	0x14c
x64	BOOTx64.EFI	0x8664
Itanium architecture	BOOTIA64.EFI	0x200
AArch32 architecture	BOOTARM.EFI	0x01c2
AArch64 architecture	BOOTAA64.EFI	0xAA64
RISC-V 32-bit architecture	BOOTRISCV32.EFI	0x5032
RISC-V 64-bit architecture	BOOTRISCV64.EFI	0x5064
RISC-V 128-bit architecture	BOOTRISCV128.EFI	0x5128

Note: * The PE Executable machine type is contained in the machine field of the COFF file header as defined in the Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0

图 5. UEFI 镜像类型

媒体可以通过简单地拥有一个 `\EFI\BOOT\BOOT{machine type short-name}.EFI` 文件来支持多种体系结构。

3.5.2 通过加载文件协议启动

当通过 `EFI_LOAD_FILE_PROTOCOL` 协议启动时, `FilePath` 是一个设备路径, 指向“说出”`EFI_LOAD_FILE_PROTOCOL` 的设备。图像直接从支持 `EFI_LOAD_FILE_PROTOCOL` 的设备加载。`FilePath` 的其余部分将包含特定于设备的信息。固件将此特定于设备的数据传递给加载的图像, 但不使用它来加载图像。如果 `FilePath` 的其余部分是空设备路径, 则加载的映像有责任实施策略以找到正确的引导设备。

`EFI_LOAD_FILE_PROTOCOL` 用于不直接支持文件系统的设备。网络设备通常以这种模式启动, 在这种模式下图像无需文件系统即可实现。

3.5.2.1 网络引导

网络引导由预引导执行环境 (Preboot eXecution Environment, PXE) BIOS 支持规范描述, 该规范是 `Wired for Management Baseline` 规范的一部分。PXE 指定引导平台可用于与智能系统加载服务器交互的 UDP、DHCP 和 TFTP 网络协议。UEFI 定义了用于实现 PXE 的特殊接口。这些接口包含在 `EFI_PXE_BASE_CODE_PROTOCOL` 中 (参见第 24.3 节)。

3.5.2.2 未来引导媒介

由于 UEFI 定义了平台和操作系统及其加载程序之间的抽象，因此随着技术的发展，应该可以添加新类型的引导媒体。操作系统加载程序不一定要更改以支持新类型的引导。UEFI 平台服务的实现可能会发生变化，但接口将保持不变。操作系统将需要驱动程序来支持新型引导媒体，以便它可以从 UEFI 引导服务过渡到引导媒体的操作系统控制。# 4-EFI-System-Table

本节介绍 UEFI 映像的入口点以及传递到该入口点的参数。符合本规范的固件可以加载和执行三种类型的 UEFI 镜像。它们是 UEFI 应用程序（参见第 2.1.2 节）、UEFI 引导服务驱动程序（参见第 2.1.4 节）和 UEFI 运行时驱动程序（参见第 2.1.4 节）。UEFI 应用程序包括 UEFI OS 加载程序（参见第 2.1.3 节）。这三种镜像类型的入口点没有区别。

3.6 UEFI Image Entry Point

传递给镜像的最重要参数是指向系统表的指针。该指针是 `EFI_IMAGE_ENTRY_POINT`（参见下面的定义），它是 UEFI 映像的主要入口点。系统表包含指向活动控制台设备的指针、指向启动服务表的指针、指向运行时服务表的指针以及指向系统配置表（如 ACPI、SMBIOS 和 SAL 系统表）列表的指针。本节详细介绍系统表。

`EFI_IMAGE_ENTRY_POINT`

概述

这是 UEFI 映像的主要入口点。此入口点对于 UEFI 应用程序和 UEFI 驱动程序是相同的。

原型

```
1 typedef
2 EFI_STATUS
3 (EFIAPI *EFI_IMAGE_ENTRY_POINT) (
4 IN EFI_HANDLE ImageHandle,
5 IN EFI_SYSTEM_TABLE *SystemTable
6 );
```

参数

- **ImageHandle**: 为 UEFI 映像分配的固件句柄
- **SystemTable**: 指向 EFI 系统表的指针

描述

此函数是 EFI 映像的入口点。EFI 映像由 EFI 引导服务 `EFI_BOOT_SERVICES.LoadImage()` 加载并重新定位在系统内存中。EFI 映像通过 EFI 引导服务 `EFI_BOOT_SERVICES.StartImage()` 调用。

传递给镜像的最重要参数是指向系统表的指针。这个指针是 `EFI_IMAGE_ENTRY_POINT`（见下面的定义），UEFI Image 的主要入口点。系统表包含指向活动控制台设备的指针、指向引导服务表的指针、指向运行时服务表的指针以及指向系统配置表列表（例如 ACPI、SMBIOS 和 SAL 系统表）的指针。本节详细介绍系统表。

`ImageHandle` 是固件分配的句柄，用于在各种功能上识别镜像。该句柄还支持镜像可以使用的一种或多种协议。所有镜像都支持 `EFI_LOADED_IMAGE_PROTOCOL` 和 `EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL`，它们返回镜像的源位置、镜像的内存位置、镜像的加载选项等。确切的 `EFI_LOADED_IMAGE_PROTOCOL` 和 `EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL` 结构在第 9 节中定义。

如果 UEFI 映像是不是 UEFI 操作系统加载程序的 UEFI 应用程序，则该应用程序将执行并返回或调用 EFI 引导服务 `EFI_BOOT_SERVICES.Exit()`。UEFI 应用程序在退出时总是从内存中卸载，并将其返回状态返回给启动该 UEFI 应用程序的组件。

如果 UEFI 映像是 UEFI 操作系统加载程序，则 UEFI 操作系统加载程序将执行并返回，调用 EFI 引导服务 `Exit()`，或调用 EFI 引导服务 `EFI_BOOT_SERVICES.ExitBootServices()`。如果 EFI OS Loader 返回或调用 `Exit()`，则 OS 加载失败，EFI OS Loader 从内存中卸载，控制权返回到尝试启动 UEFI OS Loader 的组件。如果调用了 `ExitBootServices()`，那么 UEFI OS Loader 已经控制了平台，EFI 将不会重新获得系统的控制权，直到平台被重置。重置平台的一种方法是通过 EFI 运行时服务 `ResetSystem()`。

如果 UEFI 映像是 UEFI 驱动程序，则 UEFI 驱动程序将执行并返回或调用 Boot Service `Exit()`。如果 UEFI 驱动程序返回错误，则驱动程序将从内存中卸载。如果 UEFI 驱动程序返回 `EFI_SUCCESS`，则它会驻留在内存中。如果 UEFI 驱动程序不遵循 UEFI 驱动程序模型，则它会执行任何必需的初始化并在返回之前安装其协议服务。如果驱动程序确实遵循 UEFI 驱动程序模型，则不允许入口点接触任何设备硬件。相反，入口点需要在 UEFI 驱动程序的 `ImageHandle` 上创建和安装 `EFI_DRIVER_BINDING_PROTOCOL`（请参阅第 11.1 节）。如果此过程完成，则返回 `EFI_SUCCESS`。如果资源不可用于完成 UEFI 驱动程序初始化，则返回 `EFI_OUT_OF_RESOURCES`。

返回的状态码

- `EFI_SUCCESS`: 驱动程序已初始化。
- `EFI_OUT_OF_RESOURCES`: 由于缺乏资源，请求无法完成。

3.7 EFI 表头

数据类型 `EFI_TABLE_HEADER` 是所有标准 EFI 表类型之前的数据结构。它包括对每个表类型唯一的签名、可以在扩展添加到 EFI 表类型时更新的表修订以及一个 32 位 CRC，因此 EFI 表类型的消费者可以验证 EFI 表类型的内容 EFI 表。

`EFI_TABLE_HEADER`

概述

在所有标准 EFI 表类型之前的数据结构。

原型

```
1 typedef struct {
2     UINT64 Signature;
3     UINT32 Revision;
```

```

4     UINT32 HeaderSize;
5     UINT32 CRC32;
6     UINT32 Reserved;
7 } EFI_TABLE_HEADER;

```

参数

- **Signature:** 标识后面的表类型的 64 位签名。已为 EFI 系统表、EFI 引导服务表和 EFI 运行时服务表生成唯一签名。
- **Revision:** 此表符合的 EFI 规范的修订版。该字段的高 16 位包含主修订值，低 16 位包含次修订值。次要修订值是二进制编码的十进制数，并且限制在 00..99 的范围内。
 - 当打印或显示时，UEFI 规范修订被称为（主要修订）。（次要修订上位小数）。（次要修订下位）或（主要修订）。（次要修订小数点上位）如果次要修订小数下位设置为 0。例如
 - 具有修订值 ((2«16) | (30)) 的规范将称为 2.3
 - 具有修订值 ((2«16) | (31)) 的规范将称为 2.3.1
- **HeaderSize:** 整个表的大小（以字节为单位），包括 `EFI_TABLE_HEADER`。
- **CRC32:** 整个表的 32 位 CRC。通过将此字段设置为 0 并计算 `HeaderSize` 字节的 32 位 CRC 来计算此值
- **Reserved:** 必须设置为 0 的保留字段。

描述

注：EFI 系统表、运行时表和引导服务表中的功能可能会随时间发生变化。每个表中的第一个字段是 `EFI_TABLE_HEADER`。当新的能力和功能被添加到表中的功能时，此标头的修订字段会增加。检查功能时，代码应验证 `Revision` 是否大于或等于将功能添加到 UEFI 规范时表的修订级别。

注：除非另有说明，否则 UEFI 使用标准 CCITT32 CRC 算法进行 CRC 计算，其种子多项式值为 `0x04c11db7`。

注：系统表、运行时服务表和引导服务表的大小可能会随着时间的推移而增加。始终使用 `EFI_TABLE_HEADER` 的 `HeaderSize` 字段来确定这些表的大小非常重要。

3.8 EFI 系统表

UEFI 使用 EFI 系统表，它包含指向运行时和启动服务表的指针。这个表的定义在下面的代码片段中显示。除了表头，服务表中的所有元素都是指向第 7 节和第 8 节中定义的函数的指针。在调用 `EFI_BOOT_SERVICES.ExitBootServices()` 之前，EFI 系统表的所有字段都是有效的。在操作系统通过调用 `ExitBootServices()` 控制平台后，只有 `Hdr`、`FirmwareVendor`、`FirmwareRevision`、`RuntimeServices`、`NumberOfTableEntries` 和 `ConfigurationTable` 字段有效

`EFI_SYSTEM_TABLE`

概述

包含指向运行时和引导服务表的指针。

相关定义

```
1 #define EFI_SYSTEM_TABLE_SIGNATURE 0x5453595320494249
2 #define EFI_2_90_SYSTEM_TABLE_REVISION ((2<<16) | (90))
3 #define EFI_2_80_SYSTEM_TABLE_REVISION ((2<<16) | (80))
4 #define EFI_2_70_SYSTEM_TABLE_REVISION ((2<<16) | (70))
5 #define EFI_2_60_SYSTEM_TABLE_REVISION ((2<<16) | (60))
6 #define EFI_2_50_SYSTEM_TABLE_REVISION ((2<<16) | (50))
7 #define EFI_2_40_SYSTEM_TABLE_REVISION ((2<<16) | (40))
8 #define EFI_2_31_SYSTEM_TABLE_REVISION ((2<<16) | (31))
9 #define EFI_2_30_SYSTEM_TABLE_REVISION ((2<<16) | (30))
10 #define EFI_2_20_SYSTEM_TABLE_REVISION ((2<<16) | (20))
11 #define EFI_2_10_SYSTEM_TABLE_REVISION ((2<<16) | (10))
12 #define EFI_2_00_SYSTEM_TABLE_REVISION ((2<<16) | (00))
13 #define EFI_1_10_SYSTEM_TABLE_REVISION ((1<<16) | (10))
14 #define EFI_1_02_SYSTEM_TABLE_REVISION ((1<<16) | (02))
15 #define EFI_SPECIFICATION_VERSION EFI_SYSTEM_TABLE_REVISION
16 #define EFI_SYSTEM_TABLE_REVISION EFI_2_90_SYSTEM_TABLE_REVISION
17 typedef struct {
18     EFI_TABLE_HEADER Hdr;
19     CHAR16 *FirmwareVendor;
20     UINT32 FirmwareRevision;
21     EFI_HANDLE ConsoleInHandle;
22     EFI_SIMPLE_TEXT_INPUT_PROTOCOL *ConIn;
23     EFI_HANDLE ConsoleOutHandle;
24     EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *ConOut;
25     EFI_HANDLE StandardErrorHandle;
26     EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *StdErr;
27     EFI_RUNTIME_SERVICES *RuntimeServices;
28     EFI_BOOT_SERVICES *BootServices;
29     UINTN NumberOfTableEntries;
30     EFI_CONFIGURATION_TABLE *ConfigurationTable;
31 } EFI_SYSTEM_TABLE;
```

参数

- **Hdr:** EFI 系统表的表头。此标头包含 `EFI_SYSTEM_TABLE_SIGNATURE` 和 `EFI_SYSTEM_TABLE_REVISION` 值以及 `EFI_SYSTEM_TABLE` 结构的大小和 32 位 CRC，以验证 EFI 系统表的内容是否有效
- **FirmwareVendor:** 指向空终止字符串的指针，该字符串标识为平台生产系统固件的供应商
- **FirmwareRevision:** 固件供应商特定值，用于标识平台的系统固件修订版。
- **ConsoleInHandle:** 活动控制台输入设备的句柄。此句柄必须支持 `EFI_SIMPLE_TEXT_INPUT_PROTOCOL`

和 `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL`。如果没有活动的控制台，这些协议必须仍然存在。

- **ConIn:** 指向与 `ConsoleInHandle` 关联的 `EFI_SIMPLE_TEXT_INPUT_PROTOCOL` 接口的指针。
- **ConsoleOutHandle:** 活动控制台输出设备的句柄。此句柄必须支持 `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL`。如果没有活动控制台，则此协议必须仍然存在
- **ConOut:** 指向与 `ConsoleOutHandle` 关联的 `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` 接口的指针
- **StandardErrorHandler:** 活动标准错误控制台设备的句柄。此句柄必须支持 `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL`。如果没有活动控制台，则此协议必须仍然存在
- **StdErr:** 指向与 `StandardErrorHandler` 关联的 `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` 接口的指针
- **RuntimeServices:** 指向 EFI 运行时服务表的指针。请参阅第 4.5 节。
- **BootServices:** 指向 EFI 引导服务表的指针。请参阅第 4.4 节。
- **NumberOfTableEntries:** 缓冲区 `ConfigurationTable` 中系统配置表的个数
- **ConfigurationTable:** 指向系统配置表的指针。表中的条目数是 `NumberOfTableEntries`。

3.9 EFI Boot Services Table

UEFI 使用 EFI 引导服务表，其中包含表头和指向所有引导服务的指针。该表的定义显示在以下代码片段中。除表头外，EFI 引导服务表中的所有元素都是函数指针的原型，指向第 7 节中定义的函数。在操作系统通过调用控制平台后，此表中的函数指针无效 `EFI_BOOT_SERVICES.ExitBootServices()`。

EFI_BOOT_SERVICES

概述

包含表头和指向所有引导服务的指针。

相关定义

```
1 #define EFI_BOOT_SERVICES_SIGNATURE 0x56524553544f4f42
2 #define EFI_BOOT_SERVICES_REVISION EFI_SPECIFICATION_VERSION
3 typedef struct {
4     EFI_TABLE_HEADER Hdr;
5     //
6     // Task Priority Services
7     //
8     EFI_RAISE_TPL RaiseTPL; // EFI 1.0+
9     EFI_RESTORE_TPL RestoreTPL; // EFI 1.0+
10    //
11    // Memory Services
12    //
13    EFI_ALLOCATE_PAGES AllocatePages; // EFI 1.0+
14    EFI_FREE_PAGES FreePages; // EFI 1.0+
15    EFI_GET_MEMORY_MAP GetMemoryMap; // EFI 1.0+
```

```
16    EFI_ALLOCATE_POOL AllocatePool; // EFI 1.0+
17    EFI_FREE_POOL FreePool; // EFI 1.0+
18    //
19    // Event & Timer Services
20    //
21    EFI_CREATE_EVENT CreateEvent; // EFI 1.0+
22    EFI_SET_TIMER SetTimer; // EFI 1.0+
23    EFI_WAIT_FOR_EVENT WaitForEvent; // EFI 1.0+
24    EFI_SIGNAL_EVENT SignalEvent; // EFI 1.0+
25    EFI_CLOSE_EVENT CloseEvent; // EFI 1.0+
26    EFI_CHECK_EVENT CheckEvent; // EFI 1.0+
27    //
28    // Protocol Handler Services
29    //
30    EFI_INSTALL_PROTOCOL_INTERFACE InstallProtocolInterface; // EFI 1.0+
31    EFI_REINSTALL_PROTOCOL_INTERFACE ReinstallProtocolInterface; // EFI 1.0+
32    EFI_UNINSTALL_PROTOCOL_INTERFACE UninstallProtocolInterface; // EFI 1.0+
33    EFI_HANDLE_PROTOCOL HandleProtocol; // EFI 1.0+
34    VOID* Reserved; // EFI 1.0+
35    EFI_REGISTER_PROTOCOL_NOTIFY RegisterProtocolNotify; // EFI 1.0+
36    EFI_LOCATE_HANDLE LocateHandle; // EFI 1.0+
37    EFI_LOCATE_DEVICE_PATH LocateDevicePath; // EFI 1.0+
38    EFI_INSTALL_CONFIGURATION_TABLE InstallConfigurationTable; // EFI 1.0+
39    //
40    // Image Services
41    //
42    EFI_IMAGE_LOAD LoadImage; // EFI 1.0+
43    EFI_IMAGE_START StartImage; // EFI 1.0+
44    EFI_EXIT Exit; // EFI 1.0+
45    EFI_IMAGE_UNLOAD UnloadImage; // EFI 1.0+
46    EFI_EXIT_BOOT_SERVICES ExitBootServices; // EFI 1.0+
47    //
48    // Miscellaneous Services
49    //
50    EFI_GET_NEXT_MONOTONIC_COUNT GetNextMonotonicCount; // EFI 1.0+
51    EFI_STALL Stall; // EFI 1.0+
52    EFI_SET_WATCHDOG_TIMER SetWatchdogTimer; // EFI 1.0+
53    //
54    // DriverSupport Services
55    //
56    EFI_CONNECT_CONTROLLER ConnectController; // EFI 1.1
57    EFI_DISCONNECT_CONTROLLER DisconnectController; // EFI 1.1+
58    //
```

```
59     // Open and Close Protocol Services
60     //
61     EFI_OPEN_PROTOCOL OpenProtocol; // EFI 1.1+
62     EFI_CLOSE_PROTOCOL CloseProtocol; // EFI 1.1+
63     EFI_OPEN_PROTOCOL_INFORMATION OpenProtocolInformation; // EFI 1.1+
64     //
65     // Library Services
66     //
67     EFI_PROTOCOLS_PER_HANDLE ProtocolsPerHandle; // EFI 1.1+
68     EFI_LOCATE_HANDLE_BUFFER LocateHandleBuffer; // EFI 1.1+
69     EFI_LOCATE_PROTOCOL LocateProtocol; // EFI 1.1+
70     EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES
71     InstallMultipleProtocolInterfaces; // EFI 1.1+
72     EFI_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES
73     UninstallMultipleProtocolInterfaces; // EFI 1.1+
74     //
75     // 32-bit CRC Services
76     //
77     EFI_CALCULATE_CRC32 CalculateCrc32; // EFI 1.1+
78     //
79     // Miscellaneous Services
80     //
81     EFI_COPY_MEM CopyMem; // EFI 1.1+
82     EFI_SET_MEM SetMem; // EFI 1.1+
83     EFI_CREATE_EVENT_EX CreateEventEx; // UEFI 2.0+
84 } EFI_BOOT_SERVICES;
```

参数

- **Hdr** EFI 引导服务表的表头。此标头包含 [EFI_BOOT_SERVICES_SIGNATURE](#) 和 [EFI_BOOT_SERVICES_REVISION](#) 值以及 [EFI_BOOT_SERVICES](#) 结构的大小和 32 位 CRC，以验证 EFI 引导服务表的内容是否有效。
- **RaiseTPL** 提高任务优先级。
- **RestoreTPL** 恢复/降低任务优先级。
- **AllocatePages** 分配特定类型的页面。
- **FreePages** 释放分配的页面。
- **GetMemoryMap** 返回当前引导服务内存映射和内存映射键。
- **AllocatePool** 分配特定类型的池。
- **FreePool** 释放分配的池。
- **CreateEvent** 创建通用事件结构。
- **SetTimer** 设置在特定时间发出信号的事件。
- **WaitForEvent** 停止执行，直到发出事件信号。
- **SignalEvent** 发出事件信号。

- **CloseEvent** 关闭和释放事件结构。
- **CheckEvent** 检查事件是否处于信号状态。
- **InstallProtocolInterface** 在设备句柄上安装协议接口。
- **ReinstallProtocolInterface** 在设备句柄上重新安装协议接口。
- **UninstallProtocolInterface** 从设备句柄中删除协议接口。
- **HandleProtocol** 查询句柄以确定它是否支持指定的协议。保留保留。必须为 NULL。
- **RegisterProtocolNotify** 注册一个事件，每当为指定协议安装接口时，该事件就会发出信号。
- **LocateHandle** 返回支持指定协议的句柄数组。
- **LocateDevicePath** 定位设备路径上支持指定协议的所有设备，并将句柄返回到距离该路径最近的设备。
- **InstallConfigurationTable** 在 EFI 系统表中添加、更新或删除配置表。
- **LoadImage** 将 EFI 镜像加载到内存中。
- **StartImage** 将控制转移到加载镜像的入口点。退出退出镜像的入口点。
- **UnloadImage** 卸载镜像。
- **ExitBootServices** 终止引导服务。
- **GetNextMonotonicCount** 返回平台的单调递增计数。停止处理器。
- **SetWatchdogTimer** 重置和设置引导服务期间使用的看门狗定时器。
- **ConnectController** 使用一组优先规则来找到最佳驱动程序集来管理控制器。
- **DisconnectController** 通知一组驱动程序停止管理控制器。
- **OpenProtocol** 将元素添加到使用协议接口的代理列表中。
- **CloseProtocol** 从使用协议接口的代理列表中删除元素。
- **OpenProtocolInformation** 检索当前使用协议接口的代理列表。
- **ProtocolsPerHandle** 检索句柄上安装的协议列表。返回缓冲区是自动分配的。
- **LocateHandleBuffer** 从句柄数据库中检索满足搜索条件的句柄列表。返回缓冲区是自动分配的。
- **LocateProtocol** 在句柄数据库中查找支持所请求协议的第一个句柄。
- **InstallMultipleProtocolInterfaces** 在句柄上安装一个或多个协议接口。
- **UninstallMultipleProtocolInterfaces** 从句柄中卸载一个或多个协议接口。
- **CalculateCrc32** 计算并返回数据缓冲区的 32 位 CRC。
- **CopyMem** 将一个缓冲区的内容复制到另一个缓冲区。
- **SetMem** 用指定值填充缓冲区。
- **CreateEventEx** 创建事件结构作为事件组的一部分

3.10 EFI Runtime Services Table

UEFI 使用 EFI 运行时服务表，其中包含表头和指向所有运行时服务的指针。该表的定义显示在以下代码片段中。除表头外，EFI 运行时服务表中的所有元素都是指向第 8 节中定义的函数的函数指针的原型。与 EFI 引导服务表不同，此表及其包含的函数指针在 UEFI 操作系统加载程序和操作系统通过调用 [EFI_BOOT_SERVICES](#)。

`ExitBootServices()`控制平台后有效。如果操作系统调用 `SetVirtualAddressMap()`，则此表中的函数指针将固定为指向新的虚拟映射入口点。

EFI_RUNTIME_SERVICES

概述

包含表头和指向所有运行时服务的指针。

相关定义

```
1 #define EFI_RUNTIME_SERVICES_SIGNATURE 0x56524553544e5552
2 #define EFI_RUNTIME_SERVICES_REVISION EFI_SPECIFICATION_VERSION
3 typedef struct {
4     EFI_TABLE_HEADER Hdr;
5     //
6     // Time Services
7     //
8     EFI_GET_TIME GetTime;
9     EFI_SET_TIME SetTime;
10    EFI_GET_WAKEUP_TIME GetWakeupTime;
11    EFI_SET_WAKEUP_TIME SetWakeupTime;
12    //
13    // Virtual Memory Services
14    //
15    EFI_SET_VIRTUAL_ADDRESS_MAP SetVirtualAddressMap;
16    EFI_CONVERT_POINTER ConvertPointer;
17    //
18    // Variable Services
19    //
20    EFI_GET_VARIABLE GetVariable;
21    EFI_GET_NEXT_VARIABLE_NAME GetNextVariableName;
22    EFI_SET_VARIABLE SetVariable;
23    //
24    // Miscellaneous Services
25    //
26    EFI_GET_NEXT_HIGH_MONO_COUNT GetNextHighMonotonicCount;
27    EFI_RESET_SYSTEM ResetSystem;
28    //
29    // UEFI 2.0 Capsule Services
30    //
31    EFI_UPDATE_CAPSULE UpdateCapsule;
32    EFI_QUERY_CAPSULE_CAPABILITIES QueryCapsuleCapabilities;
33    //
34    // Miscellaneous UEFI 2.0 Service
```

```

35      // 
36      EFI_QUERY_VARIABLE_INFO QueryVariableInfo;
37 } EFI_RUNTIME_SERVICES;
```

参数

- **Hdr** EFI 运行时服务表的表头。此标头包含 `EFI_RUNTIME_SERVICES_SIGNATURE` 和 `EFI_RUNTIME_SERVICES_REVISION` 值以及 `EFI_RUNTIME_SERVICES` 结构的大小和 32 位 CRC，以验证 EFI 运行时服务表的内容是否有有效。
- **GetTime** 返回当前时间和日期，以及平台的计时功能。
- **SetTime** 设置当前本地时间和日期信息。
- **GetWakeupTime** 返回当前的唤醒闹钟设置。
- **SetWakeupTime** 设置系统唤醒闹钟时间。
- **SetVirtualAddressMap** 由 UEFI 操作系统加载程序用于从物理寻址转换为虚拟寻址。
- **ConvertPointer** 由 EFI 组件用来在切换到虚拟寻址时转换内部指针。
- **GetVariable** 返回变量的值。
- **GetNextVariableName** 枚举当前变量名。
- **SetVariable** 设置变量的值。
- **GetNextHighMonotonicCount** 返回平台单调计数器的下一个高 32 位。
- **ResetSystem** 重置整个平台。
- **UpdateCapsule** 将胶囊传递给具有虚拟和物理映射的固件。
- **QueryCapsuleCapabilities** 返回是否可以通过 `UpdateCapsule()` 支持胶囊。
- **QueryVariableInfo** 返回有关 EFI 变量存储的信息

3.11 EFI Configuration Table & Properties Table

EFI 配置表是 EFI 系统表中的 `ConfigurationTable` 字段。此表包含一组 GUID 指针对。该表的每个元素都由下面的 `EFI_CONFIGURATION_TABLE` 结构描述。配置表的类型数量预计会随着时间的推移而增长。这就是使用 GUID 来标识配置表类型的原因。EFI 配置表最多包含每种表类型的一次实例。

EFI_CONFIGURATION_TABLE

概述

包含一组 GUID/指针对，由 EFI 系统表中的 `ConfigurationTable` 字段组成

相关定义

```

1  typedef struct{
2      EFI_GUID VendorGuid;
3      VOID *VendorTable;
4  } EFI_CONFIGURATION_TABLE;
```

参数

- **VendorGuid** 唯一标识系统配置表的 128 位 GUID 值。
- **VendorTable** 指向与 **VendorGuid** 关联的表的指针。用于存储表的内存类型以及该指针在运行时是物理地址还是虚拟地址（当调用 **SetVirtualAddressMap()** 时，表中报告的特定地址是否得到修复）由 **VendorGuid** 确定。除非另有说明，否则表缓冲区的内存类型由第 2 章调用约定部分中规定的指南定义。定义 **VendorTable** 的规范有责任指定额外的内存类型要求（如果有）以及是否转换表中报告的地址。任何所需的地址转换都是发布相应配置表的驱动程序的责任。指向与 **VendorGuid** 关联的表的指针。这个指针在运行时是物理地址还是虚拟地址由 **VendorGuid** 决定。与给定 **VendorTable** 指针关联的 **VendorGuid** 定义在调用 **SetVirtualAddressMap()** 时表中报告的特定地址是否得到修复。定义 **VendorTable** 的规范有责任指定是否转换表中报告的地址。

行业标准配置表

以下列表显示了一些行业标准中定义的表的 GUID。这些行业标准定义了在基于 UEFI 的系统上作为 UEFI 配置表访问的表。这些表条目中报告的所有地址都将被引用为物理地址，并且在从预引导阶段过渡到运行时阶段时不会被修复。此列表并不详尽，不会显示所有可能的 UEFI 配置表的 GUID。

```
1 #define EFI_ACPI_20_TABLE_GUID
2   \
3   {
4     0x8868e871, 0xe4f1, 0x11d3, \
5     {
6       0xbc, 0x22, 0x00, 0x80, 0xc7, 0x3c, 0x88, 0x81
7     }
8   }
9 #define ACPI_TABLE_GUID \
10  {
11    0xeb9d2d30, 0x2d88, 0x11d3, \
12    {
13      0x9a, 0x16, 0x00, 0x90, 0x27, 0x3f, 0xc1, 0x4d
14    }
15  }
16 #define SAL_SYSTEM_TABLE_GUID \
17  {
18    0xeb9d2d32, 0x2d88, 0x11d3, \
19    {
20      0x9a, 0x16, 0x00, 0x90, 0x27, 0x3f, 0xc1, 0x4d
21    }
22  }
23 #define SMBIOS_TABLE_GUID \
24  {
25    0xeb9d2d31, 0x2d88, 0x11d3, \
```

```

26     {
27         0x9a, 0x16, 0x00, 0x90, 0x27, 0x3f, 0xc1, 0x4d
28     }
29 }
30 #define SMBIOS3_TABLE_GUID \
31 {0xf2fd1544, 0x9794, 0x4a2c, \ {0x99,0x2e,0xe5,0xbb,0xcf,0x20,0xe3,0x94})#define
32     MPS_TABLE_GUID \
33     {
34         0xeb9d2d2f, 0x2d88, 0x11d3, \
35         {
36             0x9a, 0x16, 0x00, 0x90, 0x27, 0x3f, 0xc1, 0x4d
37         }
38     } //// ACPI 2.0 or newer tables should use EFI_ACPI_TABLE_GUID//#define
39     EFI_ACPI_TABLE_GUID \
40 {0x8868e871,0xe4f1,0x11d3, \
41 {0xbc,0x22,0x00,0x80,0xc7,0x3c,0x88,0x81}}
42 #define EFI_ACPI_20_TABLE_GUID EFI_ACPI_TABLE_GUID
43 #define ACPI_TABLE_GUID \
44 \
45 {
46     0xeb9d2d30, 0x2d88, 0x11d3, \
47     \
48 {
49     0x9a, 0x16, 0x00, 0x90, 0x27, 0x3f, 0xc1, 0x4d \
50 }
51 #define ACPI_10_TABLE_GUID ACPI_TABLE_GUID

```

JSON Configuration Tables

以下列表显示了为向 EFI 配置表报告固件配置数据而定义的表的 GUID，以及用于处理第 23.5 节中定义的 JSON 负载胶囊的表。在由 `EFI_JSON_CAPSULE_DATA_TABLE_GUID` 标识的表条目中报告的地址将被引用为物理地址，并且在从预引导阶段过渡到运行时阶段时不会被修复。这些由 `EFI_JSON_CONFIG_DATA_TABLE_GUID` 和 `EFI_JSON_CAPSULE_RESULT_TABLE_GUID` 标识的表条目中报告的地址将被引用为虚拟地址，并且在从预引导阶段过渡到运行时阶段时将得到修复。

```

1 # define EFI_JSON_CONFIG_DATA_TABLE_GUID
2 \
3 {
4     0x87367f87, 0x1119, 0x41ce, \
5     {
6         0xaa, 0xec, 0x8b, 0xe0, 0x11, 0x1f, 0x55, 0x8a
7     }

```

```

8     }
9 # define EFI_JSON_CAPSULE_DATA_TABLE_GUID \
10 {
11     0x35e7a725, 0x8dd2, 0x4cac, \
12     {
13         0x80, 0x11, 0x33, 0xcd, 0xa8, 0x10, 0x90, 0x56
14     }
15 }
16 # define EFI_JSON_CAPSULE_RESULT_TABLE_GUID \
17 {
18     0xdbc461c3, 0xb3de, 0x422a, \
19     {
20         0xb9, 0xb4, 0x98, 0x86, 0xfd, 0x49, 0xa1, 0xe5
21     }
22 }

```

Devicetree Tables

以下列表显示了设备树表 (DTB) 的 GUID。有关详细信息，请参阅“设备树规范”标题下的“UEFI 相关文档的链接”(<http://uefi.org/uefi>)。DTB 必须包含在 EfiACPIReclaimMemory 类型的内存中。此表条目中报告的地址将被引用为物理地址，并且在从预引导阶段过渡到运行时阶段时不会被修复。固件必须将 DTB 驻留在内存中并安装在 EFI 中在执行不属于系统固件映像的任何 UEFI 应用程序或驱动程序之前检查系统表。一旦 DTB 作为配置表安装，系统固件不得对其进行任何修改或引用 DTB 中包含的任何数据。

允许 UEFI 应用程序修改或替换加载的 DTB。系统固件不得依赖于 DTB 中包含的任何数据。如果系统固件使用 DTB 进行自己的配置，它应该使用一个单独的私有副本，该副本未安装在 EFI 系统表中或以其他方式暴露给 UEFI 应用程序。

```

1 #define CTEST_MAIN
2 /////////////////////////////////////////////////////////////////// 设备树表, 扁平化设备树 Blob (DTB) 格式// \
3 #define EFI_DTB_TABLE_GUID \
4     \
5     { \
6         0xb1b621d5, 0xf19c, 0x41a5, \
7         { \
8             0x83, 0x0b, 0xd9, 0x15, 0x2c, 0x69, 0xaa, 0xe0 \
9         } \
10    }

```

EFI_RT_PROPERTIES_TABLE

如果操作系统调用 ExitBootServices() 后平台不再支持所有 EFI 运行时服务，则该表应由平台发布。请注意，这只是对操作系统的提示，可以随意忽略，因此平台仍然需要提供不受支持的运行时服务的可调用实现，这些服务只返回 EFI_UNSUPPORTED。

```

1 #define EFI_RT_PROPERTIES_TABLE_GUID \
2     { \
3         0xeb66918a, 0x7eef, 0x402a, \
4             { \
5                 0x84, 0x2e, 0x93, 0x1d, 0x21, 0xc3, 0x8a, 0xe9 \
6             } \
7     } \
8 typedef struct { \
9     UINT16 Version; \
10    UINT16 Length; \
11    UINT32 RuntimeServicesSupported; \
12 } EFI_RT_PROPERTIES_TABLE;

```

- **Version:** 表的版本，必须是 0x1
 - `#define EFI_RT_PROPERTIES_TABLE_VERSION 0x1`

- **Length:** 整个 EFI_RT_PROPERTIES_TABLE 的大小（以字节为单位）必须为 8。
- **RuntimeServicesSupported:** 支持或不支持调用的位掩码，其中位设置为 1 表示支持该调用，0 表示不支持。

1 #define EFI_RT_SUPPORTED_GET_TIME	0x0001
2 # define EFI_RT_SUPPORTED_SET_TIME	0x0002
3 # define EFI_RT_SUPPORTED_GET_WAKEUP_TIME	0x0004
4 # define EFI_RT_SUPPORTED_SET_WAKEUP_TIME	0x0008
5 # define EFI_RT_SUPPORTED_GET_VARIABLE	0x0010
6 # define EFI_RT_SUPPORTED_GET_NEXT_VARIABLE_NAME	0x0020
7 # define EFI_RT_SUPPORTED_SET_VARIABLE	0x0040
8 # define EFI_RT_SUPPORTED_SET_VIRTUAL_ADDRESS_MAP	0x0080
9 # define EFI_RT_SUPPORTED_CONVERT_POINTER	0x0100
10 # define EFI_RT_SUPPORTED_GET_NEXT_HIGH_MONOTONIC_COUNT	0x0200
11 # define EFI_RT_SUPPORTED_RESET_SYSTEM	0x0400
12 # define EFI_RT_SUPPORTED_UPDATE_CAPSULE	0x0800
13 # define EFI_RT_SUPPORTED_QUERY_CAPSULE_CAPABILITIES	0x1000
14 # define EFI_RT_SUPPORTED_QUERY_VARIABLE_INFO	0x2000

这种类型的 EFI 配置表条目中报告的地址将被引用为物理地址，并且在从预引导过渡到运行时阶段时不会被修复。

EFI_PROPERTIES_TABLE (deprecated)

注意：此表已弃用，不应再使用！它将从规范的未来版本中删除。下面描述的 **EFI_MEMORY_ATTRIBUTES_TABLE** 提供了替代机制来实现运行时内存保护。

如果平台满足 MemoryProtectionAttributes 中列出的某些构造要求，则会发布此表。

```

1 typedef struct {
2     UINT32 Version;
3     UINT32 Length;
4     UINT64 MemoryProtectionAttribute;
5 } EFI_PROPERTIES_TABLE;
```

- **Version:** 这是表的修订。后续版本可能会填充额外的位并增加表的长度。如果是后者，Length 字段会适当调整
 - #define EFI_PROPERTIES_TABLE_VERSION 0x00010000
- **Length:** 这是整个 EFI_PROPERTIES_TABLE 结构的大小，包括版本。初始版本的长度为 16
- **MemoryProtectionAttribute:** 这个字段是一个位掩码。任何未定义的位都应被视为保留位。设置位意味着底层固件已根据给定属性构建。

```

1 // 
2 // Memory attribute (Not defined bits are reserved)
3 //
4 #define EFI_PROPERTIES_RUNTIME_MEMORY_PROTECTION_NON_EXECUTABLE_PE_DATA 0x1
5
6 // BIT 0 - description - implies the runtime data is separated from the code
```

该位意味着可执行映像的 UEFI 运行时代码和数据部分是分开的，并且必须按照第 2.3 节中的规定对齐。该位还暗示数据页没有任何可执行代码。

建议不要使用此属性，尤其是对于将运行时代码内存映射描述符分解为 UEFI 模块中的底层代码和数据部分的实现。这种拆分会导致与调用 SetVirtualAddress() 的操作系统的互操作性问题，而没有意识到这些运行时描述符之间存在关系。

EFI_MEMORY_ATTRIBUTES_TABLE

概述

当由系统固件发布时，EFI_MEMORY_ATTRIBUTES_TABLE 提供有关运行时内存块中区域的附加信息，这些内存块在从 **EFI_BOOT_SERVICES.GetMemoryMap()** 函数返回的 **EFI_MEMORY_DESCRIPTOR** 条目中定义。内存属性表当前用于描述可由操作系统或管理程序应用于 UEFI 运行时代码和数据的内存保护。此表的使用者当前必须忽略包含除 **EfiRuntimeServicesData** 和 **EfiRuntimeServicesCode** 之外的任何类型值的条目，以确保与此表的未来使用兼容。属性表可以定义多个条目来描述子区域，这些子区域包含由 **GetMemoryMap()** 返回的单个条目，但是子区域必须合计以完全描述更大的区域，并且不能跨越 **GetMemoryMap()** 报告的条目之间的边界。如果 **GetMemoryMap()** 条目中返回的运行时区域未在内存属性表中描述，则假定该区域与任何内存保护不兼容。

只有 **GetMemoryMap()** 返回的整个 **EFI_MEMORY_DESCRIPTOR** 条目可以传递给 **SetVirtualAddressMap()**。

这种类型的 EFI 配置表条目中报告的地址将被引用为物理地址，并且在从预引导阶段过渡到运行时阶段时不会被修复。

原型

```

1 #define EFI_MEMORY_ATTRIBUTES_TABLE_GUID \
2     { \
3         0xdcfa911d, 0x26eb, 0x469f, \
4             { \
5                 0xa2, 0x20, 0x38, 0xb7, 0xdc, 0x46, 0x12, 0x20 \
6             } \
7     }

```

具有以下数据结构

```

1 /***** \
2 /* EFI_MEMORY_ATTRIBUTES_TABLE \
3 /***** \
4 typedef struct {
5     UINT32 Version;
6     UINT32 NumberOfEntries;
7     UINT32 DescriptorSize;
8     UINT32 Reserved;
9     // EFI_MEMORY_DESCRIPTOR Entry [1];
10 } EFI_MEMORY_ATTRIBUTES_TABLE;

```

- **Version:** 该表的版本。当前版本为 0x00000001
- **NumberOfEntries:** 提供的 EFI_MEMORY_DESCRIPTOR 条目计数。这通常是包含 UEFI 运行时和所有 UEFI 运行时数据区域（例如运行时堆）的所有 UEFI 模块中的 PE/COFF 部分的总数。
- **Entry:** EFI_MEMORY_DESCRIPTOR 类型的条目数组。
- **DescriptorSize:** 内存描述符的大小
- **Reserved:** 保留字节

描述

对于每个数组条目，`EFI_MEMORY_DESCRIPTOR.Attribute` 字段可以通知运行时机构，例如操作系统或管理程序，关于可以在内存管理单元中为该条目定义的内存进行什么级别的保护设置。当前属性字段的唯一有效位是 `EFI_MEMORY_RO`、`EFI_MEMORY_XP` 以及 `EFI_MEMORY_RUNTIME`。无论属性隐含的内存保护如何，`EFI_MEMORY_DESCRIPTOR.Type` 字段都应与封闭的 `SetMemoryMap()` 条目中的内存类型相匹配。`PhysicalStart` 必须按照第 2.3 节中的规定对齐。该列表必须按物理起始地址升序排序。`VirtualStart` 字段必须为零并被操作系统忽略，因为它对该表没有任何用处。`NumPages` 必须覆盖保护映射的整个内存区域。`EFI_MEMORY_ATTRIBUTES_TABLE` 中具有属性 `EFI_MEMORY_RUNTIME` 的每个描述符不得与 `EFI_MEMORY_ATTRIBUTES_TABLE` 中具有属性 `EFI_MEMORY_RUNTIME` 的任何其他描述符重叠。此外，`EFI_MEMORY_ATTRIBUTES_TABLE` 中的描述

符描述的每个内存区域必须是 `GetMemoryMap()` 生成的表中描述符的子区域或等于。

	EFI_MEMORY_RO	EFI_MEMORY_XP	EFI_MEMORY_RUNTIME
No memory access protection is possible for Entry	0	0	1
Write-protected Code	1	0	1
Read/Write Data	0	1	1
Read-only Data	1	1	1

图 6. 内存属性定义的使用

其他配置表

以下列表显示了本规范中定义的其他配置表：

- `EFI_MEMORY_RANGE_CAPSULE_GUID`
- `EFI_DEBUG_IMAGE_INFO_TABLE` (Section 18.4.3)
- `EFI_SYSTEM_RESOURCE_TABLE` (Section 23.4)
- `EFI_IMAGE_EXECUTION_INFO_TABLE` (Section 32.5.3.1)
- User Information Table (Section 36.5)
- HII Database export buffer (Section 33.2.11.1, OS Runtime Utilization)

3.12 镜像入口点示例

以下部分中的示例显示了如何在 UEFI 环境中显示各种表示例。

3.12.1 镜像入口点示例

以下示例显示了 UEFI 应用程序的镜像入口点。此应用程序使用 EFI 系统表、EFI 引导服务表和 EFI 运行时服务表

```

1 EFI_SYSTEM_TABLE      *gST;
2 EFI_BOOT_SERVICES     *gBS;
3 EFI_RUNTIME_SERVICES  *gRT;
4 EfiApplicationEntryPoint(IN EFI_HANDLE          ImageHandle,
5                           IN EFI_SYSTEM_TABLE *SystemTable)
6 {
7     EFI_STATUS Status;

```

```
8     EFI_TIME *Time;
9     gST = SystemTable;
10    gBS = gST->BootServices;
11    gRT = gST->RuntimeServices;
12    //
13    // Use EFI System Table to print "Hello World" to the active console output
14    // device.
15    //
16    Status = gST->ConOut->OutputString(gST->ConOut, L"Hello World\n\r");
17    if (EFI_ERROR(Status)) {
18        return Status;
19    }
20    //
21    // Use EFI Boot Services Table to allocate a buffer to store the current time
22    // and date.
23    //
24    Status = gBS->AllocatePool(EfiBootServicesData, sizeof(EFI_TIME),
25                                (VOID **)&Time);
26    if (EFI_ERROR(Status)) {
27        return Status;
28    }
29    //
30    // Use the EFI Runtime Services Table to get the current time and date.
31    //
32    Status = gRT->GetTime(Time, NULL) if (EFI_ERROR(Status))
33    {
34        return Status;
35    }
36    return Status;
37 }
```

以下示例显示了不遵循 UEFI 驱动程序模型的驱动程序的 UEFI 映像入口点。由于此驱动程序返回 `EFI_SUCCESS`，因此它在退出后会一直驻留在内存中。

```
1 EFI_SYSTEM_TABLE      *gST;
2 EFI_BOOT_SERVICES     *gBS;
3 EFI_RUNTIME_SERVICES  *gRT;
4 EfiDriverEntryPoint(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
5 {
6     gST = SystemTable;
7     gBS = gST->BootServices;
8     gRT = gST->RuntimeServices;
9     //
```

```

10     // Implement driver initialization here.
11     //
12     return EFI_SUCCESS;
13 }
```

以下示例显示了不遵循 UEFI 驱动程序模型的驱动程序的 UEFI 映像入口点。由于此驱动返回 `EFI_DEVICE_ERROR`，退出后不会常驻内存。

```

1 EFI_SYSTEM_TABLE      *gST;
2 EFI_BOOT_SERVICES    *gBS;
3 EFI_RUNTIME_SERVICES *gRT;
4 EfiDriverEntryPoint(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
5 {
6     gST = SystemTable;
7     gBS = gST->BootServices;
8     gRT = gST->RuntimeServices;
9     //
10    // Implement driver initialization here.
11    //
12    return EFI_DEVICE_ERROR;
13 }
```

3.12.2 UEFI Driver Model Example

以下是一个 UEFI 驱动程序模型示例，它显示了 XYZ 总线上 ABC 设备控制器的驱动程序初始化例程。`EFI_DRIVER_BINDING_PROTOCOL` 和 `AbcSupported()`、`AbcStart()` 和 `AbcStop()` 的函数原型在 11.1 节中定义。该函数将驱动程序的镜像句柄和指向 EFI 引导服务表的指针保存在全局变量中，因此其他函数在同一个驱动程序可以访问这些值。然后它创建 `EFI_DRIVER_BINDING_PROTOCOL` 的实例并将其安装到驱动程序的镜像句柄上

```

1 extern EFI_GUID                  gEfiDriverBindingProtocolGuid;
2 EFI_BOOT_SERVICES               *gBS;
3 static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBinding = { AbcSupported, AbcStart,
4                                         AbcStop,        1,
5                                         NULL,          NULL };
6 AbcEntryPoint(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
7 {
8     EFI_STATUS Status;
9     gBS           = SystemTable->BootServices;
10    mAbcDriverBinding->ImageHandle       = ImageHandle;
11    mAbcDriverBinding->DriverBindingHandle = ImageHandle;
12    Status = gBS->InstallMultipleProtocolInterfaces(
```

```

13     &mAbcDriverBinding->DriverBindingHandle, &gEfiDriverBindingProtocolGuid,
14     &mAbcDriverBinding, NULL);
15 return Status;
16 }
```

3.12.3 UEFI Driver Model Example (Unloadable)

下面是与上面相同的 UEFI 驱动程序模型示例，除了它还包含允许通过引导服务 `Unload()` 卸载驱动程序所需的代码。在 `AbcEntryPoint()` 中安装的任何协议或分配的内存都必须在 `AbcUnload()` 中卸载或释放。

```

1  extern EFI_GUID                  gEfiLoadedImageProtocolGuid;
2  extern EFI_GUID                  gEfiDriverBindingProtocolGuid;
3  EFI_BOOT_SERVICES               *gBS;
4  static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBinding = { AbcSupported, AbcStart,
5                                         AbcStop,      1,
6                                         NULL,        NULL };
7  EFI_STATUS
8  AbcUnload(IN EFI_HANDLE ImageHandle);
9  AbcEntryPoint(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
10 {
11     EFI_STATUS          Status;
12     EFI_LOADED_IMAGE_PROTOCOL *LoadedImage;
13     gBS    = SystemTable->BootServices;
14     Status = gBS->OpenProtocol(ImageHandle, &gEfiLoadedImageProtocolGuid,
15                               &LoadedImage, ImageHandle, NULL,
16                               EFI_OPEN_PROTOCOL_GET_PROTOCOL);
17     if (EFI_ERROR(Status)) {
18         return Status;
19     }
20     LoadedImage->Unload           = AbcUnload;
21     mAbcDriverBinding->ImageHandle = ImageHandle;
22     mAbcDriverBinding->DriverBindingHandle = ImageHandle;
23     Status = gBS->InstallMultipleProtocolInterfaces(
24         &mAbcDriverBinding->DriverBindingHandle, &gEfiDriverBindingProtocolGuid,
25         &mAbcDriverBinding, NULL);
26     return Status;
27 }
28 EFI_STATUS
29 AbcUnload(IN EFI_HANDLE ImageHandle)
30 {
31     EFI_STATUS Status;
32     Status = gBS->UninstallMultipleProtocolInterfaces(
```

```

33     ImageHandle, &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding, NULL);
34 return Status;
35 }

```

3.12.4 EFI Driver Model Example (Multiple Instances)

以下与第一个 UEFI 驱动程序模型示例相同，只是它生成三个 `EFI_DRIVER_BINDING_PROTOCOL` 实例。第一个安装在驱动程序的镜像句柄上。另外两个安装在新创建的句柄上

```

1  extern EFI_GUID           gEfiDriverBindingProtocolGuid;
2  EFI_BOOT_SERVICES          *gBS;
3  static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBindingA = {
4      AbcSupportedA, AbcStartA, AbcStopA, 1, NULL, NULL
5  };
6  static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBindingB = {
7      AbcSupportedB, AbcStartB, AbcStopB, 1, NULL, NULL
8  };
9  static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBindingC = {
10     AbcSupportedC, AbcStartC, AbcStopC, 1, NULL, NULL
11 };
12 AbcEntryPoint(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable)
13 {
14     EFI_STATUS Status;
15     gBS = SystemTable->BootServices;
16     //
17     // Install mAbcDriverBindingA onto ImageHandle
18     //
19     mAbcDriverBindingA->ImageHandle      = ImageHandle;
20     mAbcDriverBindingA->DriverBindingHandle = ImageHandle;
21     Status = gBS->InstallMultipleProtocolInterfaces(
22         &mAbcDriverBindingA->DriverBindingHandle,
23         &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingA, NULL);
24     if (EFI_ERROR(Status)) {
25         return Status;
26     }
27     //
28     // Install mAbcDriverBindingB onto a newly created handle
29     //
30     mAbcDriverBindingB->ImageHandle      = ImageHandle;
31     mAbcDriverBindingB->DriverBindingHandle = NULL;
32     Status = gBS->InstallMultipleProtocolInterfaces(
33         &mAbcDriverBindingB->DriverBindingHandle,

```

```
34     &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingB, NULL);  
35     if (EFI_ERROR(Status)) {  
36         return Status;  
37     }  
38     //  
39     // Install mAbcDriverBindingC onto a newly created handle  
40     //  
41     mAbcDriverBindingC->ImageHandle      = ImageHandle;  
42     mAbcDriverBindingC->DriverBindingHandle = NULL;  
43     Status = gBS->InstallMultipleProtocolInterfaces(  
44         &mAbcDriverBindingC->DriverBindingHandle,  
45         &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingC, NULL);  
46     return Status;  
47 }
```

4 GUID Partition Table(GPT) Disk Layout

4.1 GPT 和 MBR 磁盘布局对比

此规范定义了 GUID 分区表 (GPT) 磁盘布局（即分区方案）。下表概述了使用 GPT 磁盘布局相对于传统主引导记录 (MBR) 磁盘布局的优势

- 逻辑块地址 (LBA) 是 64 位（而不是 32 位）。
- 支持多个分区（而不仅仅是四个主分区）。
- 提供主分区表和备份分区表以实现冗余。
- 使用版本号和大小字段以供将来扩展。
- 使用 CRC32 字段提高数据完整性。
- 定义用于唯一标识每个分区的 GUID。
- 使用 GUID 和属性来定义分区内容类型。
- 每个分区包含一个 36 个字符的人类可读名称。

4.2 LBA 0 格式

硬盘的 LBA 0（即第一个逻辑块）包含

- 传统主引导记录 (MBR)（参见第 5.2.1 节）
- 或保护性 MBR（参见第 5.2.3 节）。

4.2.1 传统主引导记录 (MBR)

如果传统 MBR 未使用 GPT 磁盘布局（即，如果它使用 MBR 磁盘布局），它可能位于磁盘的 LBA 0（即第一个逻辑块）。MBR 上的引导代码不由 UEFI 固件执行。

助记符	字节偏移	字节长度	描述
BootCode	0	424	x86 代码在非 UEFI 系统上用于选择 MBR 分区记录并加载该分区的第一个逻辑块。此代码不应在 UEFI 系统上执行。
UniqueMBRDiskSignature	4	4	Unique Disk Signature 这可能被操作系统用来从系统中的其他磁盘中识别磁盘。该值始终由操作系统写入，从不由 EFI 固件写入。
Unknown	444	2	未知。UEFI 固件不得使用该字段。
PartitionRecord	446	16*4	四个遗留 MBR 分区记录的数组（请参阅表 5-2）。
Signature	510	2	设置为 0xAA55（即字节 510 包含 0x55，字节 511 包含 0xAA）。
Reserved	512	Logical BlockSize - 512	保留逻辑块的其余部分（如果有的话）

MBR 包含四个分区记录（参见表 11），每个记录定义一个分区在磁盘上使用的开始和结束 LBA | 助记符 | 字节偏移 | 字节长度 | 描述 || :---: | :---: | :---: | :-----: || BootIndicator | 0 | 1 | 0x80 表示这是可引导的传统分区。其他值表示这不是可引导的旧分区。UEFI 固件不得使用该字段 || StartingCHS | 1 | 3 | CHS 地址格式的分区开始。UEFI 固件不得使用该字段。|| OSType | 4 | 1 | 分区类型。请参阅第 5.2.2 节。|| EndingCHS | 5 | 3 | CHS 地址格式的分区结束。UEFI 固件不得使用该字段。|| StartingLBA | 8 | 4 | 磁盘上分区的起始 LBA。UEFI 固件使用此字段来确定分区的开始。|| SizeInLBA | 12 | 4 | 以逻辑块的 LBA 为单位的分区大小。UEFI 固件使用此字段来确定分区的大小。|

如果 MBR 分区的 OSType 字段为 0xEF（即 UEFI 系统分区），则固件必须使用 InstallProtocolInterface() 将 UEFI 系统分区 GUID 添加到 MBR 分区的句柄中。这允许驱动程序和应用程序（包括操作系统加载程序）轻松搜索代表 UEFI 系统分区的句柄。

