第1章 ACPI简述

由英特尔（Intel）等业界厂商联合制订的高级配置与电源接口（ACPI）规范[1]被用来建立业界通用的接口，使得健壮的操作系统（OS）能对主板上设备的配置以及包含设备在内的整个系统的电源管理进行控制。在操作系统主导的电源管理（OSPM）功能中，ACPI是一个非常重要的组件。

ACPI将之前业界存在的电源管理BIOS代码、先进电源管理（APM）应用编程接口、多处理器规范（MPS）等进行了整合。与此同时，ACPI也提供了相应的方法，可将已存在的遗留硬件有序地转换成ACPI硬件。ACPI允许ACPI和遗留机制同时存在于单台机器中，由OS负责确定到底使用它们中的哪一个。

更进一步来说，在ACPI规范起草期间修建的系统架构摆脱了历史上即插即用接口的限制。ACPI改进了之前存在的主板配置接口，通过使用一种更加健壮、更加有效的方式来支持系统的高级架构。

在此规范中定义的接口和OSPM概念适用于所有计算机类型，包括但不限于桌面、移动平台、工作站和服务器。从电源管理的角度来说，通过将不使用的设备转换到低电量状态，包括尽可能地将整个系统放入低电量状态（睡眠状态），OSPM/ACPI发展了系统应该节约电能的这种理念。

本书描述了ACPI硬件接口、ACPI软件接口和ACPI数据结构。当平台实现了这些功能时，就可以实现对OSPM的支持。

1.1 首要目标

在实现OSPM时，ACPI是一个关键的组件。期望业界厂商能广泛地采用ACPI定义的接口，使得软、硬件供应商更加