必须执行以下测试以确定遗留 MBR 是否有效：

- 签名必须是 0xaa55。
- 包含零 OSType 值或零 SizeInLBA 值的分区记录可能会被忽略。

否则：

- 每个 MBR 分区记录定义的分区必须物理驻留在磁盘上（即不超过磁盘的容量）。
- 每个分区不得与其他分区重叠

图 5-1 显示了具有四个分区的 MBR 磁盘布局示例。

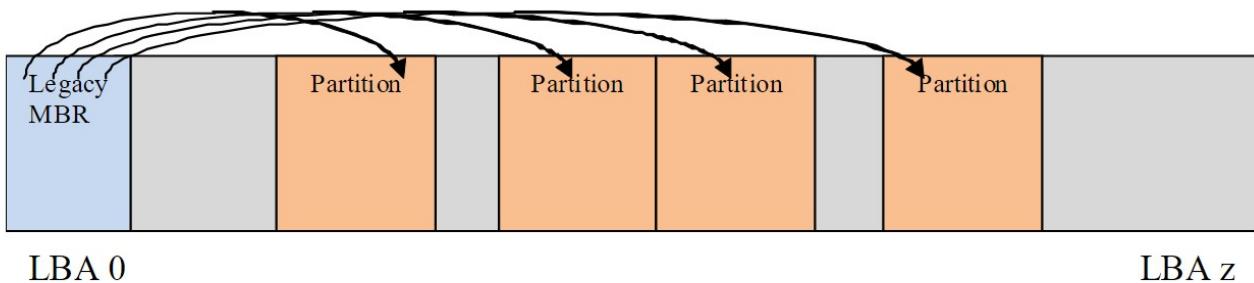


Figure 5-1 MBRDisk Layout with legacy MBR example

相关定义：

```

1 #pragma pack(1)
2 /**
3  /// MBR Partition Entry
4 /**
5  typedef struct {
6      UINT8 BootIndicator;
7      UINT8 StartHead;
8      UINT8 StartSector;
9      UINT8 StartTrack;
10     UINT8 OSIndicator;
11     UINT8 EndHead;
12     UINT8 EndSector;
13     UINT8 EndTrack;
14     UINT8 StartingLBA[4];
15     UINT8 SizeInLBA[4];
16 } MBR_PARTITION_RECORD;
17 /**
18  /// MBR Partition Table
19 /**
20  typedef struct {
21      UINT8             BootstrapCode[440];
22      UINT8             UniqueMbrSignature[4];
23      UINT8             Unknown[2];

```

```

24     MBR_PARTITION_RECORD Partition[4];
25     UINT16                 Signature;
26 } MASTER_BOOT_RECORD;
27 #pragma pack()

```

4.2.2 操作系统类型

本规范定义的唯一类型（本规范未定义其他值）：

- 0xEF（即 UEFI 系统分区）定义了一个 UEFI 系统分区。
- 0xEE（即 GPT Protective）被保护性 MBR（见 5.2.2）用来定义覆盖整个磁盘的假分区。

其他值由遗留操作系统使用，并独立于 UEFI 规范进行分配。

注意：Andries Brouwer 的“分区类型”：请参阅“MBR 磁盘布局中使用的操作系统类型值”标题下的“UEFI 相关文档链接”(<http://uefi.org/uefi>)

4.2.3 Protective MBR

对于可引导磁盘，如果它使用 GPT 磁盘布局，则保护性 MBR 必须位于磁盘的 LBA 0（即第一个逻辑块）。保护性 MBR 在 GUID 分区表头之前，以保持与不理解 GPT 分区结构的现有工具的兼容性

Table 5-3 Protective MBR | 助记符 | 字节偏移 | 字节长度 | 描述 || :-----: | :--: | :----: | :-----: | :-----: || Boot Code | 0 | 440 | 未被 UEFI 系统使用 | | Unique MBR Disk Signature | 440 | 4 | 没用过。归零 | | Unknown | 444 | 2 | 没用过。设置为零。 | | Partition Record | 446 | 16*4 | 四个 MBR 分区记录的数组。包含： • 一个分区记录，如表 5-4 所定义；和 • 三个分区记录，每个记录都设置为零 | | Signature | 510 | 2 | 设置为 0xAA55（即字节 510 包含 0x55，字节 511 包含 0xAA） | | Reserved | 512 | 逻辑块大小 - 512 | 保留逻辑块的其余部分（如果有的话）。归零 |

分区记录之一应如表 12 中所定义，在保护性 MBR 本身之后为 GPT 磁盘布局保留磁盘上的整个空间。

表 5-4 保护整个磁盘的保护性 MBR 分区记录

助记符	字节偏移	字节长度	描述
BootIndic	0	1	设置为 0x00 以指示不可引导分区。如果设置为 0x00 以外的任何值，则此标志在非 UEFI 系统上的行为未定义。UEFI 实现必须忽略。
StartingCHS	1	3	设置为 0x000200，对应 Starting LBA 字段
OSType	4	1	设置为 0xEE（即 GPT 保护）

助记符	字节 偏移	字节 长度	描述
EndingCHS	5	3	设置为磁盘上最后一个逻辑块的 CHS 地址。如果无法表示该字段中的值，则设置为 0xFFFFFFF
StartingLE	8	4	设置为 0x00000001 (即 GPT 分区标头的 LBA)
SizeInLBA	12	4	设置为磁盘大小减一。如果磁盘的大小太大而无法在此字段中表示，则设置为 0xFFFFFFFF

剩余的分区记录应分别设置为零。

图 5-2 展示了一个 GPT 磁盘布局的例子，它有四个分区和一个保护性的 MBR。

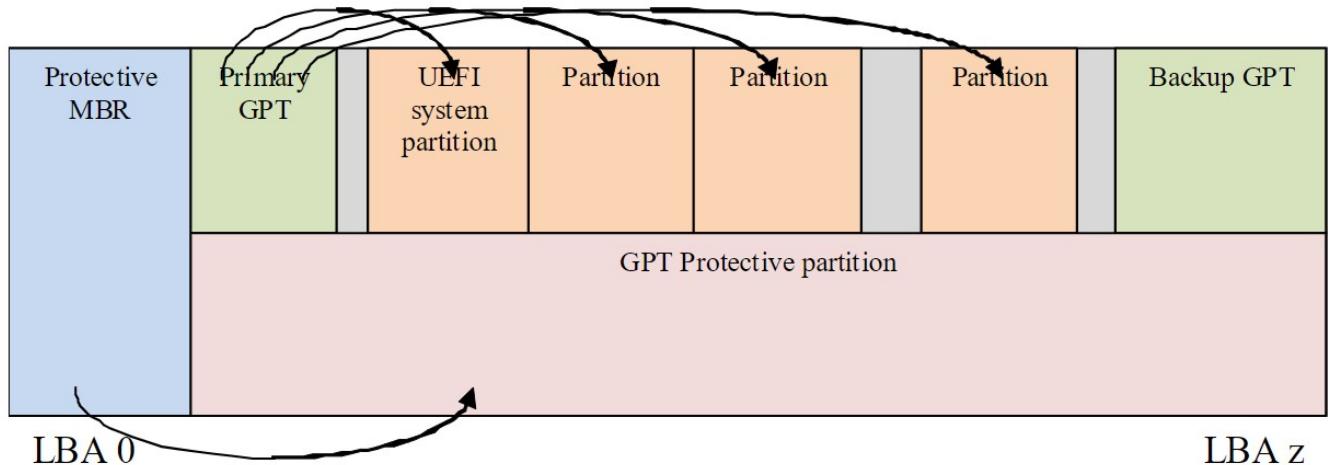


Figure 5-2 GPT disk layout with protective MBR

图 5-3 显示了具有四个分区的 GPT 磁盘布局示例，其中磁盘容量超过 LBA 0xFFFFFFFF。

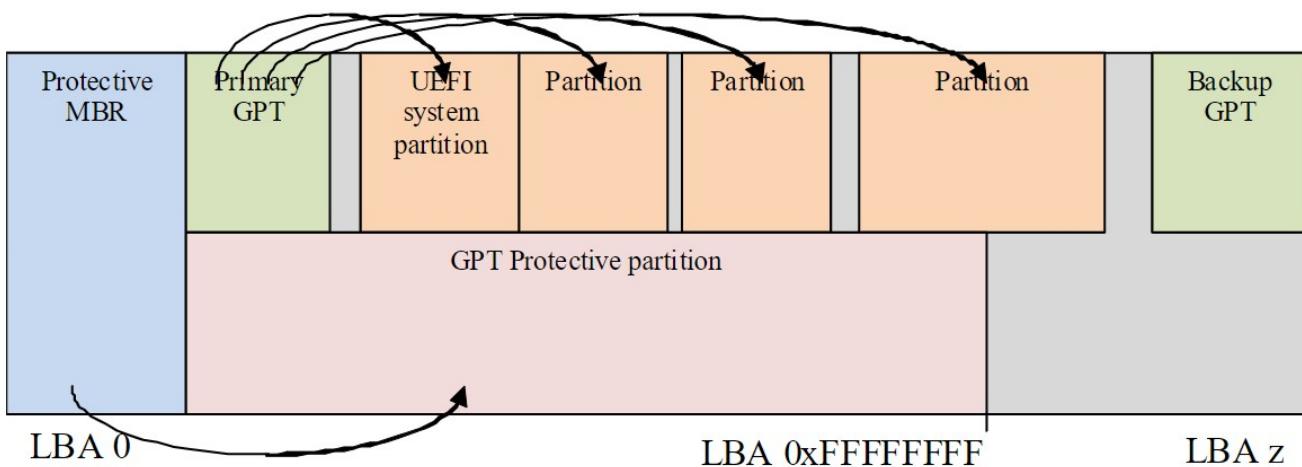


Figure 5-3 GPT disk layout with protective MBR on a disk with capacity > LBA 0xFFFFFFFF

4.2.4 分区信息

在每个安装了逻辑 `EFI_BLOCK_IO_PROTOCOL` 的设备句柄上安装 `EFI_PARTITION_INFO` 协议。

4.3 GUID 分区表 (GPT) 磁盘布局

4.3.1 GPT 概述

GPT 分区方案如图 5-4 所示。GPT 标头（参见第 5.3.2 节）包括一个签名和一个修订号，用于指定分区标头中数据字节的格式。GUID 分区表标头包含一个标头大小字段，用于计算确认 GPT 标头完整性的 CRC32。虽然 GPT 标头的大小在未来可能会增加，但它不能跨越设备上的多个逻辑块。

LBA 0（即第一个逻辑块）包含一个保护性 MBR（参见第 5.2.3 节）。

设备上存储了两个 GPT 标头结构：主要和备份。主 GPT Header 必须位于 LBA 1（即第二个逻辑块），备份 GPT Header 必须位于设备的最后一个 LBA。在 GPT 标头中，我的 LBA 字段包含 GPT 标头本身的 LBA，备用 LBA 字段包含其他 GPT 标头的 LBA。例如，主 GPT 标头的我的 LBA 值将为 1，其备用 LBA 将为设备的最后一个 LBA 的值。备份 GPT 标头的字段将被反转。GPT 标头定义了 GPT 分区条目可用的 LBA 范围。此范围定义为包括逻辑设备上的第一个可用 LBA 到最后一个可用 LBA。存储在卷上的所有数据必须存储在第一个可用 LBA 到最后一个可用 LBA 之间，并且只有 UEFI 定义的用于管理分区的数据结构可以驻留在可用空间之外。Disk GUID 的值是唯一标识整个 GPT Header 及其所有关联存储的 GUID。该值可用于唯一标识磁盘。GPT 分区条目数组的开始位于分区条目 LBA 字段指示的 LBA 处。GUID 分区条目元素的大小在“分区条目大小”字段中定义。GPT 分区条目数组的 32 位 CRC 存储在分区条目数组 CRC32 字段的 GPT 标头中。GPT 分区条目数组的大小是分区条目大小乘以分区条目数。如果 GUID 分区条目数组的大小不是逻辑块大小的偶数

倍，则最后一个逻辑块中剩余的任何空间都将保留，并且不会被分区条目数组 CRC32 字段覆盖。更新 GUID 分区条目时，必须更新分区条目数组 CRC32。当 Partition Entry Array CRC32 更新时，GPT Header CRC 也必须更新，因为 Partition Entry Array CRC32 存储在 GPT Header 中。

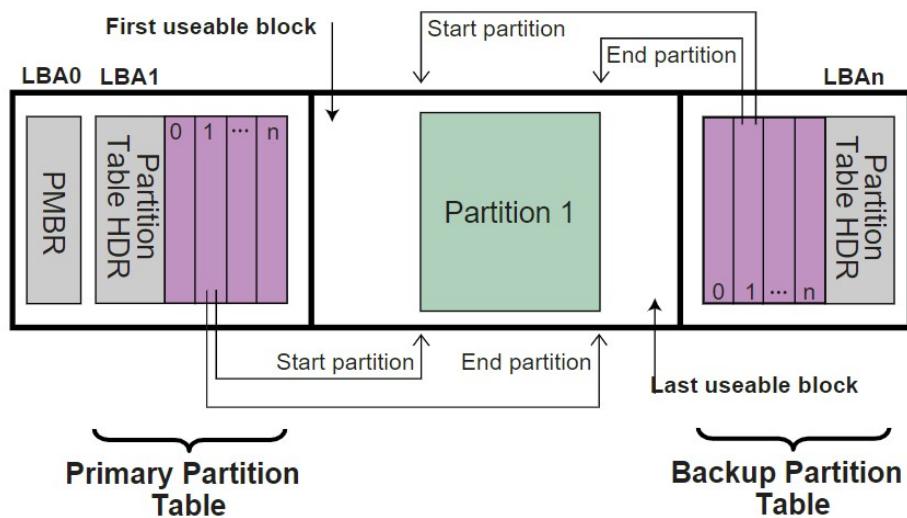


图 7. GUID 分区表 (GPT) 示例

主 GPT 分区条目数组必须位于主 GPT 标头之后并在第一个可用 LBA 之前结束。备份 GPT Partition Entry Array 必须位于 Last Usable LBA 之后并在备份 GPT Header 之前结束。

因此，主要和备份 GPT 分区 EntryArrays 存储在磁盘上的不同位置。每个 GPT 分区条目定义一个分区，该分区包含在 GPT 标头声明的可用空间内的范围内。GPT 分区条目数组中可能正在使用零个或多个 GPT 分区条目。每个定义的分区不得与任何其他定义的分区重叠。如果 GUID 分区条目的所有字段都为零，则该条目未在使用中。必须为 GPT 分区条目数组保留至少 16,384 字节的空间。

如果块大小为 512，则 First Usable LBA 必须大于或等于 34（允许 1 块用于 Protective MBR，1 块用于 Partition Table Header，32 块用于 GPT Partition Entry Array）；如果逻辑块大小为 4096，则 First Useable LBA 必须大于或等于 6（允许 1 个块用于 Protective MBR，1 个块用于 GPT Header，4 个块用于 GPT Partition Entry Array）。

设备可能会提供一个长度不是 512 字节的逻辑块大小。在 ATA 中，这称为 Long Logical Sector 功能集；ATA 设备在 IDENTIFY DEVICE 数据字 106 位 12 中报告支持此功能集，并在 IDENTIFY DEVICE 数据字 117-118 中报告每个逻辑扇区的字数（即 2 个字节）（请参阅 ATA8-ACS）。SCSI 设备在 READ CAPACITY 参数数据 Block Length In Bytes 字段中报告其逻辑块大小（请参阅 SBC-3）。设备可能呈现小于物理块大小的逻辑块大小（例如，呈现 512 字节的逻辑块大小但实现 4,096 字节的物理块大小）。在 ATA 中，这称为 Long Physical Sector 功能集；ATA 设备在 IDENTIFY DEVICE 数据字 106 位 13 中报告支持此功能集，并在 IDENTIFY DEVICE 数据字 106 位 3-0 中报告物理扇区大小/逻辑扇区大小指数比（请参阅 ATA8-ACS）。SCSI 设备在读

取容量 (16) 参数数据逻辑块每个物理块指指数字段（请参阅 SBC-3）中报告其逻辑块大小/物理块指指数比率。这些字段返回每个物理扇区 2^x 逻辑扇区（例如，3 表示 $2^3=8$ 每个物理扇区 8 个逻辑扇区）。

实现长物理块的设备可能会呈现未与底层物理块边界对齐的逻辑块。ATA 设备在 IDENTIFY DEVICE 数据字 209（参见 ATA8-ACS）中报告物理块内逻辑块的对齐情况。SCSI 设备在 READ CAPACITY (16) 参数数据最低对齐逻辑块地址字段中报告其对齐情况（请参阅 SBC-3）。请注意，ATA 和 SCSI 字段的定义不同（例如，为了使 LBA 63 对齐，ATA 返回值 1 而 SCSI 返回值 7）。在 SCSI 设备中，Block Limits VPD 页面 Optimal Transfer Length Granularity 字段（参见 SBC-3）也可能报告对对齐目的很重要的粒度（例如，RAID 控制器可能会在该字段中返回其 RAID 条带深度）。

GPT 分区应与以下较大者对齐：

- a 物理块边界，如果有的话
- b 最佳传输长度粒度，如果有的话。

例如：

- 如果逻辑块大小为 512 字节，物理块大小为 4,096 字节（即 $512 \text{ 字节} \times 8$ 个逻辑块），没有最佳传输长度粒度，并且逻辑块 0 与物理块边界对齐，则每个 GPT 分区应从 8 的倍数的 LBA 开始。
- 如果逻辑块大小为 512 字节，物理块大小为 8,192 字节（即 $512 \text{ 字节} \times 16$ 逻辑块），最佳传输长度粒度为 65,536 字节（即， $512 \text{ 字节} \times 128$ 个逻辑块），并且逻辑块 0 与物理块边界对齐，那么每个 GPT 分区应该从 LBA 开始，该 LBA 是 128 的倍数。

为避免需要确定物理块大小和最佳传输长度粒度，软件可能会在明显更大的边界处对齐 GPT 分区。例如，假设逻辑块 0 对齐，它可以使用 2,048 的倍数的 LBA 对齐到 1,048,576 字节 (1 MiB) 的边界，这支持最常见的物理块大小和 RAID 条带大小。

参考资料如下：

ISO/IEC 24739-200 [ANSI INCITS 452-2008] AT Attachment 8 - ATA/ATAPI Command Set (ATA8-ACS)。由 INCITS T13 技术委员会提供。（参见“UEFI 相关文档的链接”(<http://uefi.org/uefi>)，标题为“国际信息技术标准委员会 (INCITS)”和“INCITs T13 技术委员会”）。

ISO/IEC 14776-323 [T10/1799-D] SCSI Block Commands - 3 (SBC-3)。可从 www.incits.org 获得。由 INCITS T10 技术委员会提供（参见“UEFI 相关文档的链接”(<http://uefi.org/>) uefi 在“国际信息技术标准委员会 (INCITS)”和“SCSI 块命令”的标题下）

4.3.2 GPT Header

表 5-5 定义 GPT Header。

助记符	字节偏移	字节数度	描述
Signature	0	8	标识 EFI 兼容的分区表头。该值必须包含 ASCII 字符串 “EFI PART”，编码为 64 位常量 0x5452415020494645。
Revision	8	4	此标头的修订号。此修订值与 UEFI 规范版本无关。此标头是 1.0 版，因此正确值为 0x00010000
HeaderSize	12	4	GPT 标头的大小（以字节为单位）。HeaderSize 必须大于或等于 92，并且必须小于或等于逻辑块大小。
HeaderCRC32	16	4	GPT 标头结构的 CRC32 校验和。通过将此字段设置为 0 并计算 HeaderSize 字节的 32 位 CRC 来计算此值 s
Reserved	20	4	必须为零
MyLBA	24	8	包含这个数据结构的 LBA
AlternateLB	32	8	备用 GPT 头的 LBA 地址
FirstUsableLB	40	8	GUID 分区条目描述的分区可能使用的第一个可用逻辑块
LastUsableL	48	8	GUID 分区条目描述的分区可能使用的最后一个可用逻辑块。
DiskGUID	56	16	可用于唯一标识磁盘的 GUID。
PartitionEnt	72	8	GUID 分区条目数组的起始 LBA
NumberOfPartit	80	4	GUID 分区条目数组中的分区条目数
SizeOfPartit	84	4	GUID 分区条目数组中每个 GUID 分区条目结构的大小（以字节为单位）。该字段应设置为 $128 \times 2n$ 的值，其中 n 是大于或等于零的整数（例如，128、256、512 等）。注意：本规范的早期版本允许 8..
PartitionEntryCRC32	88	4	GUID 分区条目数组的 CRC32。从 PartitionEntryLBA 开始，计算 NumberOfPartitionEntries * SizeOfPartitionEntry 的字节长度。
Reserved	92	4	块的其余部分由 UEFI 保留并且必须为零。

必须执行以下测试以确定 GPT 是否有效：

- 检查签名
- 检查标头 CRC
- 检查 MyLBA 条目是否指向包含 GUID 分区表的 LBA
- 检查 GUID 分区条目数组的 CRC 如果 GPT 是主表，存储在 LBA 1:

- 检查 AlternateLBA 查看它是否是一个有效的 GPT

如果主 GPT 已损坏，软件必须检查设备的最后一个 LBA 以查看它是否具有有效的 GPT 标头并指向有效的 GPT 分区条目阵列。如果它指向一个有效的 GPT 分区条目数组，那么软件应该在平台策略设置允许的情况下恢复主 GPT（例如，平台可能需要用户在恢复表之前提供确认，或者可能允许自动恢复表）。

软件必须在恢复 GPT 时进行报告。软件应该在恢复主要 GPT 之前询问用户确认，并且必须在它修改媒体以恢复 GPT 时报告。如果 GPT 格式的磁盘被旧版软件重新格式化为旧版 MBR 格式，最后一个逻辑块可能不会被覆盖，并且可能仍包含陈旧的 GPT。如果 GPT 识别软件随后访问磁盘并接受陈旧的 GPT，它会误解磁盘的内容。如果遗留 MBR 包含有效分区而不是保护性 MBR，软件可能会检测到这种情况（请参阅第 5.2.1 节）。

更新主 GPT 的任何软件也必须更新备份 GPT。软件可以按任何顺序更新 GPT 标头和 GPT 分区条目数组，因为所有 CRC 都存储在 GPT 标头中。软件必须在主 GPT 之前更新备份 GPT，因此如果设备大小发生变化（例如卷扩展）并且更新中断，则备份 GPT 位于磁盘上的正确位置。

如果主 GPT 无效，则备份而是使用 GPT，它位于磁盘上的最后一个逻辑块上。如果备份 GPT 有效，则必须使用它来恢复主 GPT。如果主 GPT 有效而备份 GPT 无效，则软件必须恢复备份 GPT。如果主 GPT 和备份 GPT 都已损坏，则此块设备被定义为没有有效的 GUID 分区标头。

在尝试增加物理卷的大小之前，主 GPT 和备份 GPT 都必须有效。这是由于 GPT 恢复方案取决于在设备末端定位备份 GPT。将磁盘添加到 RAID 设备时，卷的大小可能会增加。一旦卷大小增加，备份 GPT 必须移动到卷的末尾，并且必须更新主要和备份 GPT 标头以反映新的卷大小。

4.3.3 GPT 分区条目数组

GPT 分区条目数组包含一个 GPT 分区条目数组。表 5-6 定义了 GPT 分区条目。

助记符	字节偏移	字节长度	描述
PartitionTypeGUID	0	16	定义此分区的用途和类型的唯一 ID。零值定义此分区条目未被使用。

助记符	字节偏移	字节长度	描述
UniquePartitionGUID	16	16	每个分区条目唯一的 GUID。曾经创建的每个分区都有一个唯一的 GUID。创建 GPT 分区条目时必须分配此 GUID。每当 GPT 标头中的 NumberOfPartitionsEntry 增加以包含更大范围的地址时，就会创建 GPT 分区条目
StartingLBA	32	8	此项定义的分区的起始 LBA。
EndingLBA	40	8	此条目定义的分区的结束 LBA
Attributes	48	8	属性位，UEFI 保留的所有位（见表 5-8）
PartitionName	56	72	包含人类可读的分区名称的以空字符结尾的字符串
Reserved	128	SizeOfPartitionEntry	
nEntry - 128	GPT 分区条目的其余部分（如果有）由 UEFI 保留并且必须为零。说明 GUID 值未使用条目 00000000-0000-0000- 0000-000000000000		

GPT 标头中的 SizeOfPartitionEntry 变量定义了每个 GUID 分区条目的大小。每个分区条目都包含一个唯一分区 GUID 值，该值唯一标识将要创建的每个分区。每当创建一个新的分区条目时，都必须为该分区生成一个新的 GUID，并且保证每个分区都具有唯一的 GUID。分区定义为包括 StartingLBA 和 EndingLBA 在内的所有逻辑块。

PartitionTypeGUID 字段标识分区的内容。此 GUID 类似于 MBR 中的操作系统类型字段。每个文件系统都必须发布其唯一的 GUID。实用程序可以使用 **Attributes** 字段对分区的使用进行广泛的推断，并在表 5-7 中定义。

固件必须使用 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 将 **PartitionTypeGuid** 添加到每个活动 GPT 分区的句柄。这将允许驱动程序和应用程序（包括操作系统加载程序）轻松搜索代表 EFI 系统分区或供应商特定分区类型的句柄。制作 GPT 格式磁盘和分区副本的软件必须在 GPT 标头中生成新的磁盘 GUID 值，并在每个 GPT 分区条目中生成新的唯一分区 GUID 值。如果 GPT 识别软件遇到具有相同 GUID 的两个磁盘或分区，则结果将不确定。

Table 5-7 Defined GPT Partition Entry - Partition Type GUIDs | 描述 | GUID 值 || :--: | :--: || EFI System Partition | C12A7328-F81F-11D2-BA4B-00A0C93EC93B | Partition containing a legacy MBR | 024DEE41-33E7-11D3-9D69-0008C781F39F |

操作系统供应商需要生成他们自己的分区类型 GUID 来标识他们的分区类型。

Table 5-8 Defined GPT Partition Entry - Attributes

Bits	Name	描述
Bit 0	Required	
Partition		如果设置了此位，则平台需要分区才能运行。分区的所有者/创建者表示，删除或修改内容可能会导致平台功能丢失或平台无法启动或运行。如果删除该分区，系统将无法正常运行，应将其视为系统硬件的一部分。如果删除此分区，运行诊断、系统恢复甚至操作系统安装或引导等操作可能会停止工作。除非 OS 软件或固件识别此分区，否则不应删除或修改它，因为 UEFI 固件或平台硬件可能会变得无法正常工作。
Bit 1	No Block IO	

Bits	Name	描述
Protocol		<p>如果设置了该位，则固件不得为该分区生成 EFI_BLOCK_IO_PROTOCOL 设备。有关详细信息，请参阅第 13.3.2 节。</p> <p>通过不生成 EFI_BLOCK_IO_PROTOCOL 分区，不会在 UEFI 中为该分区创建文件系统映射。</p>
Bit 2	Legacy BIOS	
Bootable		<p>此位由本规范留出，让具有传统 PC-AT BIOS 固件实现的系统通知在这些系统上运行的某些有限的专用软件 GPT 分区可能是可引导的。对于具有符合此规范的固件实现的系统，UEFI 引导管理器（请参阅第 3 章）在选择符合 UEFI 的应用程序时必须忽略此位，例如，操作系统加载程序（请参阅 2.1.3）。因此本规范无需定义该位的确切含义</p>
Bits 3-47		未定义且必须为零。为 UEFI 规范的未来版本扩展而保留
Bits 48-63		为 GUID 特定用途保留。这些位的使用将因 PartitionTypeGUID 而异。只允许 PartitionTypeGUID 的所有者修改这些位。如果修改了位 0-47，则必须保留它们

相关定义

```

1 #pragma pack(1)
2 /**
3 /// GPT Partition Entry.
4 /**
5 typedef struct {
6     EFI_GUID PartitionTypeGUID;

```

```
7     EFI_GUID UniquePartitionGUID;
8     EFI_LBA  StartingLBA;
9     EFI_LBA  EndingLBA;
10    UINT64   Attributes;
11    CHAR16   PartitionName[36];
12 } EFI_PARTITION_ENTRY;
13 #pragma pack()
```

5 区块转换表（BTT）布局

本规范定义了块转换表（BTT）元数据布局。以下各小节概述了在媒体上使用的 BTT 格式，涉及的数据结构，以及对 SW 如何解释 BTT 布局的详细描述。

5.1 区块转换表（BTT）背景

命名空间定义了非易失性存储器的连续地址范围，概念上类似于 SCSI 逻辑单元（LUN）或 NVM Express 命名空间。

任何被用于块存储的命名空间都可能包含一个块转换表（BTT），这是一个布局和一组进行块 I/O 的规则，提供单一区块的电源故障写入原子性。传统的块状存储，包括硬盘和固态硬盘，通常会防止撕裂的扇区，也就是在断电中断时部分写入的扇区。现有的软件，主要是文件系统，依赖于这种行为，往往作者没有意识到这一点。为了使这种软件能够在支持块存储访问的命名空间上正常工作，本文定义的 BTT 布局将一个命名空间细分为一个或多个 BTT 区域 (TODO)，这些命名空间的大块区域包含了提供所需写入原子性的元数据。如图 6-1 和图 6-2 所示，这些 BTT A 中的每一个都包含一个元数据布局。

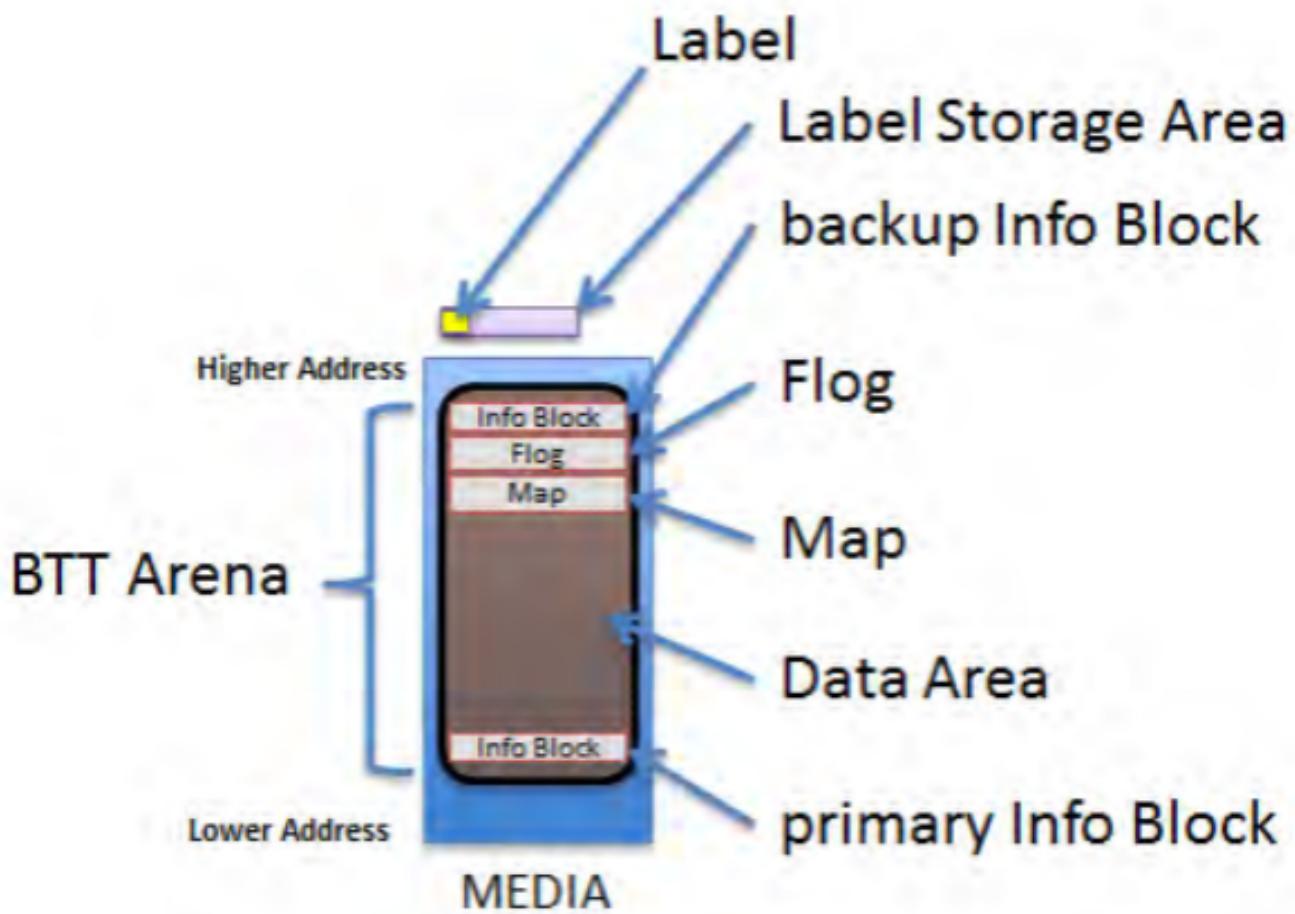


图 8. BTT 区域中的 BTT 布局

每个区域都包含图中所示的布局。BTT 区域中的 BTT 布局，主要信息块、数据区、地图、flog 和一个备份信息块。当命名空间大于 512G 时，BTT 布局需要多个区域，如图 6-2 所示。每个使用 BTT 的命名空间都被划分为尽可能多的 512G 的区域，然后是一个更小的区域，以包含任何适当的剩余空间。最小的区域大小为 16M，所以最后的区域大小应在 16M 和 512G 之间。任何小于 16M 的剩余空间都是未使用的。由于这些区域放置规则，软件可以定位每个主要信息块和每个备份信息块，而无需读取任何元数据，仅基于命名空间大小。

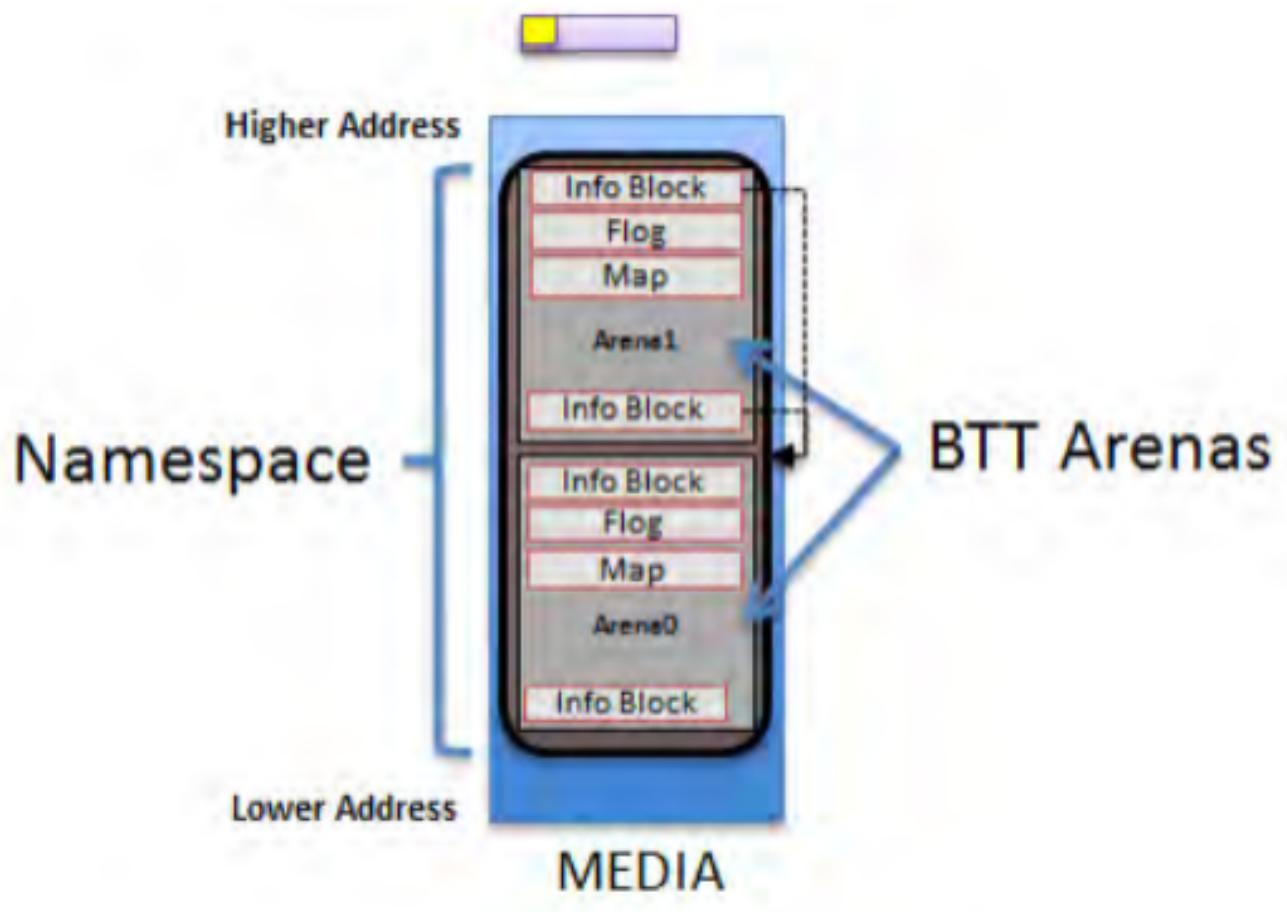


图 9. 大命名空间里包含多个区域的 BTT

5.2 区块转换表（BTT）数据结构

以下各小节概述了与 BTT 布局相关的数据结构。

5.2.1 BTT 信息块

```
1 // Alignment of all BTT structures
2 #define EFI_BTT_ALIGNMENT        4096
3 #define EFI_BTT_INFO_UNUSED_LEN   3968
4 #define EFI_BTT_INFO_BLOCK_SIG_LEN 16
5
```

```
6 // Constants for Flags field
7 #define EFI_BTT_INFO_BLOCK_FLAGS_ERROR 0x00000001
8
9 // Constants for Major and Minor version fields
10 #define EFI_BTT_INFO_BLOCK_MAJOR_VERSION 2
11 #define EFI_BTT_INFO_BLOCK_MINOR_VERSION 0
12
13 typedef struct _EFI_BTT_INFO_BLOCK {
14     CHAR8 Sig[EFI_BTT_INFO_BLOCK_SIG_LEN];
15     EFI_GUID Uuid;
16     EFI_GUID ParentUuid;
17     UINT32 Flags;
18     UINT16 Major;
19     UINT16 Minor;
20     UINT32 ExternalLbaSize;
21     UINT32 ExternalNLba;
22     UINT32 InternalLbaSize;
23     UINT32 InternalNLba;
24     UINT32 NFree;
25     UINT32 InfoSize;
26     UINT64 NextOff;
27     UINT64 DataOff;
28     UINT64 MapOff;
29     UINT64 FlogOff;
30     UINT64 InfoOff;
31     CHAR8 Unused[EFI_BTT_INFO_UNUSED_LEN];
32     UINT64 Checksum;
33 } EFI_BTT_INFO_BLOCK
```

Sig

BTT 索引块数据结构的签名。应为 BTT_ARENA_INFO\0\0。

Uuid

UUID 识别这个 BTT 实例。每次写入初始 BTT 区域时都会创建一个新的 UUID。这个值在一个命名空间的所有领域内的所有 BTT 信息块中应是相同的。

ParentUuid

包含命名空间的 UUID，在验证 BTT 信息块时使用，以确保 BTT 布局的这个实例是为当前周围的命名空间准备的，而不是从使用媒体相同区域的前一个命名空间遗留下来的。这个值在一个命名空间的所有领域内的所有 BTT 信息块中应是相同的。

Flags

该 BTT 信息块的布尔属性。参见下面关于标志的使用的补充说明。以下是定义的数值。[EFI_BTT_INFO_BLOCK_FLAGS_ERROR](#)

- BTT 区域处于错误状态。当 BTT 实现发现不一致的元数据或由于不可恢复的媒体错误而丢失的元数据等问题时，相关区域的错误位应被设置。关于[EFI_BTT_INFO_BLOCK_FLAGS_ERROR](#)的处理，请参见[BTT Theory of Operation](#)部分。

Major

主要版本号。目前为第 2 版。这个值在一个命名空间的所有区域内的所有 BTT 信息块中应是相同的。

Minor

次要版本号。目前为第 0 版。这个值在一个命名空间的所有区域内的所有 BTT 信息块中应是相同的。

ExternalLbaSize

公布的 LBA¹大小，以字节为单位。I/O 请求应在这个大小的块中。这个值在一个命名空间的所有区域内的所有 BTT 信息块中应是相同的。²: LBA(logical block address): 逻辑块地址

ExternalNLba

本区域中公布的 LBA 数量。这个字段的总和，在所有 BTT 区域中，是命名空间中可用 LBA 的总数。

InternalLbaSize

内部 LBA 大小应大于或等于 [ExternalLbaSize](#)，并且不应小于 512 字节。区域数据区的每个块都是这个大小的字节，并且正好包含一个数据块。可以选择，由于 LBA 之间的对齐填充，这可能大于[ExternalLbaSize](#)。这个值在一个命名空间的所有区域内的所有 BTT 信息块中应是相同的。

InternalNLba

区域数据区的内部块数。这应等于 [ExternalNLba](#) + [NFree](#)，因为每个内部 LBA 要么被映射到一个外部 LBA，要么在 flog(TODO) 中显示为自由。

NFree

为写到该区域而保持的空闲块的数量。[NFree](#)应等于[InternalNLba](#)-[ExternalNLba](#)。这个值在一个命名空间的所有区域内的所有 BTT 信息块中应是相同的。

InfoSize

该信息块的大小，以字节为单位。这个值在一个命名空间的所有区域内的所有 BTT 信息块中应是相同的。

NextOff

相对于这个区域的起点，下一个区域的偏移量。偏移量为 0 表示当前区域后面没有区域。提供这个字段是为了方便，因为每个区域的起点可以根据命名空间的大小来计算，如 [Theory of Operation – Validating BTT Arenas at start-up](#) 描述中所述。该值在一个区域内的主要和备用 BTT 信息块中应是相同的。

¹TPL(Task Priority Levels): 任务优先级

²TPL(Task Priority Levels): 任务优先级

DataOff

相对于这个区域的起点，本区域的数据区域的偏移量。内部 LBA number zero(TODO) 指向这个偏移处。该值在一个区域内的主要和备用 BTT 信息块中应是相同的。

MapOff

相对于这个区域的起点，本区域的地图的偏移量。该值在一个区域内的主要和备用 BTT 信息块中应是相同的。

FlogOff

相对于这个区域的起点，本区域的 flog 偏移。该值在一个区域内的主要和备用 BTT 信息块中应是相同的。

InfoOff

相对于这个区域的起点，本区域信息块的备份副本的偏移量。该值在一个区域内的主要和备用 BTT 信息块中应是相同的。

Reserved

应为 0。

Checksum

所有字段 64-bit Fletcher64 的检查和。在计算校验和时，这个字段被认为是含有零。

BTT Info Block Description

一个有效的 BTT 信息块的存在被用来确定一个命名空间是否被用作 BTT 块设备。

每个 BTT 区域包含两个 BTT 信息块，主信息块复制到 BTT 区域的开始位置，地址偏移量为 0，最后是一个相同备份 BTT 信息块，位于区域中可用的最高块，以 `EFI_BTT_ALIGNMENT` 为边界对齐。当写入 BTT 布局时，实施方案为将信息块从最高区域写到最低区域，在写主信息块之前写出备份信息块和其他 BTT 数据结构。以这种方式写入布局应确保只有在整个布局被写入后才会检测到一个有效的 BTT 布局。

5.2.2 BTT Map Entry

```
1 typedef struct _EFI_BTT_MAP_ENTRY {
2     UINT32 PostMapLba : 30;
3     UINT32 Error : 1;
4     UINT32 Zero : 1;
5 } EFI_BTT_MAP_ENTRY;
```

PostMapLba

映射后的 LBA 号码（该区域数据区的块号）

Error

当被设置且 Zero 未被设置时，对该块的读取会返回一个错误。对该块的写操作会清除该标志。

Zero

当设置和 Error 未设置时，对该块的读取会返回整块的零。对该块的写操作会清除该标志。

BTT Map Description

BTT Map 将索引到区域的 LBA 映射到其实际位置。BTT Map 在区域中的位置要尽可能高，在考虑到备用信息块和 flog（以及任何所需的排列）的空间后。术语 *pre-map LBA* 和 *post-map LBA* 被用来描述这种映射的输入和输出值。

Error(TODO) 和 **Zero(TODO)** 比特位表示不能同时为真的条件，所以该组合用于表示一个正常的地图 map entry(TODO: 地图条目)，其中没有错误或归零块的指示。只有当 **Error** 位被设置且 **Zero** 位被清除时，才会显示错误状态，零块条件的逻辑类似。当这两种情况都没有显示时，**Error** 和 **Zero** 都被设置为表示 map entry(TODO) 处于正常、非错误状态。这就留下了 **Error** 和 **Zero** 都是零的情况，这也是 BTT 布局首次写入时所有 map entry(TODO) 的初始状态。两个位都为零意味着 map entry 包含初始身份映射，其中前映射 LBA 被映射到相同的后映射 LBA。以这种方式定义映射，允许实现方案为利用已知命名空间的初始内容为零的情况，在写布局时不需要写到映射。这可以大大改善布局时间，因为 map 是布局过程中写入的最大 BTT 数据结构。

5.2.3 BTT Flog

```
1 // Alignment of each flog structure
2 #define EFI_BTT_FLOG_ENTRY_ALIGNMENT 64
3 typedef struct _EFI_BTT_FLOG {
4     UINT32 Lba0;
5     UINT32 OldMap0;
6     UINT32 NewMap0;
7     UINT32 Seq0;
8     UINT32 Lba1;
9     UINT32 OldMap1;
10    UINT32 NewMap1;
11    UINT32 Seq1;
12 } EFI_BTT_FLOG
```

Lba0

最后一次使用此 flog entry(TODO) 写入 pre-map(TODO: 预制图) 的 LBA。在更新 BTT map 以完成交易时，这个值被用作 BTT 地图的索引。

OldMap0

旧的 post-map(TODO)LBA。这是 map 中的旧 entry，当最后一次使用这个 flog entry(TODO) 的写入发生时。如果交易完成，这个 LBA 现在是与这个 flog entry(TODO) 相关的自由块。

NewMap0

新的 post-map(TODO)LBA。最后一次使用该 flog entry 的写发生时分配的块。根据定义，如果 BTT map entry 包含这个值，那么一个写交易就完成了。

Seq0

每个 flog entry 中的 Seq0 字段被用来确定两组字段中哪一组较新 (Lba0, OldMap0, NewMap0, Seq0 vs Lba1, Oldmap1, NewMap1, Seq1)。对 flog entry 的更新应总是在较早的一组字段集进行，并应谨慎执行，以便 **Seq0** 位仅在已知其他字段被提交到 persistence(TODO) 后才被写入。下图显示了 **Seq0** 位随时间变化的进程，较新的 entry 由较旧的数值的顺时针方向表示。

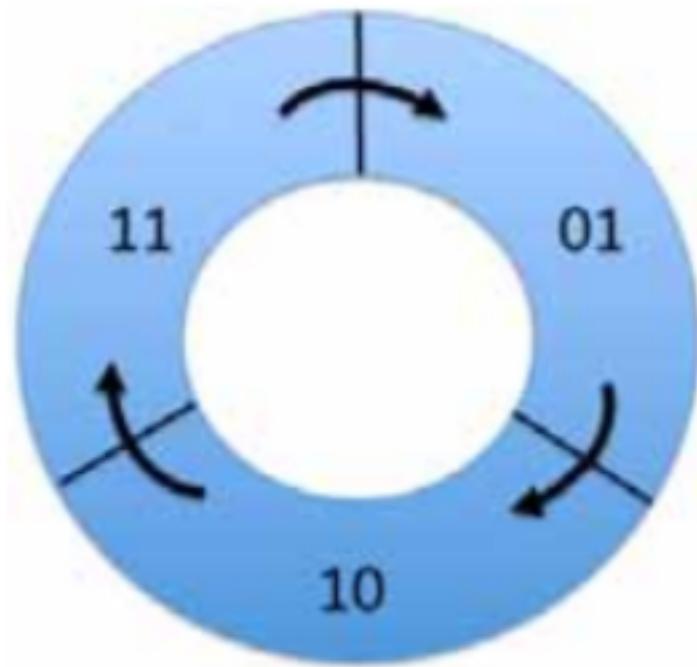


图 10. 循环序列号的 Flog entries

Lba1

备用 lba entry

OldMap1

替代旧 entry

NewMap1

备选新 entry

Seq1

替代序列 entry

BTT Flog Description

BTT Flog 如此命名是为了说明它既是一个自由列表，又是一个日志，被卷进了一个数据结构。Flog 的大小是由 BTT 信息块中的 **NFree** 字段决定的，它决定了有多少个这样的 flog entry。在考虑了备份信息块的空间和对齐要求后，flog 位置是场内的最高地址。

5.2.4 BTT 数据区

从低位地址开始到高位，BTT 数据区在 BTT 信息块之后立即开始，一直延伸到 BTT Map 数据结构的开始。一个区域可以存储的内部数据块的数量是通过以下方式计算的：首先计算 BTT 信息块、地图和 flog（加上任何需要的对齐）所需的必要空间，从总的区域大小中减去这个数量，然后计算有多少块适合于所产生的空间。

5.2.5 NVDIMM 标签协议地址抽象指南

这个版本的 BTT 布局和行为利用这个 GUID 由 UEFI NVDIMM 标签协议部分的 [AddressAbstractionGuid](#) 集体描述：

```
1 #define EFI_BTT_ABSTRACTION_GUID \
2 {0x18633bfc, 0x1735, 0x4217, \
3 {0x8a, 0xc9, 0x17, 0x23, 0x92, 0x82, 0xd3, 0xf8}}
```

5.3 BTT 操作理论

本节概述了 BTT 的操作理论，并描述了任何软件实现应遵循的责任。

BTT 布局的具体实例取决于命名空间的大小和在创建初始布局时的三个管理选择：

- **ExternalLbaSize:** 所需的区块大小
- **InternalLbaSize:** 块的大小与任何内部填充物
- **NFree:** 布局所支持的并发写的数据量

BTT 数据结构不支持小于 512 字节的 **InternalLbaSize**，所以如果 **ExternalLbaSize** 小于 512 字节，**InternalLbaSize** 应被四舍五入为 512。为了提高性能，**InternalLbaSize** 可能还包括一些填充字节。例如，一个支持 520 字节块的 BTT 布局可以在内部使用 576 字节的块，以便将大小四舍五入到 64 字节高速缓存行大小的

倍数。在这个例子中，对 BTT 软件上的软件来说，**ExternalLbaSize** 将是 520 字节，但 **InternalLbaSize** 将是 576 字节。

一旦确定了上述这些管理选择，命名空间就会被划分为区域，如 BTT 区域部分所述，每个区域对 **ExternalLbaSize**、**InternalLbaSize** 和 **Nfree** 使用相同的值。

5.3.1 BTT 区域

为了减少 BTT 元数据的大小，增加并发更新的可能性，命名空间中的 BTT 布局被划分为若干个区域。一个区域不能大于 512G 或小于 16M。一个命名空间被分为尽可能多的 512G 区域，从偏移量 0 开始，不需要填充就可以打包在一起，如果剩余的空间至少有 16M，那么后面还有一个小于 512G 的区域。如有必要，较小的区域大小将被四舍五入为 **EFI_BTT_ALIGNMENT** 的倍数。由于这些规则，命名空间中每个 BTT Arena 的位置和大小都可以从命名空间的大小中确定。

在一个区域内，用于 Flog 的空间量是每个 Flog entry 所需空间量的 **NFree** 倍。Flog entry 应在 64 字节的边界上对齐。

此外，完整的 BTT Flog 表应在 **EFI_BTT_ALIGNMENT** 边界上对齐，其大小被填充为 **EFI_BTT_ALIGNMENT** 的倍数。

```
1 FlogSize = roundup(NFree * roundup(sizeof(EFI_BTT_FLOG),
2 EFI_BTT_FLOG_ENTRY_ALIGNMENT), EFI_BTT_ALIGNMENT)
```

在一个区域内，数据块和相关 Map 的可用空间是区域大小减去用于 BTT 信息块和 Flog 的空间。

```
1 DataAndMapSize = ArenaSize - 2 * sizeof(EFI_BTT_INFO_BLOCK) - FlogSize
```

在一个区域内，数据块的数量是通过可用空间 **DataAndMapSize** 除以 **InternalLbaSize** 再加上每个块所需的 Map 开销来计算的，并对结果进行四舍五入以确保数据区在 **EFI_BTT_ALIGNMENT** 边界上对齐。

```
1 InternalNLba = (DataAndMapSize - EFI_BTT_ALIGNMENT) / (InternalLbaSize +
2 sizeof(EFI_BTT_MAP_ENTRY))
```

在知道 **InternalNLba** 值的情况下，计算外部 LBA 的数量时要减去未公布的自由块池的 **NFree**。

```
1 ExternalNLba = InternalNLba - Nfree
```

在一个区域内，BTT Map 所需的字节数为每个外部 LBA 的一个 entry，加上为保持整个 Map **EFI_BTT_ALIGNMENT** 对齐所需的任何对齐方式。

```
1 MapSize = roundup(ExternalNLba * sizeof(EFI_BTT_MAP_ENTRY),
2 EFI_BTT_ALIGNMENT)
```

一个区域允许的并发写的数量是基于 BTT 布局时选择的 **NFree** 值。例如，选择 256 的 **NFree** 意味着 BTT 区

域将有 256 个空闲块用于 `inflight(TODO)` 的写入操作。由于 BTT 区域中每个都有 **NFree** 空闲块，当有多个区域，且写入分散在多个区域之间时，一个命名空间允许的并发写入数量可能更大。

5.3.2 区域中数据块的原子性

BTT 的主要原因是为了解决在写入数据块时提供故障原子性，因此对单个数据块的任何写入都不会被断电等中断情况所撕裂。BTT 通过维护一个空闲块池来提供这种服务，这些空闲块不属于向 BTT 软件之上的软件层公布的容量。BTT 数据区足够大，可以容纳广告中的容量以及自由块池。BTT 软件将 BTT 数据区的块作为内部 LBAs 的列表进行管理，这些块的编号只在 BTT 软件内部可见。构成广告容量的块编号被称为外部 LBAs，在任何给定的时间点，这些外部 LBAs 中的每一个都被 BTT Map 映射到 BTT 数据区中的一个块。BTT 软件所做的每一个区块写入都是从分配一个空闲区块开始，将数据写入其中，只有当该区块完全 `persistent(TODO)`（包括任何需要的刷新）时，才会采取步骤使该区块处于活动状态，如[BTT Theory of Operations – Write Path](#) 部分所述。

BTT Flog（自由列表和日志的组合）是写块时原子更新的核心。在 BTT Flog 的“安静”状态下，当没有 `in-flight(TODO)` 中的写操作发生，也没有未完成的恢复步骤时，当前可用于写操作的 **NFree** 块包含在 Flog entries 的 OldMap 字段中。写入应使用这些 Flog entries 之一来寻找一个空闲的块来写入，然后 Flog 中的 Lba 和 NewMap 字段在写入的数据部分完成后被用作 BTT 地图更新的写前日志，如在[Validating the Flog at start-up](#) 部分所述。

由 BTT 软件的运行时逻辑来确保一次只使用一个 Flog entry，并且在开始使用该区块的写之前，仍在执行 OldMap entry 所指示的区块上的任何读都已完成。

5.3.3 BTT 数据结构的原子性

可字节寻址的 `persistent media(TODO)` 可能不支持大于 8 字节的原子更新，因此 BTT 中任何大于 8 字节的数据结构都使用软件实现的原子性进行更新。请注意，8 字节写的原子性，即 8 字节存储到 `persistent media(TODO)` 不会被中断，如断电，是使用本文件中描述的 BTT 的最低要求。

在 BTT 中，有四种数据结构：

- BTT 信息块
- BTT Map
- BTT Flog
- BTT 数据区

BTT Map entries 的大小为 4 字节，因此可以用一条存储指令原子式地更新。所有其他的数据结构都是按照本文描述的规则进行更新的，首先更新数据结构的非活动版本，然后是使其原子化的步骤。

对于 BTT 信息块来说，原子性是通过总是先写备份信息块来提供的，只有在该更新完全 `persistent(TODO)` 之

后（块的校验正确），主 BTT 信息块才会按照[Writing the initial BTT layout](#)部分的描述进行更新。对于 BTT Flog，每个 entry 都是双倍大小，每个字段都有两个完整的副本（Lba, OldMap, NewMap, Seq）。活跃的 entry 有更高的序列号，所以更新总是写到不活跃的字段，一旦这些字段完全 persistent，不活跃的 entry 的 Seq 字段就会被更新，使其成为活跃的 entry，这是原子性的。这将在[Validating the Flog at start-up](#)一节中描述。

对于 BTT 数据区，所有的区块写入可以被认为是分配写入，其中一个不活动的区块是从 Flog 维护的空闲列表中选择的，只有在写入该区块的新数据完全 persistent 之后，该区块才会通过更新 Flog 和 Map entry，如[Write Path](#)部分所述，以原子方式变成活动的。

5.3.4 编写初始 BTT 布局

BTT 的整体布局依赖于这样一个事实：除了最后一个区域最小为 16MB 外，所有区域的大小都应是 512GiB。在一个 BTT 的生命周期中，初始化 BTT 的 on-media(TODO) 结构只发生一次，即在它被创建时。这个序列假设软件已经确定需要创建新的 BTT 布局，并且已知命名空间的总原始大小。

在创建一个新的 BTT 布局之前，周围命名空间的 UUID 可以被更新为新生成的 UUID。根据 BTT 软件实施的需要，这一可选的步骤具有使命名空间中任何先前的 BTT 信息块无效的效果，并确保在 BTT 布局创建过程被中断时检测到无效的布局。因为父 UUID 字段，所以这种检测是有效的。

BTT 布局中的 on-media(TODO) 结构可以按任何顺序写出，但 BTT 信息块除外，BTT 信息块应作为布局的最后一步写出，从最后一个 arena（命名空间中的最高偏移量）开始到第一个 arena（命名空间中的最低偏移量），首先在每个 arena 中写入备份 BTT 信息块，然后为该 arena 写入主 BTT 信息块。这允许在执行[Validating BTT Arenas at start-up](#)部分的算法时检测出不完整的 BTT 布局。

由于一个区域内部 LBA 数量超过了 **NFree** 的外部 LBA 数量，所以有足够的内部 LBA 数量来完全初始化 BTT Map 以及 BTT Flog，其中 BTT Flog 用 **NFree** 最高的内部 LBA 数量初始化，其余的用于 BTT Map。

每个竞技场中的 BTT Map 被初始化为零。地图中的零 entry 表示所有 pre-map 的 LBAs 与相应的 post-map 的 LBAs 的身份映射。这使用了所有的内部 LBA，只有 **NFree** 的 LBA，留下 **Nfree** 的 LBA 供 BTT Flog 使用。

每个区域的 BTT Flog 的初始化方法是：整个 flog 区域从所有的零开始，将每个 flog 条目中的 Lba0 字段设置为唯一的 pre-map LBA，从 0 到 **Free-1**，每个 flog entry 中的 **OldMap0** 和 **NewMap0** 字段都设置为剩余的内部 LBA 之一。例如，flog entry 0 会将 Lba0 设置为 0，**OldMap0** 和 **NewMap0** 都设置为 Map 中没有代表的第一个内部 LBA（因为 Map 中存在 **ExternalNLbaentry**，下一个可用的内部 LBA 等于 **ExternalNLba**）。

5.3.5 启动时验证 BTT 区域

当软件准备访问命名空间中的 BTT 布局时，第一步是检查 BTT Arenas 的一致性。读取和验证 BTT 区域时依赖于这样一个事实，即所有区域的大小应是 512GiB，除了最后一个竞技场是最小的 16MiB。

在软件认为 BTT 布局有效之前，必须通过以下测试：

- 对于每一个 BTT 区域
 - 读取和验证主 BTT 信息块
 - * 如果对主 BTT 信息块读取失败，则转到读取和验证备份 BTT 信息块
 - * 如果主 BTT 信息块包含一个不正确的 **Sig** 字段，则为无效，转到读取和验证备用的 BTT 信息块
 - * 如果主 BTT 信息块的父级 **Uuid** 字段与周围命名空间的 UUID 不匹配，则转到读取和验证备用的 BTT 信息块
 - * 如果主 BTT 信息块包含一个不正确的 **Checksum(校验和)**，则为无效，转到读取和验证备用的 BTT 信息块
 - * 主 BTT 信息块有效。使用 **NextOff** 字段找到下一个区域的开始，继续 BTT 信息块的验证，转到读取和验证备用的 BTT 信息块
 - 读取和验证备份 BTT 信息块
 - * 确定备份 BTT 信息块的位置
 1. 所有区域都应是完整的 512G 数据区大小，除了最后一个区域至少是 16MB
 2. 备份 BTT 信息块是区域中最后一个 **EFI_BTT_ALIGNMENT** 对齐的块
 - * 如果在 BTT 区域的高地址上读取备份 BTT 信息块失败，**neither copy could be read**(则两个副本都无法读取)，软件应假定命名空间没有有效的 BTT 元数据布局
 - * 如果备份的 BTT 信息块父级 **Uuid** 字段与周围命名空间的 UUID 不匹配，则为无效，软件应假定该命名空间没有有效的 BTT 元数据布局。
 - * 如果备份的 BTT 信息块包含一个不正确的 **Checksum(校验和)**，则为无效，软件应假定命名空间没有有效的 BTT 元数据布局
 - * 备份 BTT 信息块是有效的。由于主副本是坏的，软件应将有效的备份 BTT 信息块的内容复制到主 BTT 信息块中，然后才能成功地完成对所有区域的 BTT 信息块的验证

5.3.6 启动时验证 Flog Entry

在按照 [Validating BTT Arenas at start-up](#) 一节所述验证 BTT 信息块后，软件应采取的下一步是验证 BTT Flog entry。当数据块被写入时，如下面 [Write Path](#) 部分所述，**persistent(持久的)** Flog 和 Map 状态不会被更新，直到空闲块被写入新数据。将部分写入的数据保留在一个空闲的块中，这确保了数据传输过程中任何时候的电源故障都是无害的。一旦 Flog 被更新（通过 Flog entry 中的 **Seq** 位使之成为原子），写入算法就致力于更新，从写入流程中的这一点开始，电源故障应通过在恢复时完成对 BTT Map 的更新来处理。Flog 包含完成 Map entry 更新所需的所有信息。

请注意，这里概述的 Flog entry 的恢复是为了在不活动的 BTT 上单线程发生（在 BTT 块命名空间被允许接受 I/O 请求之前）。恢复所需的最大时间由 **NFree** 决定，但对于发现的每个不完整的写，只需要几次加载和一次存储（以及相应的缓存刷新）。

针对每个区域中的每个 flog entry 执行以下步骤，以恢复任何中断的写入，并验证 flog entry 在启动时是否一致。在这些步骤中发现的任何一致性问题都会导致为区域设置错误状态 ([EFI_BTT_INFO_BLOCK_FLAGS_ERROR](#)) 并终止该区域的 flog 验证过程。

1. 针对 flog entry 检查 **Seq0** 和 **Seq1** 字段。如果两个字段都为零，或者两个字段彼此相等，则 flog entry 不一致。否则，较高的 Seq 字段表明在接下来的步骤中使用哪一组 flog 字段 (**Lba0**, **OldMap0**, **NewMap0** 与 **Lba1**, **OldMap1**, **NewMap1**)。从本节的这一点开始，所选字段被称为 **Lba**、**OldMap** 和 **NewMap**
2. 如果 **OldMap** 和 **NewMap** 相等，这就是一个自 BTT 的初始布局创建以来从未使用过的 Flog entry；
3. **Lba** 字段被检查以确保它是一个有效的 pre-map(预映射) LBA（范围在 0 到 **External NLba 1**）。如果检查失败，flog entry 就不一致了；
4. 获取对应 Flog entry Lba 字段中的 BTT Map Entry。由于 Map 中可包含特殊的 Zero Entry 以指示身份映射，因此当遇到零时，将获取的 Entry 调整为相应的内部 LBA（通过将 Entry 解释为与 Flog Entry LBA 字段相同的 LBA）；
5. 如果上一步调整后的 Map Entry 与 Flog Entry 中的 NewMap 字段不匹配，并且与 OldMap 字段匹配，则检测到 BTT Map 更新中断。恢复步骤是将 NewMap 字段写入 Flog entry Lba 字段索引的 BTT Map entry。

5.3.7 读取路径

以下高级序列描述了在使用 BTT 时读取单个数据块的步骤，如图所示：BTT 读取路径概述如下：

1. 如果在 Arena 的 BTT 信息块中设置了 [EFI_BTT_INFO_BLOCK_FLAGS_ERROR](#)，BTT 软件可能会返回读取错误，或者实现可能会选择继续提供只读访问并继续这些步骤；
2. 使用随读取操作提供的外部 LBA 来确定要访问的 BTT Arena。从第一个 Arena（命名空间中的最低偏移量）开始，依次循环遍历 Arena，BTT 信息块中的 **ExternalNLba** 字段描述了该区域中有多少外部 LBA。一旦确定了正确的 Arena，就从提供的 LBA 中减去包含在较低、跳过的 Arena 中的外部 LBA，以获得所选 Arena 的 pre-map LBA；
3. 使用 pre-map LBA 索引到 Arena 的 BTT Map 并获取 Map Entry；
4. 如果在映射条目中设置了零位和错误位，则这表示正常 Entry。Map Entry 中的 **PostMapLba** 字段用于通过将其乘以 **InternalLbaSize** 并将结果添加到来自 Arena 的 BTT 信息块的 **DataOff** 字段来索引到 Arena 数据区域。这提供了数据在 Arena 中的位置，然后软件将 **ExternalLbaSize** 字节复制到提供的缓冲区中以满足读取请求；

5. 否则, 如果仅在 Map Entry 中设置了错误位, 则返回读取错误;
6. 否则, 如果在 Map Entry 中仅设置了零位, 则会将一块 **ExternalLbaSize** 字节的零复制到提供的缓冲区中以满足读取请求;
7. 最后, 如果零位和错误位都被清除, 这表示初始标识映射, 并且 Pre-Map (预映射) LBA 用于通过将其乘以 **InternalLbaSize** 并将结果添加到竞技场 BTT 的 **DataOff** 字段来索引到 Arena 数据区域信息块。这提供了数据在 Arena 中的位置, 然后软件将 **ExternalLbaSize** 字节复制到提供的缓冲区中以满足读取请求。

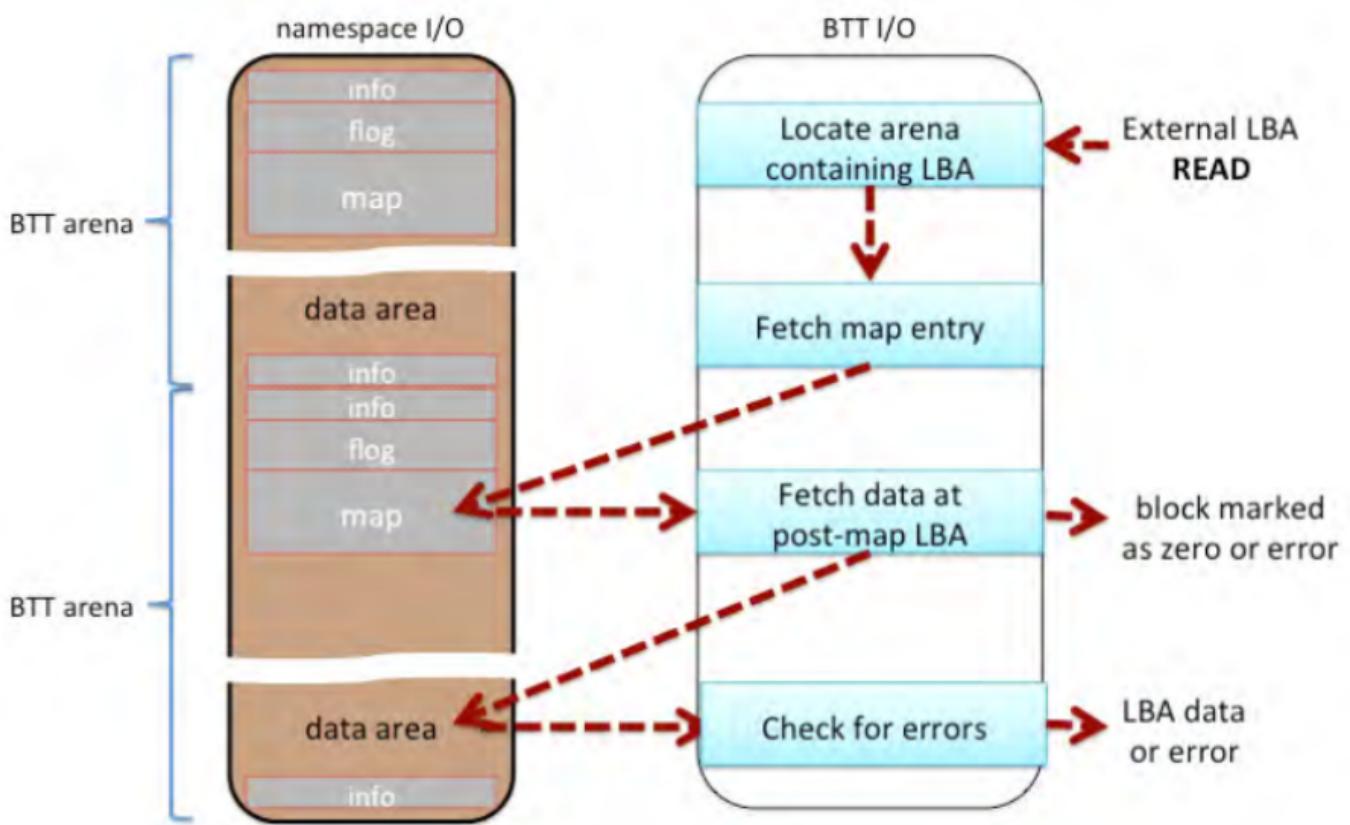


Figure 6-4 BTT Read Path Overview

图 11. BTT Read Path Overview

5.3.8 写入路径

以下高级序列描述了在使用 BTT 时写入单个数据块的步骤，如下图所示：BTT 写入路径概述：

1. 如果在 Arena 的 BTT 信息块中设置了 `EFI_BTT_INFO_BLOCK_FLAGS_ERROR`，则 BTT 软件将返回写入错误；
2. 使用随写入操作提供的外部 LBA 来确定要访问的 BTT Arena。从第一个 Arena（命名空间中的最小偏移）开始，依次循环遍历 Arena，BTT 信息块中的 **ExternalNLba** 字段描述了该区域中有多少外部 LBA。一旦识别出正确的 Arena，就从提供的 LBA 中减去包含在较低、跳过的 Arena 中的外部 LBA，以获得所选 Arena 的地图前 LBA。
3. BTT 软件在 Arena 中分配一个 Flog Entry 用于此写入。多个并发写入不能共享 Flog Entry。管理 Flog Entry 的独占使用的确切方法取决于 BTT 软件实现。媒体上没有关于 Flog Entry 当前是否分配给写入请求的指示。请注意，**OldMap** 字段中的 Flog Entry 跟踪的空闲块可能仍然有来自运行在其上的相对较慢的线程的读取。BTT 软件实施应确保在进行下一步之前已完成任何此类读取。
4. 在接下来的三个步骤中，锁定对与 Pre-Map（预映射）LBA 相关联的 BTT Map 区域的访问。锁定的粒度取决于实现；一个实现可以选择锁定单个 Map Entry、锁定整个 BTT Map 或介于两者之间的东西。
5. 使用 Pre-Map LBA 索引到 Arena 的 BTT Map 并获取旧地图 Entry。
6. 通过写入非活动的 Flog 字段集（较低的 Seq 编号）来更新 Flog Entry。首先，分别用 Pre-Map LBA、旧 Map Entry 和上面选择的空闲块更新 **LBA**、**OldMap** 和 **NewMap** 字段。一旦这些字段完全持久化（完成任何所需的刷新），**Seq** 字段就会更新，以使新字段处于活动状态。**Seq** 字段的此更新提交写入 - 在此更新之前，如果操作中断，则写入不应发生。在 **Seq** 字段更新后，即使操作中断，也应进行写入，因为下一步中的 Map 更新将在启动时发生的 BTT 恢复期间进行。
7. 使用上面选择的空闲块更新 Map Entry。
8. 删除在上述步骤 4 中获取的地图锁。写请求现在得到满足。

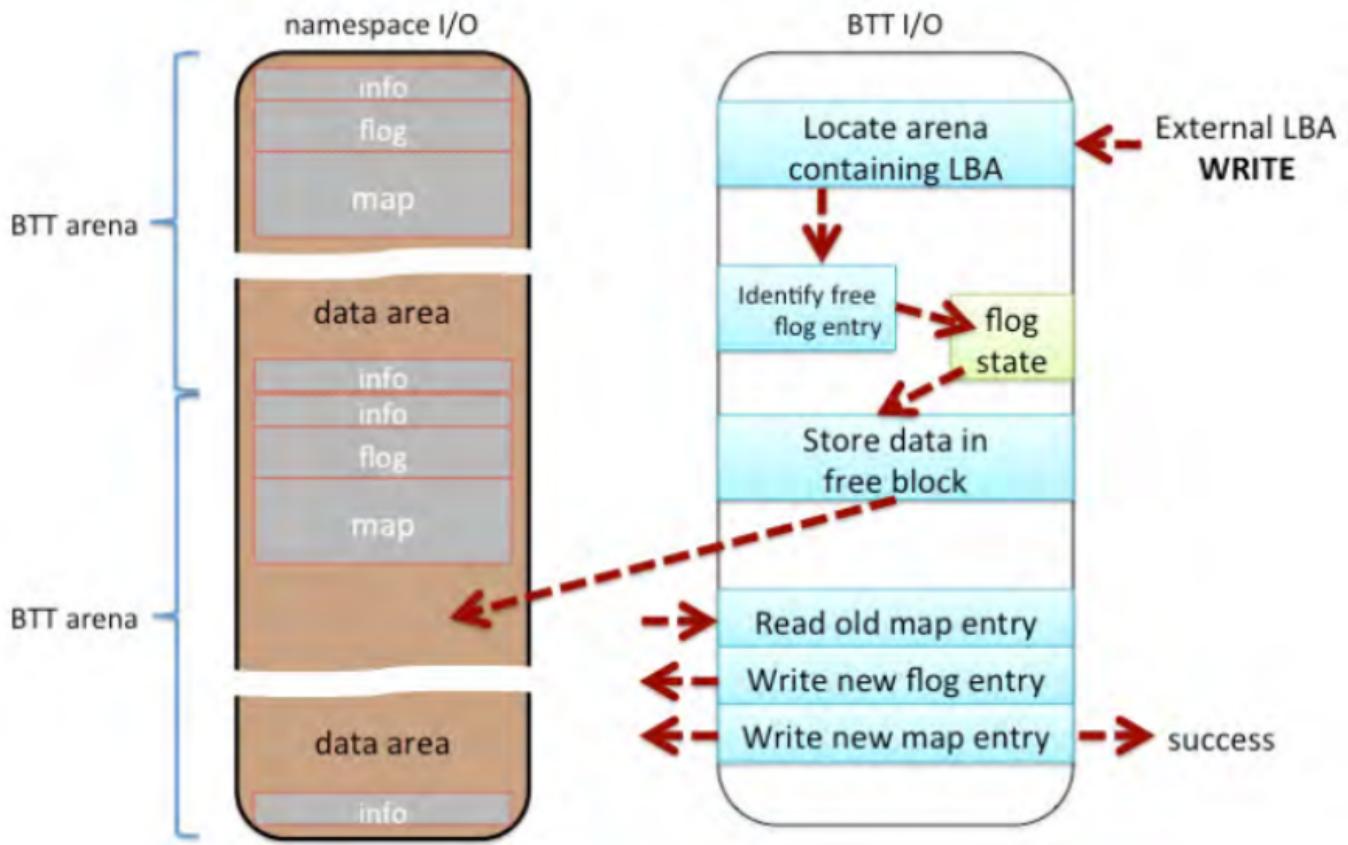
**Figure 6-5 BTT Write Path Overview**

图 12. BTT Write Path Overview

6 引导服务

本节讨论 UEFI 兼容系统中存在的基本引导服务。这些服务可由在 UEFI 环境中运行的代码使用的接口函数定义。此类代码可能包括管理设备访问或扩展平台功能的协议，以及在预引导环境中运行的应用程序和操作系统加载程序。

适用于合规系统的两种类型服务：

- 引导服务 (Boot Services)

在成功调用之前可用的函数 `EFI_BOOT_SERVICES.ExitBootServices()`。本节介绍了这些功能。

- 运行时服务 (Runtime Services)

在任何调用之前和之后可用的函数 `ExitBootServices()`。这些功能在第 8 节中描述。

在引导期间，系统资源归固件所有，并通过引导服务接口函数进行控制。这些功能可以被描述为“全局”或“基于句柄”。术语“全局”仅仅意味着一个函数访问系统服务并且在所有平台上都可用（因为所有平台都支持所有系统服务）。术语“基于句柄”是指函数访问一个特定设备或设备功能，并且可能在某些平台上不可用（因为某些设备在某些平台上不可用）。协议是动态创建的。本节讨论“全局”函数和运行时函数；后续章节讨论“基于句柄”。

UEFI 应用程序（包括 UEFI OS 加载程序）必须使用引导服务功能来访问设备和分配内存。进入时，`Image` 会提供一个指向系统表的指针，该系统表包含引导服务调度表和用于访问控制台的默认句柄。所有引导服务功能都是直到 UEFI OS 加载程序加载足够的自身环境以控制系统继续操作，然后通过调用 `ExitBootServices()` 终止引导服务。

原则上，`ExitBootServices()` 调用旨在供操作系统使用，以指示其加载程序已准备好承担对平台和所有平台资源管理的控制。因此，引导服务可用于协助 UEFI OS 加载程序准备引导操作系统。一旦 UEFI OS 加载程序控制系统并完成操作系统引导过程，就只能调用运行时服务。但是，UEFI OS 加载程序以外的代码可能会也可能不会选择调用 `ExitBootServices()`。这种选择可能部分取决于此类代码是否旨在继续使用引导服务或引导服务环境。

本节的其余部分将讨论各个功能。全局引导服务功能分为以下几类：

- 事件、计时器和任务优先级服务 (Event, Timer, and Task Priority Services) (第 7.1 节)
- 内存分配服务 (Memory Allocation Services)(第 7.2 节)
- 协议处理程序服务 (Protocol Handler Services)(第 7.3 节)
- 镜像服务 (Image Services)(第 7.4 节)
- 杂项服务 (Miscellaneous Services)(第 7.5 节)

6.1 事件、计时器和任务优先级服务

构成事件、定时器和任务优先级服务的函数在预引导期间用于创建、关闭、发出信号和等待事件；设置定时器；并提高和恢复任务优先级。见表 7-1。

Name	Type	Description
CreateEvent	Boot	Creates a general-purpose event structure
CreateEventEx	Boot	Creates an event structure as part of an event group
CloseEvent	Boot	Closes and frees an event structure
SignalEvent	Boot	Signals an event
WaitForEvent	Boot	Stops execution until an event is signaled
CheckEvent	Boot	Checks whether an event is in the signaled state
SetTimer	Boot	Sets an event to be signaled at a particular time
RaiseTPL	Boot	Raises the task priority level
RestoreTPL	Boot	Restores/lowers the task priority level

图 13. table7-1 Event, Timer, and Task Priority Functions

引导服务环境中的执行发生在不同的任务优先级或 TPL³ 上。引导服务环境仅向 UEFI 应用程序和驱动程序公开其中三个级别：

- TPL APPLICATION：最低优先级
- TPL CALLBACK：中等优先级
- TPL NOTIFY：最高优先级

以较高优先级执行的任务可能会中断以较低优先级执行的任务。例如，在 TPL_NOTIFY 级别运行的任务可能会中断在 TPL_APPLICATION 或 TPL_CALLBACK 级别运行的任务。而 TPL_NOTIFY 是暴露给引导服务的最高级别应用程序中，固件可能具有更高的任务优先级项目。例如，固件可能必须处理更高优先级的任务，如计时器滴答声（Timer Ticks）和内部设备。因此，有第四个 TPL， TPL_HIGH_LEVEL，专为固件使用而设计。

优先级的预期用途如表 7-2 所示，从最低级别 (TPL_APPLICATION) 到最高级别 (TPL_HIGH_LEVEL)。随着级别的增加，代码的持续时间和允许的阻塞量减少。执行通常发生在 TPL_APPLICATION 级别。作为触发事件通知功能的直接结果（这通常是由事件的信号引起的），执行发生在其他级别。在定时器中断期间，固件会在事件的“触发时间（Trigger Time）”到期时发出定时器事件信号。这允许事件通知功能中断较低优先级的代码以检查设备（例如）。通知功能可以根据需要通知其他事件。在所有未决事件通知函数执行后，在 TPL_APPLICATION 级别继续执行。

³TPL(Task Priority Levels): 任务优先级

Task Priority Level	Usage
---------------------	-------

图 14. table7-2-1TPL Usage

TPL_APPLICATION	This is the lowest priority level. It is the level of execution which occurs when no event notifications are pending and which interacts with the user. User I/O (and blocking on User I/O) can be performed at this level. The boot manager executes at this level and passes control to other UEFI applications at this level.
TPL_CALLBACK	Interrupts code executing below TPL_CALLBACK level Long term operations (such as file system operations and disk I/O) can occur at this level.
TPL_NOTIFY	Interrupts code executing below TPL_NOTIFY level Blocking is not allowed at this level. Code executes to completion and returns. If code requires more processing, it needs to signal an event to wait to obtain control again at whatever level it requires. This level is typically used to process low level IO to or from a device.
(Firmware Interrupts)	This level is internal to the firmware It is the level at which internal interrupts occur. Code running at this level interrupts code running at the TPL_NOTIFY level (or lower levels). If the interrupt requires extended time to complete, firmware signals another event (or events) to perform the longer term operations so that other interrupts can occur.
TPL_HIGH_LEVEL	Interrupts code executing below TPL_HIGH_LEVEL This is the highest priority level. It is not interruptible (interrupts are disabled) and is used sparingly by firmware to synchronize operations that need to be accessible from any priority level. For example, it must be possible to signal events while executing at any priority level. Therefore, firmware manipulates the internal event structure while at this priority level.

图 15. table7-2-2TPL Usage

执行代码可以通过调用 `EFI_BOOT_SERVICES` 临时提高其优先级。调用 `RaiseTPL()` 函数。这样做会屏蔽以相同或更低优先级运行的代码的事件通知，直到 `EFI_BOOT_SERVICES`。调用 `RestoreTPL()` 函数以将优先级降低到低于未决事件通知的级别。许多 UEFI 服务功能和协议接口功能可以执行的 TPL 级别存在限制。表 7-3 总结了这些限制。

Name	Restrictions	Task Priority Level
ACPI Table Protocol	<	TPL_NOTIFY
ARP	<=	TPL_CALLBACK
ARP Service Binding	<=	TPL_CALLBACK
Authentication Info	<=	TPL_NOTIFY
Block I/O Protocol	<=	TPL_CALLBACK
Block I/O 2 Protocol	<=	TPL_CALLBACK
Bluetooth Host Controller	<=	TPL_CALLBACK
Bluetooth IO Service Binding	<=	TPL_CALLBACK
Bluetooth IO	<=	TPL_CALLBACK
Bluetooth Attribute	<=	TPL_CALLBACK
Bluetooth Configuration	<=	TPL_CALLBACK
Bluetooth LE Configuration	<=	TPL_CALLBACK

图 16. table7-3-1TPL Restrictions

Name	Restrictions	Task Priority Level
CheckEvent()	<	TPL_HIGH_LEVEL
CloseEvent()	<	TPL_HIGH_LEVEL
CreateEvent()	<	TPL_HIGH_LEVEL
Deferred Image Load Protocol	<=	TPL_NOTIFY
Device Path Utilities	<=	TPL_NOTIFY
Device Path From Text	<=	TPL_NOTIFY
DHCP4 Service Binding	<=	TPL_CALLBACK
DHCP4	<=	TPL_CALLBACK
DHCP6	<=	TPL_CALLBACK
DHCP6 Service Binding	<=	TPL_CALLBACK
Disk I/O Protocol	<=	TPL_CALLBACK
Disk I/O 2 Protocol	<=	TPL_CALLBACK
DNS4 Service Binding	<=	TPL_CALLBACK
DNS4	<=	TPL_CALLBACK
DNS6 Service Binding	<=	TPL_CALLBACK
DNS6	<=	TPL_CALLBACK
Driver Health	<=	TPL_NOTIFY
EAP	<=	TPL_CALLBACK
EAP Configuration	<=	TPL_CALLBACK
EAP Management	<=	TPL_CALLBACK
EAP Management2	<=	TPL_CALLBACK
EDID Active	<=	TPL_NOTIFY
EDID Discovered	<=	TPL_NOTIFY
EFI_SIMPLE_TEXT_INPUT_PROTOCOL	<=	TPL_CALLBACK
EFI_SIMPLE_TEXT_INPUT_PROTOCOL.ReadKeyStroke	<=	TPL_APPLICATION
EFI_SIMPLE_TEXT_INPUT_PROTOCOL.Reset	<=	TPL_APPLICATION
EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL	<=	TPL_CALLBACK
EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.ReadKeyStrokeEx	<=	TPL_APPLICATION
EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.Reset	<=	TPL_APPLICATION
Event Notification Levels	> <=	TPL_APPLICATION TPL_HIGH_LEVEL
Exit()	<=	TPL_CALLBACK
ExitBootServices()	=	TPL_APPLICATION
Form Browser2 Protocol/SendForm	=	TPL_APPLICATION

Name	Restrictions	Task Priority Level
FTP	<=	TPL_CALLBACK
Graphics Output EDID Override	<=	TPL_NOTIFY
HII Protocols	<=	TPL_NOTIFY
HTTP Service Binding	<=	TPL_CALLBACK
HTTP	<=	TPL_CALLBACK
HTTP Utilities	<=	TPL_CALLBACK
IP4 Service Binding	<=	TPL_CALLBACK
IP4	<=	TPL_CALLBACK
IP4 Config	<=	TPL_CALLBACK
IP4 Config2	<=	TPL_CALLBACK
IP6	<=	TPL_CALLBACK
IP6 Config	<=	TPL_CALLBACK
IPSec Configuration	<=	TPL_CALLBACK
iSCSI Initiator Name	<=	TPL_NOTIFY
LoadImage()	<	TPL_CALLBACK
Managed Network Service Binding	<=	TPL_CALLBACK
Memory Allocation Services	<=	TPL_NOTIFY
MTFTP4 Service Binding	<=	TPL_CALLBACK
MTFTP4	<=	TPL_CALLBACK
MTFTP6	<=	TPL_CALLBACK
MTFTP6 Service Binding	<=	TPL_CALLBACK
PXE Base Code Protocol	<=	TPL_CALLBACK
Protocol Handler Services	<=	TPL_NOTIFY
REST	<=	TPL_CALLBACK
Serial I/O Protocol	<=	TPL_CALLBACK
SetTimer()	<	TPL_HIGH_LEVEL
SignalEvent()	<=	TPL_HIGH_LEVEL
Simple File System Protocol	<=	TPL_CALLBACK
Simple Network Protocol	<=	TPL_CALLBACK
Simple Text Output Protocol	<=	TPL_NOTIFY
Stall()	<=	TPL_HIGH_LEVEL
StartImage()	<	TPL_CALLBACK
Supplicant	<=	TPL_CALLBACK
Tape IO	<=	TPL_NOTIFY
TCP4 Service Binding	<=	TPL_CALLBACK
TCP4	<=	TPL_CALLBACK
TCP6	<=	TPL_CALLBACK

Name	Restrictions	Task Priority Level
TCP6 Service Binding	<=	TPL_CALLBACK
Time Services	<=	TPL_CALLBACK
TLS Service Binding	<=	TPL_CALLBACK
TLS	<=	TPL_CALLBACK
TLS Configuration	<=	TPL_CALLBACK
UDP4 Service Binding	<=	TPL_CALLBACK
UDP4	<=	TPL_CALLBACK
UDP6	<=	TPL_CALLBACK
UDP6 Service Binding	<=	TPL_CALLBACK
UnloadImage()	<=	TPL_CALLBACK
User Manager Protocol	<=	TPL_NOTIFY
User Manager Protocol/Identify()	=	TPL_APPLICATION
User Credential Protocol	<=	TPL_NOTIFY
User Info Protocol	<=	TPL_NOTIFY
Variable Services	<=	TPL_CALLBACK
VLAN Configuration	<=	TPL_CALLBACK
WaitForEvent()	=	TPL_APPLICATION
Wireless MAC Connection	<=	TPL_CALLBACK
Other protocols and services, if not listed above	<=	TPL_NOTIFY

图 19. table7-3-4TPL Restrictions

EFI_BOOT_SERVICES.CreateEvent()

• 概要 (Summary)

创建一个事件。

• 原型 (Prototype)

```

1  typedef EFI_STATUS (EFIAPI *EFI_CREATE_EVENT) (
2      IN UINT32          Type,
3      IN EFI_TPL          NotifyTpl,
4      IN EFI_EVENT_NOTIFY NotifyFunction, OPTIONAL
5      IN VOID             *NotifyContext, OPTIONAL

```

```

6     OUT EFI_EVENT      *Event
7 );

```

- 参数 (Parameters)

Type: 要创建的事件类型及其模式和属性。

NotifyTpl: 事件通知的任务优先级, 如果需要。请参阅 [EFI_BOOT_SERVICES.RaiseTPL\(\)](#)。

NotifyFunction: 指向事件通知函数的指针, 如果有。

NotifyContext: 指向通知函数上下文的指针; 对应通知函数中的参数 *Context*。

EventGroup: 指向此事件所属组的唯一标识符的指针。如果这是 `NULL`, 那么函数的行为就好像参数被传递给了 [CreateEvent](#)。

Event: 如果调用成功, 则指向新创建的事件; 否则未定义。

- 相关定义 (Related Definitions)

```

1 //*****
2 // EFI_EVENT
3 //*****
4 typedef VOID *EFI_EVENT
5 //*****
6 // Event Types
7 //*****
8 // These types can be "Ored" together as needed - for example,
9 // EVT_TIMER might be "Ored" with EVT_NOTIFY_WAIT or
10 // EVT_NOTIFY_SIGNAL.
11 #define EVT_TIMER          0x80000000
12 #define EVT_RUNTIME        0x40000000
13 #define EVT_NOTIFY_WAIT    0x000000100
14 #define EVT_NOTIFY_SIGNAL   0x000000200
15 #define EVT_SIGNAL_EXIT_BOOT_SERVICES 0x000000201
16 #define EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE 0x600000202

```

EVT_TIMER: 该事件是一个计时器事件, 可以传递给 [EFI_BOOT_SERVICES.SetTimer\(\)](#)。请注意, 计时器仅在引导服务期间起作用。

EVT_RUNTIME: 该事件是从运行时内存中分配的。如果要在调用 [EFI_BOOT_SERVICES](#) 后发出事件信号。[ExitBootServices\(\)](#) 事件的数据结构和通知函数需要从运行时内存中分配。有关详细信息, 请参阅 [SetVirtualAddressMap\(\)](#)。

EVT_NOTIFY_WAIT: 如果此类型的事件尚未处于信号状态, 则每当通过等待事件时, 事件的 *NotificationFunction* 将在事件的 *NotifyTpl* 排队 [EFI_BOOT_SERVICES.WaitForEvent\(\)](#) 或 [EFI_BOOT_SERVICES.CheckEvent\(\)](#)。

EVT_NOTIFY_SIGNAL：每当事件发出信号时，事件的 `NotifyFunction` 就会排队。

EVT_SIGNAL_EXIT_BOOT_SERVICES：此事件的类型为 `EVT_NOTIFY_SIGNAL`。它不应与任何其他事件类型结合使用。此事件类型在功能上等同于 `EFI_EVENT_GROUP_EXIT_BOOT_SERVICES` 事件组。参考 `EFI_EVENT_GROUP_EXIT_BOOT_SERVICES` 事件组 `EFI_BOOT_SERVICES.CreateEventEx()` 部分中的描述下面的更多细节。

EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE：执行 `SetVirtualAddressMap()` 时，系统将通知该事件。此事件类型是 `EVT_NOTIFY_SIGNAL`、`EVT_RUNTIME` 和 `EVT_RUNTIME_CONTEXT` 的组合，不应与任何其他事件类型组合。

```
1 //*****
2 // EFI_EVENT_NOTIFY
3 //*****
4 typedef
5 VOID
6 (EFIAPI *EFI_EVENT_NOTIFY) (
7     IN EFI_EVENT Event,
8     IN VOID *Context
9 );
```

`Event`: 正在调用其通知功能的事件。

`Context`: 指向通知函数上下文的指针，它依赖于实现。`Context` 对应于 `EFI_BOOT_SERVICES.CreateEventEx()` 中的 `NotifyContext`。

- 描述 (Description)

`CreateEvent()` 函数创建一个 `Type` 类型的新事件并将其返回到 `Event` 引用的位置。事件的通知函数、上下文和任务优先级分别由 `NotifyFunction`、`NotifyContext` 和 `NotifyTlp` 指定。事件存在于两种状态之一，“等待”或“信号”。创建事件后，固件会将其置于“等待”状态。当事件发出信号时，固件将其状态更改为“已发出信号”，如果指定了 `EVT_NOTIFY_SIGNAL`，则将对其通知函数的调用置于 FIFO 队列中。在第 7.1 节 (`TPL_CALLBACK` 和 `TPL_NOTIFY`) 中定义的每个“基本”任务优先级都有一个队列。这些队列中的函数按 FIFO 顺序调用，从最高优先级队列开始，然后到当前 TPL 未屏蔽的最低优先级队列。如果当前 TPL 等于或大于排队的通知，它将等到通过 `EFI_BOOT_SERVICES.RestoreTlp()` 降低 TPL。

一般来说，有两种“类型”的事件，同步和异步。异步事件与计时器密切相关，用于支持程序执行的周期性或定时中断。此功能通常用于设备驱动程序。例如，需要轮询新数据包存在的网络设备驱动程序可以创建一个类型包括 `EVT_TIMER` 的事件，然后调用 `EFI_BOOT_SERVICES.SetTimer()` 函数。定时器到期时，固件发出事件信号。

同步事件与定时器没有特别的关系。相反，它们用于确保在调用特定接口函数之后发生某些活动。其中一个示例是需要执行的清理以响应对 `EFI_BOOT_SERVICES.ExitBootServices()` 的调用功能。

`ExitBootServices()` 可以清理固件，因为它了解固件内部结构，但它不能代表已加载到系统中的驱动程序进行清理。驱动程序必须通过创建一个类型为 `EVT_SIGNAL_EXIT_BOOT_SERVICES` 且其通知函数是驱动程序本身内的函数的事件来自执行此操作。然后，当 `ExitBootServices()` 完成清理时，它会向每个 `EVT_SIGNAL_EXIT_BOOT_SERVICES` 类型的事件发出信号。

当 `EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE` 类型的事件与 `SetVirtualAddressMap()` 一起使用时，会发生使用同步事件的另一个示例。

`EVT_NOTIFY_WAIT` 和 `EVT_NOTIFY_SIGNAL` 标志是互斥的。如果未指定任何标志，则调用者不需要任何有关事件的通知，并且 `NotifyTpl`、`NotifyFunction` 和 `NotifyContext` 参数将被忽略。如果指定了 `EVT_NOTIFY_WAIT` 并且事件未处于信号状态，则只要事件的使用者正在等待事件（通过 `EFI_BOOT_SERVICES.WaitForEvent()` 或 `EFI_BOOT_SERVICES.CheckEvent()`），`EVT_NOTIFY_WAIT` 通知函数就会排队。如果指定了 `EVT_NOTIFY_SIGNAL` 标志，那么只要发出事件信号，事件的通知函数就会排队。

注：因为调用者不知道它的内部结构，所以调用者不能修改 `Event`。操作它的唯一方法是使用已发布的事件接口。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The event structure was created.
<code>EFI_INVALID_PARAMETER</code>	One of the parameters has an invalid value.
<code>EFI_INVALID_PARAMETER</code>	<code>Event</code> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<code>Type</code> has an unsupported bit set.
<code>EFI_INVALID_PARAMETER</code>	<code>Type</code> has both <code>EVT_NOTIFY_SIGNAL</code> and <code>EVT_NOTIFY_WAIT</code> set.
<code>EFI_INVALID_PARAMETER</code>	<code>Type</code> has either <code>EVT_NOTIFY_SIGNAL</code> or <code>EVT_NOTIFY_WAIT</code> set and <code>NotifyFunction</code> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<code>Type</code> has either <code>EVT_NOTIFY_SIGNAL</code> or <code>EVT_NOTIFY_WAIT</code> set and <code>NotifyTpL</code> is not a supported TPL level.
<code>EFI_OUT_OF_RESOURCES</code>	The event could not be allocated.

图 20. table7-3_0

`EFI_BOOT_SERVICES.CreateEventEx()`

- 概要 (Summary)

在组中创建事件。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_CREATE_EVENT_EX) (
2     IN UINT32           Type,
3     IN EFI_TPL          NotifyTpl,
4     IN EFI_EVENT_NOTIFY NotifyFunction OPTIONAL,
5     IN CONST VOID        *NotifyContext OPTIONAL,
6     IN CONST EFI_GUID    *EventGroup OPTIONAL,
7     OUT EFI_EVENT        *Event
8 );
```

- 参数 (Parameters)

Type: 要创建的事件类型及其模式和属性。

NotifyTpl: 事件通知的任务优先级 (如果需要)。请参阅 [EFI_BOOT_SERVICES.RaiseTPL\(\)](#)。

NotifyFunction: 指向事件通知函数的指针 (如果有)。

NotifyContext: 指向通知函数上下文的指针；对应于通知函数中的参数 *Context*。

EventGroup: 指向此事件所属组的唯一标识符的指针。如果这是 `NULL`，那么函数的行为就好像参数被传递给了 [CreateEvent](#)。

Event: 如果调用成功，则指向新创建的事件；否则未定义。

- 描述 (Description)

[CreateEventEx](#) 函数创建一个 *Type* 类型的新事件，并将其返回到 *Event* 指示的指定位置。事件的通知函数、上下文和任务优先级分别由 *NotifyFunction*、*NotifyContext* 和 *NotifyTpl* 指定。该事件将被添加到由 *EventGroup* 标识的事件组中。

事件组是由共享 `EFI_GUID` 标识的事件的集合，其中，当一个成员事件发出信号时，所有其他事件都会发出信号并执行它们各自的通知操作 (如 [CreateEvent](#) 中所述)。保证在采取第一个通知操作之前发出所有事件。所有通知函数将按照其 *NotifyTpl* 指定的顺序执行。

单个事件只能是单个事件组的一部分。可以使用 [CloseEvent](#) 从事件组中删除事件。

事件的类型使用与 [CreateEvent](#) 中定义的值相同的值，除了 `EVT_SIGNAL_EXIT_BOOT_SERVICES` 和 `EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE` 无效。

如果 *Type* 具有 `EVT_NOTIFY_SIGNAL` 或 `EVT_NOTIFY_WAIT`，则 *NotifyFunction* 必须为非空且 *NotifyTpl* 必须是有效的任务优先级。否则这些参数将被忽略。

多个 `EVT_TIMER` 类型的事件可能是单个事件组的一部分。然而，没有机制可以确定哪个定时器被发送信号。

- 配置表组 (Configuration Table Groups)

配置表的 `GUID` 还定义了具有相同值的相应事件组 `GUID`。如果配置表表示的数据已更改，则应调用 `InstallConfigurationTable()`。调用 `InstallConfigurationTable()` 时，会发出相应的事件信号。当发出此事件的信号时，从配置表中缓存信息的任何组件都可以选择更新其缓存状态。

例如，`EFI_ACPI_TABLE_GUID` 定义 ACPI 数据的配置表。更改 ACPI 数据时，将调用 `InstallConfigurationTable()`。在执行 `InstallConfigurationTable()` 的过程中，会发出带有 `EFI_ACPI_TABLE_GUID` 的相应事件组的信号，允许应用程序使任何缓存的 ACPI 数据无效。

- 预定义事件组 (Pre-Defined Event Groups)

本节介绍 UEFI 规范使用的预定义事件组。

- `EFI_EVENT_GROUP_EXIT_BOOT_SERVICES`: 在通知 `EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES` 事件组后调用 `ExitBootServices()` 时，系统会通知该事件组。此事件组在功能上等同于 `CreateEvent` 的 `Type` 参数的 `EVT_SIGNAL_EXIT_BOOT_SERVICES` 标志。此事件的通知功能必须符合以下要求：

- * 通知函数不允许使用内存分配服务，也不允许调用任何使用内存分配服务的函数，因为这些服务会修改当前的内存映射。注：由于服务的使用者不一定知道服务是否使用内存分配服务，这一要求实际上是将任何外部服务（在拥有通知功能的驱动程序之外实现的服务）的使用减少到执行有序过渡到运行时环境所需的绝对最低限度。使用外部服务可能会产生意想不到的结果。由于 UEFI 规范不保证任何给定的通知函数调用顺序，因此可以在提供服务的驱动程序的通知函数之前或之后调用使用服务的通知函数。因此，被通知函数调用的服务可能显示启动时行为或运行时行为（对于纯启动服务而言，这是未定义的）。
- * 通知函数不能依赖于定时器事件，因为定时器服务将在调用任何通知函数之前被停用。

有关其他详细信息，请参阅下面的 `EFI_BOOT_SERVICES.ExitBootServices()`。

- `EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES`: 当在通知 `EFI_EVENT_GROUP_EXIT_BOOT_SERVICES` 事件组之前调用 `ExitBootServices()` 时，系统会通知此事件组。该事件提供了在引导环境中使用固件接口的最后机会。此事件的通知函数不得依赖于任何类型的延迟处理（在超出通知函数时间跨度的计时器回调中发生的处理），因为系统固件会在为此事件组调度处理程序后立即停用计时器服务。

有关其他详细信息，请参阅下面的 `EFI_BOOT_SERVICES.ExitBootServices()`。

- `EFI_EVENT_GROUP_VIRTUAL_ADDRESS_CHANGE`: 当调用 `SetVirtualAddressMap()` 时，系统会通知此事件组。这在功能上等同于 `CreateEvent` 的 `Type` 参数的 `EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE` 标志。
- `EFI_EVENT_GROUP_MEMORY_MAP_CHANGE`: 当内存映射发生变化时，系统会通知该事件组。此事

件的通知功能不应使用内存分配服务以避免重入复杂性。

- **EFI_EVENT_GROUP_READY_TO_BOOT**: 当引导管理器即将加载并执行引导选项时，系统会在通知 **EFI_EVENT_GROUP_AFTER_READY_TO_BOOT** 事件组之前通知此事件组。事件组提供了在将控制权传递给引导选项之前修改设备或系统配置的最后机会。
- **EFI_EVENT_GROUP_AFTER_READY_TO_BOOT**: 当引导管理器即将加载并执行引导选项时，系统会在通知 **EFI_EVENT_GROUP_READY_TO_BOOT** 事件组后立即通知该事件组。事件组提供了在将控制权传递给引导选项之前调查设备或系统配置的最后机会。
- **EFI_EVENT_GROUP_RESET_SYSTEM**: 当调用 `ResetSystem()` 并且系统即将重置时，系统会通知此事件组。仅在调用 `ExitBootServices()` 之前通知事件组。

- 相关定义 (Related Definitions)

`EFI_EVENT` 在 `CreateEvent` 中定义。

`EVT_SIGNAL_EXIT_BOOT_SERVICES` 和 `EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE` 在 `CreateEvent` 中定义。

```

1 #define EFI_EVENT_GROUP_EXIT_BOOT_SERVICES
2 {0x27abf055, 0xb1b8, 0x4c26, 0x80, 0x48, 0x74, 0x8f, 0x37, 0xba, 0xa2, 0xdf}
3
4 #define EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES
5 { 0x8be0e274, 0x3970, 0x4b44, { 0x80, 0xc5, 0x1a, 0xb9, 0x50, 0x2f, 0x3b, 0xfc } }
6
7 #define EFI_EVENT_GROUP_VIRTUAL_ADDRESS_CHANGE
8 {0x13fa7698, 0xc831, 0x49c7, 0x87, 0xea, 0x8f, 0x43, 0xfc, 0xc2, 0x51, 0x96}
9
10 #define EFI_EVENT_GROUP_MEMORY_MAP_CHANGE
11 {0x78bee926, 0x692f, 0x48fd, 0x9e, 0xdb, 0x1, 0x42, 0x2e, 0xf0, 0xd7, 0xab}
12
13 #define EFI_EVENT_GROUP_READY_TO_BOOT
14 {0x7ce88fb3, 0x4bd7, 0x4679, 0x87, 0xa8, 0xa8, 0xd8, 0xde, 0xe5, 0xd, 0x2b}
15
16 #define EFI_EVENT_GROUP_AFTER_READY_TO_BOOT
17 { 0x3a2a00ad, 0x98b9, 0x4cdf, { 0xa4, 0x78, 0x70, 0x27, 0x77, 0xf1, 0xc1, 0xb } }
18
19 #define EFI_EVENT_GROUP_RESET_SYSTEM
20 { 0x62da6a56, 0x13fb, 0x485a, { 0xa8, 0xda, 0xa3, 0xdd, 0x79, 0x12, 0xcb, 0xb6 } }

```

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The event structure was created.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_INVALID_PARAMETER	<i>Event</i> is NULL .
EFI_INVALID_PARAMETER	<i>Type</i> has an unsupported bit set.
EFI_INVALID_PARAMETER	<i>Type</i> has both EVT_NOTIFY_SIGNAL and EVT_NOTIFY_WAIT set.
EFI_INVALID_PARAMETER	<i>Type</i> has either EVT_NOTIFY_SIGNAL or EVT_NOTIFY_WAIT set and <i>NotifyFunction</i> is NULL .
EFI_INVALID_PARAMETER	<i>Type</i> has either EVT_NOTIFY_SIGNAL or EVT_NOTIFY_WAIT set and <i>NotifyTpL</i> is not a supported TPL level.
EFI_OUT_OF_RESOURCES	The event could not be allocated.

图 21. table7-3_1

EFI_BOOT_SERVICES.CloseEvent()

• 概要 (Summary)

关闭一个事件。

• 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_CLOSE_EVENT) (
2     IN EFI_EVENT Event
3 );
```

• 参数 (Parameter)

Event: 要关闭的事件。 **EFI_EVENT** 类型在 [CreateEvent\(\)](#) 函数描述中定义。

• 描述 (Description)

[CloseEvent\(\)](#) 函数删除调用者对事件的引用，将其从它所属的任何事件组中删除，然后关闭它。一旦事件关闭，该事件就不再有效，并且不能用于任何后续函数调用。如果 *Event* 已使用 [RegisterProtocolNotify\(\)](#) 注册，则 [CloseEvent\(\)](#) 将删除相应的注册。在相应的通知函数中调用 [CloseEvent\(\)](#) 是安全的。

• 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The event has been closed.
-------------	----------------------------

图 22. table7-3_2

EFI_BOOT_SERVICES.SignalEvent()

- 概要 (Summary)

发出事件信号。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_SIGNAL_EVENT) (
2     IN EFI_EVENT Event
3 );
```

- 参数 (Parameters)

Event: 要发出信号的事件。EFI_EVENT 类型在 [EFI_BOOT_SERVICES.CheckEvent\(\)](#) 函数描述中定义。

- 描述 (Description)

提供的事件被置于信号状态。如果 *Event* 已经处于信号状态，则返回 EFI_SUCCESS。如果 *Event* 是 EVT_NOTIFY_SIGNAL 类型，那么事件的通知函数被安排在事件的通知任务优先级调用。[SignalEvent\(\)](#) 可以从任何任务优先级调用。如果提供的事件是事件组的一部分，则事件组中的所有事件也会被发送信号，并且它们的通知功能会被调度。当向事件组发送信号时，可以在组中创建一个事件，向它发出信号，然后关闭该事件以将其从组中删除。

例如：

```
1 EFI_EVENT Event;
2 EFI_GUID gMyEventGroupGuid = EFI_MY_EVENT_GROUP_GUID;
3 gBS->CreateEventEx (
4     0,
5     0,
6     NULL,
7     NULL,
8     &gMyEventGroupGuid,
9     &Event
10 );
11
12 gBS->SignalEvent (Event);
13 gBS->CloseEvent (Event);
```

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The event was signaled.
-------------	-------------------------

图 23. table7-3_3

EFI_BOOT_SERVICES.WaitForEvent()

- 概要 (Summary)

停止执行，直到发出事件信号

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_WAIT_FOR_EVENT) (
2     IN UINTN      NumberofEvents,
3     IN EFI_EVENT  *Event,
4     OUT UINTN    *Index
5 );

```

- 参数 (Parameters)

NumberofEvents: 事件数组中的事件数。

Event: `EFI_EVENT` 数组。 `EFI_EVENT` 类型在 `CreateEvent()` 函数描述中定义。

Index: 指向满足等待条件的事件索引的指针。

- 描述 (Description)

必须在优先级 `TPL_APPLICATION` 调用此函数。如果尝试以任何其他优先级调用它，则返回 `EFI_UNSUPPORTED`。

Event 数组中的事件列表按从第一个到最后一个的顺序进行评估，并重复此评估，直到发出事件信号或检测到错误。对 *Event* 数组中的每个事件执行以下检查。

- 如果事件的类型为 `EVT_NOTIFY_SIGNAL`，则返回 `EFI_INVALID_PARAMETER`，*Index* 指示导致失败的事件。
- 如果事件处于信号状态，则清除信号状态并返回 `EFI_SUCCESS`，并且 *Index* 指示已发出信号的事件。
- 如果一个事件不处于信号状态但确实具有通知功能，则通知功能将在事件的通知任务优先级排队。如果事件的通知函数的执行导致事件发出信号，则清除信号状态，返回 `EFI_SUCCESS`，*Index* 指示发出信号的事件。

要等待指定的时间，必须在 Event 数组中包含一个计时器事件。

要检查一个事件是否在没有等待的情况下发出信号，可以将已发出信号的事件用作列表中的最后一个事件，或者可以使用 `CheckEvent()` 接口。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The event indicated by <i>Index</i> was signaled.
<code>EFI_INVALID_PARAMETER</code>	<i>NumberOfEvents</i> is 0.
<code>EFI_INVALID_PARAMETER</code>	The event indicated by <i>Index</i> is of type <code>EVT_NOTIFY_SIGNAL</code> .
<code>EFI_UNSUPPORTED</code>	The current TPL is not <code>TPL_APPLICATION</code> .

图 24. table7-3_4

`EFI_BOOT_SERVICES.CheckEvent()`

- 概要 (Summary)

检查事件是否处于信号状态。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_CHECK_EVENT) (
2     IN EFI_EVENT Event
3 );
```

- 参数 (Parameters)

`Event`: 要检查的事件。`EFI_EVENT` 类型在 `CreateEvent()` 函数描述中定义。

- 描述 (Description)

`CheckEvent()` 函数检查事件是否处于信号状态。如果 `Event` 是 `EVT_NOTIFY_SIGNAL` 类型，则返回 `EFI_INVALID_PARAMETER`。否则，有三种可能：

- 如果 `Event` 处于信号状态，则将其清除并返回 `EFI_SUCCESS`。
- 如果 `Event` 不处于信号状态且没有通知功能，则返回 `EFI_NOT_READY`。
- 如果 `Event` 不处于信号状态但确实具有通知功能，则通知功能在事件的通知任务优先级排队。如果通知函数的执行导致 `Event` 被发出信号，则清除发出信号的状态并返回 `EFI_SUCCESS`；如果未发出事件信号，则返回 `EFI_NOT_READY`。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The event is in the signaled state.
<code>EFI_NOT_READY</code>	The event is not in the signaled state.
<code>EFI_INVALID_PARAMETER</code>	<i>Event</i> is of type <code>EVT_NOTIFY_SIGNAL</code> .

图 25. table7-3_5

`EFI_BOOT_SERVICES.SetTimer()`

- 概要 (Summary)

设置定时器类型和定时器事件的触发时间。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_SET_TIMER) (
2     IN EFI_EVENT           Event,
3     IN EFI_TIMER_DELAY    Type,
4     IN UINT64              TriggerTime
5 );
```

- 参数 (Parameters)

Event: 要在指定时间发出信号的计时器事件。`EFI_EVENT` 类型在 `CreateEvent()` 函数描述中定义。

Type: 在 *TriggerTime* 中指定的时间类型。请参阅“相关定义”中的定时器延迟类型。

TriggerTime: 计时器到期前的 100ns 单位数。*TriggerTime* 为 0 是合法的。如果 *Type* 为 `TimerRelative` 且 *TriggerTime* 为 0，则计时器事件将在下一个计时器滴答时发出信号。如果 *Type* 为 `TimerPeriodic` 且 *TriggerTime* 为 0，则计时器事件将在每个计时器滴答时发出信号。

- 相关定义 (Related Definitions)

```
1 //*****
2 //EFI_TIMER_DELAY
3 //*****
4 typedef enum {
5     TimerCancel,
6     TimerPeriodic,
7     TimerRelative
8 } EFI_TIMER_DELAY;
9
10 TimerCancel: 取消事件的定时器设置，不设置定时器触发。取消计时器时将忽略
                TriggerTime。
```

```

11
12 TimerPeriodic: 该事件将从当前时间开始以 TriggerTime 间隔周期性地发出信号。这是唯
      一个不需要为每个通知重置事件计时器的计时器触发器类型。所有其他计时器触发类型
      都是“一次性”。
13
14 TimerRelative: 该事件将以 TriggerTime 100ns 为单位发出信号。

```

- 描述 (Description)

`SetTimer()` 函数取消该事件之前的任何时间触发设置，并为该事件设置新的触发时间。此函数只能用于 `EVT_TIMER` 类型的事件。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The event has been set to be signaled at the requested time.
<code>EFI_INVALID_PARAMETER</code>	<i>Event</i> or <i>Type</i> is not valid.

图 26. table7-3_6

`EFI_BOOT_SERVICES.RaiseTPL()`

- 概要 (Summary)

提高任务的优先级并返回其先前的级别。

- 原型 (Prototype)

```

1 typedef
2 EFI_TPL
3 (EFIAPI *EFI_RAISE_TPL) (
4     IN EFI_TPL NewTpl
5 );

```

- 参数 (Parameters)

`NewTpl`: 新的任务优先级。它必须大于或等于当前任务优先级。请参阅“相关定义”。

- 相关定义 (Related Definitions)

```

1 //*****
2 // EFI_TPL
3 //*****
4 typedef UINTN EFI_TPL
5

```

```

6 //*****
7 // Task Priority Levels
8 //*****
9 #define TPL_APPLICATION 4
10 #define TPL_CALLBACK 8
11 #define TPL_NOTIFY 16
12 #define TPL_HIGH_LEVEL 31

```

- 描述 (Description)

`EFI_BOOT_SERVICES.RaiseTPL()` 函数提高当前执行任务的优先级并返回其先前的优先级。在启动服务执行期间，只有三个任务优先级暴露在固件之外。第一个是 `TPL_APPLICATION`，所有正常执行都会发生。该级别可能会被中断以执行各种异步中断样式通知，这些通知发生在 `TPL_CALLBACK` 或 `TPL_NOTIFY` 级别。通过将任务优先级提高到 `TPL_NOTIFY`，此类通知将被屏蔽，直到任务优先级恢复，从而与此类通知同步执行。同步阻塞 I/O 函数在 `TPL_NOTIFY` 处执行。`TPL_CALLBACK` 通常用于应用程序级通知功能。设备驱动程序通常将 `TPL_CALLBACK` 或 `TPL_NOTIFY` 用于其通知功能。应用程序和驱动程序也可以使用 `TPL_NOTIFY` 来保护代码关键部分中的数据结构。

调用者必须在返回之前使用 `EFI_BOOT_SERVICES.RestoreTPL()` 将任务优先级恢复到之前的级别。

注：如果 `NewTpl` 低于当前 `TPL` 级别，则系统行为是不确定的。此外，只能使用 `TPL_APPLICATION`、`TPL_CALLBACK`、`TPL_NOTIFY` 和 `TPL_HIGH_LEVEL`。所有其他值保留供固件使用；使用它们将导致不可预测的行为。良好的编码习惯要求所有代码都应在其可能的最低 `TPL` 级别执行，并且必须尽量减少使用高于 `TPL_APPLICATION` 的 `TPL` 级别。长时间在高于 `TPL_APPLICATION` 的 `TPL` 级别执行也可能导致不可预知的行为。

- 返回的状态码 (Status Codes Returned)

与其他 UEFI 接口函数不同，`EFI_BOOT_SERVICES.RaiseTPL()` 不返回状态码。相反，它返回先前的任务优先级，稍后将通过对 `RestoreTPL()` 的匹配调用来恢复该优先级。

`EFI_BOOT_SERVICES.RestoreTPL()`

- 概要 (Summary)

将任务的优先级恢复到之前的值。

- 原型 (Prototype)

```

1 typedef
2 VOID
3 (EFI API *EFI_RESTORE_TPL) (
4     IN EFI_TPL OldTpl
5 )

```

- 参数 (Parameters)

`OldTpl`: 要恢复的上一个任务优先级（来自上一个匹配调用 `EFI_BOOT_SERVICES.RaiseTPL()` 的值）。`EFI_TPL` 类型在 `RaiseTPL()` 函数描述中定义。

- 描述 (Description)

`RestoreTPL()` 函数将任务的优先级恢复到之前的值。对 `RestoreTPL()` 的调用与对 `RaiseTPL()` 的调用相匹配。

注：如果 `OldTpl` 高于当前 `TPL` 级别，则系统行为是不确定的。此外，只能使用 `TPL_APPLICATION`、`TPL_CALLBACK`、`TPL_NOTIFY` 和 `TPL_HIGH_LEVEL`。所有其他值保留供固件使用；使用它们将导致不可预测的行为。良好的编码习惯要求所有代码都应在其可能的最低 `TPL` 级别执行，并且必须尽量减少使用高于 `TPL_APPLICATION` 的 `TPL` 级别。长时间在高于 `TPL_APPLICATION` 的 `TPL` 级别执行也可能导致不可预知的行为。

- 返回的状态码 (Status Codes Returned)

None;

6.2 内存分配服务

组成内存分配服务的函数在预引导期间用于分配和释放内存，以及获取系统的内存映射。请参见表 7-4。

Table 7-4 Memory Allocation Functions

Name	Type	Description
AllocatePages	Boot	Allocates pages of a particular type.
FreePages	Boot	Frees allocated pages.
GetMemoryMap	Boot	Returns the current boot services memory map and memory map key.
AllocatePool	Boot	Allocates a pool of a particular type.
FreePool	Boot	Frees allocated pool.

图 27. table7-4 Memory Allocation Functions

这些函数的使用方式与 UEFI 内存设计的一个重要特性直接相关。此功能规定 EFI 固件在预引导期间拥有系统的内存映射，具有三个主要后果：

- 在预引导期间，所有组件（包括执行 EFI 镜像）必须与固件配合使用函数 `EFI_BOOT_SERVICES.AllocatePages()`、`EFI_BOOT_SERVICES.AllocatePool()`、`EFI_BOOT_SERVICES.FreePages()` 和 `EFI_BOOT_SERVICES.FreePool()`。
- 在预引导期间，正在执行的 EFI 镜像必须只使用它已分配的内存。

- 在执行的 EFI 镜像退出并将控制权返回给固件之前，它必须释放已明确分配的所有资源。这包括所有内存页面、池分配、打开的文件句柄等。卸载镜像时，固件会释放固件为加载镜像而分配的内存。当调用这些函数时，固件会动态维护内存映射。

本规范描述了由服务分配的大量内存缓冲区，调用者负责释放分配的内存。除非在本规范中另有说明，否则假定此类内存缓冲区由 `AllocatePool()` 分配并由 `FreePool()` 释放。

分配内存时，根据 `EFI_MEMORY_TYPE` 中的值对其进行“类型化”（参见 `EFI_BOOT_SERVICES.AllocatePages()` 的描述）。在调用 `EFI_BOOT_SERVICES.ExitBootServices()` 之前，某些类型的用法与调用之后的用法不同。表 7-5 列出了每种类型及其调用前的用法；表 7-6 列出了每种类型及其调用后的用法。系统固件必须遵循 EFI 内存映射布局中第 2.3.2 节和第 2.3.4 节中概述的处理器特定规则，以使操作系统能够进行所需的虚拟映射。

Table 7-5 Memory Type Usage before `ExitBootServices()`

Mnemonic	Description
EfiReservedMemoryType	Not usable.
EfiLoaderCode	The code portions of a loaded UEFI application.
EfiLoaderData	The data portions of a loaded UEFI application and the default data allocation type used by a UEFI application to allocate pool memory.
EfiBootServicesCode	The code portions of a loaded UEFI Boot Service Driver.
EfiBootServicesData	The data portions of a loaded UEFI Boot Service Driver, and the default data allocation type used by a UEFI Boot Service Driver to allocate pool memory.
EfiRuntimeServicesCode	The code portions of a loaded UEFI Runtime Driver.
EfiRuntimeServicesData	The data portions of a loaded UEFI Runtime Driver and the default data allocation type used by a UEFI Runtime Driver to allocate pool memory.
EfiConventionalMemory	Free (unallocated) memory.
EfiUnusableMemory	Memory in which errors have been detected.
EfiACPIReclaimMemory	Memory that holds the ACPI tables.
EfiACPIMemoryNVS	Address space reserved for use by the firmware.
EfiMemoryMappedIO	Used by system firmware to request that a memory-mapped IO region be mapped by the OS to a virtual address so it can be accessed by EFI runtime services.
EfiMemoryMappedIOPortSpace	System memory-mapped IO region that is used to translate memory cycles to IO cycles by the processor.
EfiPalCode	Address space reserved by the firmware for code that is part of the processor.
EfiPersistentMemory	A memory region that operates as <i>EfiConventionalMemory</i> . However, it happens to also support byte-addressable non-volatility.
EfiUnacceptedMemoryType	A memory region that represents unaccepted memory, that must be accepted by the boot target before it can be used. Unless otherwise noted, all other EFI memory types are accepted. For platforms that support unaccepted memory, all unaccepted valid memory will be reported as unaccepted in the memory map. Unreported physical address ranges must be treated as not-present memory.

图 28. table7-5

注：在 Itanium-based(TODO) 的平台的体系结构中只定义了一个类型为 `EfiMemoryMappedIOPortSpace` 的区域。因此，在 Itanium-based(TODO) 的平台的 EFI 内存映射中应该只有一个类型为 `EfiMemoryMappedIOPortSpace` 的区域。

Table 7-6 Memory Type Usage after `ExitBootServices()`

Mnemonic	Description
EfiReservedMemoryType	Not usable.
EfiLoaderCode	The UEFI OS Loader and/or OS may use this memory as they see fit. Note: the UEFI OS loader that called <code>EFI_BOOT_SERVICES.ExitBootServices()</code> is utilizing one or more EfiLoaderCode ranges.

图 29. table7-6-1

EfiLoaderData	The Loader and/or OS may use this memory as they see fit. Note: the OS loader that called ExitBootServices() is utilizing one or more EfiLoaderData ranges.
EfiBootServicesCode	Memory available for general use.
EfiBootServicesData	Memory available for general use.
EfiRuntimeServicesCode	The memory in this range is to be preserved by the UEFI OS loader and OS in the working and ACPI S1–S3 states.
EfiRuntimeServicesData	The memory in this range is to be preserved by the UEFI OS loader and OS in the working and ACPI S1–S3 states.
EfiConventionalMemory	Memory available for general use.
EfiUnusableMemory	Memory that contains errors and is not to be used.
EfiACPIReclaimMemory	This memory is to be preserved by the UEFI OS loader and OS until ACPI is enabled. Once ACPI is enabled, the memory in this range is available for general use.
EfiACPIMemoryNVS	This memory is to be preserved by the UEFI OS loader and OS in the working and ACPI S1–S3 states.
EfiMemoryMappedIO	This memory is not used by the OS. All system memory-mapped IO information should come from ACPI tables.
EfiMemoryMappedIOPortSpace	This memory is not used by the OS. All system memory-mapped IO port space information should come from ACPI tables.
EfiPalCode	This memory is to be preserved by the UEFI OS loader and OS in the working and ACPI S1–S4 states. This memory may also have other attributes that are defined by the processor implementation.
EfiPersistentMemory	A memory region that operates as <i>EfiConventionalMemory</i> . However, it happens to also support byte-addressable non-volatility.
EfiUnacceptedMemoryType	A memory region that represents unaccepted memory, that must be accepted by the boot target before it can be used. Unless otherwise noted, all other EFI memory types are accepted. For platforms that support unaccepted memory, all unaccepted valid memory will be reported as unaccepted in the memory map. Unreported physical address ranges must be treated as not-present memory.

图 30. table7-6-2

注：调用 `ExitBootServices()` 的镜像（即 UEFI OS Loader）首先调用 `EFI_BOOT_SERVICES.GetMemoryMap()` 以获取当前内存映射。在 `ExitBootServices()` 调用之后，镜像隐含地拥有映射中所有未使用的内存。这包括内存类型 `EfiLoaderCode`、`EfiLoaderData`、`EfiBootServicesCode`、`EfiBootServicesData` 和 `EfiConventionalMemory`。UEFI OS Loader 和 OS 必须保留标记为 `EfiRuntimeServicesCode` 和 `EfiRuntimeServicesData` 的内存。

EFI_BOOT_SERVICES.AllocatePages()

- 概要 (Summary)

从系统分配内存页。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_ALLOCATE_PAGES) (
2     IN EFI_ALLOCATE_TYPE           Type,
3     IN EFI_MEMORY_TYPE            MemoryType,
4     IN UINTN                      Pages,
5     IN OUT EFI_PHYSICAL_ADDRESS *Memory
6 );
```

- 参数 (Parameters)

Type: 要执行的分配类型。请参阅“相关定义”。

MemoryType: 要分配的内存类型。`EFI_MEMORY_TYPE` 类型在下面的“相关定义”中定义。表 7-5 和表 7-6 中也更详细地描述了这些存储器类型。正常分配（即任何 UEFI 应用程序的分配）的类型为 `EfiLoaderData`。`0x70000000 .. 0x7FFFFFFF` 范围内的 *MemoryType* 值保留供 OEM 使用。`0x80000000 .. 0xFFFFFFFF` 范围内的 *MemoryType* 值保留供操作系统供应商提供的 UEFI OS 加载程序使用。

Pages: 要分配的连续 4 KiB 页面的数量。

Memory: 指向物理地址的指针。在输入时，地址的使用方式取决于 *Type* 的值。有关详细信息，请参阅“说明”。在输出时，地址被设置为分配的页面范围的基础。请参阅“相关定义”。

注：UEFI 应用程序、UEFI 驱动程序和 UEFI 操作系统加载程序不得分配 `EfiReservedMemoryType`、`EfiMemoryMappedIO` 和 `EfiUnacceptedMemoryType` 类型的内存。

- 相关定义 (Related Definitions)

```
1 //*****
2 //EFI_ALLOCATE_TYPE
3 //*****
4 // These types are discussed in the "Description" section below.
5 typedef enum {
6     AllocateAnyPages,
7     AllocateMaxAddress,
8     AllocateAddress,
9     MaxAllocateType
10 } EFI_ALLOCATE_TYPE;
11
12 //*****
13 //EFI_MEMORY_TYPE
```

```
14 //*****
15 // These type values are discussed in Table 7-5 and Table 7-6.
16 typedef enum {
17     EfiReservedMemoryType,
18     EfiLoaderCode,
19     EfiLoaderData,
20     EfiBootServicesCode,
21     EfiBootServicesData,
22     EfiRuntimeServicesCode,
23     EfiRuntimeServicesData,
24     EfiConventionalMemory,
25     EfiUnusableMemory,
26     EfiACPIReclaimMemory,
27     EfiACPIMemoryNVS,
28     EfiMemoryMappedIO,
29     EfiMemoryMappedIOPortSpace,
30     EfiPalCode,
31     EfiPersistentMemory,
32     EfiUnacceptedMemoryType
33     EfiMaxMemoryType
34 } EFI_MEMORY_TYPE;
35
36 //*****
37 //EFI_PHYSICAL_ADDRESS
38 //*****
39 typedef UINT64 EFI_PHYSICAL_ADDRESS;
```

- 描述 (Description)

`AllocatePages()` 函数分配请求的页数，并返回一个指针，该指针指向 `Memory` 引用的位置中的页范围的基地址。该函数扫描内存映射以定位空闲页面。当它发现一个物理上连续的页面块足够大并且也满足 `Type` 的分配要求时，它会更改内存映射以指示页面现在是 `MemoryType` 类型。

通常，UEFI OS 加载程序和 UEFI 应用程序应分配 `EfiLoaderData` 类型的内存（和池）。UEFI 引导服务驱动程序必须分配 `EfiBootServicesData` 类型的内存（和池）。UREFI 运行时驱动程序应该分配类型为 `EfiRuntimeServicesData` 的内存（和池）（尽管这种分配只能在启动服务期间进行）。

`AllocateAnyPages` 类型的分配请求分配满足请求的任何可用页面范围。在输入时，`Memory` 指向的地址被忽略。

`AllocateMaxAddress` 类型的分配请求分配任何可用范围的页面，其最高地址小于或等于输入时 `Memory` 指向的地址。

`AllocateAddress` 类型的分配请求在输入时 `Memory` 指向的地址分配页面。

注：不针对特定实现的 UEFI 驱动程序和 UEFI 应用程序必须使用 [AllocateAnyPages](#) 地址模式为以下运行时类型执行内存分配：

```

1 EfiACPIReclaimMemory,
2
3 EfiACPIMemoryNVS,
4
5 EfiRuntimeServicesCode,
6
7 EfiRuntimeServicesData,
8
9 EfiReservedMemoryType.
```

- 返回的状态码（Status Codes Returned）

EFI_SUCCESS	The requested pages were allocated.
EFI_OUT_OF_RESOURCES	The pages could not be allocated.
EFI_INVALID_PARAMETER	<i>Type</i> is not AllocateAnyPages or AllocateMaxAddress or AllocateAddress .
EFI_INVALID_PARAMETER	<i>MemoryType</i> is in the range EfiMaxMemoryType..0x6FFFFFFF .
EFI_INVALID_PARAMETER	<i>MemoryType</i> is EfiPersistentMemory or EfiUnacceptedMemoryType .
EFI_INVALID_PARAMETER	<i>Memory</i> is NULL .
EFI_NOT_FOUND	The requested pages could not be found.

图 31. table7-6_0

EFI_BOOT_SERVICES.FreePages()

- 概要（Summary）

释放内存页面。

- 原型（Prototype）

```

1 typedef EFI_STATUS (EFIAPI *EFI_FREE_PAGES) (
2     IN EFI_PHYSICAL_ADDRESS Memory,
3     IN UINTN                 Pages
4 );
```

- 参数（Parameters）

Memory: 要释放的页面的基本物理地址。[EFI_PHYSICAL_ADDRESS](#) 类型在 [EFI_BOOT_SERVICES.AllocatePages](#)

(*)* 函数描述中定义。

Pages: 要释放的连续 4 KiB 页面的数量。

- 描述 (Description)

`FreePages()` 函数将 `AllocatePages()` 分配的内存返回给固件。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The requested memory pages were freed.
<code>EFI_NOT_FOUND</code>	The requested memory pages were not allocated with <code>AllocatePages()</code> .
<code>EFI_INVALID_PARAMETER</code>	<i>Memory</i> is not a page-aligned address or <i>Pages</i> is invalid.

图 32. table7-6_1

`EFI_BOOT_SERVICES.GetMemoryMap()`

- 概要 (Summary)

返回当前的内存映射。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_GET_MEMORY_MAP) (
2     IN OUT UINTN           *MemoryMapSize,
3     OUT EFI_MEMORY_DESCRIPTOR *MemoryMap,
4     OUT UINTN               *MapKey,
5     OUT UINTN               *DescriptorSize,
6     OUT UINT32              *DescriptorVersion
7 );

```

- 参数 (Parameters)

MemoryMapSize: 指向 *MemoryMap* 缓冲区大小 (以字节为单位) 的指针。在输入时, 这是调用者分配的缓冲区的大小。在输出时, 如果缓冲区足够大, 则为固件返回的缓冲区大小; 如果缓冲区太小, 则为包含映射所需的缓冲区大小。

MemoryMap: 指向固件放置当前内存映射的缓冲区的指针。该映射是一个 `EFI_MEMORY_DESCRIPTOR` 数组。请参阅 “相关定义”。

MapKey: 指向固件返回当前内存映射密钥的位置的指针。

DescriptorSize: 指向固件返回单个 `EFI_MEMORY_DESCRIPTOR` 大小 (以字节为单位) 的位置的指针。

DescriptorVersion: 指向固件返回与 `EFI_MEMORY_DESCRIPTOR` 关联的版本号的位置的指针。请参阅“[相关定义](#)”。

- [相关定义 \(Related Definitions\)](#)

```

1 //*****
2 //EFI_MEMORY_DESCRIPTOR
3 //*****
4 typedef struct {
5     UINT32             Type;
6     EFI_PHYSICAL_ADDRESS PhysicalStart;
7     EFI_VIRTUAL_ADDRESS VirtualStart;
8     UINT64            NumberOfPages;
9     UINT64            Attribute;
10 } EFI_MEMORY_DESCRIPTOR;
```

Type: 内存区域的类型。`EFI_MEMORY_TYPE` 类型在 `AllocatePages()` 函数描述中定义。

PhysicalStart: 内存区域中第一个字节的物理地址。`PhysicalStart` 必须在 4 KiB 边界上对齐，并且不得高于 `0xfffffffffffff000`。`EFI_PHYSICAL_ADDRESS` 类型在 `AllocatePages()` 函数描述中定义。

VirtualStart: 内存区域中第一个字节的虚拟地址。`VirtualStart` 必须在 4 KiB 边界上对齐，并且不得高于 `0xfffffffffffff000`。`EFI_VIRTUAL_ADDRESS` 类型在“[相关定义](#)”中定义。

NumberOfPages: 内存区域中 4 KiB 页的数量。`NumberOfPages` 不能为 0，并且不能是任何可以表示内存页的起始地址（物理或虚拟）大于 `0xfffffffffffff000` 的值。

Attribute: 描述该内存区域功能位掩码的内存区域属性，不一定是该内存区域的当前设置。请参阅以下“[内存属性定义](#)”。

```

1 //*****
2 // Memory Attribute Definitions
3 //*****
4 // These types can be "ORed" together as needed.
5 #define EFI_MEMORY_UC          0x0000000000000001
6 #define EFI_MEMORY_WC          0x0000000000000002
7 #define EFI_MEMORY_WT          0x0000000000000004
8 #define EFI_MEMORY_WB          0x0000000000000008
9 #define EFI_MEMORY_UCE         0x0000000000000010
10 #define EFI_MEMORY_WP          0x0000000000001000
11 #define EFI_MEMORY_RP          0x0000000000002000
12 #define EFI_MEMORY_XP          0x0000000000004000
13 #define EFI_MEMORY_NV          0x0000000000008000
14 #define EFI_MEMORY_MORE_RELIABLE 0x00000000000010000
15 #define EFI_MEMORY_RO          0x00000000000020000
```

```
16 #define EFI_MEMORY_SP          0x00000000000040000
17 #define EFI_MEMORY_CPU_CRYPTO   0x00000000000080000
18 #define EFI_MEMORY_RUNTIME       0x80000000000000000
```

EFI_MEMORY_UC: 内存缓存属性: 内存区域支持配置为不可缓存。

EFI_MEMORY_WC: 内存缓存属性: 内存区域支持配置为写合并。

EFI_MEMORY_WT: 内存缓存属性: 内存区域支持使用“直写”策略配置为可缓存。在缓存中命中的写入也将写入主内存。

EFI_MEMORY_WB: 内存缓存属性: 内存区域支持通过“写回”策略配置为可缓存。在缓存中命中的读取和写入不会传播到主内存。当分配新的高速缓存行时，脏数据被写回主存储器。

EFI_MEMORY_UCE: 内存缓存属性: 内存区域支持配置为不可缓存、导出，并支持“获取和添加”信号量机制。

EFI_MEMORY_WP: 物理内存保护属性: 内存区域支持被系统硬件配置为写保护。这通常用作当前的可缓存性属性。内存区域支持使用“写保护”策略配置为可缓存。读取尽可能来自缓存行，读取未命中会导致缓存填充。写入被传播到系统总线并导致总线上所有处理器上的相应高速缓存行无效。

EFI_MEMORY_SP: 专用内存 (Specific-purpose memory:SPM)。内存被指定用于特定目的，例如特定设备驱动程序或应用程序。SPM 属性用作提示操作系统避免为核心操作系统数据或无法重定位的代码分配此内存。出于预期目的以外的目的长时间使用此内存可能会导致平台性能欠佳。

EFI_MEMORY_CPU_CRYPTO: 如果设置了此标志，则内存区能够受到 CPU 的内存加密功能的保护。如果清除此标志，则内存区无法使用 CPU 的内存加密功能进行保护，或者 CPU 不支持 CPU 内存加密功能。

注:UEFI 规范 2.5 及以下: 使用 **EFI_MEMORY_RO** 作为写保护的物理内存保护属性。此外,**EFI_MEMORY_WP** 表示可缓存性属性。

EFI_MEMORY_RP: 物理内存保护属性: 内存区域支持被系统硬件配置为读保护。

EFI_MEMORY_XP: 物理内存保护属性: 内存区域支持配置，因此它受到系统硬件的保护，不会执行代码。

EFI_MEMORY_NV: 运行时内存属性: 内存区域是指持久内存。

EFI_MEMORY_MORE_RELIABLE: 相对于系统中的其他内存，内存区域提供了更高的可靠性。如果所有存储器具有相同的可靠性，则不使用该位。

EFI_MEMORY_RO: 物理内存保护属性: 内存区域支持系统硬件将此内存范围设为只读。

EFI_MEMORY_RUNTIME: 运行时内存属性: 当调用 `SetVirtualAddressMap()` 时（在第 8.4 节中描述），操作系统需要给内存区域一个虚拟映射。

```
1 //*****
```

```
2 //EFI_VIRTUAL_ADDRESS
3 //*****
4 typedef UINT64 EFI_VIRTUAL_ADDRESS;
5
6 //*****
7 // Memory Descriptor Version Number
8 //*****
9 #define EFI_MEMORY_DESCRIPTOR_VERSION 1
```

- 描述 (Description)

`GetMemoryMap()` 函数返回当前内存映射的副本。该映射是一个内存描述符数组，每个描述符描述一个连续的内存块。该映射描述了所有内存，无论它是如何使用的。也就是说，它包括由 `EFI_BOOT_SERVICES.AllocatePages()` 和 `EFI_BOOT_SERVICES.AllocatePool()` 分配的块，以及固件用于其自身目的的块。内存映射仅用于描述系统中存在的内存。固件不会返回不受物理硬件支持的地址空间区域的范围描述。由物理硬件支持但不应由操作系统访问的区域必须作为 `EfiReservedMemoryType` 返回。操作系统可以自行决定使用未在内存映射中描述的内存范围的地址。

在调用 `EFI_BOOT_SERVICES.ExitBootServices()` 之前，内存映射归固件所有，当前执行的 UEFI 镜像应仅使用已显式分配的内存页。如果 `MemoryMap` 缓冲区太小，则返回 `EFI_BUFFER_TOO_SMALL` 错误代码，并且 `MemoryMapSize` 值包含包含当前内存映射所需的缓冲区大小。为后续调用 `GetMemoryMap()` 分配的缓冲区的实际大小应该大于 `MemoryMapSize` 中返回的值，因为分配新缓冲区可能会增加内存映射大小。

成功时返回一个标识当前内存映射的 `MapKey`。每次内存映射中的某些内容发生更改时，固件的密钥都会更改。为了成功调用 `EFI_BOOT_SERVICES.ExitBootServices()` 调用者必须提供当前内存映射密钥。

`GetMemoryMap()` 函数还返回 `EFI_MEMORY_DESCRIPTOR` 的大小和修订号。`DescriptorSize` 表示在 `MemoryMap` 中返回的 `EFI_MEMORY_DESCRIPTOR` 数组元素的大小（以字节为单位）。返回大小以允许将来扩展 `EFI_MEMORY_DESCRIPTOR` 以响应硬件 innovation（创新）。`EFI_MEMORY_DESCRIPTOR` 的结构将来可能会扩展，但它将保持与当前定义的向后兼容。因此操作系统软件必须使用 `DescriptorSize` 来找到 `MemoryMap` 数组中每个 `EFI_MEMORY_DESCRIPTOR` 的开始。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The memory map was returned in the <code>MemoryMap</code> buffer.
<code>EFI_BUFFER_TOO_SMALL</code>	The <code>MemoryMap</code> buffer was too small. The current buffer size needed to hold the memory map is returned in <code>MemoryMapSize</code> .
<code>EFI_INVALID_PARAMETER</code>	<code>MemoryMapSize</code> is NULL .
<code>EFI_INVALID_PARAMETER</code>	The <code>MemoryMap</code> buffer is not too small and <code>MemoryMap</code> is NULL .

图 33. table7-6_2

EFI_BOOT_SERVICES.AllocatePool()

- 概要 (Summary)
- 分配池内存。
- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_ALLOCATE_POOL) (
2     IN EFI_MEMORY_TYPE  PoolType,
3     IN UINTN           Size,
4     OUT VOID            **Buffer
5 );

```

- 参数 (Parameters)

PoolType: 要分配的池类型。`EFI_MEMORY_TYPE` 类型在 `EFI_BOOT_SERVICES.AllocatePages()` 函数描述中定义。`0x70000000 .. 0x7FFFFFFF` 范围内的 *PoolType* 值保留供 OEM 使用。`0x80000000 .. 0xFFFFFFFF` 范围内的 *PoolType* 值保留供操作系统供应商提供的 UEFI OS 加载程序使用。

Size: 要从池中分配的字节数。

Buffer: 如果调用成功，则指向分配缓冲区的指针；否则未定义。

注：UEFI 应用程序和 UEFI 驱动程序不得分配 `EfiReservedMemoryType` 类型的内存。

- 描述 (Description)

`AllocatePool()` 函数从 *PoolType* 类型的内存中分配 *Size* 字节的内存区域，并返回 *Buffer* 引用的位置中分配内存的地址。此函数根据需要从 `EfiConventionalMemory` 分配页面以增加请求的池类型。所有分配都是 8-byte 对齐的。

分配的池内存使用 `EFI_BOOT_SERVICES.FreePool()` 函数返回到可用池。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The requested number of bytes was allocated.
EFI_OUT_OF_RESOURCES	The pool requested could not be allocated.
EFI_INVALID_PARAMETER	<i>Pool Type</i> is in the range EfiMaxMemoryType..0x6FFFFFFF .
EFI_INVALID_PARAMETER	<i>Pool Type</i> is EfiPersistentMemory .
EFI_INVALID_PARAMETER	<i>Buffer</i> is NULL.

图 34. table7-6_3

EFI_BOOT_SERVICES.FreePool()

• 概要 (Summary)

将池内存返回给系统。

• 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_FREE_POOL) (
2     IN VOID  *Buffer
3 );
```

• 参数 (Parameters)

Buffer: 指向要释放的缓冲区的指针。

• 描述 (Description)

`FreePool()` 函数将 *Buffer* 指定的内存返回给系统。返回时，内存的类型是 `EfiConventionalMemory`。释放的 *Buffer* 必须由 `AllocatePool()` 分配。

• 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The memory was returned to the system.
EFI_INVALID_PARAMETER	<i>Buffer</i> was invalid.

图 35. table7-6_4

6.3 协议处理器服务 (Protocol Handler Services)

抽象地说，一个协议由一个 128 位的全局唯一标识符 (GUID) 和一个协议接口结构组成。该结构包含用于访问设备的函数和实例数据。组成协议处理器服务的功能允许应用程序在句柄上安装协议、识别支持给定协议的句柄、确定句柄是否支持给定协议等等。请参见表 7-7。

Table 7-7 Protocol Interface Functions

Name	Type	Description
InstallProtocolInterface	Boot	Installs a protocol interface on a device handle.
UninstallProtocolInterface	Boot	Removes a protocol interface from a device handle.
ReinstallProtocolInterface	Boot	Reinstalls a protocol interface on a device handle.
RegisterProtocolNotify	Boot	Registers an event that is to be signaled whenever an interface is installed for a specified protocol.
LocateHandle	Boot	Returns an array of handles that support a specified protocol.
HandleProtocol	Boot	Queries a handle to determine if it supports a specified protocol.
LocateDevicePath	Boot	Locates all devices on a device path that support a specified protocol and returns the handle to the device that is closest to the path.
OpenProtocol	Boot	Adds elements to the list of agents consuming a protocol interface.
CloseProtocol	Boot	Removes elements from the list of agents consuming a protocol interface.
OpenProtocolInformation	Boot	Retrieves the list of agents that are currently consuming a protocol interface.
ConnectController	Boot	Uses a set of precedence rules to find the best set of drivers to manage a controller.
DisconnectController	Boot	Informs a set of drivers to stop managing a controller.
ProtocolsPerHandle	Boot	Retrieves the list of protocols installed on a handle. The return buffer is automatically allocated.
LocateHandleBuffer	Boot	Retrieves the list of handles from the handle database that meet the search criteria. The return buffer is automatically allocated.
LocateProtocol	Boot	Finds the first handle in the handle database that supports the requested protocol.
InstallMultipleProtocolInterfaces	Boot	Installs one or more protocol interfaces onto a handle.
UninstallMultipleProtocolInterfaces	Boot	Uninstalls one or more protocol interfaces from a handle.

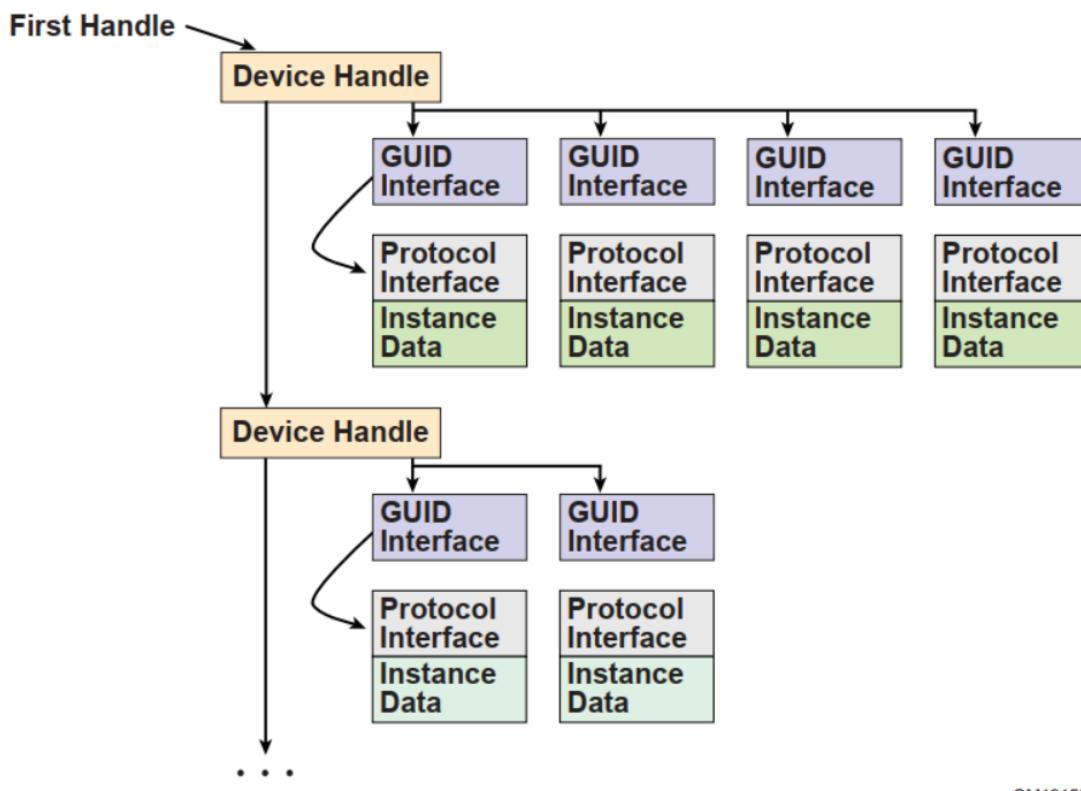
图 36. table7-7

协议处理程序启动服务已被修改,利用`EFI_BOOT_SERVICES.OpenProtocol()`和`EFI_BOOT_SERVICES.CloseProtocol()`引导服务正在跟踪的信息。由于这些新的引导服务正在跟踪协议接口的使用情况,因此现在可以安全地卸载和重新安装UEFI驱动程序正在使用的协议接口。

如图 7-1 所示，固件负责维护一个“数据库”，该数据库显示哪些协议附加到每个设备句柄。（图中将“数据库”描述为链表，但数据结构的选择取决于实现。）“数据库”是通过调用 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数动态构建的。协议只能由 UEFI 驱动程序或固件本身安装。在图中，设备句柄 (`EFI_HANDLE`) 是指该句柄的一个或多个已注册协议接口的列表。系统中的第一个句柄有四个附加协议，第二个句柄有两个附加协议。每个附加的协议都表示为一个 GUID/接口指针。GUID 是协议的名称，Interface 指向一个协议实例。这个数据结构通常包含一个接口函数列表和一些实例数据。

通过调用 `EFI_BOOT_SERVICES.HandleProtocol()` 函数来启动对设备的访问，该函数确定句柄是否支持给定协议。如果是，则返回一个指向匹配协议接口结构的指针。

当一个协议被添加到系统中时，它可以被添加到现有的设备句柄中，也可以被添加以创建一个新的设备句柄。图 7-1 显示了为每个设备句柄列出了协议处理程序，并且每个协议处理程序在逻辑上都是一个 UEFI 驱动程序。



OM13155

图 37. Figure 7-1 Device Handle to Protocol Handler Mapping

将新协议接口添加为新句柄或在现有接口上分层的能力提供了极大的灵活性。分层使添加基于设备基本协议

的新协议成为可能。这方面的一个示例可能是在 `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` 支持上进行分层，该支持将建立在句柄的底层 `EFI_SERIAL_IO_PROTOCOL` 之上。

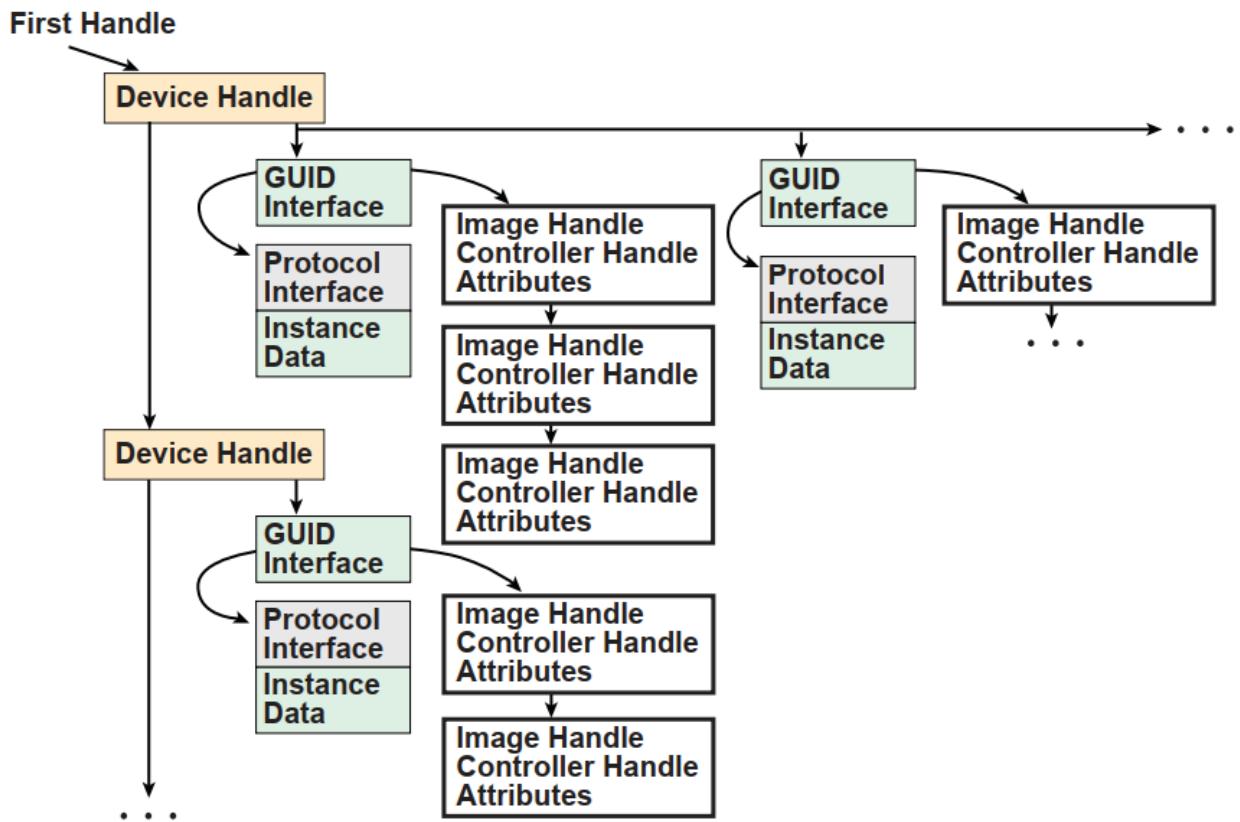
添加新句柄的能力可用于在发现新设备时生成新设备，甚至生成抽象设备。这方面的一个示例可能是添加一个复用设备，将 `ConsoleOut` 替换为将 `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` 协议多路复用到多个底层设备句柄的虚拟设备。

驱动程序模型引导服务（Driver Model Boot Services）

以下是 UEFI 驱动程序模型所需的新 UEFI 引导服务的详细说明。添加这些引导服务是为了减少总线驱动程序和设备驱动程序的大小和复杂性。反过来，这将减少驱动程序所需的 ROM 空间量，这些驱动程序被编程到适配器上的 ROM 或系统闪存中，并减少驱动程序编写者所需的开发和测试时间。

这些新服务分为两类。第一组用于跟踪系统中不同代理对协议接口的使用情况。协议接口存储在句柄数据库中。句柄数据库由一系列句柄组成，每个句柄上都有一个或多个协议接口的列表。引导服务 `EFI_BOOT_SERVICES.InstallProtocolInterface()`、`EFI_BOOT_SERVICES.UninstallProtocolInterface()` 和 `EFI_BOOT_SERVICES.ReinstallProtocolInterface()` 用于在句柄数据库中添加、删除和替换协议接口。引导服务 `EFI_BOOT_SERVICES.HandleProtocol()` 用于在句柄数据库中查找协议接口。但是，调用 `HandleProtocol()` 的代理不会被跟踪，因此调用 `UninstallProtocolInterface()` 或 `ReinstallProtocolInterface()` 是不安全的，因为代理可能正在使用正在被删除或替换的协议接口。

解决方案是在句柄数据库本身中跟踪协议接口的使用情况。为此，每个协议接口都包含一个正在使用该协议接口的代理列表。图 7-2 显示了带有这些新代理列表的示例句柄数据库。代理由镜像句柄、控制器句柄和一些属性组成。镜像句柄标识正在使用协议接口的驱动程序或应用程序。控制器句柄标识正在使用协议接口的控制器。由于驱动程序可能管理多个控制器，驱动程序的镜像句柄和控制器的控制器句柄的组合唯一地标识正在使用协议接口的代理。这些属性显示了协议接口是如何被使用的。



OM13156

Figure 7-2 Handle Database

图 38. Figure 7-2 Handle Database

为了在句柄数据库中维护这些代理列表，需要一些新的引导服务。它们是 `EFI_BOOT_SERVICES.OpenProtocol()`、`EFI_BOOT_SERVICES.CloseProtocol()` 和 `EFI_BOOT_SERVICES.OpenProtocolInformation()`。`OpenProtocol()` 将元素添加到使用协议接口的代理列表中。`CloseProtocol()` 从使用协议接口的代理列表中删除元素，并且 `EFI_BOOT_SERVICES.OpenProtocolInformation()` 检索当前正在使用协议接口的代理的整个列表。

第二组引导服务于确定性地连接和断开驱动程序与控制器。该组中的引导服务是 `EFI_BOOT_SERVICES.ConnectController()` 和 `EFI_BOOT_SERVICES.DisconnectController()`。这些服务利用句柄数据库的新功能以及本文档中描述的新协议来管理系统中存在的驱动程序和控制器。`ConnectController()` 使用一组严格的优先级规则来找到控制器的最佳驱动程序集。这通过 OEM、IBV 和 IHV 的可扩展机制提供了驱动程序与控制器的确定性匹配。`DisconnectController()` 允许驱动程序以受控方式与控制器断开连接，并且通过使用句柄数据库的新功能，可能会使断开连接请求失败，因为在断开连接请求时无法释放协议接口。

第三组引导服务旨在帮助简化驱动程序的实现，并生成可执行文件占用空间更小的驱动程序。`EFI_BOOT_SERVICES.LocateHandleBuffer()` 是 `EFI_BOOT_SERVICES.LocateHandle()` 的新版本，它为调用者分配所需的缓冲区。这消除了调用方代码中对 `LocateHandle()` 的两次调用和对 `EFI_BOOT_SERVICES.AllocatePool()` 的调用。`EFI_BOOT_SERVICES.LocateProtocol()` 在句柄数据库中搜索与搜索条件匹配的第一个协议实例。`EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces()` 和 `EFI_BOOT_SERVICES.UninstallMultipleProtocolInterfaces()` 对驱动程序编写者非常有用。这些引导服务允许从句柄中添加或删除一个或多个协议接口。此外，`InstallMultipleProtocolInterfaces()` 保证永远不会将重复的设备路径添加到句柄数据库中。这对于一次可以创建一个子句柄的总线驱动程序非常有用，因为它保证总线驱动程序不会无意中创建同一个子句柄的两个实例。

`EFI_BOOT_SERVICES.InstallProtocolInterface()`

- 概要 (Summary)

在设备句柄上安装协议接口。如果句柄不存在，则将其创建并添加到系统中的句柄列表中。`InstallMultipleProtocolInterfaces()` 比 `InstallProtocolInterface()` 执行更多错误检查，因此建议使用 `InstallMultipleProtocolInterfaces()` 代替 `InstallProtocolInterface()`。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_INSTALL_PROTOCOL_INTERFACE) (
2     IN OUT EFI_HANDLE      *Handle,
3     IN EFI_GUID            *Protocol,
4     IN EFI_INTERFACE_TYPE   InterfaceType,
5     IN VOID                *Interface
6 );

```

- 参数 (Parameters)

Handle: 指向要在其上安装接口的 `EFI_HANDLE` 的指针。如果 **Handle*** 在输入时为 `NULL`，则创建一个新句柄并在输出时返回。如果 **Handle*** 在输入时不为 `NULL`，则将协议添加到句柄中，并且句柄不加修改地返回。`EFI_HANDLE` 类型在“相关定义”中定义。如果 `**Handle*` 不是有效句柄，则返回 `EFI_INVALID_PARAMETER`。

Protocol: 协议接口的数字 ID。`EFI_GUID` 类型在“相关定义”中定义。传入有效的 GUID 是调用者的责任。有关有效 GUID 值的说明，请参阅“Wired For Management Baseline”。

InterfaceType: 指示接口是否以本机形式提供。该值表示请求的原始执行环境。请参阅“相关定义”。

Interface: 指向协议接口的指针。接口必须遵守协议定义的结构。如果结构与协议无关，则可以使用 `NULL`。

- 相关定义 (Related Definitions)

```

1 //*****
2 //EFI_HANDLE

```

```

3 //*****
4 typedef VOID *EFI_HANDLE;
5
6 //*****
7 //EFI_GUID
8 //*****
9 typedef struct {
10     UINT32 Data1;
11     UINT16 Data2;
12     UINT16 Data3;
13     UINT8 Data4[8];
14 } EFI_GUID;
15
16 //*****
17 //EFI_INTERFACE_TYPE
18 //*****
19 typedef enum {
20     EFI_NATIVE_INTERFACE
21 } EFI_INTERFACE_TYPE;

```

- 描述 (Description)

`InstallProtocolInterface()` 函数在设备句柄上安装协议接口 (GUID/协议接口结构对)。同一个 GUID 不能多次安装到同一个句柄上。如果尝试在句柄上安装重复的 GUID，则会产生 `EFI_INVALID_PARAMETER`。安装协议接口允许其他组件定位 `Handle`，以及安装在其上的接口。

安装协议接口时，固件会调用所有已注册的通知函数等待协议的安装。有关详细信息，请参阅 `EFI_BOOT_SERVICES.RegisterProtocolNotify()` 函数说明。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The protocol interface was installed.
<code>EFI_OUT_OF_RESOURCES</code>	Space for a new handle could not be allocated.
<code>EFI_INVALID_PARAMETER</code>	<code>Handle</code> is NULL
<code>EFI_INVALID_PARAMETER</code>	<code>Protocol</code> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<code>InterfaceType</code> is not EFI_NATIVE_INTERFACE .
<code>EFI_INVALID_PARAMETER</code>	<code>Protocol</code> is already installed on the handle specified by <code>Handle</code> .

图 39. table7-7_0

EFI_BOOT_SERVICES.UninstallProtocolInterface()

- 概要 (Summary)

从设备句柄中删除协议接口。建议使用 [UninstallMultipleProtocolInterfaces\(\)](#) 代替 [UninstallProtocolInterface\(\)](#)。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_UNINSTALL_PROTOCOL_INTERFACE) (
2     IN EFI_HANDLE    Handle,
3     IN EFI_GUID      *Protocol,
4     IN VOID          *Interface
5 );
```

- 参数 (Parameters)

Handle: 安装接口的句柄。如果 *Handle* 不是有效句柄，则返回 [EFI_INVALID_PARAMETER](#)。[EFI_HANDLE](#) 类型在 [EFI_BOOT_SERVICES.InstallProtocolInterface\(\)](#) 函数描述中定义。

Protocol: 接口的数字 ID。传入有效的 GUID 是调用者的责任。有关有效 GUID 值的说明，请参阅 “Wired For Management Baseline”。类型 [EFI_GUID](#) 在 [InstallProtocolInterface\(\)](#) 函数描述中定义。

Interface: 指向接口的指针。如果结构与 *Protocol* 无关，则可以使用 [NULL](#)。

- 描述 (Description)

[UninstallProtocolInterface\(\)](#) 函数从之前安装的句柄中删除协议接口。*Protocol* 和 *Interface* 值定义要从句柄中删除的协议接口。

调用者负责确保没有对已删除的协议接口的引用。在某些情况下，协议中没有突出的参考信息，因此协议一旦添加，就无法删除。示例包括控制台 I/O、块 I/O、磁盘 I/O 和（通常）设备协议句柄。

如果从句柄中删除最后一个协议接口，则该句柄将被释放并且不再有效。

此服务的扩展直接解决了上述部分中描述的限制。可能有一些驱动程序当前正在使用需要卸载的协议接口，因此盲目地从系统中删除协议接口可能是危险的。由于现在正在跟踪使用 [EFI_BOOT_SERVICES.OpenProtocol\(\)](#) 和 [EFI_BOOT_SERVICES.CloseProtocol\(\)](#) 引导服务的组件的协议接口的使用，因此可以实现此函数的安全版本。在删除协议接口之前，会尝试强制所有使用该协议接口的驱动程序停止使用该协议接口。这是通过调用启动服务 [EFI_BOOT_SERVICES.DisconnectController\(\)](#) 来完成的，该驱动程序当前打开了具有 [EFI_OPEN_PROTOCOL_BY_DRIVER](#) 或 [EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE](#) 属性的协议接口。

如果断开连接成功，那么这些代理将调用启动服务 [EFI_BOOT_SERVICES.CloseProtocol\(\)](#) 以释放协议接口。最后，所有具有 [EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL](#)、[EFI_OPEN_PROTOCOL_GET_PROTOCOL](#) 或 [EFI_OPEN_PROTOCOL_TEST_PROTOCOL](#) 属性的协议接口打开的代理都将关闭。如果剩余的任何代

理仍然打开协议接口，则不会从句柄中删除协议接口并返回 `EFI_ACCESS_DENIED`。此外，之前与引导服务 `DisconnectController()` 断开连接的所有驱动程序都将与引导服务 `EFI_BOOT_SERVICES.ConnectController()` 重新连接。如果没有剩余的代理正在使用协议接口，则协议接口从句柄中删除，如上所述。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The interface was removed.
<code>EFI_NOT_FOUND</code>	The interface was not found.
<code>EFI_ACCESS_DENIED</code>	The interface was not removed because the interface is still being used by a driver.
<code>EFI_INVALID_PARAMETER</code>	<i>Handle</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>Protocol</i> is NULL .

图 40. table7-7_1

`EFI_BOOT_SERVICES.ReinstallProtocolInterface()`

- 概要 (Summary)

在设备句柄上重新安装协议接口。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_REINSTALL_PROTOCOL_INTERFACE) (
2     IN EFI_HANDLE    Handle,
3     IN EFI_GUID      *Protocol,
4     IN VOID          *OldInterface,
5     IN VOID          *NewInterface
6 );

```

- 参数 (Parameters)

Handle: 要重新安装接口的句柄。如果 *Handle* 不是有效句柄，则返回 `EFI_INVALID_PARAMETER`。
`EFI_HANDLE` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

Protocol: 接口的数字 ID。传递有效的 GUID 是调用者的责任。有关有效 GUID 值的说明，请参阅“有线管理基线”。`EFI_GUID` 类型在 `InstallProtocolInterface()` 函数描述中定义。

OldInterface: 指向旧接口的指针。如果结构不与协议相关联，则可以使用 `NULL`。

NewInterface: 指向新接口的指针。如果结构不与协议相关联，则可以使用 `NULL`。

- 描述 (Description)

`ReinstallProtocolInterface()` 函数在设备句柄上重新安装协议接口。协议的 *OldInterface* 被 *NewInterface* 取代。*NewInterface* 可能与 *OldInterface* 相同。如果是，注册的协议通知会发生在句柄上，而不用替换句柄上的接口。

与 `InstallProtocolInterface()` 一样，任何已注册等待接口安装的进程都会收到通知。

调用者负责确保没有对要删除的 *OldInterface* 的引用。

- EFI 1.10 扩展 (EFI 1.10 Extension)

此服务的扩展直接解决了上一节中描述的限制。当前可能有一些驱动程序正在使用正在重新安装的协议接口。在这种情况下，替换系统中的协议接口可能是危险的。它可能导致状态不稳定，因为驱动程序可能会在重新安装新协议接口后尝试使用旧协议接口。由于现在正在跟踪使用 `EFI_BOOT_SERVICES.OpenProtocol()` 和 `EFI_BOOT_SERVICES.CloseProtocol()` 引导服务的组件的协议接口的使用情况，因此可以实现此功能的安全版本。

调用此函数时，首先调用引导服务 `UninstallProtocolInterface()`。这将保证所有代理当前正在使用协议接口 *OldInterface* 将停止使用 *OldInterface*。如果 `UninstallProtocolInterface()` 返回 `EFI_ACCESS_DENIED`，则此函数返回 `EFI_ACCESS_DENIED`，*OldInterface* 保留在 *Handle* 上，并且协议通知未被处理，因为 *NewInterface* 从未安装过。

如果 `UninstallProtocolInterface()` 成功，则调用引导服务 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 以将 *NewInterface* 放到 *Handle* 上。

最后，启动服务 `EFI_BOOT_SERVICES.ConnectController()` 被调用，因此所有被迫使用 `UninstallProtocolInterface()` 释放 *OldInterface* 的代理现在可以使用使用 `InstallProtocolInterface()` 安装的协议接口 *NewInterface*。在 *OldInterface* 被 *NewInterface* 替换后，任何已注册等待接口安装的进程都会收到通知。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The protocol interface was reinstalled.
<code>EFI_NOT_FOUND</code>	The <i>OldInterface</i> on the handle was not found.
<code>EFI_ACCESS_DENIED</code>	The protocol interface could not be reinstalled, because <i>OldInterface</i> is still being used by a driver that will not release it.
<code>EFI_INVALID_PARAMETER</code>	<i>Handle</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>Protocol</i> is NULL .

图 41. table7-7_2

`EFI_BOOT_SERVICES.RegisterProtocolNotify()`

- 概要 (Summary)

创建一个事件，每当为指定协议安装接口时，该事件就会发出信号。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_REGISTER_PROTOCOL_NOTIFY) (
2     IN EFI_GUID    *Protocol,
3     IN EFI_EVENT   Event,
4     OUT VOID       **Registration
5 );
```

- 参数 (Parameters)

Protocol: 事件要注册的协议的数字 ID。`EFI_GUID` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

Event: 每当为协议注册协议接口时，要发出信号的事件。`EFI_EVENT` 类型在 `CreateEvent()` 函数描述中定义。同一个 `EFI_EVENT` 可以用于多个协议通知注册。

Registration: 指向接收注册值（registration value）的内存位置的指针。该值必须被保存，并由 *Event* 的通知函数使用，以检索已添加 *Protocol* 类型的协议接口的句柄列表。

- 描述 (Description)

`RegisterProtocolNotify()` 函数创建一个事件，每当通过 `InstallProtocolInterface()` 或 `EFI_BOOT_SERVICES.ReinstallProtocolInterface()` 为 *Protocol* 安装协议接口时，该事件就会发出信号。

一旦发出事件信号，就可以调用 `EFI_BOOT_SERVICES.LocateHandle()` 函数来识别新安装或重新安装的支持协议的句柄。`EFI_BOOT_SERVICES.RegisterProtocolNotify()` 中的 *Registration* 参数对应 `LocateHandle()` 中的 *SearchKey* 参数。请注意，如果句柄多次重新安装目标协议 ID，则可能会多次返回同一个句柄。这对于可移动媒体设备来说是典型的，因为当这样的设备再次出现时，它会重新安装 Block I/O 协议以指示需要再次检查该设备。作为响应，分层磁盘 I/O 和简单文件系统协议可能会重新安装它们的协议以指示它们可以被重新检查，等等。

已为协议接口通知注册的事件可以通过调用 `CloseEvent()` 注销。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The notification event has been registered.
<code>EFI_OUT_OF_RESOURCES</code>	Space for the notification event could not be allocated.
<code>EFI_INVALID_PARAMETER</code>	<i>Protocol</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>Event</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>Registration</i> is NULL .

图 42. table7-7_3

EFI_BOOT_SERVICES.LocateHandle()

• 概要 (Summary)

返回支持指定协议的句柄数组。

• 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_LOCATE_HANDLE) (
2     IN EFI_LOCATE_SEARCH_TYPE  SearchType,
3     IN EFI_GUID                 *Protocol OPTIONAL,
4     IN VOID                     *SearchKey OPTIONAL,
5     IN OUT UINTN                *BufferSize,
6     OUT EFI_HANDLE               *Buffer
7 );

```

• 参数 (Parameters)

SearchType: 指定要返回的句柄。`EFI_LOCATE_SEARCH_TYPE` 类型在 “相关定义” 中定义。

Protocol: 指定要搜索的协议。此参数仅在 *SearchType* 为 `ByProtocol` 时有效。`EFI_GUID` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

SearchKey: 指定搜索关键字。如果 *SearchType* 是 `AllHandles` 或 `ByProtocol`, 则忽略此参数。如果 *SearchType* 是 `ByRegisterNotify`, 则参数必须是函数 `EFI_BOOT_SERVICES.RegisterProtocolNotify()` 返回的 *Registration* (注册) 值。

BufferSize: 输入时, *Buffer* 的大小 (以字节为单位)。在输出时, *Buffer* 中返回的数组的字节大小 (如果缓冲区足够大) 或获取数组所需的缓冲区大小 (以字节为单位) (如果缓冲区不够大)。

Buffer: 返回数组的缓冲区。`EFI_HANDLE` 类型在 `InstallProtocolInterface()` 函数描述中定义。

• 相关定义 (Related Definitions)

```
1 //*****
```

```

2 // EFI_LOCATE_SEARCH_TYPE
3 //*****
4 typedef enum {
5     AllHandles,
6     ByRegisterNotify,
7     ByProtocol
8 } EFI_LOCATE_SEARCH_TYPE;

```

AllHandles: *Protocol* 和 *SearchKey* 被忽略，函数返回系统中每个句柄的数组。

ByRegisterNotify: *SearchKey* 提供由 `EFI_BOOT_SERVICES.RegisterProtocolNotify()` 返回的注册值。该函数返回下一个新的注册句柄。一次只返回一个句柄，从第一个开始，调用者必须循环直到不再返回句柄。此搜索类型忽略协议。

ByProtocol: 返回所有支持协议的句柄。此搜索类型忽略 *SearchKey*。

- 描述 (Description)

`LocateHandle()` 函数返回匹配 *SearchType* 请求的句柄数组。如果 *BufferSize* 的输入值太小，函数返回 `EFI_BUFFER_TOO_SMALL` 并将 *BufferSize* 更新为获取数组所需的缓冲区大小。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The array of handles was returned.
<code>EFI_NOT_FOUND</code>	No handles match the search.
<code>EFI_BUFFER_TOO_SMALL</code>	The <i>BufferSize</i> is too small for the result. <i>BufferSize</i> has been updated with the size needed to complete the request.
<code>EFI_INVALID_PARAMETER</code>	<i>SearchType</i> is not a member of <code>EFI_LOCATE_SEARCH_TYPE</code> .
<code>EFI_INVALID_PARAMETER</code>	<i>SearchType</i> is <code>ByRegisterNotify</code> and <i>SearchKey</i> is <code>NULL</code> .
<code>EFI_INVALID_PARAMETER</code>	<i>SearchType</i> is <code>ByProtocol</code> and <i>Protocol</i> is <code>NULL</code> .
<code>EFI_INVALID_PARAMETER</code>	One or more matches are found and <i>BufferSize</i> is <code>NULL</code> .
<code>EFI_INVALID_PARAMETER</code>	<i>BufferSize</i> is large enough for the result and <i>Buffer</i> is <code>NULL</code> .

图 43. table7-7_4

`EFI_BOOT_SERVICES.HandleProtocol()`

- 概要 (Summary)

查询句柄以确定它是否支持指定的协议。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_HANDLE_PROTOCOL) (
2     IN EFI_HANDLE    Handle,
3     IN EFI_GUID     *Protocol,
4     OUT VOID        **Interface
5 );
```

- 参数 (Parameters)

Handle: 被查询的句柄。如果 *Handle* 为 `NULL`, 则返回 `EFI_INVALID_PARAMETER`。`EFI_HANDLE` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

Protocol: 协议的已发布唯一标识符。传递有效的 GUID 是调用者的责任。有关有效 GUID 值的说明, 请参阅“有线管理基线”。`EFI_GUID` 类型在 `InstallProtocolInterface()` 函数描述中定义。

Interface: 提供返回指向相应协议接口的指针的地址。如果结构未与 *Protocol* 相关联, 则 `**Interface`* 将返回 `NULL`。

- 描述 (Description)

`HandleProtocol()` 函数查询 *Handle* 以确定它是否支持协议。如果是, 则返回接口指向相应协议接口的指针。然后可以将接口传递给任何协议服务以识别请求的上下文。

- EFI 1.10 扩展 (EFI 1.10 Extension)

`HandleProtocol()` 函数仍然可供旧的 EFI 应用程序和驱动程序使用。但是, 所有新的应用程序和驱动程序都应该使用 `EFI_BOOT_SERVICES.OpenProtocol()` 代替 `HandleProtocol()`。以下代码片段显示了使用 `OpenProtocol()` 的 `HandleProtocol()` 的可能实现。变量 `EfiCoreImageHandle` 是 EFI 核心的镜像句柄。

```
1 typedef EFI_STATUS HandleProtocol (
2     IN EFI_HANDLE    Handle,
3     IN EFI_GUID     *Protocol,
4     OUT VOID        **Interface
5 )
6 {
7     return OpenProtocol (
8         Handle,
9         Protocol,
10        Interface,
11        EfiCoreImageHandle,
12        NULL,
13        EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL
14 );
```

```
14 );
15 }
```

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The interface information for the specified protocol was returned.
<code>EFI_UNSUPPORTED</code>	The device does not support the specified protocol.
<code>EFI_INVALID_PARAMETER</code>	<i>Handle</i> is NULL ..
<code>EFI_INVALID_PARAMETER</code>	<i>Protocol</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>Interface</i> is NULL .

图 44. table7-7_5

`EFI_BOOT_SERVICES.LocateDevicePath()`

- 概要 (Summary)

在支持指定协议的设备路径上找到设备的句柄。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_LOCATE_DEVICE_PATH) (
2     IN EFI_GUID                 *Protocol,
3     IN OUT EFI_DEVICE_PATH_PROTOCOL **DevicePath,
4     OUT EFI_HANDLE               *Device
5 );
```

- 参数 (Parameters)

Protocol: 要搜索的协议。`EFI_GUID` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

DevicePath: 输入时，指向指向设备路径的指针。在输出时，设备路径指针被修改为指向设备路径的剩余部分——也就是说，当函数找到最近的句柄时，它将设备路径分成两部分，剥离前面的部分，并返回剩余部分。`EFI_DEVICE_PATH_PROTOCOL` 在第 10.2 节中定义。

Device: 指向返回的设备句柄的指针。`EFI_HANDLE` 类型在 `InstallProtocolInterface()` 函数描述中定义。

- 描述 (Description)

`LocateDevicePath()` 函数在 *DevicePath* 上定位所有支持协议的设备，并将句柄返回到离 *DevicePath* 最近的设备。*DevicePath* 在匹配的设备路径节点上前进。

此函数对于定位要从逻辑父设备驱动程序使用的协议接口的正确实例很有用。例如，目标设备驱动程序可能会发出带有自己设备路径的请求，并定位接口在其总线上执行 I/O。它还可以与包含文件路径的设备路径一起使用，以剥离设备路径的文件系统部分，将文件路径和句柄留给访问文件所需的文件系统驱动程序。

如果 *DevicePath* 的句柄支持协议（直接匹配），则生成的设备路径将前进到设备路径终止符节点。如果 *DevicePath* 是多实例设备路径，该函数将在第一个实例上运行。

- 返回的状态码（Status Codes Returned）

<code>EFI_SUCCESS</code>	The resulting handle was returned.
<code>EFI_NOT_FOUND</code>	No handles matched the search.
<code>EFI_INVALID_PARAMETER</code>	<i>Protocol</i> is NULL
<code>EFI_INVALID_PARAMETER</code>	<i>DevicePath</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	A handle matched the search and <i>Device</i> is NULL .

图 45. table7-7_6

EFI_BOOT_SERVICES.OpenProtocol()

- 概要（Summary）

查询句柄以确定它是否支持指定的协议。如果句柄支持该协议，它将代表调用代理打开该协议。这是 EFI 引导服务 `EFI_BOOT_SERVICES.HandleProtocol()` 的扩展版本。

- 原型（Prototype）

```

1 typedef EFI_STATUS (EFIAPI *EFI_OPEN_PROTOCOL) (
2     IN EFI_HANDLE    Handle,
3     IN EFI_GUID      *Protocol,
4     OUT VOID        **Interface OPTIONAL,
5     IN EFI_HANDLE   AgentHandle,
6     IN EFI_HANDLE   ControllerHandle,
7     IN UINT32       Attributes
8 );

```

- 参数（Parameters）

Handle: 正在打开的协议接口的句柄。

Protocol: 协议的已发布唯一标识符。传递有效的 GUID 是调用者的责任。有关有效 GUID 值的说明，请参阅“有线管理基线”。

Interface: 提供返回指向相应协议接口的指针的地址。如果结构未与协议相关联，则 **Interface*** 将返回 **NULL**。此参数是可选的，如果 **Attributes** 是 **EFI_OPEN_PROTOCOL_TEST_PROTOCOL**，将被忽略。

AgentHandle: 打开由 **Protocol** 和 **Interface** 指定的协议接口的代理的句柄。对于遵循 UEFI 驱动程序模型的代理，此参数是包含 **EFI_DRIVER_BINDING_PROTOCOL** 实例的句柄，该实例由打开协议接口的 UEFI 驱动程序生成。对于 UEFI 应用程序，这是打开协议接口的 UEFI 应用程序的镜像句柄。对于使用 **HandleProtocol()** 打开协议接口的应用程序，此参数是 **EFI** 固件的镜像句柄。

ControllerHandle: 如果打开协议的代理是遵循 UEFI 驱动程序模型的驱动程序，则此参数是需要协议接口的控制器句柄。如果代理不遵循 UEFI 驱动程序模型，则此参数是可选的，可以为 **NULL**。

Attributes: **Handle** 和 **Protocol** 指定的协议接口的打开方式。有关合法属性列表，请参阅“相关定义”。

- 描述 (Description)

此函数在 **handle** 指定的句柄上为 **protocol** 指定的协议打开协议接口。前三个参数与 **EFI_BOOT_SERVICES.HandleProtocol()** 相同。唯一的区别是打开协议接口的代理在 **EFI** 的内部句柄数据库中被跟踪。跟踪由 UEFI 驱动程序模型使用，还用于确定卸载或重新安装协议接口是否安全。

打开协议接口的代理由 **AgentHandle**、**ControllerHandle** 和 **Attributes** 指定。如果可以打开协议接口，则将 **AgentHandle**、**ControllerHandle** 和 **Attributes** 添加到正在使用 **Handle** 和 **Protocol** 指定的协议接口的代理列表中。另外，在 **Interface** 中返回协议接口，返回 **EFI_SUCCESS**。如果 **Attributes** 是 **TEST_PROTOCOL**，那么 **Interface** 是可选的，可以为 **NULL**。

此函数调用可能返回错误的原因有很多。如果返回错误，则不会将 **AgentHandle**、**ControllerHandle** 和 **Attributes** 添加到正在使用 **Handle** 和 **Protocol** 指定的协议接口的代理列表中。对于除 **EFI_UNSUPPORTED** 和 **EFI_ALREADY_STARTED** 之外的所有错误情况，接口均未修改地返回，当 **EFI_UNSUPPORTED** 和 **Attributes** 不是 **EFI_OPEN_PROTOCOL_TEST_PROTOCOL** 时，**Interface*** 将返回 **NULL**，当 **EFI_ALREADY_STARTED** 时，协议接口将在 **Interface*** 中返回。

以下是此函数返回 **EFI_SUCCESS** 之前必须检查的条件列表：

- 如果 **Protocol** 为 **NULL**，则返回 **EFI_INVALID_PARAMETER**；
- 如果 **Interface** 为 **NULL** 且 **Attributes** 不是 **TEST_PROTOCOL**，则返回 **EFI_INVALID_PARAMETER**；
- 如果 **Handle** 为 **NULL**，则返回 **EFI_INVALID_PARAMETER**；
- 如果句柄不支持协议，则返回 **EFI_UNSUPPORTED**；
- 如果 **Attributes** 不是合法值，则返回 **EFI_INVALID_PARAMETER**。“相关定义”中列出了合法值；
- 如果 **Attributes** 为 **BY_CHILD_CONTROLLER**、**BY_DRIVER**、**EXCLUSIVE** 或 **BY_DRIVER|EXCLUSIVE**，并且 **AgentHandle** 为 **NULL**，则返回 **EFI_INVALID_PARAMETER**；
- 如果 **Attributes** 为 **BY_CHILD_CONTROLLER**、**BY_DRIVER** 或 **BY_DRIVER|EXCLUSIVE**，并且 **ControllerHandle** 为 **NULL**，则返回 **EFI_INVALID_PARAMETER**；

- 如果 *Attributes* 是 `BY_CHILD_CONTROLLER` 并且 *Handle* 与 *ControllerHandle* 相同, 则返回 `EFI_INVALID_PARAMETER`;
- 如果 *Attributes* 是 `BY_DRIVER`、`BY_DRIVER | EXCLUSIVE` 或 `EXCLUSIVE`, 并且协议接口的打开列表中有任何项的 *Attributes* 为 `EXCLUSIVE` 或 `BY_DRIVER | EXCLUSIVE`, 则返回 `EFI_ACCESS_DENIED`;
- 如果 *Attributes* 为 `BY_DRIVER`, 并且协议接口的打开列表中有任何项的属性为 `BY_DRIVER`, 并且 *AgentHandle* 是打开列表项中相同的代理句柄, 则返回 `EFI_ALREADY_STARTED`;
- 如果 *Attributes* 为 `BY_DRIVER`, 并且协议接口的打开列表中有任何项的属性为 `BY_DRIVER`, 并且 *AgentHandle* 与打开列表项中的代理句柄不同, 则返回 `EFI_ACCESS_DENIED`;
- 如果属性为 `BY_DRIVER | EXCLUSIVE`, 并且协议接口的打开列表中有任何项目的属性为 `BY_DRIVER | EXCLUSIVE`, 并且 *AgentHandle* 是打开列表项目中的相同代理句柄, 则返回 `EFI_ALREADY_STARTED`;
- 如果属性为 `BY_DRIVER | EXCLUSIVE`, 并且协议接口的打开列表中有任何项目的属性为 `BY_DRIVER | EXCLUSIVE`, 并且 *AgentHandle* 与打开列表项目中的代理句柄不同, 则返回 `EFI_ACCESS_DENIED`;
- 如果属性为 `BY_DRIVER | EXCLUSIV`, 并且协议接口的打开列表中有一个项目的属性为 `BY_DRIVER`, 则启动服务 `EFI_boot_SERVICES`。为打开列表上的驱动程序调用 `DisconnectController()`。如果在调用 `DisconnectController()` 后协议接口的打开列表中还有一个 *Attributes* 为 `BY_DRIVER` 的项, 则返回 `EFI_ACCESS_DENIED`;

- 相关定义 (Related Definitions)

```

1 #define EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL      0x00000001
2 #define EFI_OPEN_PROTOCOL_GET_PROTOCOL           0x00000002
3 #define EFI_OPEN_PROTOCOL_TEST_PROTOCOL          0x00000004
4 #define EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER    0x00000008
5 #define EFI_OPEN_PROTOCOL_BY_DRIVER              0x00000010
6 #define EFI_OPEN_PROTOCOL_EXCLUSIVE             0x00000020

```

以下是 *Attributes* 参数的合法值列表, 以及每个值的使用方式。

`BY_HANDLE_PROTOCOL`: 用于 `EFI_BOOT_SERVICES.HandleProtocol()` 的实现。由于 `EFI_BOOT_SERVICES.OpenProtocol()` 执行与 `HandleProtocol()` 相同的功能并具有附加功能, 因此 `HandleProtocol()` 可以使用此属性值简单地调用 `OpenProtocol()`。

`GET_PROTOCOL`: 由驱动程序用于从句柄获取协议接口。使用这种打开方式时必须小心, 因为如果协议接口被卸载或重新安装, 以这种方式打开协议接口的驱动程序将不会得到通知。调用者也不需要使用 `EFI_BOOT_SERVICES.CloseProtocol()` 关闭协议接口。

`TEST_PROTOCOL`: 由驱动程序用来测试句柄上是否存在协议接口。接口对于这个属性值是可选的, 所

以被忽略，调用者应该只使用返回状态码。调用者也不需要使用 `CloseProtocol()` 关闭协议接口。

BY_CHILD_CONTROLLER: 总线驱动程序使用它来表明协议接口正在被总线的子控制器之一使用。此信息由引导服务 `EFI_BOOT_SERVICES.ConnectController()` 用于递归连接所有子控制器，由引导服务 `EFI_BOOT_SERVICES.DisconnectController()` 用于获取总线驱动程序创建的子控制器列表。

BY_DRIVER: 由驱动程序用来访问协议接口。使用此模式时，如果重新安装或卸载协议接口，`EFI_BOOT_SERVICES.DisconnectController()` 将调用驱动程序的 `Stop()` 函数。一旦一个协议接口被具有该属性的驱动程序打开，其他驱动程序将不允许打开具有 **BY_DRIVER** 属性的相同协议接口。

BY_DRIVER|EXCLUSIVE: 由驱动程序用来获得对协议接口的独占访问权。如果任何其他驱动程序使用 **BY_DRIVER** 属性打开协议接口，则将尝试使用 `DisconnectController()` 删除它们。

EXCLUSIVE: 应用程序使用它来获得对协议接口的独占访问权。如果任何驱动程序使用 **BY_DRIVER** 属性打开了协议接口，则将尝试通过调用驱动程序的 `Stop()` 函数来删除它们。

- 返回的状态码（Status Codes Returned）

EFI_SUCCESS	An item was added to the open list for the protocol interface, and the protocol interface was returned in <i>Interface</i> .
EFI_INVALID_PARAMETER	<i>Protocol</i> is NULL .
EFI_INVALID_PARAMETER	<i>Interface</i> is NULL , and <i>Attributes</i> is not TEST_PROTOCOL .
EFI_INVALID_PARAMETER	<i>Handle</i> is NULL ..
EFI_UNSUPPORTED	<i>Handle</i> does not support <i>Protocol</i> .
EFI_INVALID_PARAMETER	<i>Attributes</i> is not a legal value.
EFI_INVALID_PARAMETER	<i>Attributes</i> is BY_CHILD_CONTROLLER and <i>AgentHandle</i> is NULL ..
EFI_INVALID_PARAMETER	<i>Attributes</i> is BY_DRIVER and <i>AgentHandle</i> is NULL .
EFI_INVALID_PARAMETER	<i>Attributes</i> is BY_DRIVER EXCLUSIVE and <i>AgentHandle</i> is NULL .
EFI_INVALID_PARAMETER	<i>Attributes</i> is EXCLUSIVE and <i>AgentHandle</i> is NULL .
EFI_INVALID_PARAMETER	<i>Attributes</i> is BY_CHILD_CONTROLLER and <i>ControllerHandle</i> is NULL .
EFI_INVALID_PARAMETER	<i>Attributes</i> is BY_DRIVER and <i>ControllerHandle</i> is NULL .
EFI_INVALID_PARAMETER	<i>Attributes</i> is BY_DRIVER EXCLUSIVE and <i>ControllerHandle</i> is NULL .
EFI_INVALID_PARAMETER	<i>Attributes</i> is BY_CHILD_CONTROLLER and <i>Handle</i> is identical to <i>ControllerHandle</i> .
EFI_ACCESS_DENIED	<i>Attributes</i> is BY_DRIVER and there is an item on the open list with an attribute of BY_DRIVER EXCLUSIVE or EXCLUSIVE .
EFI_ACCESS_DENIED	<i>Attributes</i> is BY_DRIVER EXCLUSIVE and there is an item on the open list with an attribute of EXCLUSIVE .
EFI_ACCESS_DENIED	<i>Attributes</i> is EXCLUSIVE and there is an item on the open list with an attribute of BY_DRIVER EXCLUSIVE or EXCLUSIVE .
EFI_ALREADY_STARTED	<i>Attributes</i> is BY_DRIVER and there is an item on the open list with an attribute of BY_DRIVER whose agent handle is the same as <i>AgentHandle</i> .
EFI_ACCESS_DENIED	<i>Attributes</i> is BY_DRIVER and there is an item on the open list with an attribute of BY_DRIVER whose agent handle is different than <i>AgentHandle</i> .
EFI_ALREADY_STARTED	<i>Attributes</i> is BY_DRIVER EXCLUSIVE and there is an item on the open list with an attribute of BY_DRIVER EXCLUSIVE whose agent handle is the same as <i>AgentHandle</i> .
EFI_ACCESS_DENIED	<i>Attributes</i> is BY_DRIVER EXCLUSIVE and there is an item on the open list with an attribute of BY_DRIVER EXCLUSIVE whose agent handle is different than <i>AgentHandle</i> .
EFI_ACCESS_DENIED	<i>Attributes</i> is BY_DRIVER EXCLUSIVE or EXCLUSIVE and there are items in the open list with an attribute of BY_DRIVER that could not be removed when EFI_BOOT_SERVICES.DisconnectController() was called for that open item.

- 示例 (Examples)

```
1 EFI_BOOT_SERVICES           *gBS;
2 EFI_HANDLE                  ImageHandle;
3 EFI_DRIVER_BINDING_PROTOCOL *This;
4 IN EFI_HANDLE                ControllerHandle,
5 extern EFI_GUID              gEfiXyzIoProtocol;
6 EFI_XYZ_IO_PROTOCOL          *XyzIo;
7 EFI_STATUS                   Status;
8
9 //
10 // EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL example
11 // Retrieves the XYZ I/O Protocol instance from ControllerHandle
12 // The application that is opening the protocol is identified by ImageHandle
13 // Possible return status codes:
14 // EFI_SUCCESS : The protocol was opened and returned in XyzIo
15 // EFI_UNSUPPORTED : The protocol is not present on ControllerHandle
16 //
17 Status = gBS->OpenProtocol (
18     ControllerHandle,
19     &gEfiXyzIoProtocol,
20     &XyzIo,
21     ImageHandle,
22     NULL,
23     EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL
24 );
25
26 //
27 // EFI_OPEN_PROTOCOL_GET_PROTOCOL example
28 // Retrieves the XYZ I/O Protocol instance from ControllerHandle
29 // The driver that is opening the protocol is identified by the
30 // Driver Binding Protocol instance This. This->DriverBindingHandle
31 // identifies the agent that is opening the protocol interface, and it
32 // is opening this protocol on behalf of ControllerHandle.
33 // Possible return status codes:
34 // EFI_SUCCESS : The protocol was opened and returned in XyzIo
35 // EFI_UNSUPPORTED : The protocol is not present on ControllerHandle
36 //
37 Status = gBS->OpenProtocol (
38     ControllerHandle,
39     &gEfiXyzIoProtocol,
40     &XyzIo,
41     This->DriverBindingHandle,
```

```
42         ControllerHandle,
43         EFI_OPEN_PROTOCOL_GET_PROTOCOL
44     );
45
46 // 
47 // EFI_OPEN_PROTOCOL_TEST_PROTOCOL example
48 // Tests to see if the XYZ I/O Protocol is present on ControllerHandle
49 // The driver that is opening the protocol is identified by the
50 // Driver Binding Protocol instance This. This->DriverBindingHandle
51 // identifies the agent that is opening the protocol interface, and it
52 // is opening this protocol on behalf of ControllerHandle.
53 // EFI_SUCCESS : The protocol was opened and returned in XyzIo
54 // EFI_UNSUPPORTED : The protocol is not present on ControllerHandle
55 //
56 Status = gBS->OpenProtocol (
57     ControllerHandle,
58     &gEfiXyzIoProtocol,
59     NULL,
60     This->DriverBindingHandle,
61     ControllerHandle,
62     EFI_OPEN_PROTOCOL_TEST_PROTOCOL
63 );
64
65 //
66 // EFI_OPEN_PROTOCOL_BY_DRIVER example
67 // Opens the XYZ I/O Protocol on ControllerHandle
68 // The driver that is opening the protocol is identified by the
69 // Driver Binding Protocol instance This. This->DriverBindingHandle
70 // identifies the agent that is opening the protocol interface, and it
71 // is opening this protocol on behalf of ControllerHandle.
72 // Possible return status codes:
73 // EFI_SUCCESS : The protocol was opened and returned in XyzIo
74 // EFI_UNSUPPORTED : The protocol is not present on ControllerHandle
75 // EFI_ALREADY_STARTED : The protocol is already opened by the driver
76 // EFI_ACCESS_DENIED : The protocol is managed by a different driver
77 //
78 Status = gBS->OpenProtocol (
79     ControllerHandle,
80     &gEfiXyzIoProtocol,
81     &XyzIo,
82     This->DriverBindingHandle,
83     ControllerHandle,
84     EFI_OPEN_PROTOCOL_BY_DRIVER
```

```
85      );
86 //
87 // EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE example
88 // Opens the XYZ I/O Protocol on ControllerHandle
89 // The driver that is opening the protocol is identified by the
90 // Driver Binding Protocol instance This. This->DriverBindingHandle
91 // identifies the agent that is opening the protocol interface, and it
92 // is opening this protocol on behalf of ControllerHandle.
93 // Possible return status codes:
94 // EFI_SUCCESS : The protocol was opened and returned in XyzIo. If
95 // a different driver had the XYZ I/O Protocol opened
96 // BY_DRIVER, then that driver was disconnected to
97 // allow this driver to open the XYZ I/O Protocol.
98 // EFI_UNSUPPORTED : The protocol is not present on ControllerHandle
99 // EFI_ALREADY_STARTED : The protocol is already opened by the driver
100 // EFI_ACCESS_DENIED : The protocol is managed by a different driver that
101 // already has the protocol opened with an EXCLUSIVE attribute.
102 //
103 //
104 Status = gBS->OpenProtocol (
105     ControllerHandle,
106     &gEfiXyzIoProtocol,
107     &XyzIo,
108     This->DriverBindingHandle,
109     ControllerHandle,
110     EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE
111 );
```

EFI_BOOT_SERVICES.CloseProtocol()

- 概要 (Summary)

关闭使用 `EFI_BOOT_SERVICES.OpenProtocol()` 打开的句柄上的协议。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_CLOSE_PROTOCOL) (
2     IN EFI_HANDLE    Handle,
3     IN EFI_GUID     *Protocol,
4     IN EFI_HANDLE    AgentHandle,
5     IN EFI_HANDLE    ControllerHandle
6 );
```

- 参数 (Parameters)

Handle: 以前用 `OpenProtocol()` 打开的协议接口的句柄，现在正在关闭。

Protocol: 协议的已发布唯一标识符。传递有效的 GUID 是调用者的责任。有关有效 GUID 值的说明，请参阅“有线管理基线”。

AgentHandle: 关闭协议接口的代理句柄。对于遵循 UEFI 驱动程序模型的代理，此参数是包含 `EFI_DRIVER_BINDING_PROTOCOL` 实例的句柄，该实例由打开协议接口的 UEFI 驱动程序生成。对于 UEFI 应用程序，这是 UEFI 应用程序的镜像句柄。对于使用 `EFI_BOOT_SERVICES.HandleProtocol()` 打开协议接口的应用程序，这将是 EFI 固件的镜像句柄。

ControllerHandle: 如果打开协议的代理是遵循 UEFI 驱动模型的驱动，那么这个参数就是需要协议接口的控制器句柄。如果代理不遵循 UEFI 驱动程序模型，则此参数是可选的，可以为 `NULL`。

- 描述 (Description)

此函数更新句柄数据库以显示由 *Handle* 和 *Protocol* 指定的协议实例不再被指定的 *AgentHandle* 和 *ControllerHandle* 的代理和控制器所需要。如果 *Handle* 或 *AgentHandle* 为 `NULL`，则返回 `EFI_INVALID_PARAMETER`。

如果 *ControllerHandle* 不为 `NULL`，并且 *ControllerHandle* 为 `NULL`，则返回 `EFI_INVALID_PARAMETER`。如果 *Protocol* 为 `NULL`，则返回 `EFI_INVALID_PARAMETER`。

如果 *Handle* 指定的句柄不支持 *Protocol* 指定的接口，则返回 `EFI_NOT_FOUND`。

如果 *Handle* 指定的句柄支持 *Protocol* 指定的接口，则检查 *Protocol* 和 *Handle* 指定的协议实例是否由 *AgentHandle* 和 *ControllerHandle* 使用 `EFI_BOOT_SERVICES.OpenProtocol()` 打开。如果协议实例未被 *AgentHandle* 和 *ControllerHandle* 打开，则返回 `EFI_NOT_FOUND`。如果协议实例由 *AgentHandle* 和 *ControllerHandle* 打开，则所有这些引用都将从句柄数据库中删除，并返回 `EFI_SUCCESS`。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The protocol instance was closed.
EFI_INVALID_PARAMETER	<i>Handle</i> is NULL .
EFI_INVALID_PARAMETER	<i>AgentHandle</i> is NULL .
EFI_INVALID_PARAMETER	<i>ControllerHandle</i> is not NULL and <i>ControllerHandle</i> is NULL .
EFI_INVALID_PARAMETER	<i>Protocol</i> is NULL .
EFI_NOT_FOUND	<i>Handle</i> does not support the protocol specified by <i>Protocol</i> .
EFI_NOT_FOUND	The protocol interface specified by <i>Handle</i> and <i>Protocol</i> is not currently open by <i>AgentHandle</i> and <i>ControllerHandle</i> .

图 47. table7-7_8

- 示例 (Examples)

```

1 EFI_BOOT_SERVICES           *gBS;
2 EFI_HANDLE                  ImageHandle;
3 EFI_DRIVER_BINDING_PROTOCOL *This;
4 IN EFI_HANDLE                ControllerHandle,
5 extern EFI_GUID              gEfiXyzIoProtocol;
6 EFI_STATUS                  Status;
7
8 //
9 // Close the XYZ I/O Protocol that was opened on behalf of ControllerHandle
10 //
11 Status = gBS->CloseProtocol (
12     ControllerHandle,
13     &gEfiXyzIoProtocol,
14     This->DriverBindingHandle,
15     ControllerHandle
16 );
17
18 //
19 // Close the XYZ I/O Protocol that was opened with BY_HANDLE_PROTOCOL
20 //
21 Status = gBS->CloseProtocol (
22     ControllerHandle,
23     &gEfiXyzIoProtocol,
24     ImageHandle,

```

```

25      NULL
26 );

```

EFI_BOOT_SERVICES.OpenProtocolInformation()

- 概要 (Summary)

检索当前已打开协议接口的代理列表。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_OPEN_PROTOCOL_INFORMATION) (
2     IN EFI_HANDLE                               Handle,
3     IN EFI_GUID                                *Protocol,
4     OUT EFI_OPEN_PROTOCOL_INFORMATION_ENTRY **EntryBuffer,
5     OUT UINTN                                  *EntryCount
6 );

```

- 参数 (Parameters)

Handle: 正在查询的协议接口的句柄。

Protocol: 协议的已发布唯一标识符。传递有效的 GUID 是调用者的责任。有关有效 GUID 值的说明, 请参阅 “有线管理基线”。

EntryBuffer: 指向 `EFI_OPEN_PROTOCOL_INFORMATION_ENTRY` 结构形式的开放协议信息缓冲区的指针。该类型的声明见 “相关定义”。缓冲区由该服务分配, 当调用者不再需要缓冲区的内容时, 调用者有责任释放该缓冲区。

EntryCount: 指向 *EntryBuffer* 中 *Entries* 数目的指针。

- 相关定义 (Related Definitions)

```

1 typedef struct {
2     EFI_HANDLE AgentHandle;
3     EFI_HANDLE ControllerHandle;
4     UINT32 Attributes;
5     UINT32 OpenCount;
6 } EFI_OPEN_PROTOCOL_INFORMATION_ENTRY;

```

- 描述 (Description)

此函数分配并返回 `EFI_OPEN_PROTOCOL_INFORMATION_ENTRY` 结构的缓冲区。缓冲区在 *EntryBuffer* 中返回, *Entries* 数量在 *EntryCount* 中返回。

如果 *Handle* 指定的句柄不支持 *Protocol* 指定的接口, 则返回 `EFI_NOT_FOUND`。

如果 *Handle* 指定的句柄支持 *Protocol* 指定的接口, 则 *EntryBuffer* 被分配启动服务 `EFI_BOOT_SERVICES`

.AllocatePool(), EntryCount 被设置为 EntryBuffer 中的 Entries 数量。EntryBuffer 的每个 Entry 都填充了打开协议接口时传递给 EFI_BOOT_SERVICES.OpenProtocol() 的镜像句柄、控制器句柄和属性。OpenCount 字段显示协议接口已被 ImageHandle、ControllerHandle 和 Attributes 指定的代理打开的次数。EntryBuffer 的内容填满后，返回 EFI_SUCCESS。当调用者不再需要 EntryBuffer 的内容时，调用者有责任在 EntryBuffer 上调用 EFI_BOOT_SERVICES.FreePool()。

如果没有足够的资源可用于分配 EntryBuffer，则返回 EFI_OUT_OF_RESOURCES。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The open protocol information was returned in <i>EntryBuffer</i> , and the number of entries was returned <i>EntryCount</i> .
EFI_NOT_FOUND	<i>Handle</i> does not support the protocol specified by <i>Protocol</i> .
EFI_OUT_OF_RESOURCES	There are not enough resources available to allocate <i>EntryBuffer</i> .

图 48. table7-7_9

- 示例 (Examples)

有关如何使用 LocateHandleBuffer()、EFI_BOOT_SERVICES.ProtocolsPerHandle()、OpenProtocol() 和 EFI_BOOT_SERVICES.OpenProtocolInformation() 遍历整个句柄数据库的示例，请参阅 EFI_BOOT_SERVICES.LocateHandleBuffer() 函数描述中的示例。

EFI_BOOT_SERVICES.ConnectController()

- 概要 (Summary)

将一个或多个驱动程序连接到控制器。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_CONNECT_CONTROLLER) (
2     IN EFI_HANDLE           ControllerHandle,
3     IN EFI_HANDLE           *DriverImageHandle OPTIONAL,
4     IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL,
5     IN BOOLEAN              Recursive
6 );

```

- 参数 (Parameters)

ControllerHandle: 驱动程序要连接到的控制器的句柄。

DriverImageHandle: 指向支持 EFI_DRIVER_BINDING_PROTOCOL 的有序列表句柄的指针。该列表以 NULL 句柄值终止。这些句柄是驱动程序绑定协议的候选对象，该协议将管理由 *ControllerHandle* 指定的控

制器。这是一个可选参数，可以为 `NULL`。此参数通常用于调试新驱动程序。

`RemainingDevicePath`: 指向设备路径的指针，该路径指定由 `ControllerHandle` 指定的控制器的子级。这是一个可选参数，可以为 `NULL`。如果它为 `NULL`，则将创建 `ControllerHandle` 的所有子项的句柄。此参数不变的传递给附加到 `ControllerHandle` 的 `EFI_DRIVER_BINDING_PROTOCOL` 的 `Supported()` 和 `Start()` 服务。

`Recursive`: 如果为 `TRUE`，则递归调用 `ConnectController()`，直到创建了由 `ControllerHandle` 指定的控制器下的整个控制器树。如果为 `FALSE`，则控制器树仅展开一级。

- 描述 (Description)

此函数将一个或多个驱动程序连接到 `ControllerHandle` 指定的控制器。如果 `ControllerHandle` 为 `NULL`，则返回 `EFI_INVALID_PARAMETER`。如果系统中不存在 `EFI_DRIVER_BINDING_PROTOCOL` 实例，则返回 `EFI_NOT_FOUND`。如果没有足够的可用资源来完成此功能，则返回 `EFI_OUT_OF_RESOURCES`。

如果平台支持用户身份验证，如第 36 节所述，则根据当前用户配置文件中的连接权限检查与 `ControllerHandle` 关联的设备路径。如果禁止，则返回 `EFI_SECURITY_VIOLATION`。然后，在连接任何 `DriverImageHandles` 之前，根据当前用户配置文件中的连接权限检查与句柄关联的设备路径。

如果 `Recursive` 为 `FALSE`，则此函数在所有驱动程序都已连接到 `ControllerHandle` 后返回。如果 `Recursive` 为 `TRUE`，则在 `ControllerHandle` 的所有子控制器上递归调用 `ConnectController()`。可以通过在句柄数据库中搜索所有已打开 `ControllerHandle` 且属性为 `EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER` 的控制器来识别子控制器。

在决定针对控制器测试驱动程序的顺序时，此函数使用五个优先规则。这五个规则从最高优先级到最低优先级如下：

- 上下文覆盖 (Context Override): `DriverImageHandle` 是支持 `EFI_DRIVER_BINDING_PROTOCOL` 的句柄的有序列表。优先级最高的镜像句柄是列表的第一个元素，优先级最低的镜像句柄是列表的最后一个元素。该列表以 `NULL` 镜像句柄终止。
- 平台驱动程序覆盖 (Platform Driver Override): 如果系统中存在 `EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL` 实例，则此协议的 `GetDriver()` 服务用于检索 `ControllerHandle` 的镜像句柄的有序列表。从此列表中，删除了上述规则 (1) 中找到的镜像句柄。从 `GetDriver()` 返回的第一个镜像句柄具有最高优先级，而从 `GetDriver()` 返回的最后一个镜像句柄具有最低优先级。当 `GetDriver()` 返回 `EFI_NOT_FOUND` 时，有序列表终止。`GetDriver()` 不返回镜像句柄是合法的。`EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL` 的系统中最多只能有一个实例。如果不止一个，则系统行为不是确定性的。
- 驱动程序家族覆盖搜索 (Driver Family Override Search): 可用的驱动程序镜像句柄列表可以通过使用引导服务 `EFI_BOOT_SERVICES.LocateHandle()` 以及 `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL` 的 GUID 的 `SearchType` 的 `ByProtocol` 找到。从此列表中，删除了上述规则 (1) 和 (2) 中找到的镜像句柄。其余镜像句柄根据与每个镜像句柄关联的 `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL`

的 `GetVersion()` 函数返回的值从高到低排序。

- 总线特定驱动程序覆盖(*Bus Specific Driver Override*):如果有一个 `EFI_BUS_SPECIFIC_DRIVER_OVERRIDE` 的实例附加到 `ControllerHandle`, 则该协议的 `GetDriver()` 服务用于检索 `ControllerHandle` 的镜像句柄的有序列表。从此列表中, 删除了上述规则 (1)、(2) 和 (3) 中找到的镜像句柄。从 `GetDriver()` 返回的第一个镜像句柄具有最高优先级, 而从 `GetDriver()` 返回的最后一个镜像句柄具有最低优先级。当 `GetDriver()` 返回 `EFI_NOT_FOUND` 时, 有序列表终止。`GetDriver()` 不返回镜像句柄是合法的。
- 驱动程序绑定搜索 (*Driver Binding Search*): 可用的驱动程序镜像句柄列表可以通过使用引导服务 `EFI_BOOT_SERVICES.LocateHandle()` 以及 `EFI_DRIVER_BINDING_PROTOCOL` 的 `GUID` 的 `SearchType` 的 `ByProtocol` 找到。从此列表中, 删除了上述规则 (1)、(2)、(3) 和 (4) 中找到的镜像句柄。其余镜像句柄根据与每个镜像句柄关联的 `EFI_DRIVER_BINDING_PROTOCOL` 实例的 `Version` 字段从高到低排序。

使用 `EFI_DRIVER_BINDING_PROTOCOL` 服务 `Supported()` 按顺序针对 `ControllerHandle` 测试上面列出的五组镜像句柄中的每一组。`RemainingDevicePath` 未经修改地传递到 `Supported()`。`Supported()` 服务返回 `EFI_SUCCESS` 的第一个镜像句柄被标记, 因此在调用 `ConnectController()` 期间不会再次尝试镜像句柄。然后, 为 `ControllerHandle` 调用 `EFI_DRIVER_BINDING_PROTOCOL` 的 `Start()` 服务。再一次, `RemainingDevicePath` 未经修改地传入。每次 `Supported()` 返回 `EFI_SUCCESS` 时, 都会以最高优先级镜像句柄重新开始搜索驱动程序。重复此过程, 直到没有镜像句柄通过 `Supported()` 检查。

如果至少一个镜像句柄从其 `Start()` 服务返回 `EFI_SUCCESS`, 则返回 `EFI_SUCCESS`。

如果没有镜像句柄从它们的 `Start()` 服务返回 `EFI_SUCCESS`, 则返回 `EFI_NOT_FOUND`, 除非 `RemainingDevicePath` 不为 `NULL`, 并且 `RemainingDevicePath` 是一个结束节点。在这种特殊情况下, 返回 `EFI_SUCCESS`, 因为无法启动终端设备路径节点指定的子控制器不是错误。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	One or more drivers were connected to <i>ControllerHandle</i> .
EFI_SUCCESS	No drivers were connected to <i>ControllerHandle</i> , but <i>RemainingDevicePath</i> is not NULL , and it is an End Device Path Node.
EFI_INVALID_PARAMETER	<i>ControllerHandle</i> is NULL .
EFI_NOT_FOUND	There are no EFI_DRIVER_BINDING_PROTOCOL instances present in the system.
EFI_NOT_FOUND	No drivers were connected to <i>ControllerHandle</i> .
EFI_SECURITY_VIOLATION	The user has no permission to start UEFI device drivers on the device path associated with the <i>ControllerHandle</i> or specified by the <i>RemainingDevicePath</i> .

图 49. table7-7_10

- 示例 (Examples)

```

1  //
2  // Connect All Handles Example
3  // The following example recursively connects all controllers in a platform.
4  //
5
6  EFI_STATUS      Status;
7  EFI_BOOT_SERVICES *gBS;
8  UINTN           HandleCount;
9  EFI_HANDLE      *HandleBuffer;
10  UINTN          HandleIndex;
11
12 //
13 // Retrieve the list of all handles from the handle database
14 //
15 Status = gBS->LocateHandleBuffer (
16     AllHandles,
17     NULL,
18     NULL,
19     &HandleCount,
20     &HandleBuffer
21 );
22 if (!EFI_ERROR (Status)) {
23     for (HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++) {

```

```
24 Status = gBS->ConnectController (
25     HandleBuffer[HandleIndex],
26     NULL,
27     NULL,
28     TRUE
29 );
30 }
31 gBS->FreePool(HandleBuffer);
32 }
33
34 //
35 // Connect Device Path Example
36 // The following example walks the device path nodes of a device path, and
37 // connects only the drivers required to force a handle with that device path
38 // to be present in the handle database. This algorithm guarantees that
39 // only the minimum number of devices and drivers are initialized.
40 //
41
42 EFI_STATUS          Status;
43 EFI_DEVICE_PATH_PROTOCOL *DevicePath;
44 EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath;
45 EFI_HANDLE           Handle;
46
47 //
48 // Find the handle that best matches the Device Path. If it is only a
49 // partial match the remaining part of the device path is returned in
50 // RemainingDevicePath.
51 //
52
53 do {
54     RemainingDevicePath = DevicePath;
55     Status = gBS->LocateDevicePath (
56         &gEfiDevicePathProtocolGuid,
57         &RemainingDevicePath,
58         &Handle
59     );
60     if (EFI_ERROR(Status)) {
61         return EFI_NOT_FOUND;
62     }
63
64 //
65 // Connect all drivers that apply to Handle and RemainingDevicePath
66 // If no drivers are connected Handle, then return EFI_NOT_FOUND
```

```
67 // The Recursive flag is FALSE so only one level will be expanded.
68 //
69
70 Status = gBS->ConnectController (
71     Handle,
72     NULL,
73     RemainingDevicePath,
74     FALSE
75 );
76 if (EFI_ERROR(Status)) {
77     return EFI_NOT_FOUND;
78 }
79
80 //
81 // Loop until RemainingDevicePath is an empty device path
82 //
83
84 } while (!IsDevicePathEnd (RemainingDevicePath));
85
86 //
87 // A handle with DevicePath exists in the handle database
88 //
89
90 return EFI_SUCCESS;
```

EFI_BOOT_SERVICES.DisconnectController()

- 概要 (Summary)

从控制器断开一个或多个驱动程序。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_DISCONNECT_CONTROLLER) (
2     IN EFI_HANDLE ControllerHandle,
3     IN EFI_HANDLE DriverImageHandle OPTIONAL,
4     IN EFI_HANDLE ChildHandle OPTIONAL
5 );
```

- 参数 (Parameters)

ControllerHandle: 要断开驱动程序连接的控制器的句柄。

DriverImageHandle: 与 *ControllerHandle* 断开连接的驱动程序。如果 *DriverImageHandle* 为 `NULL`, 则当前管理 *ControllerHandle* 的所有驱动程序都与 *ControllerHandle* 断开连接。

ChildHandle: 要销毁的子句柄。如果 *ChildHandle* 为 `NULL`, 则在驱动程序与 *ControllerHandle* 断开连接之前, *ControllerHandle* 的所有子项都将被销毁。

- 描述 (Description)

此函数断开一个或多个驱动程序与 *ControllerHandle* 指定的控制器的连接。如果 *DriverImageHandle* 为 `NULL`, 则当前管理 *ControllerHandle* 的所有驱动程序都与 *ControllerHandle* 断开连接。如果 *DriverImageHandle* 不为 `NULL`, 则只有 *DriverImageHandle* 指定的驱动程序与 *ControllerHandle* 断开连接。如果 *ChildHandle* 为 `NULL`, 则在驱动程序与 *ControllerHandle* 断开连接之前, *ControllerHandle* 的所有子项都将被销毁。如果 *ChildHandle* 不为 `NULL`, 则仅销毁 *ChildHandle* 指定的子控制器。如果 *ChildHandle* 是 *ControllerHandle* 的唯一子节点, 则 *DriverImageHandle* 指定的驱动程序将与 *ControllerHandle* 断开连接。通过调用 `EFI_DRIVER_BINDING_PROTOCOL` 的 `Stop()` 服务, 驱动程序与控制器断开连接。`EFI_DRIVER_BINDING_PROTOCOL` 在驱动程序镜像句柄上, 控制器的句柄被传递到 `Stop()` 服务中。可以使用引导服务 `EFI_BOOT_SERVICES.OpenProtocolInformation()` 从句柄数据库中检索管理控制器的驱动程序列表以及特定控制器的子级列表。如果所有必需的驱动程序都与 *ControllerHandle* 断开连接, 则返回 `EFI_SUCCESS`。

如果 *ControllerHandle* 为 `NULL`, 则返回 `EFI_INVALID_PARAMETER`。如果没有驱动程序正在管理 *ControllerHandle*, 则返回 `EFI_SUCCESS`。如果 *DriverImageHandle* 不为 `NULL`, 并且 *DriverImageHandle* 不是有效的 `EFI_HANDLE`, 则返回 `EFI_INVALID_PARAMETER`。如果 *DriverImageHandle* 不为 `NULL`, 并且 *DriverImageHandle* 当前未管理 *ControllerHandle*, 则返回 `EFI_SUCCESS`。如果 *ChildHandle* 不为 `NULL`, 并且 *ChildHandle* 不是有效的 `EFI_HANDLE`, 则返回 `EFI_INVALID_PARAMETER`。如果没有足够的资源可用于断开驱动程序与 *ControllerHandle* 的连接, 则返回 `EFI_OUT_OF_RESOURCES`。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	One or more drivers were disconnected from the controller.
EFI_SUCCESS	On entry, no drivers are managing <i>ControllerHandle</i> .
EFI_SUCCESS	DriverImageHandle is not NULL , and on entry DriverImageHandle is not managing ControllerHandle.
EFI_INVALID_PARAMETER	<i>ControllerHandle</i> is NULL .
EFI_INVALID_PARAMETER	<i>DriverImageHandle</i> is not NULL , and it is not a valid EFI_HANDLE .
EFI_INVALID_PARAMETER	<i>ChildHandle</i> is not NULL , and it is not a valid EFI_HANDLE .
EFI_OUT_OF_RESOURCES	There are not enough resources available to disconnect any drivers from <i>ControllerHandle</i> .
EFI_DEVICE_ERROR	The controller could not be disconnected because of a device error.
EFI_INVALID_PARAMETER	<i>DriverImageHandle</i> does not support the EFI_DRIVER_BINDING_PROTOCOL .

图 50. table7-7_11

- 示例 (Examples)

```

1 // 
2 // Disconnect All Handles Example
3 // The following example recursively disconnects all drivers from all
4 // controllers in a platform.
5 //
6
7 EFI_STATUS      Status;
8 EFI_BOOT_SERVICES *gBS;
9 UINTN           HandleCount;
10 EFI_HANDLE     *HandleBuffer;
11 UINTN           HandleIndex;
12
13 //
14 // Retrieve the list of all handles from the handle database
15 //
16
17 Status = gBS->LocateHandleBuffer (
18     AllHandles,
19     NULL,
20     NULL,
```

```

21     &HandleCount,
22     &HandleBuffer
23 );
24 if (!EFI_ERROR (Status)) {
25 for (HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++) {
26 Status = gBS->DisconnectController (
27     HandleBuffer[HandleIndex],
28     NULL,
29     NULL
30 );
31 }
32 gBS->FreePool(HandleBuffer);

```

EFI_BOOT_SERVICES.ProtocolsPerHandle()

- 概要 (Summary)

检索安装在从池分配的缓冲区中的句柄上的协议接口 GUID 的列表。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_PROTOCOLS_PER_HANDLE) (
2     IN EFI_HANDLE    Handle,
3     OUT EFI_GUID    ***ProtocolBuffer,
4     OUT UINTN       *ProtocolBufferCount
5 );

```

- 参数 (Parameters)

Handle: 从中检索协议接口 GUID 列表的句柄。

ProtocolBuffer: 指向安装在 *Handle* 上的协议接口 GUID 指针列表的指针。此缓冲区是通过调用引导服务 [EFI_BOOT_SERVICES.AllocatePool\(\)](#) 来分配的。当调用者不再需要 *ProtocolBuffer* 的内容时，调用者有责任调用引导服务 [EFI_BOOT_SERVICES.FreePool\(\)](#)。

ProtocolBufferCount: 指向 *ProtocolBuffer* 中存在的 GUID 指针数的指针。

- 描述 (Description)

[ProtocolsPerHandle\(\)](#) 函数检索安装在 *Handle* 上的协议接口 GUID 的列表。该列表在 *ProtocolBuffer* 中返回，*ProtocolBuffer* 中 GUID 指针的数量在 *ProtocolBufferCount* 中返回。

如果 *Handle* 为 [NULL](#) 或 *Handle* 为 [NULL](#)，则返回 [EFI_INVALID_PARAMETER](#)。

如果 *ProtocolBuffer* 为 [NULL](#)，则返回 [EFI_INVALID_PARAMETER](#)。

如果 *ProtocolBufferCount* 为 [NULL](#)，则返回 [EFI_INVALID_PARAMETER](#)。

如果没有足够的资源可用于分配 *ProtocolBuffer*, 则返回 `EFI_OUT_OF_RESOURCES`。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The list of protocol interface GUIDs installed on <i>Handle</i> was returned in <i>ProtocolBuffer</i> . The number of protocol interface GUIDs was returned in <i>ProtocolBufferCount</i> .
<code>EFI_INVALID_PARAMETER</code>	<i>Handle</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>ProtocolBuffer</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>ProtocolBufferCount</i> is NULL .
<code>EFI_OUT_OF_RESOURCES</code>	There is not enough pool memory to store the results.

图 51. table7-7_12

- 示例 (Examples)

有关如何使用 `LocateHandleBuffer()`、`EFI_BOOT_SERVICES.ProtocolsPerHandle()`、`EFI_BOOT_SERVICES.OpenProtocol()` 和 `EFI_BOOT_SERVICES.OpenProtocolInformation()` 遍历整个句柄数据库的示例, 请参阅 `EFI_BOOT_SERVICES.LocateHandleBuffer()` 函数描述中的示例。

EFI_BOOT_SERVICES.LocateHandleBuffer()

- 概要 (Summary)

在从池中分配的缓冲区中返回支持所请求协议的句柄数组。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_LOCATE_HANDLE_BUFFER) (
2     IN EFI_LOCATE_SEARCH_TYPE   SearchType,
3     IN EFI_GUID                 *Protocol OPTIONAL,
4     IN VOID *SearchKey         OPTIONAL,
5     OUT UINTN                  *NoHandles,
6     OUT EFI_HANDLE              **Buffer
7 );

```

- 参数 (Parameters)

SearchType: 指定要返回的句柄。

Protocol: 提供用于搜索的协议。此参数仅对 *ByProtocol* 的 *SearchType* 有效。

SearchKey: 根据 *SearchType* 提供搜索关键字

NoHandles: Buffer 中返回的句柄数。

Buffer: 指向缓冲区的指针，用于返回请求的支持协议的句柄数组。此缓冲区是通过调用引导服务 `EFI_BOOT_SERVICES.AllocatePool()` 来分配的。当调用者不再需要 *Buffer* 的内容时，调用者有责任调用引导服务 `EFI_BOOT_SERVICES.FreePool()`。

- 描述 (Description)

`LocateHandleBuffer()` 函数返回一个或多个与 *SearchType* 请求匹配的句柄。*Buffer* 从 *pool* 中分配，*NoHandles* 返回 *Buffer* 中的 *Entries* 数量。每个 *SearchType* 描述如下：

- `AllHandles`: *Protocol* 和 *SearchKey* 被忽略，函数返回系统中每个句柄的数组。
- `ByRegisterNotify`: *SearchKey* 提供由 `EFI_BOOT_SERVICES.RegisterProtocolNotify()` 返回的 *Registration*。该函数返回 *Registration* 的新句柄。一次只返回一个句柄，调用者必须循环直到不再返回句柄。此搜索类型忽略协议。
- `ByProtocol`: 返回所有支持 *Protocol* 的句柄。此搜索类型忽略 *SearchKey*。

如果 *NoHandles* 为 `NULL`，则返回 `EFI_INVALID_PARAMETER`。

如果 *Buffer* 为 `NULL`，则返回 `EFI_INVALID_PARAMETER`。

如果句柄数据库中没有匹配搜索条件的句柄，则返回 `EFI_NOT_FOUND`。

如果没有足够的资源可用于分配 *Buffer*，则返回 `EFI_OUT_OF_RESOURCES`。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The array of handles was returned in <i>Buffer</i> , and the number of handles in <i>Buffer</i> was returned in <i>NoHandles</i> .
<code>EFI_INVALID_PARAMETER</code>	<i>NoHandles</i> is <code>NULL</code>
<code>EFI_INVALID_PARAMETER</code>	<i>Buffer</i> is <code>NULL</code>
<code>EFI_NOT_FOUND</code>	No handles match the search.
<code>EFI_OUT_OF_RESOURCES</code>	There is not enough pool memory to store the matching results.

图 52. table7-7_13

- 示例 (Examples)

```

1 // 
2 // The following example traverses the entire handle database. First all of
3 // the handles in the handle database are retrieved by using
4 // LocateHandleBuffer(). Then it uses ProtocolsPerHandle() to retrieve the

```

```
5 // list of protocol GUIDs attached to each handle. Then it uses OpenProtocol()
6 // to get the protocol instance associated with each protocol GUID on the
7 // handle. Finally, it uses OpenProtocolInformation() to retrieve the list of
8 // agents that have opened the protocol on the handle. The caller of these
9 // functions must make sure that they free the return buffers with FreePool()
10 // when they are done.
11 //
12
13 EFI_STATUS Status;
14 EFI_BOOT_SERVICES *gBS;
15 EFI_HANDLE ImageHandle;
16 UINTN HandleCount;
17 EFI_HANDLE *HandleBuffer;
18 UINTN HandleIndex;
19 EFI_GUID **ProtocolGuidArray;
20 UINTN ArrayCount;
21 UINTN ProtocolIndex;
22 EFI_OPEN_PROTOCOL_INFORMATION_ENTRY *OpenInfo;
23 UINTN OpenInfoCount;
24 UINTN OpenInfoIndex;
25
26 //
27 // Retrieve the list of all handles from the handle database
28 //
29 Status = gBS->LocateHandleBuffer (
30     AllHandles,
31     NULL,
32     NULL,
33     &HandleCount,
34     &HandleBuffer
35 );
36 if (!EFI_ERROR (Status)) {
37     for (HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++) {
38
39         //
40         // Retrieve the list of all the protocols on each handle
41         //
42
43         Status = gBS->ProtocolsPerHandle (
44             HandleBuffer[HandleIndex],
45             &ProtocolGuidArray,
46             &ArrayCount
47         );

```

```
48     if (!EFI_ERROR (Status)) {
49         for (ProtocolIndex = 0; ProtocolIndex < ArrayCount; ProtocolIndex++) {
50             //
51             // Retrieve the protocol instance for each protocol
52             //
53
54             Status = gBS->OpenProtocol (
55                 HandleBuffer[HandleIndex],
56                 ProtocolGuidArray[ProtocolIndex],
57                 &Instance,
58                 ImageHandle,
59                 NULL,
60                 EFI_OPEN_PROTOCOL_GET_PROTOCOL
61             );
62
63             //
64             // Retrieve the list of agents that have opened each protocol
65             //
66
67             Status = gBS->OpenProtocolInformation (
68                 HandleBuffer[HandleIndex],
69                 ProtocolGuidArray[ProtocolIndex],
70                 &OpenInfo,
71                 &OpenInfoCount
72             );
73
74         if (!EFI_ERROR (Status)) {
75             for (OpenInfoIndex=0;OpenInfoIndex<OpenInfoCount;
76                 OpenInfoIndex++) {
77
78                 //
79                 // HandleBuffer[HandleIndex] is the handle
80                 // ProtocolGuidArray[ProtocolIndex] is the protocol GUID
81                 // Instance is the protocol instance for the protocol
82                 // OpenInfo[OpenInfoIndex] is an agent that has opened a
83                 // protocol
84
85             }
86             if (OpenInfo != NULL) {
87                 gBS->FreePool(OpenInfo);
88             }
89
90         }
91     }
92 }
```

```

88         }
89     }
90     if (ProtocolGuidArray != NULL) {
91         gBS->FreePool(ProtocolGuidArray);
92     }
93 }
94 }
95 if (HandleBuffer != NULL) {
96     gBS->FreePool (HandleBuffer);
97 }
98 }

```

EFI_BOOT_SERVICES.LocateProtocol()

- 概要 (Summary)

返回与给定协议匹配的第一个协议实例。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_LOCATE_PROTOCOL) (
2     IN EFI_GUID    *Protocol,
3     IN VOID        *Registration OPTIONAL,
4     OUT VOID       **Interface
5 );

```

- 参数 (Parameters)

Protocol: 提供要搜索的协议。

Registration: 从 [EFI_BOOT_SERVICES.RegisterProtocolNotify\(\)](#) 返回的可选 *Registration* (注册) 密钥。如果 *Registration* 为 **NULL**, 则忽略它。

Interface: 返回时, 指向匹配 *Protocol* 和 *Registration* 的第一个接口的指针。

- 描述 (Description)

[LocateProtocol\(\)](#) 函数找到第一个支持 *Protocol* 的设备句柄, 并从接口中的该句柄返回指向协议接口的指针。如果未找到协议实例, 则将 *Interface* 设置为 **NULL**。

如果 *Interface* 为 **NULL**, 则返回 [EFI_INVALID_PARAMETER](#)。

如果 *Protocol* 为 **NULL**, 则返回 [EFI_INVALID_PARAMETER](#)。

如果 *Registration* 为 **NULL**, 并且句柄数据库中没有支持 *Protocol* 的句柄, 则返回 [EFI_NOT_FOUND](#)。

如果 *Registration* 不为 **NULL**, 并且没有新的 *Registration* 句柄, 则返回 [EFI_NOT_FOUND](#)。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	A protocol instance matching <i>Protocol</i> was found and returned in <i>Interface</i> .
<code>EFI_INVALID_PARAMETER</code>	<i>Interface</i> is NULL . <i>Protocol</i> is NULL .
<code>EFI_NOT_FOUND</code>	No protocol instances were found that match <i>Protocol</i> and <i>Registration</i> .

图 53. table7-7_14

EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces()

• 概要 (Summary)

将一个或多个协议接口安装到引导服务环境中。

• 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES) (
2     IN OUT EFI_HANDLE *Handle,
3     ...
4 );

```

• 参数 (Parameters)

Handle: 指向安装新协议接口的句柄的指针，或者如果要分配新句柄则指向 **NULL** 的指针。包含协议 GUID 和协议接口对的可变参数列表。

• 描述 (Description)

此函数将一组协议接口安装到引导服务环境中。它成对地从可变参数列表中删除参数。第一项始终是指向协议 GUID 的指针，第二项始终是指向协议接口的指针。这些对用于调用引导服务 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 以将协议接口添加到 *Handle*。如果 *Handle* 在入口处为 **NULL**，则将分配一个新句柄。从变量参数列表中按顺序删除参数对，直到找到 **NULL** 协议 GUID 值。如果在安装协议接口时产生任何错误，则在返回错误之前，将使用引导服务 `EFI_BOOT_SERVICES.UninstallProtocolInterface()` 卸载错误之前安装的所有协议。同一个 GUID 不能在同一个句柄上安装多次。

在句柄数据库中有两个具有相同设备路径的句柄是非法的。此服务执行测试以确保不会在两个不同的句柄上无意中安装重复的设备路径。在将任何协议接口安装到 *Handle* 之前，会搜索 `GUID/Pointer` 对参数列表以查看是否正在安装设备路径协议实例。如果要将设备路径协议实例安装到句柄上，则进行检查以查看句柄数据库中是否已经存在具有相同设备路径协议实例的句柄。如果 *handle* 数据库中已

经存在相同的设备路径协议实例，则不会在 *Handle* 上安装任何协议，并返回 `EFI_ALREADY_STARTED`。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	All the protocol interfaces were installed.
<code>EFI_ALREADY_STARTED</code>	A Device Path Protocol instance was passed in that is already present in the handle database.
<code>EFI_OUT_OF_RESOURCES</code>	There was not enough memory in pool to install all the protocols.
<code>EFI_INVALID_PARAMETER</code>	Handle is NULL.
<code>EFI_INVALID_PARAMETER</code>	Protocol is already installed on the handle specified by Handle.

图 54. table7-7_15

`EFI_BOOT_SERVICES.UninstallMultipleProtocolInterfaces()`

- 概要 (Summary)

将一个或多个协议接口移除到引导服务环境中。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES) (
2     IN EFI_HANDLE Handle,
3     ...
4 );

```

- 参数 (Parameters)

Handle: 从中删除协议接口的句柄。包含成对的协议 GUID 和协议接口的可变参数列表。

- 描述 (Description)

此函数从引导服务环境中删除一组协议接口。它成对地从可变参数列表中删除参数。第一项始终是指向协议 GUID 的指针，第二项始终是指向协议接口的指针。这些对于调用引导服务 `EFI_BOOT_SERVICES.UninstallProtocolInterface()` 以从句柄中删除协议接口。从变量参数列表中按顺序删除参数对，直到找到 NULL 协议 GUID 值。如果所有协议都从 *Handle* 中卸载，则返回 `EFI_SUCCESS`。如果在卸载协议接口时生成任何错误，则将使用引导服务 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 重新安装错误之前卸载的协议，并返回状态代码 `EFI_INVALID_PARAMETER`。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	All the protocol interfaces were removed.
EFI_INVALID_PARAMETER	One of the protocol interfaces was not previously installed on <i>Handle</i> .

图 55. table7-7_16

6.4 镜像服务 (Image Services)

可以加载三种类型的镜像：编写的 UEFI 应用程序（参见第 2.1.2 节）、UEFI 引导服务驱动程序（参见第 2.1.4 节）和 EFI 运行时驱动程序（参见第 2.1.4 节）。UEFI 操作系统加载程序（请参阅第 2.1.3 节）是一种 UEFI 应用程序。这些镜像类型之间最显着的区别是固件加载程序将它们加载到的内存类型。表 7-8 总结了镜像之间的差异。

Table 7-8 Image Type Differences Summary

	UEFI Application	UEFI Boot Service Driver	UEFI Runtime Driver
Description	A transient application that is loaded during boot services time. UEFI applications are either unloaded when they complete (see Section 2.1.2), or they take responsibility for the continued operation of the system via ExitBootServices () (see Section 2.1.3) . The UEFI applications are loaded in sequential order by the boot manager, but one UEFI application may dynamically load another.	A program that is loaded into boot services memory and stays resident until boot services terminate .See Section 2.1.4 .	A program that is loaded into runtime services memory and stays resident during runtime. The memory required for a UEFI runtime services driver must be performed in a single memory allocation, and marked as EfiRuntimeServicesData . (Note that the memory only stays resident when booting an EFI-compatible operating system. Legacy operating systems will reuse the memory.) See Section 2.1.4 .
Loaded into memory type	EfiLoaderCode , EfiLoaderData	EfiBootServicesCode , EfiBootServicesData	EfiRuntimeServicesCode, EfiRuntimeServicesData
Default pool allocations from memory type	EfiLoaderData	EfiBootServicesData	EfiRuntimeServicesData

图 56. Table7-8-1

	UEFI Application	UEFI Boot Service Driver	UEFI Runtime Driver
Exit behavior	When an application exits, firmware frees the memory used to hold its image.	When a UEFI boot service driver exits with an error code, firmware frees the memory used to hold its image. When a UEFI boot service driver's entry point completes with EFI_SUCCESS , the image is retained in memory.	When a UEFI runtime driver exits with an error code, firmware frees the memory used to hold its image. When a UEFI runtime services driver's entry point completes with EFI_SUCCESS , the image is retained in memory.
Notes	This type of image would not install any protocol interfaces or handles.	This type of image would typically use InstallProtocolInterface() .	A UEFI runtime driver can only allocate runtime memory during boot services time. Due to the complexity of performing a virtual relocation for a runtime image, this driver type is discouraged unless it is absolutely required.

图 57. Table7-8-1

大多数 UEFI 镜像由引导管理器加载。安装 UEFI 应用程序或 UEFI 驱动程序时，安装过程会向引导管理器注册自身以进行加载。但是，在某些情况下，UEFI 应用程序或 UEFI 驱动程序可能希望以编程方式加载和启动另一个 UEFI 镜像。这可以通过 `EFI_BOOT_SERVICES.LoadImage()` 和 `EFI_BOOT_SERVICES.StartImage()` 接口来完成。UEFI 驱动程序只能在 UEFI 驱动程序的初始化入口点期间加载 UEFI 应用程序。表 7-9 列出了构成镜像服务的函数。

Name	Type	Description
<code>LoadImage</code>	Boot	Loads an EFI image into memory.
<code>StartImage</code>	Boot	Transfers control to a loaded image's entry point.
<code>UnloadImage</code>	Boot	Unloads an image.
<code>EFI_IMAGE_ENTRY_POINT</code>	Boot	Prototype of an EFI Image's entry point.
<code>Exit</code>	Boot	Exits the image's entry point.
<code>ExitBootServices</code>	Boot	Terminates boot services.

图 58. Table7-9

镜像引导服务已被修改以利用现在使用 `EFI_BOOT_SERVICES.OpenProtocol()` 和 `EFI_BOOT_SERVICES.CloseProtocol()` 引导服务跟踪的信息。由于正在使用这些新的引导服务跟踪协议接口的使用情况，因此现在可以在卸载或

退出 UEFI 应用程序或 UEFI 驱动程序时自动关闭协议接口。

EFI_BOOT_SERVICES.LoadImage()

- 概要 (Summary)

将 EFI 镜像加载到内存中。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_IMAGE_LOAD) (
2     IN BOOLEAN             BootPolicy,
3     IN EFI_HANDLE          ParentImageHandle,
4     IN EFI_DEVICE_PATH_PROTOCOL *DevicePath,
5     IN VOID *SourceBuffer   OPTIONAL,
6     IN UINTN               SourceSize,
7     OUT EFI_HANDLE         *ImageHandle
8 );

```

- 参数 (Parameters)

BootPolicy: 如果为 `TRUE`, 则表示请求源自引导管理器, 并且引导管理器正在尝试加载 *DevicePath* 作为引导选择。如果 *SourceBuffer* 不为 `NULL`, 则忽略。

ParentImageHandle: 调用者的镜像句柄。`EFI_HANDLE` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。该字段用于为正在加载的镜像初始化 `EFI_LOADED_IMAGE_PROTOCOL` 的 *ParentHandle* 字段。

DevicePath: 从中加载镜像的 *DeviceHandle* 特定文件路径。`EFI_DEVICE_PATH_PROTOCOL` 在第 10.2 节中定义。

SourceBuffer: 如果不是 `NULL`, 则指向包含要加载的镜像副本的内存位置。

SourceSize: *SourceBuffer* 的字节大小。如果 *SourceBuffer* 为 `NULL`, 则忽略。

ImageHandle: 指向成功加载镜像时创建的返回镜像句柄的指针。`EFI_HANDLE` 类型在 `InstallProtocolInterface()` 函数描述中定义。

- 相关定义 (Related Definitions)

```

1 #define EFI_HII_PACKAGE_LIST_PROTOCOL_GUID \
2     { 0x6a1ee763, 0xd47a, 0x43b4, \
3     { 0xaa, 0xbe, 0xef, 0x1d, 0xe2, 0xab, 0x56, 0xfc } }
4
5 typedef EFI_HII_PACKAGE_LIST_HEADER *EFI_HII_PACKAGE_LIST_PROTOCOL;

```

- 描述 (Description)

`LoadImage()` 函数将 EFI 镜像加载到内存中并返回镜像的句柄。镜像以两种方式之一加载。

- 如果 *SourceBuffer* 不为 `NULL`, 则该函数是内存到内存加载, 其中 *SourceBuffer* 指向要加载的镜像, *SourceSize* 指示镜像的大小 (以字节为单位)。在这种情况下, 调用者已将镜像复制到 *SourceBuffer* 中, 并且可以在加载完成后释放缓冲区。
- 如果 *SourceBuffer* 为 `NULL`, 则该函数是使用 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 的文件复制操作。

如果没有与文件路径关联的 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 实例, 则此函数将尝试使用 `EFI_LOAD_FILE_PROTOCOL` (*BootPolicy* 为 `TRUE`) 或 `EFI_LOAD_FILE2_PROTOCOL`, 然后是 `EFI_LOAD_FILE_PROTOCOL` (*BootPolicy* 为 `FALSE`)。

在所有情况下, 此函数将使用与将使用的 *DevicePath* 最匹配的句柄关联的这些协议的实例。有关如何定位最近的句柄的更多信息, 请参阅引导服务描述。

- 在 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 的情况下, 使用 *DevicePath* 的文件路径媒体设备路径节点的路径名。
- 在 `EFI_LOAD_FILE_PROTOCOL` 的情况下, 将 *DevicePath* 的其余设备路径节点和 *BootPolicy* 标志传递给 `EFI_LOAD_FILE_PROTOCOL.LoadFile()` 函数。当 *DevicePath* 仅指定设备 (并且没有其他设备节点) 时, 将加载负责引导的默认镜像。有关详细信息, 请参阅第 13.1 节中对 `EFI_LOAD_FILE_PROTOCOL` 的讨论。
- 在 `EFI_LOAD_FILE2_PROTOCOL` 的情况下, 行为与上面相同, 只是它仅在 *BootOption* 为 `FALSE` 时使用。有关详细信息, 请参阅 `EFI_LOAD_FILE2_PROTOCOL` 的讨论。
- 如果平台支持驱动程序签名, 如第 32.4.2 节所述, 并且镜像签名无效, 则有关镜像的信息将记录在 `EFI_IMAGE_EXECUTION_INFO_TABLE` (参见第 32.4.2 节) 中, 并返回 `EFI_SECURITY_VIOLATION`。
- 如果平台支持用户身份验证, 如第 36 节所述, 并且在当前用户配置文件中禁止在指定的文件路径上加载镜像, 则记录有关镜像的信息 (参见第 36.1.5 节中的延迟执行) 并且 `EFI_SECURITY_VIOLATION` 是回来。

加载镜像后, 固件会创建并返回一个 `EFI_HANDLE`, 用于标识镜像并支持 `EFI_LOADED_IMAGE_PROTOCOL` 和 `EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL`。调用者可以填写镜像的“加载选项”数据, 或者在通过调用 `EFI_BOOT_SERVICES.StartImage()` 将控制权传递给新加载的镜像之前向句柄添加额外的协议支持。此外, 一旦镜像被加载, 调用者要么通过调用 `StartImage()` 启动它, 要么通过调用 `EFI_BOOT_SERVICES.UnloadImage()` 卸载它。

加载镜像后, 如果镜像包含类型为“HII”的自定义 `PE/COFF` 资源, 则 `LoadImage()` 会在句柄上安装 `EFI_HII_PACKAGE_LIST_PROTOCOL`。协议的接口指针指向包含在资源数据中的 `HII` 包列表。其格式在第 33.3.1 节中。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	Image was loaded into memory correctly.
<code>EFI_NOT_FOUND</code>	Both <code>SourceBuffer</code> and <code>DevicePath</code> are NULL .
<code>EFI_INVALID_PARAMETER</code>	One of the parameters has an invalid value.
<code>EFI_INVALID_PARAMETER</code>	<code>ImageHandle</code> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<code>ParentImageHandle</code> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<code>ParentImageHandle</code> is NULL .
<code>EFI_UNSUPPORTED</code>	The image type is not supported.
<code>EFI_OUT_OF_RESOURCES</code>	Image was not loaded due to insufficient resources.
<code>EFI_LOAD_ERROR</code>	Image was not loaded because the image format was corrupt or not understood.
<code>EFI_DEVICE_ERROR</code>	Image was not loaded because the device returned a read error.
<code>EFI_ACCESS_DENIED</code>	Image was not loaded because the platform policy prohibits the image from being loaded. NULL is returned in <code>*ImageHandle</code> .
<code>EFI_SECURITY_VIOLATION</code>	Image was loaded and an <code>ImageHandle</code> was created with a valid <code>EFI_LOADED_IMAGE_PROTOCOL</code> . However, the current platform policy specifies that the image should not be started.

图 59. table7-9_1.jpg

`EFI_BOOT_SERVICES.StartImage()`

- 概要 (Summary)

将控制转移到加载镜像的入口点。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_IMAGE_START) (
2     IN EFI_HANDLE    ImageHandle,
3     OUT UINTN       *ExitDataSize,
4     OUT CHAR16      **ExitData OPTIONAL
5 );

```

- 参数 (Parameters)

`ImageHandle`: 要启动的镜像句柄。`EFI_HANDLE` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

`ExitDataSize`: 指向 `ExitData` 大小 (以字节为单位) 的指针。如果 `ExitData` 为 `NULL`, 则忽略此参数并

且不修改 *ExitDataSize* 的内容。

ExitData: 指向数据缓冲区的指针，该缓冲区包含以 `Null` 结尾的字符串，后面可以选择附加二进制数据。该字符串是调用者可以用来进一步指示镜像退出原因的描述。

- 描述 (Description)

`StartImage()` 函数将控制转移到由 `EFI_BOOT_SERVICES.LoadImage()` 加载的镜像的入口点。镜像只能启动一次。

当加载镜像的 `EFI_IMAGE_ENTRY_POINT` 返回或加载镜像调用 `EFI_BOOT_SERVICES.Exit()` 时，控制从 `StartImage()` 返回。进行该调用时，`Exit()` 中的 *ExitData* 缓冲区和 *ExitDataSize* 通过此函数中的 *ExitData* 缓冲区和 *ExitDataSize* 传回。当不再需要缓冲区时，此函数的调用者负责通过调用 `EFI_BOOT_SERVICES.FreePool()` 将 *ExitData* 缓冲区返回到池中。使用 `Exit()` 类似于从镜像的 `EFI_IMAGE_ENTRY_POINT` 返回，除了 `Exit()` 还可能返回额外的 *ExitData*。`Exit()` 函数描述定义了固件在加载镜像返回控制后执行的清理过程。

- EFI 1.10 扩展 (EFI 1.10 Extension)

为了保持与写入 EFI 1.02 规范的 UEFI 驱动程序的兼容性，`StartImage()` 必须在启动每个镜像之前和之后监视句柄数据库。如果在启动镜像时创建或修改了任何句柄，则必须在 `StartImage()` 返回之前为每个新创建或修改的句柄调用 `EFI_BOOT_SERVICES.ConnectController()` 并将 *Recursive* 参数设置为 `TRUE`。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	Image was loaded into memory correctly.
<code>EFI_NOT_FOUND</code>	Both <code>SourceBuffer</code> and <code>DevicePath</code> are NULL .
<code>EFI_INVALID_PARAMETER</code>	One of the parameters has an invalid value.
<code>EFI_INVALID_PARAMETER</code>	<code>ImageHandle</code> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<code>ParentImageHandle</code> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<code>ParentImageHandle</code> is NULL .
<code>EFI_UNSUPPORTED</code>	The image type is not supported.
<code>EFI_OUT_OF_RESOURCES</code>	Image was not loaded due to insufficient resources.
<code>EFI_LOAD_ERROR</code>	Image was not loaded because the image format was corrupt or not understood.
<code>EFI_DEVICE_ERROR</code>	Image was not loaded because the device returned a read error.
<code>EFI_ACCESS_DENIED</code>	Image was not loaded because the platform policy prohibits the image from being loaded. NULL is returned in <code>*ImageHandle</code> .
<code>EFI_SECURITY_VIOLATION</code>	Image was loaded and an <code>ImageHandle</code> was created with a valid <code>EFI_LOADED_IMAGE_PROTOCOL</code> . However, the current platform policy specifies that the image should not be started.

图 60. table7-9_1

`EFI_BOOT_SERVICES.UnloadImage()`

- 概要 (Summary)

卸载镜像。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_IMAGE_UNLOAD) (
2     IN EFI_HANDLE  ImageHandle
3 );
```

- 参数 (Parameters)

`ImageHandle`: 标识待卸载镜像的句柄。

- 描述 (Description)

`UnloadImage()` 函数卸载先前加载的镜像。

存在三种可能的情况。如果镜像尚未启动，函数将卸载镜像并返回 `EFI_SUCCESS`。

如果镜像已启动并且具有 `Unload()` 入口点，则控制权将传递到该入口点。

如果镜像的卸载函数返回 `EFI_SUCCESS`，镜像被卸载；否则，镜像卸载函数返回的错误返回给调用者。镜像卸载函数负责释放所有分配的内存，并确保在返回 `EFI_SUCCESS` 之前没有对任何已释放内存或镜像本身的引用。

如果镜像已启动且没有 `Unload()` 入口点，则该函数返回 `EFI_UNSUPPORTED`。

- **EFI 1.10 扩展 (EFI 1.10 Extension)**

ImageHandle 使用启动服务 `EFI_BOOT_SERVICES.OpenProtocol()` 打开的所有协议都将自动关闭启动服务 `EFI_BOOT_SERVICES.CloseProtocol()`。如果关闭所有打开的协议，则返回 `EFI_SUCCESS`。如果对 `CloseProtocol()` 的任何调用失败，则返回 `CloseProtocol()` 的错误代码。

- **返回的状态码 (Status Codes Returned)**

<code>EFI_SUCCESS</code>	The image has been unloaded.
<code>EFI_UNSUPPORTED</code>	The image has been started, and does not support unload.
<code>EFI_INVALID_PARAMETER</code>	<i>ImageHandle</i> is not a valid image handle.
Exit code from Unload handler	Exit code from the image's unload function.

图 61. table7-9_3

EFI_IMAGE_ENTRY_POINT

- **概要 (Summary)**

这是 EFI 镜像入口点的声明。这可以是写入此规范的应用程序、EFI 引导服务驱动程序或 EFI 运行时驱动程序的入口点。

- **原型 (Prototype)**

```

1 typedef EFI_STATUS (EFIAPI *EFI_IMAGE_ENTRY_POINT) (
2     IN EFI_HANDLE      ImageHandle,
3     IN EFI_SYSTEM_TABLE *SystemTable
4 );

```

- **参数 (Parameters)**

ImageHandle: 标识加载镜像的句柄。`EFI_HANDLE` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

SystemTable: 此镜像的系统表。`EFI_SYSTEM_TABLE` 类型在第 4 节中定义。

- **描述 (Description)**

图像的入口点是 `EFI_IMAGE_ENTRY_POINT` 类型。固件将图像加载到内存后，控制权将传递到图像的入口点。入口点负责初始化图像。图像的 `ImageHandle` 被传递给图像。`ImageHandle` 为图像提供它需要的所有绑定和数据信息。此信息可通过协议接口获得。但是，要访问 `ImageHandle` 上的协议接口，需要访问引导服务功能。因此，`EFI_BOOT_SERVICES.LoadImage()` 将一个从 `LoadImage()` 当前范围继承的 `SystemTable` 传递给 `EFI_IMAGE_ENTRY_POINT`。

所有图像句柄都支持 `EFI_LOADED_IMAGE_PROTOCOL` 和 `EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL`。这些协议可用于获取有关已加载图像状态的信息，例如，从中加载图像的设备和图像的加载选项。此外，`ImageHandle` 可能支持父图像提供的其他协议。

如果图像支持动态卸载，它必须在从其入口点返回控制之前在 `EFI_LOADED_IMAGE_PROTOCOL` 结构中提供一个卸载函数。

通常，镜像通过调用 `EFI_BOOT_SERVICES.Exit()` 或通过从其入口点返回控制来从其初始化入口点返回控制。如果图像从其入口点返回控制，则固件将控制传递给 `Exit()`，使用返回代码作为 `Exit()` 的 `ExitStatus` 参数。

请参阅下面的 `Exit()` 以了解入口点退出条件。

`EFI_BOOT_SERVICES.Exit()`

- 概要 (Summary)

终止加载的 EFI 镜像并将控制权返回给引导服务。

- 原型 (Prototype)

```
1 typedef
2 EFI_STATUS (EFIAPI *EFI_EXIT) (
3     IN EFI_HANDLE  ImageHandle,
4     IN EFI_STATUS   ExitStatus,
5     IN UINTN       ExitDataSize,
6     IN CHAR16      *ExitData OPTIONAL
7 );
```

- 参数 (Parameters)

`ImageHandle`: 标识图像的句柄。此参数在输入时传递给图像。

`ExitStatus`: 图像的退出代码。

`ExitDataSize`: `ExitData` 的大小（以字节为单位）。如果 `ExitStatus` 为 `EFI_SUCCESS`，则忽略。

`ExitData`: 指向包含 `Null` 终止字符串的数据缓冲区的指针，后面可以选择附加二进制数据。该字符串是调用者可以用来进一步指示图像退出原因的描述。只有当 `ExitStatus` 不是 `EFI_SUCCESS` 时，`ExitData` 才有效。必须通过调用 `EFI_BOOT_SERVICES.AllocatePool()` 来分配 `ExitData` 缓冲区。

- 描述 (Description)

`Exit()` 函数终止 *ImageHandle* 引用的图像并将控制权返回给启动服务。如果图像已经从其入口点 (`EFI_IMAGE_ENTRY_POINT`) 返回, 或者如果它加载了任何尚未退出的子图像 (所有子图像必须在此图像退出之前退出), 则可能不会调用此函数。

使用 `Exit()` 类似于从图像的 `EFI_IMAGE_ENTRY_POINT` 返回, 除了 `Exit()` 还可能返回额外的 `ExitData`。

当应用程序退出兼容系统时, 固件会释放用于保存图像的内存。固件还释放它对 *ImageHandle* 和句柄本身的引用。在退出之前, 应用程序负责释放它分配的任何资源。这包括内存 (页面和/或池)、打开的文件系统句柄等。此规则的唯一例外是 `ExitData` 缓冲区, 它必须由 `EFI_BOOT_SERVICES.StartImage()` 的调用者释放。(如果需要缓冲区, 固件必须通过调用 `EFI_BOOT_SERVICES.AllocatePool()` 分配它, 并且必须将指向它的指针返回给 `StartImage()` 的调用者。)

当 EFI 引导服务驱动程序或运行时服务驱动程序退出时, 固件仅在 `ExitStatus` 为错误代码时释放映像; 否则图像将驻留在内存中。如果驱动程序已将任何协议处理程序或其他活动回调安装到尚未 (或无法) 清理的系统中, 则驱动程序不得返回错误代码。如果驱动程序以错误代码退出, 则它负责在退出前释放所有资源。这包括任何分配的内存 (页面和/或池)、打开的文件系统句柄等。

在调用 `EFI_BOOT_SERVICES.StartImage()` 之前, 为 `EFI_BOOT_SERVICES.LoadImage()` 加载的图像调用 `Exit()` 或 `UnloadImage()` 是有效的。这将在不启动图像的情况下从内存中释放图像。

- **EFI 1.10 扩展 (EFI 1.10 Extension)**

如果 *ImageHandle* 是 UEFI 应用程序, 则 *ImageHandle* 使用引导服务 `EFI_BOOT_SERVICES.OpenProtocol()` 打开的所有协议将自动使用引导服务 `EFI_BOOT_SERVICES.CloseProtocol()` 关闭。如果 *ImageHandle* 是 UEFI 引导服务驱动程序或 UEFI 运行时服务驱动程序, 并且 `ExitStatus` 是错误代码, 则 *ImageHandle* 使用引导服务 `OpenProtocol()` 打开的所有协议将自动使用引导服务 `CloseProtocol()` 关闭。如果 *ImageHandle* 是 UEFI 引导服务驱动程序或 UEFI 运行时服务驱动程序, 并且 `ExitStatus` 不是错误代码, 则此服务不会自动关闭任何协议。

- **返回的状态码 (Status Codes Returned)**

EFI_SUCCESS	The array of handles was returned.
EFI_NOT_FOUND	No handles match the search.
EFI_BUFFER_TOO_SMALL	The <i>BufferSize</i> is too small for the result. <i>BufferSize</i> has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	<i>SearchType</i> is not a member of EFI_LOCATE_SEARCH_TYPE .
EFI_INVALID_PARAMETER	<i>SearchType</i> is ByRegisterNotify and <i>SearchKey</i> is NULL .
EFI_INVALID_PARAMETER	<i>SearchType</i> is ByProtocol and <i>Protocol</i> is NULL .
EFI_INVALID_PARAMETER	One or more matches are found and <i>BufferSize</i> is NULL .
EFI_INVALID_PARAMETER	<i>BufferSize</i> is large enough for the result and <i>Buffer</i> is NULL .

图 62. table7-7_4

EFI_BOOT_SERVICES.ExitBootServices()

• 概要 (Summary)

终止所有引导服务。

• 原型 (Prototype)

```

1  typedef EFI_STATUS (EFIAPI *EFI_EXIT_BOOT_SERVICES) (
2      IN EFI_HANDLE ImageHandle,
3      IN UINTN MapKey
4  );

```

• 参数 (Parameters)

ImageHandle: 标识加载镜像的句柄。**EFI_HANDLE** 类型在 **EFI_BOOT_SERVICES.InstallProtocolInterface()** 函数描述中定义。

MapKey: 最新内存映射的 *Key*。

• 描述 (Description)

ExitBootServices() 函数由当前正在执行的 UEFI OS 加载程序映像调用以终止所有引导服务。成功后, UEFI OSloader 将负责系统的持续运行。来自 **EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES** 和 **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** 事件通知组的所有事件以及 **EVT_SIGNAL_EXIT_BOOT_SERVICES** 类型的事件必须在 **ExitBootServices()** 返回 **EFI_SUCCESS** 之前发出信号。即使多次调用 **ExitBootServices()**, 事件也只会发出一次信号。

UEFI 操作系统加载程序必须确保它在调用 `ExitBootServices()` 时具有系统的当前内存映射。这是通过传入 `EFI_BOOT_SERVICES.GetMemoryMap()` 返回的当前内存映射的 `MapKey` 值来完成的。必须注意确保内存映射在这两个调用之间不会发生变化。建议在调用 `ExitBootServices()` 之前立即调用 `GetMemoryMap()`。如果 `MapKey` 值不正确，则 `ExitBootServices()` 返回 `EFI_INVALID_PARAMETER` 并且必须再次调用带有 `ExitBootServices()` 的 `GetMemoryMap()`。固件实现可能会选择在第一次调用 `ExitBootServices()` 期间部分关闭引导服务。在第一次调用 `ExitBootServices()` 之后，UEFI 操作系统加载程序不应调用内存分配服务以外的任何引导服务函数。

成功后，UEFI 操作系统加载程序拥有系统中的所有可用内存。此外，UEFI 操作系统加载程序可以将映射中标记为 `EfiBootServicesCode` 和 `EfiBootServicesData` 的所有内存视为可用空闲内存。不能进一步调用引导服务函数或基于 EFI 设备句柄的协议，并且引导服务看门狗定时器被禁用。成功时，EFI 系统表的几个字段应设置为 `NULL`。这些包括 `ConsoleInHandle`、`ConIn`、`ConsoleOutHandle`、`ConOut`、`StandardErrorHandler`、`StdErr` 和 `BootServicesTable`。此外，由于正在修改 EFI 系统表的字段，因此必须重新计算 EFI 系统表的 32 位 CRC。

固件必须保证以下处理顺序：

- `EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES` 处理程序被调用；
- 定时器服务被停用（定时器事件活动停止）；
- 调用 `EVT_SIGNAL_EXIT_BOOT_SERVICES` 和 `EFI_EVENT_GROUP_EXIT_BOOT_SERVICES` 处理程序。

注意：`EVT_SIGNAL_EXIT_BOOT_SERVICES` 事件类型和 `EFI_EVENT_GROUP_EXIT_BOOT_SERVICES` 事件组在功能上是等效的。固件在订购处理程序时不区分两者。

请参阅上面 `EFI_BOOT_SERVICES.CreateEventEx()` 部分中的 `EFI_EVENT_GROUP_EXIT_BOOT_SERVICES` 描述，了解对 `EXIT_BOOT_SERVICES` 处理程序的其他限制。

- 返回的状态码（Status Codes Returned）

<code>EFI_SUCCESS</code>	The interface information for the specified protocol was returned.
<code>EFI_UNSUPPORTED</code>	The device does not support the specified protocol.
<code>EFI_INVALID_PARAMETER</code>	<i>Handle</i> is NULL ..
<code>EFI_INVALID_PARAMETER</code>	<i>Protocol</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>Interface</i> is NULL .

图 63. table7-7_5

6.5 杂项启动服务 (Miscellaneous Boot Services)

本节包含其他地方未定义但完成 EFI 环境定义所需的引导服务的其余函数定义。表 7-10 列出了其他引导服务功能。

Name	Type	Description
SetWatchDogTimer	Boot	Resets and sets a watchdog timer used during boot services time.
Stall	Boot	Stalls the processor.
CopyMem	Boot	Copies the contents of one buffer to another buffer.
SetMem	Boot	Fills a buffer with a specified value.
GetNextMonotonicCount	Boot	Returns a monotonically increasing count for the platform.
InstallConfigurationTable	Boot	Adds, updates, or removes a configuration table from the EFI System Table.
CalculateCrc32	Boot	Computes and returns a 32-bit CRC for a data buffer.

图 64. table7-10

添加 `EFI_BOOT_SERVICES.CalculateCrc32()` 服务是因为 EFI 中有多个地方使用了 32 位 CRC。其中包括 EFI 系统表、EFI 引导服务表、EFI 运行时服务表和 GUID 分区表 (GPT) 结构。`CalculateCrc32()` 服务允许计算新的 32 位 CRC，并验证现有的 32 位 CRC。

EFI_BOOT_SERVICES.SetWatchdogTimer()

- 概要 (Summary)

设置系统的看门狗定时器。

- 原型 (Prototype)

```

1 typedef EFI_STATUS (EFIAPI *EFI_SET_WATCHDOG_TIMER) (((
2     IN UINTN    Timeout,
3     IN UINT64   WatchdogCode,
4     IN UINTN    DataSize,
5     IN CHAR16   *WatchdogData OPTIONAL
6 );

```

- 参数 (Parameters)

Timeout: 设置看门狗定时器的秒数。零值禁用计时器。

WatchdogCode: 用于记录看门狗定时器超时事件的数字代码。固件保留代码 `0x0000` 到 `0xFFFF`。加载程序和操作系统可能会使用其他超时代码。

DataSize: *WatchdogData* 的大小（以字节为单位）。

WatchdogData: 包含以 `Null` 结尾的字符串的数据缓冲区，后面可以选择附加二进制数据。该字符串是调用可用于进一步指示使用看门狗事件记录的原因的描述。

- 描述 (Description)

`SetWatchdogTimer()` 函数设置系统的看门狗定时器。

如果看门狗定时器到期，固件会记录该事件。然后，系统可以使用运行时服务 `ResetSystem()` 进行重置，或者执行最终必须导致平台重置的平台特定操作。看门狗定时器在固件的引导管理器调用 EFI 引导选项之前准备就绪。看门狗必须设置为 5 分钟的周期。EFI 镜像可以根据需要重置或禁用看门狗定时器。如果控制权返回到固件的引导管理器，则必须禁用看门狗定时器。

看门狗定时器仅在引导服务期间使用。在 `EFI_BOOT_SERVICES.ExitBootServices()` 成功完成后，看门狗定时器被禁用。

看门狗定时器的精度为请求超时 $+/- 1$ 秒。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The timeout has been set.
<code>EFI_INVALID_PARAMETER</code>	The supplied <i>WatchdogCode</i> is invalid.
<code>EFI_UNSUPPORTED</code>	The system does not have a watchdog timer.
<code>EFI_DEVICE_ERROR</code>	The watch dog timer could not be programmed due to a hardware error.

图 65. table7-10_1

EFI_BOOT_SERVICES.Stall()

- 概要 (Summary)

导致细粒度的停顿。(TODO:Induces a fine-grained stall.)

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_STALL) (
2     IN UINTN Microseconds
3 );
```

- 参数 (Parameters)

Microseconds: 停止执行的微秒数。

- 描述 (Description)

`Stall()` 函数至少在请求的微秒数内停止处理器的执行。在停顿期间不会让出处理器的执行。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	Execution was stalled at least the requested number of <i>Microseconds</i> .
-------------	--

图 66. table7-10_2

EFI_BOOT_SERVICES.CopyMem()

- 概要 (Summary)

`CopyMem()` 函数将一个缓冲区的内容复制到另一个缓冲区。

- 原型 (Prototype)

```

1 typedef VOID (EFIAPI *EFI_COPY_MEM) (
2     IN VOID    *Destination,
3     IN VOID    *Source,
4     IN UINTN   Length
5 );

```

- 参数 (Parameters)

Destination: 指向内存复制的目标缓冲区的指针。

Source: 指向内存复制的源缓冲区的指针。

Length: 从源缓冲区复制到目标缓冲区的字节数。

- 描述 (Description)

`CopyMem()` 函数将 *Length* 字节从缓冲区 *Source* 复制到缓冲区 *Destination*。

`CopyMem()` 的实现必须是可重入的，并且它必须处理重叠的源缓冲区和目标缓冲区。这意味着 `CopyMem()` 的实现必须根据源缓冲区和目标缓冲区之间存在的重叠类型选择正确的复制操作方向。如果源缓冲区或目标缓冲区越过处理器地址空间的顶部，则复制操作的结果是不可预测的。

退出此服务时目标缓冲区的内容必须与进入此服务时源缓冲区的内容相匹配。由于潜在的重叠，此服务可能会修改源缓冲区的内容。以下规则可用于保证正确的行为：

- 如果 *Destination* 和 *Source* 相同，则不应执行任何操作；
- 如果 *Destination* > *Source* 且 *Destination* < (*Source* + *Length*)，则数据应从 *Source* 缓冲区复制到 *Destination* 缓冲区，从缓冲区的末尾开始，一直到缓冲区的开头；
- 否则，数据应该从源缓冲区复制到目标缓冲区，从缓冲区的开头开始，一直到缓冲区的结尾。

- 返回的状态码 (Status Codes Returned)

无。

EFI_BOOT_SERVICES.SetMem()

- 概要 (Summary)

`SetMem()` 函数用指定值填充缓冲区。

- 原型 (Prototype)

```
1 typedef VOID (EFIAPI *EFI_SET_MEM) (
2     IN VOID    *Buffer,
3     IN UINTN   Size,
4     IN UINT8   Value
5 );
```

- 参数 (Parameters)

Buffer: 指向要填充的缓冲区的指针。

Size: 缓冲区中要填充的字节数。

Value: 填充缓冲区的值。

- 描述 (Description)

此函数用 *Value* 填充 *Buffer* 的 *Size* 字节。`SetMem()` 的实现必须是可重入的。如果 *Buffer* 越过处理器地址空间的顶部，则 `SetMem()` 操作的结果是不可预测的。

- 返回的状态码 (Status Codes Returned)

无。

EFI_BOOT_SERVICES.GetNextMonotonicCount()

- 概要 (Summary)

返回平台的单调递增计数。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_GET_NEXT_MONOTONIC_COUNT) (
2     OUT UINT64  *Count
3 );
```

- 参数 (Parameters)

Count: 指向返回值的指针。

- 描述 (Description)

`GetNextMonotonicCount()` 函数返回一个 64 位值，该值的数值大于上次调用该函数时的数值。

该平台的单调计数器由两部分组成：高 32 位和低 32 位。低 32 位值是易失性的，在每次系统复位时复位为零。每次调用 `GetNextMonotonicCount()` 时它都会增加 1。高 32 位值是非易失性的，每当系统复位或低 32 位计数器溢出时都会加 1。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The next monotonic count was returned.
<code>EFI_DEVICE_ERROR</code>	The device is not functioning properly.
<code>EFI_INVALID_PARAMETER</code>	<i>Count</i> is NULL .

图 67. table7-10_5

`EFI_BOOT_SERVICES.InstallConfigurationTable()`

- 概要 (Summary)

从 EFI 系统表中添加、更新或删除配置表 Entry(TODO: 项)。

- 原型 (Prototype)

```
1 typedef EFI_STATUS (EFIAPI *EFI_INSTALL_CONFIGURATION_TABLE) (
2     IN EFI_GUID  *Guid,
3     IN VOID      *Table
4 );
```

- 参数 (Parameters)

Guid: 指向要添加、更新或删除的条目的 GUID 的指针。

Table: 指向要添加、更新或删除的条目的配置表的指针。可以为 `NULL`。

- 描述 (Description)

`InstallConfigurationTable()` 函数用于维护存储在 EFI 系统表中的配置表列表。该列表存储为 (GUID, Pointer) 对数组。该列表必须从 *PoolType* 设置为 `EfiRuntimeServicesData` 的池内存中分配。

如果 *Guid* 为 `NULL`，则返回 `EFI_INVALID_PARAMETER`。如果 *Guid* 有效，则有四种可能：

- 如果 *Guid* 不存在于系统表中，并且 *Table* 不为 `NULL`，则将 (*Guid*, *Table*) 对添加到系统表中。请参阅下面的注释；
- 如果系统表中不存在 *Guid*，并且 *Table* 为 `NULL`，则返回 `EFI_NOT_FOUND`；
- 如果 *Guid* 存在于系统表中，并且 *Table* 不为 `NULL`，则 (*Guid*, *Table*) 对将更新为新的 *Table* 值；

- 如果 *Guid* 存在于系统表中，并且 *Table* 为 `NULL`，则与 *Guid* 关联的条目将从系统表中删除。

如果添加、修改或删除操作完成，则返回 `EFI_SUCCESS`。

注：如果没有足够的内存来执行添加操作，则返回 `EFI_OUT_OF_RESOURCES`。

- 返回的状态码（Status Codes Returned）

<code>EFI_SUCCESS</code>	The (<i>Guid</i> , <i>Table</i>) pair was added, updated, or removed.
<code>EFI_INVALID_PARAMETER</code>	<i>Guid</i> is NULL .
<code>EFI_NOT_FOUND</code>	An attempt was made to delete a nonexistent entry.
<code>EFI_OUT_OF_RESOURCES</code>	There is not enough memory available to complete the operation.

图 68. table7-10_6

EFI_BOOT_SERVICES.CalculateCrc32()

- 概要（Summary）

计算并返回数据缓冲区的 32 位 CRC。

- 原型（Prototype）

```

1 typedef EFI_STATUS (EFIAPI *EFI_CALCULATE_CRC32)
2     IN VOID      *Data,
3     IN UINTN     DataSize,
4     OUT UINT32   *Crc32
5 );

```

- 参数（Parameters）

Data: 指向要在其上计算 32 位 CRC 的缓冲区的指针。

DataSize: 缓冲区数据中的字节数。

Crc32: 为 *Data* 和 *DataSize* 指定的数据缓冲区计算的 32 位 CRC。

- 描述（Description）

此函数计算由 *Data* 和 *DataSize* 指定的数据缓冲区的 32 位 CRC。如果计算出 32 位 CRC，则在 *Crc32* 中返回，并返回 `EFI_SUCCESS`。

如果 *Data* 为 `NULL`，则返回 `EFI_INVALID_PARAMETER`。

如果 *Crc32* 为 `NULL`，则返回 `EFI_INVALID_PARAMETER`。

如果 *DataSize* 为 0，则返回 `EFI_INVALID_PARAMETER`。

- 返回的状态码（Status Codes Returned）

<code>EFI_SUCCESS</code>	The 32-bit CRC was computed for the data buffer and returned in <i>Crc32</i> .
<code>EFI_INVALID_PARAMETER</code>	<i>Data</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>Crc32</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>DataSize</i> is 0.

图 69. table7-10_7

7 运行时服务

本节讨论兼容系统中存在的基本服务。这些服务由在 EFI 环境中运行的代码可以使用的接口函数定义。此类代码可能包括管理设备访问或扩展平台功能的协议，以及在预引导环境和 EFI OS 加载程序中运行的应用程序。

此处描述了两种类型的服务：

- 引导服务。在成功调用 `EFI_BOOT_SERVICES.ExitBootServices()` 之前可用的函数。这些功能在第 7 节中描述。
- 运行时服务。在任何调用 `ExitBootServices()` 之前和之后可用的函数。这些功能在本节中描述。

在引导期间，系统资源由固件拥有，并通过引导服务接口函数进行控制。这些功能可以描述为“全局”或“基于句柄”。术语“全局”仅表示功能访问系统服务并且在所有平台上都可用（因为所有平台都支持所有系统服务）。术语“基于句柄”意味着该函数访问特定设备或设备功能，并且在某些平台上可能不可用（因为某些设备在某些平台上不可用）。协议是动态创建的。本节讨论“全局”功能和运行时功能；随后的部分讨论“基于句柄”。

UEFI 应用程序（包括 UEFI 操作系统加载程序）必须使用引导服务功能来访问设备和分配内存。进入时，图像会提供一个指向系统表的指针，该表包含引导服务分派表和用于访问控制台的默认句柄。所有引导服务功能都可用，直到 UEFI 操作系统加载程序加载了足够多的自身环境以控制系统的持续运行，然后通过调用 `ExitBootServices()` 终止引导服务。

原则上，`ExitBootServices()` 调用旨在供操作系统使用，以指示其加载程序已准备好接管平台和所有平台资源管理的控制权。因此，到目前为止，引导服务可用以协助 UEFI OS 加载程序准备引导操作系统。一旦 UEFI 操作系统加载程序控制了系统并完成了操作系统启动过程，就只能调用运行时服务。但是，UEFI 操作系统加载程序以外的代码可能会也可能不会选择调用 `ExitBootServices()`。这种选择可能部分取决于此类代码是否旨在继续使用 EFI 引导服务或引导服务环境。

本节的其余部分将讨论各个函数。运行时服务分为以下几类：

- 运行时规则和限制（第 8.1 节）
- 可变服务（第 8.1.1 节）
- 时间服务（第 8.3 节）
- 虚拟内存服务（第 8.4 节）
- 杂项服务（第 8.5 节）

7.1 运行时服务规则和限制（Runtime Services Rules and Restrictions）

如果需要，可以在启用中断的情况下调用所有运行时服务。当需要保护对硬件资源的访问时，运行时服务功能将在内部禁用中断。

对关键硬件资源的访问完成后，中断使能控制位将返回其入口状态。

运行时服务的所有调用者都被限制在先前运行时服务调用完成和返回之前调用相同或某些其他运行时服务功能。这些限制适用于：

- 被中断的运行时服务；
- 在另一个处理器上处于活动状态的运行时服务。

如表 8-1（重新进入运行时服务的规则）中所述，调用者被禁止在中断后使用来自另一个处理器或同一处理器的某些其他服务。对于此表，“忙碌”定义为运行时服务已进入且尚未返回给调用者时的状态。

调用者违反这些限制的后果是未定义的，除了下面描述的某些特殊情况。

If previous call is busy in	Forbidden to call
Any	SetVirtualAddressMap()
ConvertPointer()	ConvertPointer()
SetVariable(), UpdateCapsule(), SetTime() SetWakeupTime(), GetNextHighMonotonicCount()	ResetSystem()
GetVariable() GetNextVariableName() SetVariable() QueryVariableInfo() UpdateCapsule() QueryCapsuleCapabilities() GetNextHighMonotonicCount()	GetVariable(), GetNextVariableName(), SetVariable(), QueryVariableInfo(), UpdateCapsule(), QueryCapsuleCapabilities(), GetNextHighMonotonicCount()
GetTime() SetTime() GetWakeupTime() SetWakeupTime()	GetTime() SetTime() GetWakeupTime() SetWakeupTime()

图 70. table8-1

如果操作系统在运行时不支持使用任何 `EFI_RUNTIME_SERVICES` 调用，则应发布 `EFI_RT_PROPERTIES_TABLE` 配置表，描述在运行时支持哪些运行时服务（请参阅第 4.6 节）。请注意，这只是对操作系统的提示，可以随意忽略，因此平台仍然需要提供不受支持的运行时服务的可调用实现，这些服务只返回 `EFI_UNSUPPORTED`。

7.1.1 Machine Check, INIT and NMI⁴ 异常 (Exception for Machine Check, INIT and NMI)

⁴1: NMI(Non-Maskable Interrupt): 不可屏蔽中断

某些异步事件（例如，IA-32 和 x64 系统上的 NMI、Itanium 系统上的 Machine Check 和 INIT）无法被屏蔽，并且可能在启用任何中断设置的情况下发生。这些事件还可能需要 OS 级处理程序的参与，这可能涉及调用某些运行时服务（见下文）。

如果调用了 `SetVirtualAddressMap()`，则在 Machine Check、INIT 或 NMI 之后对运行时服务的所有调用都必须使用该调用设置的虚拟地址映射进行。

Machine Check 可能中断了运行时服务（见下文）。如果 OS 确定 Machine Check 是可恢复的，则 OS 级别的处理程序必须遵循表 8-1 中的正常限制。

⁴TPL(Task Priority Levels): 任务优先级

如果 OS 确定 Machine Check 是不可恢复的，则 OS 级别的处理程序可能会忽略正常限制，并且可能会调用表 8-2（Machine Check、INIT 和 NMI 之后可能调用的函数）中描述的运行时服务，即使在先前调用繁忙的情况下也是如此。系统固件将接受新的运行时服务调用，并且不保证先前中断的调用的操作。任何中断的运行时函数都不会重新启动。

INIT 和 NMI 事件遵循相同的限制。

注：在 [Itanium](#) 系统上，操作系统机器检查处理程序不得调用 [ResetSystem\(\)](#)。如果需要重置，操作系统机器检查处理程序可能会请求 [SAL](#) [^2] 在返回到 [SAL_CHECK](#) 时重置。

^{^2}: SAL(System Abstraction Layer): 系统抽象层

平台实现需要清除任何正在进行的运行时服务，以便操作系统处理程序能够调用这些运行时服务，即使在先前的调用繁忙的情况下也是如此。在这种情况下，不能保证原来中断的呼叫的正常运行。

Function	Called after Machine Check, INIT and NMI
<code>GetTime()</code>	Yes, even if previously busy
<code>GetVariable()</code>	Yes, even if previously busy
<code>GetNextVariableName()</code>	Yes, even if previously busy
<code>QueryVariableInfo()</code>	Yes, even if previously busy
<code>SetVariable()</code>	Yes, even if previously busy
<code>UpdateCapsule()</code>	Yes, even if previously busy
<code>QueryCapsuleCapabilities()</code>	Yes, even if previously busy
<code>ResetSystem()</code>	Yes, even if previously busy

图 71. table8-2

7.2 可变服务 (Variable Services)

变量被定义为键/值对，由标识信息加上属性（键）和任意数据（值）组成。变量旨在用作存储在平台中实现的 EFI 环境与 EFI 操作系统加载程序和在 EFI 环境中运行的其他应用程序之间传递的数据的一种方式。

虽然本规范没有定义变量存储的实现，但在大多数情况下变量必须是持久的。这意味着平台上的 EFI 实现必须对其进行安排，以便在每次系统启动时保留并可供使用，至少在它们被明确删除或覆盖之前，传递给存储的变量都可以使用。在某些平台上提供这种类型的非易失性存储可能非常有限，因此在无法使用其他通信信息方式的情况下应谨慎使用变量。

表 8-3 列出了本节中描述的变量服务函数：

Name	Type	Description
GetVariable	Runtime	Returns the value of a variable.
GetNextVariableName	Runtime	Enumerates the current variable names.
SetVariable	Runtime	Sets the value of a variable.
QueryVariableInfo	Runtime	Returns information about the EFI variables

图 72. table8-3

GetVariable()

- 概要 (Summary)

返回变量的值。

- 原型 (Prototype)

```

1 typedef
2 EFI_STATUS GetVariable (
3     IN CHAR16      *VariableName,
4     IN EFI_GUID    *VendorGuid,
5     OUT UINT32     *Attributes OPTIONAL,
6     IN OUT UINTN   *DataSize,
7     OUT VOID       *Data OPTIONAL
8 );

```

- 参数 (Parameters)

VariableName: 一个以 `Null` 结尾的字符串，它是供应商变量的名称。

VendorGuid: 供应商的唯一标识符。`EFI_GUID` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

Attributes: 如果不是 `NULL`，则指向内存位置的指针以返回变量的属性位掩码。参见“相关定义”。如果不为 `NULL`，则在返回 `EFI_SUCCESS` 和 `EFI_BUFFER_TOO_SMALL` 时都在输出上设置 *Attributes*。

DataSize: 输入时，返回 *data* 缓冲区的大小（以字节为单位）。输出数据中返回的 *data* 大小。

Data: 返回变量内容的缓冲区。可以为 `NULL` 且 *DataSize* 为零，以确定所需的缓冲区大小。

- 相关定义 (Related Definitions)

```

1 //*****
2 // Variable Attributes
3 //*****

```

```

4 #define EFI_VARIABLE_NON_VOLATILE           0x00000001
5 #define EFI_VARIABLE_BOOTSERVICE_ACCESS     0x00000002
6 #define EFI_VARIABLE_RUNTIME_ACCESS          0x00000004
7 #define EFI_VARIABLE_HARDWARE_ERROR_RECORD   0x00000008 \
8
9 //This attribute is identified by the mnemonic 'HR' elsewhere
10 //in this specification.
11 #define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
12 //NOTE: EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS is deprecated
13 //and should be considered reserved.
14 #define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \ 0x00000020
15 #define EFI_VARIABLE_APPEND_WRITE            0x00000040
16 #define EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS 0x00000080
17 //This attribute indicates that the variable payload begins
18 //with an EFI_VARIABLE_AUTHENTICATION_3 structure, and
19 //potentially more structures as indicated by fields of this
20 //structure. See definition below and in SetVariable().

```

- 描述 (Description)

每个供应商都可以通过使用唯一的 *VendorGuid* 创建和管理自己的变量，而不会出现名称冲突的风险。设置变量时，将提供其 *Attributes* 以指示系统应如何存储和维护数据变量。这些属性会影响何时可以访问变量和数据的易变性。如果 `EFI_BOOT_SERVICES.ExitBootServices()` 已经执行，没有设置 `EFI_VARIABLE_RUNTIME_ACCESS` 属性的数据变量将对 `GetVariable()` 不可见，并将返回 `EFI_NOT_FOUND` 错误。

如果数据缓冲区太小无法容纳变量的内容，则返回错误 `EFI_BUFFER_TOO_SMALL` 并将 *DataSize* 设置为获取数据所需的缓冲区大小。

`EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` 和 `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` 属性都可以在 `GetVariable()` 调用的返回属性位掩码参数中设置，但应注意 `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` 属性已弃用，不应再使用。`EFI_VARIABLE_APPEND_WRITE` 属性永远不会在返回的 *Attributes* 位掩码参数中设置。

当调用 `GetVariable()` 时，使用 `EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS` 属性集存储的变量将返回元数据以及变量数据。如果 `GetVariable()` 调用指示设置了此属性，则必须根据元数据标头解释 `GetVariable()` 有效负载。除了 `SetVariable()` 中描述的标头之外，以下标头用于指示当前可能与变量关联的证书。

```

1 //
2 // EFI_VARIABLE_AUTHENTICATION_3_CERT_ID descriptor
3 //
4 // An extensible structure to identify a unique x509 cert
5 // associated with a given variable

```

```
6 //  
7 #define EFI_VARIABLE_AUTHENTICATION_3_CERT_ID_SHA256 1  
8  
9 typedef struct {  
10     UINT8    Type;  
11     UINT32   IdSize  
12 // UINT8 Id[IdSize];  
13 } EFI_VARIABLE_AUTHENTICATION_3_CERT_ID;
```

Type: 标识返回的 ID 类型以及应如何解释 ID。

IdSize: 指示结构中此字段后面的 Id 缓冲区的大小。

Id(*Not a formal structure member*): 这是 Type 字段定义的关联证书的唯一标识符。对于 CERT_ID_SHA256，缓冲区将是证书的 tbsCertificate (x509 中定义的待签名证书数据) 数据的 SHA-256 摘要。

当设置 EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS 属性时，数据缓冲区应解释如下：

```
1 // NOTE: “||” indicates concatenation.  
2  
3 // Example: EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE  
4 EFI_VARIABLE_AUTHENTICATION_3 || EFI_TIME ||  
    EFI_VARIABLE_AUTHENTICATION_3_CERT_ID || Data  
5  
6 // Example: EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE  
7 EFI_VARIABLE_AUTHENTICATION_3 || EFI_VARIABLE_AUTHENTICATION_3_NONCE ||  
    EFI_VARIABLE_AUTHENTICATION_3_CERT_ID || Data
```

注：每个示例中的 EFI_VARIABLE_AUTHENTICATION_3 结构的 MetadataSize 字段不包含任何 WIN_CERTIFICATE_UE

结构。这些结构用于 SetVariable() 接口，而不是 GetVariable()，如上例中所述。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The variable was not found.
EFI_BUFFER_TOO_SMALL	The <i>DataSize</i> is too small for the result. <i>DataSize</i> has been updated with the size needed to complete the request. If <i>Attributes</i> is not NULL, then the attributes bitmask for the variable has been stored to the memory location pointed-to by <i>Attributes</i> .
EFI_INVALID_PARAMETER	<i>VariableName</i> is NULL.
EFI_INVALID_PARAMETER	<i>VendorGuid</i> is NULL.
EFI_INVALID_PARAMETER	<i>DataSize</i> is NULL.
EFI_INVALID_PARAMETER	The <i>DataSize</i> is not too small and <i>Data</i> is NULL.
EFI_DEVICE_ERROR	The variable could not be retrieved due to a hardware error.
EFI_SECURITY_VIOLATION	The variable could not be retrieved due to an authentication failure.
EFI_UNSUPPORTED	After ExitBootServices() has been called, this return code may be returned if no variable storage is supported. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.

图 73. table8-3_1

GetNextVariableName()

• 概要 (Summary)

枚举当前变量名称。

• 原型 (Prototype)

```

1  typedef
2  EFI_STATUS GetNextVariableName (
3      IN OUT UINTN    *VariableNameSize,
4      IN OUT CHAR16   *VariableName,
5      IN OUT EFI_GUID *VendorGuid
6  );

```

• 参数 (Parameters)

VariableNameSize: 变量名缓冲区的大小。大小必须足够大以适合 *VariableName* 缓冲区中提供的输入字符串。

VariableName: 输入时, 提供由 `GetNextVariableName()` 返回的最后一个 *VariableName*。在输出时, 返回当前变量的 `Null` 终止字符串。

VendorGuid: 在输入时, 提供由 `GetNextVariableName()` 返回的最后一个 *VendorGuid*。在输出时, 返回当前变量的 *VendorGuid*。`EFI_GUID` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

- 描述 (Description)

多次调用 `GetNextVariableName()` 以检索系统中当前可用的所有变量的 *VariableName* 和 *VendorGuid*。在每次调用 `GetNextVariableName()` 时, 先前的结果都会传递到接口中, 而在输出时, 接口会返回下一个变量名称数据。返回整个变量列表后, 将返回错误 `EFI_NOT_FOUND`。

请注意, 如果返回 `EFI_BUFFER_TOO_SMALL`, 则 *VariableName* 缓冲区对于下一个变量来说太小了。发生此类错误时, 将更新 *VariableNameSize* 以反映所需缓冲区的大小。在所有情况下, 当调用 `GetNextVariableName()` 时, *VariableNameSize* 不得超过为 *VariableName* 分配的实际缓冲区大小。*VariableNameSize* 不得小于在 *VariableName* 缓冲区输入时传递给 `GetNextVariableName()` 的变量名称字符串的大小。

要开始搜索, 将以 `Null` 结尾的字符串传递到 *VariableName* 中; 也就是说, *VariableName* 是指向 `Null` 字符的指针。这始终在首次调用 `GetNextVariableName()` 时完成。当 *VariableName* 是指向 `Null` 字符的指针时, *VendorGuid* 将被忽略。`GetNextVariableName()` 不能用作过滤器以返回具有特定 GUID 的变量名称。相反, 必须检索整个变量列表, 并且调用者可以选择充当过滤器。在调用 `GetNextVariableName()` 之间调用 `SetVariable()` 可能会产生不可预知的结果。如果输入的 *VariableName* 缓冲区不是以 `Null` 结尾的字符串, 则返回 `EFI_INVALID_PARAMETER`。如果 *VariableName* 和 *VendorGuid* 的输入值不是现有变量的名称和 GUID, 则返回 `EFI_INVALID_PARAMETER`。

一旦执行 `EFI_BOOT_SERVICES.ExitBootServices()`, 将不再返回仅在引导服务期间可见的变量。要获取由 `GetNextVariableName()` 返回的变量的数据内容或属性, 可使用 `GetVariable()` 接口。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The next variable was not found.
EFI_BUFFER_TOO_SMALL	The <i>VariableNameSize</i> is too small for the result. <i>VariableNameSize</i> has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	<i>VariableNameSize</i> is NULL .
EFI_INVALID_PARAMETER	<i>VariableName</i> is NULL .
EFI_INVALID_PARAMETER	<i>VendorGuid</i> is NULL .
EFI_INVALID_PARAMETER	The input values of <i>VariableName</i> and <i>VendorGuid</i> are not a name and GUID of an existing variable.
EFI_INVALID_PARAMETER	Null-terminator is not found in the first <i>VariableNameSize</i> bytes of the input <i>VariableName</i> buffer.
EFI_DEVICE_ERROR	The variable name could not be retrieved due to a hardware error.
EFI_UNSUPPORTED	After <i>ExitBootServices()</i> has been called, this return code may be returned if no variable storage is supported. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.

图 74. table8-3_2

SetVariable()

- 概要 (Summary)

设置变量的值。此服务可用于创建新变量、修改现有变量的值或删除现有变量。

- 原型 (Prototype)

```

1 typedef EFI_STATUS SetVariable (
2     IN CHAR16    *VariableName,
3     IN EFI_GUID  *VendorGuid,
4     IN UINT32    Attributes,
5     IN UINTN     DataSize,
6     IN VOID      *Data
7 );

```

- 参数 (Parameters)

VariableName: 一个以 **Null** 结尾的字符串，它是供应商变量的名称。每个 *VariableName* 对于每个 *VendorGuid* 都是唯一的。变量名必须包含 1 个或多个字符。如果 *VariableName* 是空字符串，则返回

`EFI_INVALID_PARAMETER`。

VendorGuid: 供应商的唯一标识符。`EFI_GUID` 类型在 `EFI_BOOT_SERVICES.InstallProtocolInterface()` 函数描述中定义。

Attributes: 为变量设置的属性位掩码。请参阅 `GetVariable()` 函数说明。

DataSize: 数据缓冲区的大小(以字节为单位)。除非设置了 `EFI_VARIABLE_APPEND_WRITE`、`EFI_VARIABLE_AUTHENTICATED_ACCESS`、`EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS` 或 `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` 属性，否则大小为零会导致变量被删除。当设置了 `EFI_VARIABLE_APPEND_WRITE` 属性时，`DataSize` 为零的 `SetVariable()` 调用将不会导致变量值发生任何变化(然而，与变量关联的时间戳可能会更新，即使没有提供新的数据值；请参阅下面对 `EFI_VARIABLE_AUTHENTICATION_2` 描述符的描述)。在这种情况下，`DataSize` 不会为零，因为将填充 `EFI_VARIABLE_AUTHENTICATION_2` 描述符)。

Data: 变量的内容。

- 相关定义 (Related Definitions)

```
1 //*****
2 // Variable Attributes
3 //*****
4
5 // NOTE: This interface is deprecated and should no longer be used!
6 //
7 // EFI_VARIABLE_AUTHENTICATION descriptor
8 //
9 // A counter-based authentication method descriptor template
10 //
11 typedef struct {
12     UINT64 MonotonicCount;
13     WIN_CERTIFICATE_UEFI_GUID AuthInfo
14 } EFI_VARIABLE_AUTHENTICATION;
```

MonotonicCount: 包含在 **AuthInfo** 的签名中。用于确保新鲜度/无重播。在每次“写入”访问期间递增。

AuthInfo: 提供变量访问权限。它是跨可变数据和单调计数值的签名。调用方使用与通过密钥交换提供的公钥相关联的私钥。

```
1 //
2 // EFI_VARIABLE_AUTHENTICATION_2 descriptor
3 //
4 // A time-based authentication method descriptor template
5 //
6 typedef struct {
7     EFI_TIME TimeStamp;
```

```

8     WIN_CERTIFICATE_UEFI_GUID  AuthInfo
9 } EFI_VARIABLE_AUTHENTICATION_2;

```

TimeStamp: 与身份验证描述符关联的时间。对于 *TimeStamp* 值, 组件 *Pad1*、*Nanosecond*、*TimeZone*、*Daylight* 和 *Pad2* 应设置为 0。这意味着时间应始终以 GMT [^3] 表示。

^3: GMT(Greenwich Mean Time): 格林威治标准时间。

AuthInfo: 提供变量访问权限。仅接受 `EFI_CERT_TYPE_PKCS7_GUID` 的 *CertType*。

```

1 //
2 // EFI_VARIABLE_AUTHENTICATION_3 descriptor
3 //
4 // An extensible implementation of the Variable Authentication
5 // structure.
6 //
7 #define EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE 1
8 #define EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE      2
9
10 typedef struct {
11     UINT8    Version;
12     UINT8    Type;
13     UINT32   MetadataSize;
14     UINT32   Flags
15 } EFI_VARIABLE_AUTHENTICATION_3;

```

Version: 该字段用于 `EFI_VARIABLE_AUTHENTICATION_3` 结构本身需要更新的情况。目前, 它被硬编码为 “0x1”。

Type: 在可变数据有效负载中声明紧跟在该结构之后的结构。对于 `EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP`, 它将是 `EFI_TIME`(对于 *TimeStamp*)的一个实例。对于 `EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE`, 该结构将是 `EFI_VARIABLE_AUTHENTICATION_3_NONCE` 的一个实例。该结构定义如下。请注意, 这些结构都不包含 `WIN_CERTIFICATE_UEFI_GUID` 结构。有关结构排序的说明, 请参见第 8.2.1 节。

MetadataSize: 声明所有变量身份验证元数据 (与变量身份验证相关的数据本身不是变量数据) 的大小, 包括此标头结构和特定于类型的结构 (例如 `EFI_VARIABLE_AUTHENTICATION_3_NONCE`) 和任何 `WIN_CERTIFICATE_UEFI_GUID` 结构。

Flags: 指示此调用的任何可选配置的位字段。目前, 唯一定义的值是:

```

1 #define EFI_VARIABLE_ENHANCED_AUTH_FLAG_UPDATE_CERT 0x00000001

```

`SetVariable()` 上此标志的存在表明在特定类型的结构之后有 `WIN_CERTIFICATE_UEFI_GUID` 结构的两个实例。第一个实例描述了要设置为变量权限的新证书。第二个是授权当前更新的签名数据。注意: 所有其他位当前在 `SetVariable()` 上保留。注意: 所有标志都保留在 `GetVariable()` 上。

```
1 //  
2 // EFI_VARIABLE_AUTHENTICATION_3_NONCE descriptor  
3 //  
4 // A nonce-based authentication method descriptor template. This  
5 // structure will always be followed by a  
6 // WIN_CERTIFICATE_UEFI_GUID structure.  
7 //  
8 typedef struct {  
9     UINT32 NonceSize  
10    // UINT8 Nonce[NonceSize];  
11 } EFI_VARIABLE_AUTHENTICATION_3_NONCE;
```

NonceSize: 指示结构中此字段后面的 **Nonce** 缓冲区的大小。不得为 0。

Nonce (Not a formal structure member): 保证签名有效载荷的唯一随机值不能在多台机器或机器系列之间共享。在 **SetVariable()** 上, 如果 **Nonce** 字段全为 0, 主机将尝试使用内部生成的随机数。如果不可能, 将返回 **EFI_UNSUPPORTED**。此外, 在 **SetVariable()** 上, 如果变量已经存在并且随机数与当前随机数相同, 将返回 **EFI_INVALID_PARAMETER**。

- 描述 (Description)

变量由固件存储, 并且可以在电源循环期间保持它们的值。每个供应商都可以通过使用唯一的 **VendorGuid** 创建和管理自己的变量, 而不会出现名称冲突的风险。

每个变量都有定义固件如何存储和维护数据值的属性。如果未设置 **EFI_VARIABLE_NON_VOLATILE** 属性, 则固件会将变量存储在普通内存中, 并且不会在电源循环期间保持不变。这些变量用于将信息从一个组件传递到另一个组件。这方面的一个例子是固件的语言代码支持变量。它是在固件初始化时创建的, 供可能需要该信息的 **EFI** 组件访问, 但不需要备份到非易失性存储器。

EFI_VARIABLE_NON_VOLATILE 变量存储在存储容量有限的固定硬件中; 有时能力严重受限。软件只应在绝对必要时才使用非易失性变量。此外, 如果软件使用非易失性变量, 它应该尽可能使用只能在引导服务时访问的变量。

变量必须包含一个或多个字节的数据。除非设置了 **EFI_VARIABLE_APPEND_WRITE**、**EFI_VARIABLE_TIME_BASED_AUTHENTICATED_ACCESS** 或 **EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS** 属性 (见下文), 否则使用 **DataSize** 为零的 **SetVariable()** 将导致删除整个变量。被删除的变量占用的空间可能在下一次电源循环之前不可用。

如果具有匹配名称、GUID 和属性的变量已经存在, 则更新其值。

属性具有以下使用规则:

- 如果用不同的属性重写一个预先存在的变量, **SetVariable()** 不应修改该变量并应返回 **EFI_INVALID_PARAMETER**。唯一的例外是唯一不同的属性是 **EFI_VARIABLE_APPEND_WRITE**。在这种情况下, 调用的成功与否取决于写入的实际值。此规则有两个例外:

- * 如果一个预先存在的变量在没有指定访问属性的情况下被重写，该变量将被删除。
- * `EFI_VARIABLE_APPEND_WRITE` 属性代表了一种特殊情况。可以使用或不使用 `EFI_VARIABLE_APPEND_WRITE` 属性重写变量。
- * 设置没有访问属性的数据变量会导致它被删除。
- * `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` 已弃用，不应使用。如果 `SetVariable()` 的调用方指定了此属性，平台应返回 `EFI_UNSUPPORTED`。
- * 除非设置了 `EFI_VARIABLE_APPEND_WRITE`、`EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` 或 `EFI_VARIABLE_ENHANCED_AUTHENTICATED_WRITE_ACCESS` 属性，设置指定为零 `DataSize` 的数据变量会导致其被删除。
- * 对数据变量的运行时访问意味着引导服务访问。设置了 `EFI_VARIABLE_RUNTIME_ACCESS` 的属性也必须设置 `EFI_VARIABLE_BOOTSERVICE_ACCESS`。调用者有责任遵守此规则。
- * 一旦执行了 `EFI_BOOT_SERVICES.ExitBootServices()`，没有设置 `EFI_VARIABLE_RUNTIME_ACCESS` 的数据变量将不再对 `GetVariable()` 可见。
- * 执行 `ExitBootServices()` 后，只有设置了 `EFI_VARIABLE_RUNTIME_ACCESS` 和 `EFI_VARIABLE_NON_VOLATILE` 的变量才能使用 `SetVariable()` 进行设置。执行 `ExitBootServices()` 后，具有运行时访问权限但非易失性的变量是只读数据变量。当在 `SetVariable()` 调用中设置 `EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS` 属性时，身份验证应使用 `EFI_VARIABLE_AUTHENTICATION_3` 描述符，其后将是类型和标志字段中指示的任何描述符。
- * 当在 `SetVariable()` 调用中设置 `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` 属性时，验证应使用 `EFI_VARIABLE_AUTHENTICATION_2` 描述符。
- * 如果在 `SetVariable()` 调用中设置了 `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` 和 `EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS` 属性，则固件必须返回 `EFI_INVALID_PARAMETER`。
- * 如果在 `SetVariable()` 调用中设置了 `EFI_VARIABLE_APPEND_WRITE` 属性，则任何现有变量值都应附加 `Data` 参数的值。如果固件不支持附加操作，则 `SetVariable()` 调用应返回 `EFI_INVALID_PARAMETER`。
- * 如果在 `SetVariable()` 调用中设置了 `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` 属性，并且固件不支持包含在 `EFI_VARIABLE_AUTHENTICATION_2` 描述符中的证书的签名类型，则 `SetVariable()` 调用应返回 `EFI_INVALID_PARAMETER`。固件支持的签名类型列表由 `SignatureSupport` 变量定义。证书的签名类型由其摘要和加密算法定义。
- * 如果设置了 `EFI_VARIABLE_HARDWARE_ERROR_RECORD` 属性，`VariableName` 和 `VendorGuid` 必须符合 **Section 8.2.4.2** 和 **附录 P** 中规定的规则。否则，`SetVariable()` 调用应返回 `EFI_INVALID_PARAMETER`。

- * 必须使用 [Boot Manager](#) 这一章 **Table 3-1** 中定义的属性创建 **Globally Defined Variables**。如果使用错误的属性创建全局定义的变量，则结果是不确定的，并且可能因实现而异。
- * 如果使用 [EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS](#) 接口更新给定变量的证书颁发机构，有效负载的 *Data* 区域为空是有效的。这将在不修改数据本身的情况下更新证书。如果 *Data* 区域为空且未指定 *NewCert*，则将删除该变量（假设已验证所有授权）。
- * 必须使用 [EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS](#) 属性集创建安全启动策略变量，并且身份验证应使用 [EFI_VARIABLE_AUTHENTICATION_2](#) 描述符。如果未设置适当的属性位，则固件应返回 [EFI_INVALID_PARAMETER](#)。

固件在保存非易失性变量时必须执行的唯一规则是它在返回 [EFI_SUCCESS](#) 之前实际上已经保存到非易失性存储中，并且不执行部分保存。如果在调用 [SetVariable\(\)](#) 期间发生电源故障，则变量可能包含其先前值或新值。此外，没有读取、写入或删除安全保护。

要删除使用 [EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS](#) 属性创建的变量，必须将 [SetVariable](#) 与与现有变量匹配的属性一起使用，并将 *DataSize* 设置为 *AuthInfo* 描述符的大小。数据缓冲区必须包含 *AuthInfo* 描述符的实例，该实例将根据上面相应部分中参考更新已验证变量的步骤进行验证。尝试删除使用 [EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS](#) 属性创建的变量时，规定的 *AuthInfo* 验证失败或使用 **0** 值的 *DataSize* 调用时将失败，状态为 [EFI_SECURITY_VIOLATION](#)。

要删除使用 [EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS](#) 属性创建的变量，必须将 [SetVariable](#) 与与现有变量匹配的属性一起使用，并将 *DataSize* 设置为整个有效负载的大小，包括所有描述符和证书。数据缓冲区必须包含 [EFI_VARIABLE_AUTHENTICATION_3](#) 描述符的实例，它将指示如何根据 **Section 8.2.1** 中的描述验证有效负载。尝试删除使用 [EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS](#) 属性创建的变量时，规定的验证失败或使用 **0** 值的 *DataSize* 调用时将失败，状态为 [EFI_SECURITY_VIOLATION](#)。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	The firmware has successfully stored the variable and its data as defined by the Attributes.
EFI_INVALID_PARAMETER	An invalid combination of attribute bits, name, and GUID was supplied, or the <i>DataSize</i> exceeds the maximum allowed.
EFI_INVALID_PARAMETER	<i>VariableName</i> is an empty string.
EFI_OUT_OF_RESOURCES	Not enough storage is available to hold the variable and its data.
EFI_DEVICE_ERROR	The variable could not be saved due to a hardware failure.
EFI_WRITE_PROTECTED	The variable in question is read-only.
EFI_WRITE_PROTECTED	The variable in question cannot be deleted.
EFI_SECURITY_VIOLATION	The variable could not be written due to EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS or EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS being set, but the payload does NOT pass the validation check carried out by the firmware.
EFI_NOT_FOUND	The variable trying to be updated or deleted was not found.
EFI_UNSUPPORTED	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.

图 75. table8-3_3

QueryVariableInfo()

- 概要 (Summary)

返回有关 EFI 变量的信息。

- 原型 (Prototype)

```

1  typedef EFI_STATUS QueryVariableInfo (
2      IN UINT32    Attributes,
3      OUT UINT64   *MaximumVariableStorageSize,
4      OUT UINT64   *RemainingVariableStorageSize,
5      OUT UINT64   *MaximumVariableSize
6  );

```

- 参数 (Parameters)

Attributes: 属性位掩码指定返回信息的变量类型。请参阅 [GetVariable\(\)](#) 函数说明。**EFI_VARIABLE_APPEND_WRITE**

属性，如果在属性位掩码中设置，将被忽略。

MaximumVariableStorageSize: 输出时可用于与指定属性关联的 EFI 变量的存储空间的最大大小。

RemainingVariableStorageSize: 返回可用于与指定属性关联的 EFI 变量的存储空间的剩余大小。

MaximumVariableSize: 返回与指定属性关联的单个 EFI 变量的最大大小。

- 描述 (Description)

`QueryVariableInfo()` 函数允许调用者获取有关 EFI 变量可用存储空间的最大大小、EFI 变量可用存储空间的剩余大小以及每个单独的 EFI 变量的最大大小的信息，与指定的属性。

MaximumVariableSize 值将反映与保存单个 EFI 变量相关的开销，与 EFI 变量的字符串名称的长度相关的开销除外。

返回的 *MaximumVariableStorageSize*、*RemainingVariableStorageSize*、*MaximumVariableSize* 信息可能会在调用后根据其他运行时活动（包括异步错误事件）立即更改。此外，这些与不同属性关联的值在本质上不是相加的。

系统转换到运行时后（调用 `ExitBootServices()` 后），实现可能无法准确返回有关引导服务变量存储的信息。在这种情况下，应返回 `EFI_INVALID_PARAMETER`。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	Valid answer returned.
<code>EFI_INVALID_PARAMETER</code>	An invalid combination of attribute bits was supplied
<code>EFI_UNSUPPORTED</code>	The attribute is not supported on this platform, and the <i>MaximumVariableStorageSize</i> , <i>RemainingVariableStorageSize</i> , <i>MaximumVariableSize</i> are undefined.

图 76. table8-3_4

7.2.1 使用 `EFI_VARIABLE_AUTHENTICATION_3` 描述符 (Using the `EFI_VARIABLE_AUTHENTICATION_3` descriptor)

当设置 `EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS` 属性时，有效负载缓冲区（作为“数据”传递到 `SetVariable()` 中）应按如下方式构造：

```

1 // NOTE: "||" indicates concatenation.
2 // NOTE: "[" indicates an optional element.
3
4 // Example: EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE

```

```

5 EFI_VARIABLE_AUTHENTICATION_3 || EFI_TIME || [ NewCert ] || SigningCert || Data
6
7 // Example: EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE
8 EFI_VARIABLE_AUTHENTICATION_3 || EFI_VARIABLE_AUTHENTICATION_3_NONCE || [ NewCert ] ||
  SigningCert || Data

```

在此示例中, *NewCert* 和 *SigningCert* 都是 [WIN_CERTIFICATE_UEFI_GUID](#) 的实例。*NewCert* 的存在由 [EFI_VARIABLE_AUTHENTICATION_3](#) 的 *Flags* 字段指示 (请参阅 [SetVariable\(\)](#) 中的定义)。如果提供 - 并假设有效负载通过所有完整性和安全验证 - 此证书将被设置为基础变量的新权限, 即使变量是新创建的。

NewCert 元素必须具有 [EFI_CERT_TYPE_PKCS7_GUID](#) 的 *CertType*, 并且 *CertData* 必须是符合 **PKCS#7 version 1.5 (RFC 2315)** 的 [DER-encoded SignedData](#) 结构, 根据 **PKCS#7 version 1.5**, 无论是否使用 [DER-encoded ContentInfo](#) 结构都应支持该结构。创建 *SignedData* 结构时, 应遵循以下步骤:

1. 创建 [WIN_CERTIFICATE_UEFI_GUID](#) 结构, 其中 *CertType* 设置为 [EFI_CERT_TYPE_PKCS7_GUID](#)。
2. 使用添加的 **x509** 证书作为新权限来签署自己的 *tbsCertificate* 数据。
3. 构造一个 **PKCS#7 version 1.5 SignedData** (参见 [RFC2315]), 其签名内容如下:
 - *SignedData.version* 应设置为 1。
 - *SignedData.digestAlgorithms* 应包含准备签名时使用的摘要算法。仅接受 **SHA-256** 的摘要算法。
 - *SignedData.contentInfo.contentType* 应设置为 **id-data**。
 - *SignedData.contentInfo.content* 应该是为新的 **x509** 证书签名的 *tbsCertificate* 数据。
 - *SignedData.certificates* 应至少包含签名者的 [DER-encoded X.509](#) 证书。
 - *SignedData.crls* 是可选的。
 - *SignedData.signerInfos* 应构造为:
 - *SignerInfo.version* 应设置为 1;
 - *SignerInfo.issuerAndSerial* 应存在并与签名者的证书相同;
 - *SignerInfo.authenticatedAttributes* 不应存在;
 - *SignerInfo.digestEncryptionAlgorithm* 应设置为用于签署数据的算法。仅接受具有 **PKCS #1 v1.5 padding (RSASSA_PKCS1v1_5)** 的 RSA 摘要加密算法;
 - *SignerInfo.encryptedDigest* 应存在;
 - *SignerInfo.unauthenticatedAttributes* 不应存在;
 - 将 *CertData* 字段设置为 [DER-encoded PKCS#7 SignedData](#) 值。
4. 将 *CertData* 字段设置为 [DER-encoded PKCS#7 SignedData](#) 值。

尝试创建、更新或删除带有 `EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS` 集的变量的 `SetVariable()` 调用者应执行以下步骤来为 `SignedCert` 创建 `SignedData` 结构：

1. 使用以下值创建 `EFI_VARIABLE_AUTHENTICATION_3` 主描述符：

- 版本应设置为适合正在使用的元数据头的版本（当前为 1）；
- 类型应根据调用者规范进行设置（请参阅 `SetVariable()` 下的 `EFI_VARIABLE_AUTHENTICATION_3` 描述符）；
- `MetadataSize` 暂时可以忽略，会在构建最终 `payload` 时更新；
- 应根据调用方规范设置标志。

2. 可能需要根据类型创建二级描述符：

- 对于 `EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE` 类型，这将是 `EFI_TIME` 设置为当前时间的一个实例；
- 对于 `EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE` 类型，这将是 `EFI_VARIABLE_AUTHENTICATION_3_N` 的实例，根据调用者规范（不得为零）设置 `NonceSize` 更新，并将 `Nonce`（非正式结构成员）设置为：
 - 全零请求平台创建随机数；
 - 调用者为预生成的随机数指定值。
- 散列有效负载的序列化。序列化应按此顺序包含以下元素：
 - `VariableName`、`VendorGuid`、`Attributes` 和 `Secondary` 描述符（如果存在）；
 - 变量的新值（即 `Data` 参数的新变量内容）；
 - 如果这是对类型为 `EFI_VARIABLE_AUTHENTICATION_3_NONCE` 的变量的更新或删除，请序列化当前随机数。当前随机数是当前与该变量关联的随机数，而不是二级描述符中的随机数。仅序列化 `nonce` 缓冲区内容，而不序列化大小或任何其他数据。如果这是尝试创建一个新变量（即没有当前随机数），请跳过此步骤；
- 签署生成的摘要。
- 创建 `WIN_CERTIFICATE_UFBI_GUID` 结构，其中 `CertType` 设置为 `EFI_CERT_TYPE_PKCS7_GUID`。
- 按照上面为 `NewCert`（步骤 3）描述的步骤构建一个 `DER-encoded PKCS #7 version 1.5 SignedData`（请参阅 [RFC2315]），但有以下例外：
 - `SignedData.contentInfo.content` 应不存在（内容在 `Data` 参数中提供给 `SetVariable()` 调用）。
- 根据本节开头对“payload buffer”的描述，构造 `SetVariable()` 的最终 `payload`。

- 更新 `EFI_VARIABLE_AUTHENTICATION_3.MetadataSize` 字段以包含最终有效负载的所有部分，但“数据”除外。

实现 `SetVariable()` 服务并支持 `EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS` 属性的固件应执行以下操作以响应被调用：

1. 阅读 `EFI_VARIABLE_AUTHENTICATION_3` 描述符以确定正在执行的身份验证类型以及如何解析其余有效负载。
2. 验证 `SigningCert.CertType` 是否为 `EFI_CERT_TYPE_PKCS7_GUID`。
 - 如果 `EFI_VARIABLE_AUTHENTICATION_3.Flags` 字段指示存在 `NewCert`, 请验证 `NewCert.CertType` 是否为 `EFI_CERT_TYPE_PKCS7_GUID`;
 - 如果任一失败, 则返回 `EFI_INVALID_PARAMETER`。
3. 如果该变量已存在, 请验证传入类型是否与现有类型匹配。
4. 确认任何 `EFI_TIME` 结构都将 `Pad1`、`Nanosecond`、`TimeZone`、`Daylight` 和 `Pad2` 字段设置为零。
5. 如果 `EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE`:
 - 验证 `NonceSize` 是否大于零。如果为零, 则返回 `EFI_INVALID_PARAMETER`;
 - 如果传入的 `nonce` 全为 0, 确认平台支持生成随机 `nonce`。如果不支持, 则返回 `EFI_UNSUPPORTED`;
 - 如果指定了 `nonce` 并且变量已经存在, 请验证传入的 `nonce` 是否与现有的 `nonce` 不匹配。如果相同, 则返回 `EFI_INVALID_PARAMETER`。
6. 如果 `EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE` 和变量已经存在, 请验证新时间戳在时间上是否大于当前 `Timestamp`。
7. 通过以下方式验证有效负载签名：
 - 根据描述符解析整个有效载荷;
 - 使用描述符内容 (以及必要时来自现有变量的元数据) 构建本节前面描述的序列化 (`SetVariable()` 指令的第 3 步);
 - 计算摘要并与将 `SigningCert` 的公钥应用于签名的结果进行比较。
8. 如果该变量已存在, 请验证 `SigningCert` 权限是否与已与该变量关联的权限相同。
9. 如果提供了 `NewCert`, 请通过以下方式验证 `NewCert` 签名：
 - 根据描述符解析整个有效载荷;
 - 计算 `NewCert` 中 **x509** 证书的 `tbsCertificate` 摘要, 并与将 `NewCert` 的公钥应用于签名的结果进行比较;

- 如果失败，返回 `EFI_SECURITY_VIOLATION`。

7.2.2 使用 `EFI_VARIABLE_AUTHENTICATION_2` 描述符 (Using the `EFI_VARIABLE_AUTHENTICATION_2` descriptor)

当设置了 `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` 属性时，数据缓冲区应以一个完整（和序列化）的实例开始。

`EFI_VARIABLE_AUTHENTICATION_2` 描述符。描述符后应跟新变量值，`DataSize` 应反映描述符和新变量值的组合大小。身份验证描述符不是变量数据的一部分，并且不会由对 `GetVariable()` 的后续调用返回。

使用 `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` 属性集调用 `SetVariable()` 服务的调用方应在调用服务之前执行以下操作：

1. 创建描述符

创建 `EFI_VARIABLE_AUTHENTICATION_2` 描述符，其中：

- `TimeStamp` 设置为当前时间；

注：在某些环境中，可靠的时间源可能不可用。在这种情况下，由于 `EFI_VARIABLE_APPEND_WRITE` 属性在设置时会禁用时间戳验证（见下文），因此实现可能仍会向经过身份验证的变量添加值。在这些情况下，应使用 `EFI_TIME` 结构的每个组件（包括日和月）都设置为 0 的特殊时间值。

- `AuthInfo.CertType` 设置为 `EFI_CERT_TYPE_PKCS7_GUID`。

2. 散列 `SetVariable()` 调用的 `VariableName`、`VendorGuid` 和 `Attributes` 参数值的序列化以及 `EFI_VARIABLE_AUTHENTICATION_2` 描述符的 `TimeStamp` 组件，后跟变量的新值（即 `Data` 参数的新变量内容）。即 `digest = hash(VariableName, VendorGuid, Attributes, TimeStamp, DataNew_variable_content)`。终止 `VariableName` 值的 `NULL` 字符不应包含在散列计算中。

3. 使用选定的签名方案（例如 `PKCS #1 v1.5`）对生成的摘要进行签名。

4. 根据 `PKCS#7 v1.5 (RFC 2315)` 构造一个 `DER-encoded SignedData` 结构，无论是否使用 `PKCS#7 v1.5` 的 `DER-encoded ContentInfo` 结构都应支持，签名内容如下：

- `SignedData.version` 应设置为 1；
- `SignedData.digestAlgorithms` 应包含准备签名时使用的摘要算法。仅接受 **SHA-256** 的摘要算法；
- `SignedData.contentInfo.contentType` 应设置为 `id-data`；
- `SignedData.contentInfo.content` 应不存在（内容在 `Data` 参数中提供给 `SetVariable()` 调用）；
- `SignedData.certificates` 应至少包含签名者的 `DER-encoded X.509` 证书；

- `SignedData.crls` 可选;
- `SignedData.signerInfos` 应构造为:
 - `SignerInfo.version` 应设置为 1;
 - `SignerInfo.issuerAndSerial` 应存在, 并且在签名者的证书中 `SignerInfo.authenticatedAttributes` 不应存在;
 - `SignerInfo.digestEncryptionAlgorithm` 应设置为用于签署数据的算法。仅接受具有 PKCS #1 v1.5 padding (RSASSA_PKCS1v1_5) 的 RSA 摘要加密算法;
 - `SignerInfo.encryptedDigest` 应存在;
 - `SignerInfo.unauthenticatedAttributes` 不应存在;
- 将 `AuthInfo.CertData` 设置为 DER-encoded PKCS #7 `SignedData` 值;
- 构造数据参数: 通过将完整的序列化 `EFI_VARIABLE_AUTHENTICATION_2` 描述符与变量的新值 (`DataNew_variable_content`) 连接起来构造 `SetVariable()` 的数据参数。

实现 `SetVariable()` 服务并支持 `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` 属性的固件应执行以下操作以响应调用:

1. 验证是否使用了正确的 `AuthInfo.CertType` (`EFI_CERT_TYPE_PKCS7_GUID`) 以及 `AuthInfo.CertData` 值是否正确解析为 PKCS #7 `SignedData` 值;
2. 验证 `TimeStamp` 值的 `Pad1`、`Nanosecond`、`TimeZone`、`Daylight` 和 `Pad2` 组件是否设置为零。除非设置了 `EFI_VARIABLE_APPEND_WRITE` 属性, 否则验证 `TimeStamp` 值是否晚于与变量关联的当前时间戳值;
3. 如果变量 `SetupMode=1`, 并且该变量是安全启动策略变量, 则固件实现应认为已通过以下步骤 4 和 5 中的检查, 并继续更新变量值, 如下所述。
4. 通过以下方式验证签名:
 - 从数据缓冲区中提取 `EFI_VARIABLE_AUTHENTICATION_2` 描述符;
 - 使用描述符内容和其他参数来: (a) 构造摘要算法的输入; (b) 计算摘要; (c) 将摘要与将签名者的公钥应用于签名的结果进行比较;
5. 如果变量是全局 PK 变量或全局 KEK 变量, 请验证是否使用当前 `Platform Key` 进行了签名;
 - 如果变量是步骤 3 中提到的 “db”、“dbt”、“dbr” 或 “dbx” 变量, 请验证签名者的证书链接到密钥交换密钥数据库中的证书 (或者签名是使用当前的平台密钥);
 - 如果变量是步骤 3 中提到的 “OsRecoveryOrder” 或 “OsRecover#####” 变量, 请验证签名者的证书是否链接到 “dbr” 数据库或密钥交换密钥数据库中的证书, 或者签名是使用当前平台密钥进行的;
 - 否则, 如果变量不是上述任何一种, 则应将其指定为私有验证变量。如果私有身份验证变量

不存在，则签名证书的主题的 **CN** 和顶级颁发者证书的 **tbsCertificate** 的哈希值（如果不存在其他证书或证书链长度为 1，则为签名证书本身）在 **SignedData.certificates** 中注册用于此变量的后续验证。实现可能只存储这两个元素的单个散列以减少存储需求。如果 **PrivateAuthenticated** 变量以前存在，则签名者的证书链接到以前与该变量关联的信息。请注意，由于不存在针对它们的撤销列表，如果证书链的任何成员遭到破坏，则撤销对私有身份验证变量证书的信任的唯一方法是删除该变量，重新颁发链中的所有证书颁发机构，并使用新的证书链重新创建变量。因此，剩下的好处可能是对发起者的强烈识别，或遵守某些证书颁发机构的政策。进一步注意，经过身份验证的变量更新的 PKCS7 包必须包含签名证书链，包括所需信任锚的完整证书。信任锚可能是中级证书或根，但由于它们为不同目的颁发的 CA 数量，许多根可能不适合作为信任锚。一些工具需要非默认参数来包含信任锚证书。

只有当所有这些检查都通过时，驱动程序才会更新变量的值。如果任何检查失败，固件必须返回 **EFI_SECURITY_VIOLATION**。仅当设置了 **EFI_VARIABLE_APPEND_WRITE** 属性时，固件才应执行对现有变量值的追加。

对于具有 **GUID EFI_IMAGE_SECURITY_DATABASE_GUID** 的变量（即数据缓冲区格式为 **EFI_SIGNATURE_LIST** 的地方），驱动程序不应执行已经是现有变量值的一部分的 **EFI_SIGNATURE_DATA** 值的追加。

注意：这种情况不被视为错误，并且本身不会导致返回 **EFI_SUCCESS** 以外的状态代码或不更新与变量关联的时间戳。

固件应将新 **TimeStamp** 与更新值相关联（在设置 **EFI_VARIABLE_APPEND_WRITE** 属性的情况下，这仅适用于新时间戳值晚于与变量关联的当前时间戳的情况）。

如果该变量以前不存在，并且不是上面第 3 步中列出的变量之一，则固件应将签名者的公钥与该变量相关联，以供将来验证之用。

7.2.3 使用 **EFI_VARIABLE_AUTHENTICATION** 描述符（Using the **EFI_VARIABLE_AUTHENTICATION** descriptor）

注意：此接口已弃用，不应再使用！它将从规范的未来版本中删除。

7.2.4 硬件错误记录持久化（Hardware Error Record Persistence）

本节定义如何实现硬件错误记录持久性。通过实现对硬件错误记录持久性的支持，该平台使操作系统能够利用 **EFI** 变量服务来保存硬件错误记录，因此它们是持久的，并且在操作系统会话中保持可用，直到它们被其创建者明确清除或覆盖。

7.2.4.1 硬件错误记录非易失性存储（Hardware Error Record Non-Volatile Store）

需要一个实现支持硬件错误记录持久化的平台，以保证一定数量的 NVR 可供 OS 用于保存硬件错误记录。平台通过 `QueryVariableInfo` 例程传达为错误记录分配的空间量，如 [附录 P](#) 中所述。

7.2.4.2 硬件错误记录变量（Hardware Error Record Variables）

本节定义了一组具有架构定义含义的硬件错误记录变量。除了定义的数据内容之外，每个这样的变量都有一个体系结构定义的属性，指示何时可以访问数据变量。具有 HR 属性的变量存储在 NVR 分配给错误记录的部分。NV、BS 和 RT 具有第 3.2 节中定义的含义。所有硬件错误记录变量都使用 `EFI_HARDWARE_ERROR_VARIABLE VendorGuid`：

```
1 #define EFI_HARDWARE_ERROR_VARIABLE \
2 {0x414E6BDD, 0xE47B, 0x47cc, {0xB2, 0x44, 0xBB, 0x61, 0x02, 0x0C, 0xF5, 0x16}}
```

Variable Name	Attribute	Description
HwErrRec####	NV, BS, RT, HR	A hardware error record. ##### is a printed hex value. No 0x or h is included in the hex value

图 77. table8-4 Hardware Error Record Persistence Variables

`HwErrRec####` 变量包含硬件错误记录。每个 `HwErrRec####` 变量都是名称 “`HwErrRec`” 附加一个唯一的 4 位十六进制数。例如 `HwErrRec0001`、`HwErrRec0002`、`HwErrRecF31A` 等。HR 属性表示此变量将存储在分配给错误记录的 NVR 部分中。

7.2.4.3 通用平台错误记录格式（Common Platform Error Record Format）

使用此接口保留的错误记录变量以通用平台错误记录格式进行编码，该格式在 UEFI 规范的附录 N 中进行了描述。由于使用此接口保存的错误记录符合此标准格式，因此错误信息可能会被 OS 以外的实体使用。

7.3 时间服务（Time Services）

本节包含与时间相关的功能的功能定义，操作系统通常需要这些功能在运行时访问管理时间信息和服务的底层硬件。这些接口的目的是为操作系统编写者提供硬件时间设备的抽象，从而减轻直接访问遗留硬件设备的需要。还有一个用于预引导环境的停止功能。表 8-5 列出了本节中描述的时间服务功能：

Name	Type	Description
GetTime	Runtime	Returns the current time and date, and the time-keeping capabilities of the platform.
SetTime	Runtime	Sets the current local time and date information.
GetWakeupTime	Runtime	Returns the current wakeup alarm clock setting.
SetWakeupTime	Runtime	Sets the system wakeup alarm clock time.

图 78. table8-5 Time Services Functions

GetTime()

- 概要 (Summary)

返回当前时间和日期信息，以及硬件平台的计时能力。

- 原型 (Prototype)

```

1 typedef EFI_STATUS GetTime (
2     OUT EFI_TIME           *Time,
3     OUT EFI_TIME_CAPABILITIES *Capabilities OPTIONAL
4 );

```

- 参数 (Parameters)

Time: 指向存储的指针以接收当前时间的快照。[EFI_TIME](#) 类型在“相关定义”中定义。

Capabilities: 指向缓冲区的可选指针，用于接收实时时钟设备的功能。[EFI_TIME_CAPABILITIES](#) 类型在“相关定义”中定义。

- 相关定义 (Related Definitions)

```

1 //*****
2 //EFI_TIME
3 //*****
4 // This represents the current time information
5 typedef struct {
6     UINT16 Year;          // 1900 - 9999
7     UINT8 Month;         // 1 - 12
8     UINT8 Day;           // 1 - 31
9     UINT8 Hour;          // 0 - 23
10    UINT8 Minute;        // 0 - 59
11    UINT8 Second;        // 0 - 59
12    UINT8 Pad1;

```

```

13     UINT32 Nanosecond; // 0 - 999,999,999
14     INT16 TimeZone; // -1440 to 1440 or 2047
15     UINT8 Daylight;
16     UINT8 Pad2
17 } EFI_TIME;
18
19 //*****
20 // Bit Definitions for EFI_TIME.Daylight. See below.
21 //*****
22 #define EFI_TIME_ADJUST_DAYLIGHT 0x01
23 #define EFI_TIME_IN_DAYLIGHT 0x02
24 //*****
25 // Value Definition for EFI_TIME.TimeZone. See below.
26 //*****
27 #define EFI_UNSPECIFIED_TIMEZONE 0x07FF

```

Year, Month, Day: 当前本地日期。

Hour, Minute, Second, Nanosecond: 当前当地时间。纳秒报告设备中的当前秒数。时间格式为 `hh:mm:ss.nnnnnnnnnn`。电池供电的实时时钟设备保持日期和时间。

TimeZone: 时间与 UTC 的偏移量（以分钟为单位）。如果值为 `EFI_UNSPECIFIED_TIMEZONE`, 则时间被解释为本地时间。*TimeZone* 是本地时间相对于 UTC 的分钟数。要计算 *TimeZone* 值, 请遵循以下等式: `Localtime = UTC - TimeZone`。为了进一步说明这一点, 下面给出了一个例子: `PST` (太平洋标准时间是中午 12 点) = `UTC` (晚上 8 点) - 8 小时 (480 分钟)。在这种情况下, 如果引用 `PST`, *TimeZone* 的值将为 480。

Daylight: 包含时间的夏令时信息的位掩码。`EFI_TIME_ADJUST_DAYLIGHT` 位指示时间是否受夏令时影响。此值并不表示时间已针对夏令时进行了调整。仅表示 `EFI_TIME` 进入夏令时时需要调整。如果设置了 `EFI_TIME_IN_DAYLIGHT`, 则时间已针对夏令时进行了调整。所有其他位必须为零。输入夏令时时, 如果时间受到影响, 但没有调整 (`DST = 1`), 则使用新计算:

1. 日期/时间适当增加;
2. *TimeZone* 应适当减少 (例如: 从 `PST` 移动到 `PDT` 时 +480 变为 +420);
3. 日光值变为 3;

退出夏令时时, 如果时间受到影响并已调整 (`DST = 3`), 则使用新计算:

1. 日期/时间应适当减少;
2. *TimeZone* 要适当增加;
3. 日光值变为 1;

```
1 //*****
2 // EFI_TIME_CAPABILITIES
3 //*****
4 // This provides the capabilities of the
5 // real time clock device as exposed through the EFI interfaces.
6 typedef struct {
7     UINT32 Resolution;
8     UINT32 Accuracy;
9     BOOLEAN SetsToZero
10 } EFI_TIME_CAPABILITIES;
```

Resolution: 提供实时时钟设备每秒计数的报告分辨率。对于普通的 PC-AT CMOS RTC 设备，此值将为 1 Hz 或 1，以指示该设备仅报告 1 秒分辨率的时间。

Accuracy: 提供实时时钟的计时精度，误差率为百万分之 1E-6。对于精度为百万分之 50 的时钟，此字段中的值为 50,000,000。

SetsToZero: **TRUE** 表示时间设置操作将设备的时间清除到 **Resolution** 报告级别以下。**FALSE** 表示在设置时间时不清除设备 **Resolution** 级别以下的状态。普通 PC-AT CMOS RTC 设备将此值设置为 **FALSE**。

- 描述 (Description)

GetTime() 函数返回一个在函数调用期间有效的时间。虽然返回的 **EFI_TIME** 结构包含 **TimeZone** 和夏令时信息，但实际时钟不维护这些值。**GetTime()** 返回的当前时区和夏令时信息是上次通过 **SetTime()** 设置的值。

GetTime() 函数在每次调用时应该花费大致相同的时间来读取时间。所有报告的设备功能都将被四舍五入。

在运行时期间，如果平台中存在 PC-AT CMOS 设备，则调用者必须在调用 **GetTime()** 之前同步对设备的访问。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The operation completed successfully.
<code>EFI_INVALID_PARAMETER</code>	<i>Time</i> is NULL .
<code>EFI_DEVICE_ERROR</code>	The time could not be retrieved due to a hardware error.
<code>EFI_UNSUPPORTED</code>	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.

图 79. table8-5_1

SetTime()

- 概要 (Summary)

设置当前本地时间和日期信息。

- 原型 (Prototype)

```
1 typedef EFI_STATUS SetTime (
2     IN EFI_TIME *Time
3 );
```

- 参数 (Parameters)

Time: 指向当前时间的指针。`EFI_TIME` 类型在 `GetTime()` 函数描述中定义。对 `EFI_TIME` 结构的不同字段执行完整的错误检查 (有关完整详细信息, 请参阅 `GetTime()` 函数描述中的 `EFI_TIME` 定义), 如果任何字段超出范围, 则返回 `EFI_INVALID_PARAMETER`。

- 描述 (Description)

`SetTime()` 函数将实时时钟设备设置为提供的时间, 并记录当前时区和夏令时信息。`SetTime()` 函数不允许根据当前时间循环。例如, 如果设备不支持亚分辨率时间的硬件重置, 则代码不会通过等待时间回绕来实现该功能。

在运行期间, 如果平台中存在 PC-AT CMOS 设备, 则调用者必须在调用 `SetTime()` 之前同步对设备的访问。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The operation completed successfully.
<code>EFI_INVALID_PARAMETER</code>	A time field is out of range.
<code>EFI_DEVICE_ERROR</code>	The time could not be set due to a hardware error.
<code>EFI_UNSUPPORTED</code>	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.

图 80. table8-5_2

GetWakeupTime()

- 概要 (Summary)

返回当前唤醒闹钟设置。

- 原型 (Prototype)

```

1 typedef EFI_STATUS GetWakeupTime (
2     OUT BOOLEAN    *Enabled,
3     OUT BOOLEAN    *Pending,
4     OUT EFI_TIME   *Time
5 );

```

- 参数 (Parameters)

Enabled: 指示警报当前是启用还是禁用;

Pending: 指示警报信号是否未决并需要确认;

Time: 当前闹钟设置。`EFI_TIME` 类型在 `GetTime()` 函数描述中定义。

- 描述 (Description)

闹钟时间可以从设置的闹钟时间四舍五入到闹钟设备的分辨率内。闹钟设备的分辨率定义为一秒。

在运行时期间，如果平台中存在 PC-AT CMOS 设备，则调用者必须在调用 `GetWakeupTime()` 之前同步对设备的访问。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The alarm settings were returned.
<code>EFI_INVALID_PARAMETER</code>	<i>Enabled</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>Pending</i> is NULL .
<code>EFI_INVALID_PARAMETER</code>	<i>Time</i> is NULL .
<code>EFI_DEVICE_ERROR</code>	The wakeup time could not be retrieved due to a hardware error.
<code>EFI_UNSUPPORTED</code>	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.

图 81. table8-5_3

SetWakeupTime()

- 概要 (Summary)

设置系统唤醒闹钟时间。

- 原型 (Prototype)

```

1 typedef EFI_STATUS SetWakeupTime (
2     IN BOOLEAN    Enable,
3     IN EFI_TIME   *Time OPTIONAL
4 );

```

- 参数 (Parameters)

Enable: 启用或禁用唤醒警报;

Time: 如果 *Enable* 为 `TRUE`, 则为设置唤醒警报的时间。`EFI_TIME` 类型在 `GetTime()` 函数描述中定义。如果 *Enable* 为 `FALSE`, 则此参数是可选的, 并且可以为 `NULL`。

- 描述 (Description)

设置系统唤醒警报会使系统在设置的时间唤醒或开机。当警报触发时, 警报信号将被锁存, 直到通过调用 `SetWakeupTime()` 确认以禁用警报。如果警报在系统进入休眠或关闭状态之前触发, 由于警报信号被锁存, 系统将立即唤醒。如果在系统关闭且没有足够的电力启动系统时发出警报, 系统会在电源恢复时启动。

对于 ACPI 感知操作系统, 此函数仅处理为所需的唤醒时间编程唤醒警报。操作系统仍然像往常一样通过 ACPI 电源管理寄存器组控制唤醒事件。

唤醒警报的分辨率定义为 1 秒。

在运行期间，如果平台中存在 PC-AT CMOS 设备，则调用者必须在调用 `SetWakeupTime()` 之前同步对设备的访问。

- 返回的状态码（Status Codes Returned）

<code>EFI_SUCCESS</code>	The function completed successfully.
<code>EFI_NOT_FOUND</code>	The variable was not found.
<code>EFI_BUFFER_TOO_SMALL</code>	The <code>DataSize</code> is too small for the result. <code>DataSize</code> has been updated with the size needed to complete the request. If <code>Attributes</code> is not NULL, then the attributes bitmask for the variable has been stored to the memory location pointed-to by <code>Attributes</code> .
<code>EFI_INVALID_PARAMETER</code>	<code>VariableName</code> is NULL.
<code>EFI_INVALID_PARAMETER</code>	<code>VendorGuid</code> is NULL.
<code>EFI_INVALID_PARAMETER</code>	<code>DataSize</code> is NULL.
<code>EFI_INVALID_PARAMETER</code>	The <code>DataSize</code> is not too small and <code>Data</code> is NULL.
<code>EFI_DEVICE_ERROR</code>	The variable could not be retrieved due to a hardware error.
<code>EFI_SECURITY_VIOLATION</code>	The variable could not be retrieved due to an authentication failure.
<code>EFI_UNSUPPORTED</code>	After <code>ExitBootServices()</code> has been called, this return code may be returned if no variable storage is supported. The platform should describe this runtime service as unsupported at runtime via an <code>EFI_RT_PROPERTIES_TABLE</code> configuration table.

图 82. table8-5_4

7.4 虚拟内存服务（Virtual Memory Services）

本节包含操作系统在运行时可选择使用的虚拟内存支持的函数定义。如果操作系统选择以虚拟寻址模式而不是平面物理模式进行 EFI 运行时服务调用，则操作系统必须使用本节中的服务将 EFI 运行时服务从平面物理寻址切换到虚拟寻址。Table 8-6 列出了本节中描述的虚拟内存服务函数。系统固件必须遵循 EFI 内存映射布局中 Section 2.3.2 到 Section 2.3.2 中概述的特定于处理器的规则，以使操作系统能够进行所需的虚拟映射。

Name	Type	Description
SetVirtualAddressMap	Runtime	Used by an OS loader to convert from physical addressing to virtual addressing.
ConvertPointer	Runtime	Used by EFI components to convert internal pointers when switching to virtual addressing.

图 83. table8-6 Virtual Memory Functions

SetVirtualAddressMap()

• 概要 (Summary)

将 EFI 固件的运行时寻址模式从物理更改为虚拟。

• 原型 (Prototype)

```

1 typedef EFI_STATUS SetVirtualAddressMap (
2     IN UINTN             MemoryMapSize,
3     IN UINTN             DescriptorSize,
4     IN UINT32            DescriptorVersion,
5     IN EFI_MEMORY_DESCRIPTOR *VirtualMap
6 );

```

• 参数 (Parameters)

MemoryMapSize: *VirtualMap* 的字节大小;

DescriptorSize: *VirtualMap* 中 *entry* 的大小 (以字节为单位);

DescriptorVersion: *VirtualMap* 中结构 *entry* 的版本;

VirtualMap: 包含所有运行时范围的新虚拟地址映射信息的内存描述符数组。*EFI_MEMORY_DESCRIPTOR* 类型在 [EFI_BOOT_SERVICES.GetMemoryMap\(\)](#) 函数描述中定义。

• 描述 (Description)

OS 加载程序使用 [SetVirtualAddressMap\(\)](#) 函数。该函数只能在运行时调用，并由系统内存映射的所有者调用：即调用 [EFI_BOOT_SERVICES.ExitBootServices\(\)](#) 的组件。所有类型为 [EVT_SIGNAL_VIRTUAL_ADDRESS_C](#) 的事件都必须在 [SetVirtualAddressMap\(\)](#) 返回之前发出信号。

此调用将 EFI 固件的运行时组件的地址更改为 *VirtualMap* 中提供的新虚拟地址。提供的 *VirtualMap* 必须在 [ExitBootServices\(\)](#) 处为内存映射中的每个条目提供一个新的虚拟地址，这些条目被标记为需要运行时使用。*VirtualMap* 中的所有虚拟地址字段都必须在 4 KiB 边界上对齐。

必须使用物理映射来调用 [SetVirtualAddressMap\(\)](#)。从此函数成功返回后，系统必须使用新分配的虚

拟映射进行任何未来调用。所有地址空间映射都必须根据原始地址映射中指定的可缓存性标志来完成。调用此函数时，将通知所有注册为在地址映射更改时发出信号的事件。收到通知的每个组件必须更新其新地址的任何内部指针。这可以通过 `ConvertPointer()` 函数来完成。通知所有事件后，EFI 固件会重新应用映像“修复”信息，以虚拟方式将所有运行时映像重新定位到它们的新地址。此外，EFI 运行时服务表中除 `SetVirtualAddressMap` 和 `ConvertPointer` 之外的所有字段都必须使用 `ConvertPointer()` 服务从物理指针转换为虚拟指针。`SetVirtualAddressMap()` 和 `ConvertPointer()` 服务只能在物理模式下调用，因此不需要将它们从物理指针转换为虚拟指针。EFI 系统表的几个字段必须使用 `ConvertPointer()` 服务从物理指针转换为虚拟指针。这些字段包括 `FirmwareVendor`、`RuntimeServices` 和 `ConfigurationTable`。由于 EFI 运行时服务表和 EFI 系统表的内容都被该服务修改，因此必须重新计算 EFI 运行时服务表和 EFI 系统表的 32 位 CRC。

虚拟地址映射只能应用一次。一旦运行时系统处于虚拟模式，对该函数的调用将返回 `EFI_UNSUPPORTED`。

- 返回的状态码（Status Codes Returned）

<code>EFI_SUCCESS</code>	The virtual address map has been applied.
<code>EFI_UNSUPPORTED</code>	EFI firmware is not at runtime, or the EFI firmware is already in virtual address mapped mode.
<code>EFI_INVALID_PARAMETER</code>	<code>DescriptorSize</code> or <code>DescriptorVersion</code> is invalid.
<code>EFI_NO_MAPPING</code>	A virtual address was not supplied for a range in the memory map that requires a mapping.
<code>EFI_NOT_FOUND</code>	A virtual address was supplied for an address that is not found in the memory map.
<code>EFI_UNSUPPORTED</code>	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an <code>EFI_RT_PROPERTIES_TABLE</code> configuration table.

图 84. table8-6_1

ConvertPointer()

- 概要（Summary）

确定要在后续内存访问中使用的新虚拟地址。

- 原型（Prototype）

```
1 typedef EFI_STATUS ConvertPointer (
2     IN UINTN  DebugDisposition,
```

```

3     IN VOID  **Address
4 );

```

- 参数 (Parameters)

DebugDisposition: 为正在转换的指针提供类型信息。参见“相关定义”。

Address: 指向要固定为正在应用的新虚拟地址映射所需的值的指针的指针。

- 相关定义 (Related Definitions)

```

1 //*****
2 // EFI OPTIONAL_PTR
3 //*****
4 #define EFI OPTIONAL_PTR 0x00000001

```

- 描述 (Description)

EFI 组件在 `SetVirtualAddressMap()` 操作期间使用 `ConvertPointer()` 函数。在执行 `SetVirtualAddressMap()` 期间，必须使用物理地址指针调用 `ConvertPointer()`。

`ConvertPointer()` 函数将 *Address* 指向的当前指针更新为新地址映射的正确值。只有运行时组件需要执行此操作。`EFI_BOOT_SERVICES.CreateEvent()` 函数用于创建一个事件，该事件将在地址映射发生变化时被通知。必须更新组件已分配或指定的所有指针。

如果指定了 `EFI OPTIONAL_PTR` 标志，则允许被转换的指针为 `NULL`。

一旦所有组件都收到地址映射更改的通知，固件就会修复嵌入任何运行时映像中的任何编译指针。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The pointer pointed to by <i>Address</i> was modified.
<code>EFI_NOT_FOUND</code>	The pointer pointed to by <i>Address</i> was not found to be part of the current memory map. This is normally fatal.
<code>EFI_INVALID_PARAMETER</code>	<i>Address</i> is <code>NULL</code> .
<code>EFI_INVALID_PARAMETER</code>	* <i>Address</i> is <code>NULL</code> and <i>DebugDisposition</i> does not have the <code>EFI OPTIONAL_PTR</code> bit set.
<code>EFI_UNSUPPORTED</code>	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an <code>EFI_RT_PROPERTIES_TABLE</code> configuration table.

图 85. table8-6_2

7.5 杂项运行时服务（Miscellaneous Runtime Services）

本节包含其他地方未定义但需要完成 EFI 环境定义的运行时服务的其余函数定义。Table 8-7 列出了杂项运行时服务。

Name	Type	Description
GetNextHighMonotonicCount	Runtime	Returns the next high 32 bits of the platform's monotonic counter.
ResetSystem	Runtime	Resets the entire platform.
UpdateCapsule	Runtime	Pass capsules to the firmware. The firmware may process the capsules immediately or return a value to be passed into <code>ResetSystem()</code> that will cause the capsule to be processed by the firmware as part of the reset process.
QueryCapsuleCapabilities	Runtime	Returns if the capsule can be supported via <code>UpdateCapsule()</code>

图 86. table8-7 Miscellaneous Runtime Services

7.5.1 重置系统（Reset System）

ResetSystem()

- 概要（Summary）

重置整个平台。如果平台支持 `EFI_RESET_NOTIFICATION_PROTOCOL`，则在完成平台重置之前，必须调用所有挂起的通知。

- 原型（Prototype）

```

1 typedef VOID (EFIAPI *EFI_RESET_SYSTEM) (
2     IN EFI_RESET_TYPE    ResetType,
3     IN EFI_STATUS        ResetStatus,
4     IN UINTN             DataSize,
5     IN VOID              *ResetData OPTIONAL
6 );

```

- 参数（Parameters）

`ResetType`: 要执行的重置类型。`EFI_RESET_TYPE` 类型在下面的“相关定义”中定义；

`ResetStatus`: 重置的状态代码。如果系统重置是正常操作的一部分，则状态代码将为 `EFI_SUCCESS`。如果系统重置是由于某种类型的故障，将使用最合适的 EFI 状态代码；

`DataSize`: `ResetData` 的大小（以字节为单位）；

ResetData: 对于 *EfiResetCold*、*EfiResetWarm* 或 *EfiResetShutdown* 的 *ResetType*, 数据缓冲区以 *Null* 终止字符串开头, 后面可以选择附加二进制数据。该字符串是调用者可以用来进一步指示系统重置原因的描述。对于 *EfiResetPlatformSpecific* 的 *ResetType*, 数据缓冲区也以 *Null* 终止的字符串开头, 后跟描述要执行的特定重置类型的 *EFI_GUID*。

- 相关定义 (Related Definitions)

```

1 //*****
2 // EFI_RESET_TYPE
3 //*****
4 typedef enum {
5     EfiResetCold,
6     EfiResetWarm,
7     EfiResetShutdown,
8     EfiResetPlatformSpecific
9 } EFI_RESET_TYPE;

```

- 描述 (Description)

ResetSystem() 函数重置整个平台, 包括所有处理器和设备, 并重新启动系统。

使用 *EfiResetCold* 的 *ResetType* 调用此接口会导致系统范围的重置。这会将系统内的所有电路设置为其初始状态。这种类型的复位与系统操作异步, 并且在不考虑周期边界的情况下进行操作。*EfiResetCold* 等同于系统电源循环。

使用 *EfiResetWarm* 的 *ResetType* 调用此接口会导致系统范围的初始化。处理器被设置为它们的初始状态, 挂起的周期没有被破坏。如果系统不支持此重置类型, 则必须执行 *EfiResetCold*。

使用 *EfiResetShutdown* 的 *ResetType* 调用此接口会导致系统进入等效于 ACPI G2/S5 或 G3 状态的电源状态。如果系统不支持这种重置类型, 那么当系统重新启动时, 它应该显示 *EfiResetCold* 属性。

使用 *EfiResetPlatformSpecific* 的 *ResetType* 调用此接口会导致系统范围的重置。重置的确切类型由 *EFI_GUID* 定义, 该 *EFI_GUID* 跟随传递到 *ResetData* 的 *Null* 终止的 *Unicode* 字符串。如果平台无法识别 *ResetData* 中的 *EFI_GUID*, 则平台必须选择支持的重置类型来执行。该平台可以选择记录来自发生的任何非正常重置的参数。

ResetSystem() 函数不返回。

7.5.2 获取下一个高单调计数 (Get Next High Monotonic Count)

本节介绍 *GetNextHighMonotonicCount* 运行时服务及其关联的数据结构。

GetNextHighMonotonicCount()

- 概要 (Summary)

返回平台单调计数器的下一个高 32 位。

- 原型 (Prototype)

```
1 typedef EFI_STATUS GetNextHighMonotonicCount (
2     OUT UINT32 *HighCount
3 );
```

- 参数 (Parameters)

HighCount: 指向返回值的指针。

- 描述 (Description)

`GetNextHighMonotonicCount()` 函数返回平台单调计数器的下一个高 32 位。

该平台的单调计数器由两个 32 位量组成：高 32 位和低 32 位。在引导服务期间，低 32 位值是易变的：它在每次系统重置时重置为零，并在每次调用 `GetNextMonotonicCount()` 时增加 1。高 32 位值是非易失性的，每当系统重置、调用 `GetNextHighMonotonicCount()` 或低 32 位计数（由 `GetNextMonotonicCount()` 返回）溢出时都会增加 1。

`EFI_BOOT_SERVICES.GetNextMonotonicCount()` 函数仅在引导服务时可用。如果操作系统希望将平台单调计数器扩展到运行时，它可以通过使用 `GetNextHighMonotonicCount()` 来实现。为此，在调用 `EFI_BOOT_SERVICES.ExitBootServices()` 之前，操作系统将调用 `GetNextMonotonicCount()` 以获取当前平台单调计数。然后操作系统将提供一个接口，通过以下方式返回下一个计数：

1. 最后一次计数加 1；
2. 在计数的低 32 位溢出之前，调用 `GetNextHighMonotonicCount()`。这会将平台的单调计数的非易失性部分的高 32 位增加 1。

该函数只能在运行时调用。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	The next high monotonic count was returned.
<code>EFI_DEVICE_ERROR</code>	The device is not functioning properly.
<code>EFI_INVALID_PARAMETER</code>	<i>HighCount</i> is NULL .
<code>EFI_UNSUPPORTED</code>	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.

图 87. table8-7_1

7.5.3 更新胶囊 (Update Capsule)

此运行时函数允许调用者将信息传递给固件。*Update Capsule* 通常用于更新固件 FLASH 或操作系统，让信息在系统重置时保持不变。

UpdateCapsule()

- 概要 (Summary)

通过虚拟和物理映射将胶囊传递给固件。根据预期消耗量，固件可能会立即处理胶囊。如果有效负载应在系统重置期间持续存在，则必须将 `EFI_QueryCapsuleCapabilities` 返回的重置值传递到 `ResetSystem()` 并将导致胶囊作为重置过程的一部分由固件处理。

- 原型 (Prototype)

```
1 typedef EFI_STATUS UpdateCapsule (
2     IN EFI_CAPSULE_HEADER    **CapsuleHeaderArray,
3     IN UINTN                 CapsuleCount,
4     IN EFI_PHYSICAL_ADDRESS  ScatterGatherList OPTIONAL
5 );
```

- 参数 (Parameters)

CapsuleHeaderArray: 指向被传递到更新胶囊的胶囊的虚拟指针数组的虚拟指针。假定每个胶囊存储在连续的虚拟内存中。*CapsuleHeaderArray* 中的胶囊必须与 *ScatterGatherList* 中的胶囊相同。*CapsuleHeaderArray* 必须具有与 *ScatterGatherList* 相同顺序的胶囊。

CapsuleCount: *CapsuleHeaderArray* 中指向 `EFI_CAPSULE_HEADER` 的指针数。

ScatterGatherList: 指向一组 `EFI_CAPSULE_BLOCK_DESCRIPTOR` 的物理指针，描述一组胶囊在物理内存中的位置。请参阅“相关定义”以了解如何通过此接口传递一个以上的胶囊。*ScatterGatherList* 中的胶囊必须与 *CapsuleHeaderArray* 的顺序相同。仅当 *capsules* 被定义为在系统重置时持续存在时才引用此参数。

- 相关定义 (Related Definitions)

```
1 typedef struct {
2     UINT64 Length;
3     union {
4         EFI_PHYSICAL_ADDRESS DataBlock;
5         EFI_PHYSICAL_ADDRESS ContinuationPointer;
6     } Union;
7 } EFI_CAPSULE_BLOCK_DESCRIPTOR;
```

Length: *DataBlock/ContinuationPointer* 指向的数据的字节长度；

DataBlock: 数据块的物理地址。如果 *Length* 不等于零，则使用该联合成员；

ContinuationPointer: 另一个 `EFI_CAPSULE_BLOCK_DESCRIPTOR` 结构块的物理地址。如果 *Length* 等于零，则使用该联合成员。如果 *ContinuationPointer* 为零，则此 *entry* 表示列表的末尾。

- 相关定义 (Related Definitions)

此数据结构定义操作系统传递给固件的 *ScatterGatherList* 列表。*ScatterGatherList* 表示一个结构体数组，以长度为 0 且 *DataBlock* 物理地址为 0 的结构体成员结束。如果 *Length* 为 0 且 *DataBlock* 物理地址不为 0，则指定的物理地址称为“连续指针”，它指向 `EFI_CAPSULE_BLOCK_DESCRIPTOR` 结构的进一步列表。连续指针用于允许分散收集列表包含在不连续的物理内存中。它还用于允许一次通过多个胶囊。

```
1 typedef struct {
2     EFI_GUID    CapsuleGuid;
3     UINT32      HeaderSize;
4     UINT32      Flags;
5     UINT32      CapsuleImageSize;
6 } EFI_CAPSULE_HEADER;
```

CapsuleGuid: 定义胶囊内容的 GUID;

HeaderSize: 胶囊标头的大小。这可能大于 `EFI_CAPSULE_HEADER` 的大小，因为 *CapsuleGuid* 可能暗示扩展的标头 *entry*;

Flags: 标志位 [15:0] 由 *CapsuleGuid* 定义，标志位 [31:16] 由本规范定义;

CapsuleImageSize: 胶囊的字节大小（包括胶囊标头）。

```
1 #define CAPSULE_FLAGS_PERSIST_ACROSS_RESET 0x00010000
2 #define CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE 0x00020000
3 #define CAPSULE_FLAGS_INITIATE_RESET        0x00040000
```

注：具有 `CAPSULE_FLAGS_INITIATE_RESET` 标志的胶囊必须在其标头中也设置 `CAPSULE_FLAGS_PERSIST_ACROSS_RESET`。遇到在其标头中设置了 `CAPSULE_FLAGS_INITIATE_RESET` 标志的胶囊的固件将启动与传入的胶囊请求兼容的平台重置，并且不会返回给调用者。

```
1 typedef struct {
2     UINT32    CapsuleArrayNumber;
3     VOID*     CapsulePtr[1];
4 } EFI_CAPSULE_TABLE;
```

CapsuleArrayNumber: 胶囊数组中的 *entry* 数;

CapsulePtr: 指向包含相同 *CapsuleGuid* 值的胶囊数组的指针。每个 *CapsulePtr* 都指向 `EFI_CAPSULE_HEADER` 的一个实例，胶囊数据连接在其末端。

- 描述 (Description)

`UpdateCapsule()` 函数允许操作系统将信息传递给固件。`UpdateCapsule()` 函数支持将操作系统虚拟

内存中的胶囊传递回固件。每个 *capsule* 都包含在操作系统的连续虚拟内存范围内，但 *capsules* 的虚拟和物理映射都会传递给固件。

如果 *capsule* 在其标头设置了 `CAPSULE_FLAGS_PERSIST_ACROSS_RESET` 标志，则固件将在系统重置后处理 *capsules*。调用者必须确保使用从 `QueryCapsuleCapabilities` 获得的所需重置值来重置系统。如果未设置此标志，固件将立即处理胶囊。

具有 `CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE` 标志的胶囊必须在其标头中也设置 `CAPSULE_FLAGS_PERSIST_ACROSS_RESET`。

处理在其标头中设置了 `CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE` 标志的胶囊的固件会将胶囊的内容从 *ScatterGatherList* 合并到一个连续的缓冲区中，然后必须在系统重置后在 EFI 系统表中放置指向该合并胶囊的指针。搜索此胶囊的代理将在 `EFI_CONFIGURATION_TABLE` 中查找并搜索胶囊的 GUID 和关联的指针以在重置后检索数据。

Flags	Firmware Behavior
No Specification defined flags	Firmware attempts to immediately process or launch the capsule. If capsule is not recognized, can expect an error.
CAPSULE_FLAGS_PERSIST_ACROSS_RESET	Firmware will attempt to process or launch the capsule across a reset. If capsule is not recognized, can expect an error. If the processing requires a reset which is unsupported by the platform, expect an error.
CAPSULE_FLAGS_PERSIST_ACROSS_RESET + CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE	Firmware will coalesce the capsule from the ScatterGatherList into a contiguous buffer and place a pointer to the coalesced capsule in the EFI System Table. Platform recognition of the capsule type is not required. If the action requires a reset which is unsupported by the platform, expect an error.
CAPSULE_FLAGS_PERSIST_ACROSS_RESET + CAPSULE_FLAGS_INITIATE_RESET	Firmware will attempt to process or launch the capsule across a reset. The firmware will initiate a reset which is compatible with the passed-in capsule request and will not return back to the caller. If the capsule is not recognized, can expect an error. If the processing requires a reset which is unsupported by the platform, expect an error.
CAPSULE_FLAGS_PERSIST_ACROSS_RESET + CAPSULE_FLAGS_INITIATE_RESET + CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE	The firmware will initiate a reset which is compatible with the passed-in capsule request and not return back to the caller. Upon resetting, the firmware will coalesce the capsule from the ScatterGatherList into a contiguous buffer and place a pointer to the coalesced capsule in the EFI System Table. Platform recognition of the capsule type is not required. If the action requires a reset which is unsupported by the platform, expect an error.

图 88. table8-8 Flag Firmware Behavior

EFI 系统表条目必须使用 [EFI_CAPSULE_HEADER](#) 的 *CapsuleGuid* 字段中的 GUID。EFI 系统表 *entry* 必须指向包含相同 *CapsuleGuid* 值的胶囊数组。该数组必须以表示胶囊数组大小的 *UINT32* 为前缀。

这组胶囊由 *ScatterGatherList* 和 *CapsuleHeaderArray* 指向，因此固件将知道操作系统分配的缓冲区的物理地址和虚拟地址。*scatter-gather* 列表支持 *capsule* 的虚拟地址范围是连续的，但物理地址不连续的情况。

在禁用内存管理单元时，处理器的主内存视图与缓存不一致的架构上，[UpdateCapsule\(\)](#) 的调用方必须在调用 [UpdateCapsul\(\)](#) 之前对每个 *ScatterGatherList* 元素的主内存执行缓存维护。此要求仅在操作系统调用 [ExitBootServices\(\)](#) 后适用。

如果传入此函数的任何胶囊遇到错误，则不会处理整组胶囊，并将遇到的错误返回给调用者。

- 返回的状态码 (Status Codes Returned)

<code>EFI_SUCCESS</code>	Valid capsule was passed. If <code>CAPSULE_FLAGS_PERSIST_ACROSS_RESET</code> is not set, the capsule has been successfully processed by the firmware.
<code>EFI_INVALID_PARAMETER</code>	<code>CapsuleSize</code> , or an incompatible set of flags were set in the capsule header.
<code>EFI_INVALID_PARAMETER</code>	<code>CapsuleCount</code> is 0
<code>EFI_DEVICE_ERROR</code>	The capsule update was started, but failed due to a device error.
<code>EFI_UNSUPPORTED</code>	The capsule type is not supported on this platform.
<code>EFI_OUT_OF_RESOURCES</code>	When <code>ExitBootServices()</code> has been previously called this error indicates the capsule is compatible with this platform but is not capable of being submitted or processed in runtime. The caller may resubmit the capsule prior to <code>ExitBootServices()</code> .
<code>EFI_OUT_OF_RESOURCES</code>	When <code>ExitBootServices()</code> has not been previously called then this error indicates the capsule is compatible with this platform but there are insufficient resources to process.
<code>EFI_UNSUPPORTED</code>	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an <code>EFI_RT_PROPERTIES_TABLE</code> configuration table.

图 89. table8-8_1

7.5.3.1 胶囊定义 (Capsule Definition)

胶囊只是一组以 `EFI_CAPSULE_HEADER` 开头的连续数据。标头中的 `CapsuleGuid` 字段定义了胶囊的格式。

胶囊内容旨在从存在操作系统的环境与系统固件进行通信。为了允许 `capsule` 在系统重置时持续存在，`capsule` 的描述需要一定程度的间接性，因为操作系统主要使用虚拟内存，而固件在启动时使用物理内存。这个抽象级别是通过 `EFI_CAPSULE_BLOCK_DESCRIPTOR` 完成的。`EFI_CAPSULE_BLOCK_DESCRIPTOR` 允许操作系统分配连续的虚拟地址空间，并将该地址空间描述为一组不连续的物理地址范围给固件。向固件传递物理地址和虚拟地址以及指针来描述胶囊，因此固件可以立即处理胶囊或将胶囊的处理推迟到系统重置之后。

在大多数指令集和操作系统架构中，物理内存的分配只能在“页面”粒度（范围从 4 KiB 到至少 1 MiB）上进行。`EFI_CAPSULE_BLOCK_DESCRIPTOR` 必须具有以下属性以确保数据的安全和定义良好的转换：

- 每个新胶囊必须从新的内存页开始；

- 除最后一页外，所有页面都必须被胶囊完全填满；
 - 填充标头以使其消耗整页数据以允许通过胶囊传递页面对齐的数据结构是合法的。最后一页必须至少有一个字节的胶囊；
- 页面必须自然对齐；
- 页面不能重叠；
- 固件可能永远不会假设操作系统正在使用的页面大小。

多个胶囊可以连接在一起，并通过一次调用 `UpdateCapsule()` 传递。胶囊的物理地址描述通过将第一个胶囊的终止 `EFI_CAPSULE_BLOCK_DESCRIPTOR entry` 转换为一个延续指针，使其指向代表第二个胶囊开始的 `EFI_CAPSULE_BLOCK_DESCRIPTOR` 来连接。只有一个终止 `EFI_CAPSULE_BLOCK_DESCRIPTOR entry`，它位于链中最后一个胶囊的末尾。

必须使用以下算法在单个分散收集列表中查找多个胶囊：

- 查看 `capsule` 标头以确定 `capsule` 的大小；
 - 第一个 `Capsule` 标头始终由第一个 `EFI_CAPSULE_BLOCK_DESCRIPTOR entry` 指向；
- 遍历 `EFI_CAPSULE_BLOCK_DESCRIPTOR` 列表，保持每个 `entry` 代表的大小的运行计数；
- 如果 `EFI_CAPSULE_BLOCK_DESCRIPTOR entry` 是一个延续指针，并且正在运行的当前胶囊大小计数大于或等于当前胶囊的大小，则这是下一个胶囊的开始；
- 使新胶囊成为当前胶囊并重复该算法。

Figure 8-1 显示了描述两个胶囊的 `EFI_CAPSULE_BLOCK_DESCRIPTOR` 结构的 *Scatter-Gather* 列表。图的左侧显示了胶囊的操作系统视图，作为两个独立的连续虚拟内存缓冲区。图中的中心显示了系统内存中数据的布局。该图的右侧显示了传递到固件中的 *ScatterGatherList* 列表。由于有两个胶囊，因此存在两个独立的 `EFI_CAPSULE_BLOCK_DESCRIPTOR` 列表，它们通过第一个列表中的延续指针连接在一起。

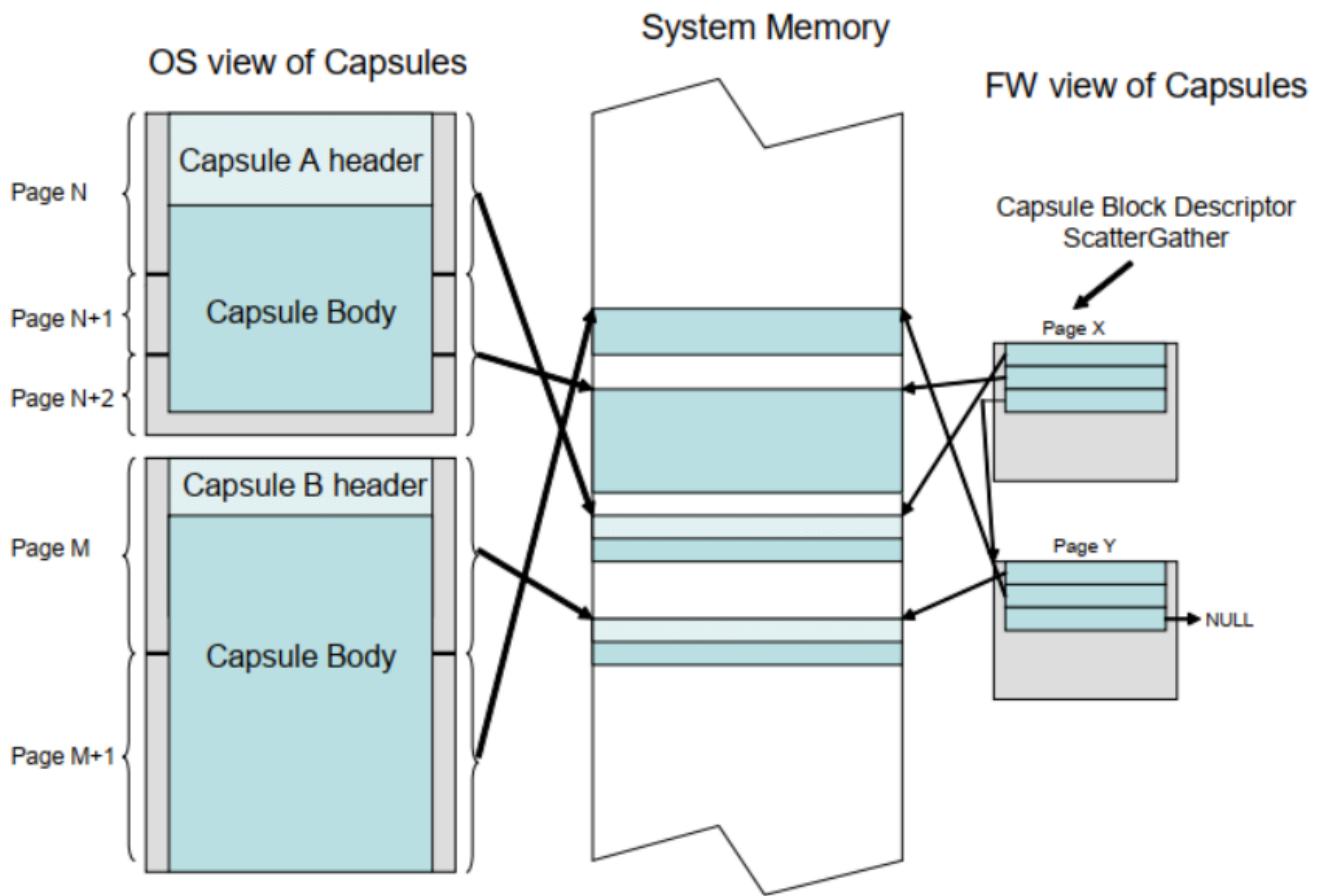


图 90. Figure 8-1 Scatter-Gather List of EFI_CAPSULE_BLOCK_DESCRIPTOR Structures

EFI_MEMORY_RANGE_CAPSULE_GUID

这种胶囊结构定义提供了一种方法，第三方组件（例如操作系统）可以通过这种方法向固件描述内存中的哪些区域在下一次重置时应该保持不变。

对此 *capsule* 的支持是可选的。对于支持此胶囊的平台，它们必须使用 `EFI_MEMORY_RANGE_CAPSULE_GUID` 作为 GUID/指针对中的 GUID 在 EFI 配置表中公布 `EFI_MEMORY_RANGE_CAPSULE`。

```

1 // {0DE9F0EC-88B6-428F-977A-258F1D0E5E72}
2 #define EFI_MEMORY_RANGE_CAPSULE_GUID \
3 { 0xde9f0ec, 0x88b6, 0x428f, \
4 { 0x97, 0x7a, 0x25, 0x8f, 0x1d, 0xe, 0x5e, 0x72 } }
```

内存范围描述符。

```

1 typedef struct {
2     EFI_PHYSICAL_ADDRESS Address;
3     UINT64 Length
4 } EFI_MEMORY_RANGE;

```

Address: 正在描述的内存位置的物理地址。

Length: 以字节为单位的长度。

描述平台固件应保持不变的内存范围的胶囊描述符。

```

1 typedef struct {
2     EFI_CAPSULE_HEADER Header;
3     UINT32 OsRequestedMemoryType;
4     UINT64 NumberOfMemoryRanges;
5     EFI_MEMORY_RANGE MemoryRanges[]
6 } EFI_MEMORY_RANGE_CAPSULE;

```

Header:
Header.CapsuleGuid = `EFI_MEMORY_RANGE_CAPSULE_GUI`,
Header.Flags = `CAPSULE_FLAGS_PERSIST_ACROSS_RESET`。

OsRequestedMemoryType: 必须在 `0x80000000-0xFFFFFFFF` 范围内。当 UEFI 固件处理胶囊时, *MemoryRanges[]* 中描述的内容将在 EFI 内存映射中显示为 *OsRequestedMemoryType* 值。

NumberofMemoryRanges: *MemoryRanges[] entry* 的数量。必须是 1 或更大的值。

MemoryRanges[]: 内存范围数组。等同于 *MemoryRanges[NumberOfMemoryRanges]*。

对于打算支持 `EFI_MEMORY_RANGE_CAPSULE` 的平台, 它必须使用 `EFI_MEMORY_RANGE_CAPSULE_GUID` 作为 GUID/指针对中的 GUID 在 EFI 配置表中公布 `EFI_MEMORY_RANGE_CAPSULE_RESULT`。

```

1 typedef struct {
2     UINT64 FirmwareMemoryRequirement;
3     UINT64 NumberOfMemoryRanges
4 } EFI_MEMORY_RANGE_CAPSULE_RESULT

```

FirmwareMemoryRequirement: UEFI 固件初始化所需的最大内存量 (以字节为单位)。

NumberofMemoryRanges: 如果没有处理 `EFI_MEMORY_RANGE_CAPSULE`, 则为 0。如果处理了 `EFI_MEMORY_RANGE_CAPSULE`, 则此数字将与 `EFI_MEMORY_RANGE_CAPSULE.NumberOfMemoryRanges` 值相同。

QueryCapsuleCapabilities()

- 概要 (Summary)

返回是否可以通过 `UpdateCapsule()` 支持胶囊。

- 原型 (Prototype)

```
1 typedef EFI_STATUS QueryCapsuleCapabilities {
2     IN EFI_CAPSULE_HEADER **CapsuleHeaderArray,
3     IN UINTN             CapsuleCount,
4     OUT UINT64            *MaximumCapsuleSize,
5     OUT EFI_RESET_TYPE    *ResetType
6 };
```

- 参数 (Parameters)

CapsuleHeaderArray: 指向被传递到更新胶囊的胶囊的虚拟指针数组的虚拟指针。假定胶囊存储在连续的虚拟内存中。

CapsuleCount: *CapsuleHeaderArray* 中指向 `EFI_CAPSULE_HEADER` 的指针数。

MaximumCapsuleSize: 在输出时, `UpdateCapsule()` 可以通过 *CapsuleHeaderArray* 和 `ScatterGatherList` 作为 `UpdateCapsule()` 的参数支持的最大大小 (以字节为单位)。输入未定义。

ResetType: 返回胶囊更新所需的重置类型。输入未定义。

- 描述 (Description)

`QueryCapsuleCapabilities()` 函数允许调用者测试是否可以通过 `UpdateCapsule()` 更新一个或多个胶囊。检查胶囊标头中的标志值和整个胶囊的大小。

如果调用者需要查询通用胶囊功能, 则可以构造一个伪造的 `EFI_CAPSULE_HEADER`, 其中 *CapsuleImageSize* 等于 *HeaderSize*, 即等于 `sizeof (EFI_CAPSULE_HEADER)`。要确定重置要求, 应在 `EFI_CAPSULE_HEADER` 的标志字段中设置 `CAPSULE_FLAGS_PERSIST_ACROSS_RESET`。

- 返回的状态码 (Status Codes Returned)

EFI_SUCCESS	Valid answer returned.
EFI_INVALID_PARAMETER	<i>MaximumCapsuleSize</i> is NULL .
EFI_UNSUPPORTED	The capsule type is not supported on this platform, and <i>MaximumCapsuleSize</i> and <i>ResetType</i> are undefined.
EFI_OUT_OF_RESOURCES	When ExitBootServices() has been previously called this error indicates the capsule is compatible with this platform but is not capable of being submitted or processed in runtime. The caller may resubmit the capsule prior to ExitBootServices() .
EFI_OUT_OF_RESOURCES	When ExitBootServices() has not been previously called then this error indicates the capsule is compatible with this platform but there are insufficient resources to process.
EFI_UNSUPPORTED	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.

图 91. table8-8_2

7.5.3.2 在操作系统和固件之间交换信息（Exchanging information between the OS and Firmware）

固件和操作系统可以通过 *OsIndicationsSupported* 和 *OSIndications* 变量交换信息，如下所示：

- *OsIndications* 变量返回操作系统拥有的 UINT64 位掩码，用于指示操作系统希望固件启用哪些功能或操作系统希望固件执行哪些操作。操作系统将通过 *SetVariable()* 调用提供此数据；
- *OsIndicationsSupported* 变量返回固件拥有的 UINT64 位掩码，并指示固件支持哪些操作系统指示功能和操作。这个变量在每次启动时由固件重新创建，并且不能被操作系统修改。

如果固件支持操作系统请求在固件用户界面停止，则固件可以在 *OsIndicationsSupported* 变量中设置 **EFI_OS_INDICATIONS_BOOT_TO_FW_UI** 位。如果操作系统希望固件在下次启动时在固件用户界面停止，则 **EFI_OS_INDICATIONS_BOOT_TO_FW_UI** 位可以由操作系统在 *OsIndications* 变量中设置。一旦固件使用 *OsIndications* 变量中的该位并在固件用户界面停止，固件应从 *OsIndications* 变量中清除该位，以便向操作系统确认信息已被使用，更重要的是，防止固件用户在后续启动时再次显示界面。

如果固件支持基于时间戳的撤销和“dbt”授权时间戳数据库变量，则固件可以在 *OSIndicationsSupported* 变量中设置 **EFI_OS_INDICATIONS_TIMESTAMP_REVOCATION** 位。

如果平台支持处理 **Section 23.2** 中定义的固件管理协议更新胶囊，则 **EFI_OS_INDICATIONS_FMP_CAPSULE_SUPPORTED** 位在 *OsIndicationsSupported* 变量中设置。如果在 *OsIndications* 变量中设置，则 **EFI_OS_INDICATIONS_FMP_CAPSULE_SUPPORTED** 位没有任何功能，并在下次重新启动时被清除。

如果平台支持根据 **Section 8.5.5** 处理文件胶囊, 则设置 *OsIndicationsSupported* 变量中的 `EFI_OS_INDICATIONS_FILE_CAPSULE_SUPPORTED` 位。

当通过 **Section 8.5.5** 的大容量存储设备方法提交胶囊时, *OsIndications* 变量中的 `EFI_OS_INDICATIONS_FILE_CAPSULE_DELIVERY_SUPPORTED` 位必须由提交者设置, 以在下次重启时触发提交的胶囊处理。在重启后的处理过程中, 系统固件会在所有情况下从 *OsIndications* 中清除该位。

`EFI_OS_INDICATIONS_CAPSULE_RESULT_VAR_SUPPORTED` 位在 *OsIndicationsSupported* 变量中设置, 如果平台支持通过创建 **Section 8.5.6** 中定义的结果变量来报告延迟胶囊处理。如果在 *OsIndications* 中设置, 则此位无效。

如果平台既支持操作系统指示重新启动时应开始操作系统定义的恢复的能力, 也支持短格式文件路径媒体设备路径, 则在 *OsIndicationsSupported* 变量中设置 `EFI_OS_INDICATIONS_START_OS_RECOVERY` 位 (参见 **Section 3.1.2**)。如果在 *OsIndications* 中设置了该位, 则平台固件必须在引导期间绕过 *BootOrder* 变量的处理, 并直接跳到操作系统定义的恢复 (参见 **Section 3.4.1**), 然后是平台定义的恢复 (参见 **Section 3.4.2**)。系统固件在启动操作系统定义的恢复时必须清除 *OsIndications* 中的该位。

如果平台既支持操作系统指示重新启动时应开始平台定义的恢复的能力, 也支持短格式文件路径媒体设备路径, 则在 *OsIndicationsSupported* 变量中设置 `EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY` 位 (参见 **Section 3.1.2**)。如果在 *OsIndications* 中设置了该位, 则平台固件必须在引导期间绕过 *BootOrder* 变量的处理, 并直接跳到平台定义的恢复 (参见 **Section 3.4.2**)。系统固件在启动平台定义的恢复时必须清除 *OsIndications* 中的该位。

在所有情况下, 如果在 *OsIndicationsSupported* 中设置了 `EFI_OS_INDICATIONS_START_OS_RECOVERY` 或 `EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY` 之一, 则两者都必须设置并受支持。

`EFI_OS_INDICATIONS_JSON_CONFIG_DATA_REFRESH` 位由提交者在 *OsIndications* 变量中设置, 以触发收集当前配置并在下次启动时将刷新的数据报告给 EFI 系统配置表。如果未设置, 平台将不会收集当前配置, 而是将缓存的配置数据报告给 EFI 系统配置表。配置数据应使用 **Section 23.5.2** 中定义的 `EFI_JSON_CAPSULE_CONFIG_DATA` 格式安装到 EFI 系统配置表。一旦报告了刷新的数据, 该位将被系统固件从 *OsIndications* 中清除。

如果在 *OsIndicationsSupported* 变量中设置, 则 `EFI_OS_INDICATIONS_JSON_CONFIG_DATA_REFRESH` 位没有任何功能, 并在下次重新启动时被清除。

- 相关定义 (Related Definitions)

1 #define	<code>EFI_OS_INDICATIONS_BOOT_TO_FW_UI</code>	0x00000000000000000001
2 #define	<code>EFI_OS_INDICATIONS_TIMESTAMP_REVOCATION</code>	0x00000000000000000002
3 #define	<code>EFI_OS_INDICATIONS_FILE_CAPSULE_DELIVERY_SUPPORTED</code>	0x00000000000000000004
4 #define	<code>EFI_OS_INDICATIONS_FMP_CAPSULE_SUPPORTED</code>	0x00000000000000000008
5 #define	<code>EFI_OS_INDICATIONS_CAPSULE_RESULT_VAR_SUPPORTED</code>	0x00000000000000000010
6 #define	<code>EFI_OS_INDICATIONS_START_OS_RECOVERY</code>	0x00000000000000000020
7 #define	<code>EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY</code>	0x00000000000000000040
8 #define	<code>EFI_OS_INDICATIONS_JSON_CONFIG_DATA_REFRESH</code>	0x00000000000000000080

7.5.3.3 通过大容量存储设备上的文件交付胶囊 (Delivery of Capsules via file on Mass Storage device)

作为 `UpdateCapsule()` 运行时 API 的替代方案，平台支持的任何类型的胶囊也可以通过目标为引导的大容量存储设备上的 EFI 系统分区内的文件传送到固件。使用此方法暂存的胶囊将在下次系统重新启动时处理。此方法仅在从使用 GPT 格式化（参见 **Section 5.3**）并且在设备映像中包含 EFI 系统分区的大容量存储设备引导时可用。当 `OsIndications` 中的 `EFI_OS_INDICATIONS_FILE_CAPSULE_DELIVERY_SUPPORTED` 位设置时，系统固件将搜索胶囊，如 **Section 8.5.4** 所述。

活动 EFI 系统分区中的目录 `\EFI\UpdateCapsule`（忽略字母大小写）被定义用于将封装体传送到固件。

大容量存储设备上的胶囊文件的二进制结构与通过 EFI RunTime API 传送的胶囊内容相同，只是不支持使用 `EFI_CAPSULE_BLOCK_DESCRIPTOR` 进行分段，并且单个胶囊必须存储在以 `EFI_CAPSULE_HEADER` 开头的文件中的连续字节中。文件的大小必须等于 `EFI_CAPSULE_HEADER.CapsuleImageSize` 否则将生成错误并忽略胶囊。在一个文件中只能提交带有单个 `EFI_CAPSULE_HEADER` 的单个胶囊，但在单次重启期间可以提交多个文件，每个文件都包含一个胶囊。

压缩包的文件名应由提交者使用适合 EFI 系统分区文件系统的 8 位 ASCII 字符选择（第 13.3.1 节）。检查和处理放置在该目录中的文件后，固件将（如果可能）删除该文件。删除在成功处理的情况下执行，也在错误的情况下执行，但未能成功删除本身不是可报告的错误。

指定目录中可以存储多个胶囊文件，每个胶囊文件包含一个胶囊图像。在多个文件的情况下，系统固件将使用基于文件名字符的 CHAR16 数值排序的字母顺序处理文件，从左到右进行比较。小写字母字符将在比较前转换为大写字母。比较不等长的文件名时，应使用空格字符来填充较短的文件名。如果文件名包含一个或多个句点字符（.），则文件名中最右边的句点和最右边的句点右侧的文本将在比较前被删除。如果文件名在排除最右边的句点之后的任何文本后具有相同的文本，则处理顺序将通过在文件名字符串中最右边的句点右侧找到的任何文本的排序来确定。

如果胶囊处理因错误而终止，则将正常处理任何剩余的附加胶囊文件。

目录 `\EFI\UpdateCapsule` 仅在活动引导选项中指定的设备上的 EFI 系统分区检查胶囊，通过引用 `BootNext` 变量或 `BootOrder` 变量处理确定。活动引导变量是具有最高优先级 `BootNext` 或 `BootOrder` 中的变量，它指的是发现存在的设备。`BootOrder` 中但引用不存在的设备的引导变量在确定活动引导变量时将被忽略。

要检查 `\EFI\UpdateCapsule` 的设备通过引用所选活动 `Boot####` 变量中的 `FilePathList` 字段来标识。系统固件不需要检查不包含最高引导优先级的引导目标的大容量存储设备，也不需要检查第二个 EFI 系统分区而不是活动引导变量的目标。

在所有情况下，一个 `capsule` 被识别用于处理，系统在 `capsule` 处理完成后重新启动。如果设置了 `BootNext` 变量，则在执行胶囊处理时清除该变量，而无需指示变量的实际引导。

7.5.3.4 UEFI 变量报告成功或重启后处理胶囊时遇到的任何错误 (UEFI variable reporting on the Success or any Errors encountered in processing of capsules after restart)

如果 *capsule* 的处理是（1）通过调用 `UpdateCapsule()` API 传递，但延迟到下次重新启动，或者（2）当胶囊通过大容量存储设备传递时，固件会创建 UEFI 变量，以向 *capsule* 提供商指示胶囊处理的状态。

在调用 `UpdateCapsule()` 时传递多个胶囊的情况下，或如 **Section 8.5.5** 所述的磁盘上的多个文件，或如 **Section 23.2** 所述当胶囊包含多个有效负载时，将为处理的每个胶囊有效载荷创建一个单独的结果变量。当计算出的变量名称已经存在时，固件将覆盖结果变量。然而，为了避免不必要的消耗系统变量存储，结果变量应该在检查结果状态后由胶囊提供者删除。

当整个胶囊处理发生在对 `UpdateCapsule()` 函数的调用中时，将不会使用 UEFI 变量报告。

报告变量属性将为 `EFI_VARIABLE_NON_VOLATILE + EFI_VARIABLE_BOOTSERVICE_ACCESS + EFI_VARIABLE_RUNTIME_ACCESS`。

报告变量的供应商 GUID 将为 `EFI_CAPSULE_REPORT_GUID`。报告变量的名称将是 *CapsuleNNNN*，其中 NNNN 是由固件选择的 4 位十六进制数。NNNN 的值将由固件从 *Capsule0000* 开始递增，一直持续到平台定义的最大值。

平台将在名为 `EFI_CAPSULE_REPORT_GUID:CapsuleMax` 的只读变量中发布平台最大值。*CapsuleMax* 的内容将是字符串“*CapsuleNNNN*”，其中 NNNN 是平台在滚动到 *Capsule0000* 之前使用的最高值。该平台还将发布在 `EFI_CAPSULE_REPORT_GUID:CapsuleLast` 中创建的最后一个变量的名称。

创建新的结果变量时，将覆盖任何先前具有相同名称的变量。在变量存储有限的情况下，系统固件可以选择删除最旧的报告变量以创建可用空间。如果无法释放足够的变量空间，则不会创建变量。

Variable Name	Attributes	Internal Format
<code>Capsule0000, Capsule0001, ... up to max</code>	NV, BS, RT	<code>EFI_CAPSULE_RESULT_VARIABLE</code>
<code>CapsuleMax</code>	BS, RT, Read-Only	<code>CHAR16[11] (no zero terminator)</code>
<code>CapsuleLast</code>	NV, BS, RT, Read-Only	<code>CHAR16[11] (no zero terminator)</code>

图 92. Table 8-9 Variables Using `EFI_CAPSULE_REPORT_GUID`

EFI_CAPSULE_REPORT_GUID

```

1 // {39B68C46-F7FB-441B-B6EC-16B0F69821F3}
2 #define EFI_CAPSULE_REPORT_GUID \
3 { 0x39b68c46, 0xf7fb, 0x441b, \
4 {0xb6, 0xec, 0x16, 0xb0, 0xf6, 0x98, 0x21, 0xf3 }};

```

胶囊处理结果变量的结构 (Structure of the Capsule Processing Result Variable)

胶囊处理结果变量的内容总是以 `EFI_CAPSULE_RESULT_VARIABLE_HEADER` 结构开头。`CapsuleGuid` 的值决定了结果变量内容实例中可能跟随的任何附加数据。对于 `CapsuleGuid` 的某些值，可能无法定义其他数据。

如下所述，**VariableTotalSize** 是完整结果变量的大小，包括整个标头和特定 `CapsuleGuid` 类型所需的任何其他数据。

```

1 typedef struct {
2     UINT32      VariableTotalSize;
3     UINT32      Reserved; //for alignment
4     EFI_GUID    CapsuleGuid;
5     EFI_TIME   CapsuleProcessed;
6     EFI_STATUS CapsuleStatus
7 } EFI_CAPSULE_RESULT_VARIABLE_HEADER;

```

`VariableTotalSize`: 变量的大小（以字节为单位），包括 `CapsuleGuid` 指定的标头之外的任何数据。

`CapsuleGuid`: 来自 `EFI_CAPSULE_HEADER` 的引导。

`CapsuleProcessed`: 处理完成时使用系统时间的 `TimeStamp`。

`CapsuleStatus`: 胶囊处理的结果。任何错误代码的准确解释可能取决于处理的胶囊类型。

`CapsuleGuid` 为 `EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID` 时的附加结构 (**Additional Structure When CapsuleGuid is EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID**)

胶囊处理结果变量的内容总是以 `EFI_CAPSULE_RESULT_VARIABLE_HEADER` 开头。当 `CapsuleGuid` 是 `EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID` 时，标头后面是 `EFI_CAPSULE_RESULT_VARIABLE_FMP` 定义的附加数据。

```

1 typedef struct {
2     UINT16      Version;
3     UINT8       PayloadIndex;
4     UINT8       UpdateImageIndex;
5     EFI_GUID    UpdateImageTypeId
6     // CHAR16 CapsuleFileName[];
7     // CHAR16 CapsuleTarget[];
8 } EFI_CAPSULE_RESULT_VARIABLE_FMP;

```

`Version`: 此结构的版本，当前为 `0x00000001`。

`PayloadIndex`: FMP 胶囊中有效载荷的索引，从零开始，经过处理以生成此报告。

`UpdateImageIndex`: 来自 `EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER` 的 `UpdateImageIndex` (从 `UINT8` 无符号转换为 `UINT16` 后)。

`UpdateImageTypeId`: 来自 `EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER` 的 `UpdateImageTypeId Guid`。

`CapsuleFileName`: 如果是从磁盘加载的胶囊，则以零结尾的数组包含已处理的胶囊文件名。如果胶囊直接提交给 `UpdateCapsule()`，则没有文件名，并且此字段需要包含一个 16 位零字符，该字符包含在 `VariableTotalSize`

中。

CapsuleTarget: 该字段将包含一个以零结尾的 CHAR16 字符串，其中包含设备发布固件管理协议（如果存在）的设备路径的文本表示。如果设备路径不存在并且目标不为固件所知，或者有效载荷被策略阻止或跳过，则此字段需要包含一个 16 位零字符，该字符包含在 **VariableTotalSize** 中。

CapsuleGuid 为 EFI_JSON_CAPSULE_ID_GUID 时的附加结构（Additional Structure When CapsuleGuid is EFI_JSON_CAPSULE_ID_GUID）

胶囊处理结果变量的内容总是以 [EFI_CAPSULE_RESULT_VARIABLE_HEADER](#) 开头。当 CapsuleGuid 为 [EFI_JSON_CAPSULE_ID](#) 时，标头后跟 [EFI_CAPSULE_RESULT_VARIABLE_JSON](#) 定义的附加数据。

```
1 typedef struct {
2     UINT32 Version;
3     UINT32 CapsuleId;
4     UINT32 RespLength;
5     UINT8 Resp[];
6 } EFI\_CAPSULE\_RESULT\_VARIABLE\_JSON;
```

Version: 此结构的版本，当前为 0x00000001。

CapsuleId: 处理结果记录在该变量中的胶囊的唯一标识符。0x00000000 – 0xFFFFFFFF – 实现保留，0xF0000000 – 0xFFFFFFFF – 规范保留，[#define REDFISH_DEFINED_JSON_SCHEMA 0xF000000](#)，JSON 负载应符合 Redfish-defined JSON 架构，请参阅 DMTF-Redfish 规范。

RespLength: **Resp** 的字节长度。

Resp: 可变长度缓冲区，其中包含向将 JSON 胶囊传送到系统的调用方回复的 JSON 负载。回复有效负载中使用的 JSON 模式的定义超出了本规范的范围。

CapsuleStatus 中返回的状态码（Status Codes Returned in CapsuleStatus）

EFI_SUCCESS	Valid capsule was passed and the capsule has been successfully processed by the firmware.
EFI_INVALID_PARAMETER	Invalid capsule size, or an incompatible set of flags were set in the capsule header. In the case of a capsule file, the file size was not valid or an error was detected in the internal structure of the file.
EFI_DEVICE_ERROR	The capsule update was started, but failed due to a device error.
EFI_ACCESS_DENIED	Image within capsule was not loaded because the platform policy prohibits the image from being loaded.
EFI_LOAD_ERROR	For capsule with included driver, no driver with correct format for the platform was found.
EFI_UNSUPPORTED	The capsule type is not supported on this platform. Or the capsule internal structures were not recognized as valid by the platform.
EFI_OUT_OF_RESOURCES	There were insufficient resources to process the capsule.
EFI_NOT_READY	Capsule payload blocked by platform policy.
EFI_ABORTED	Capsule payload was skipped.
EFI_UNSUPPORTED	This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.

图 93. table8-9_1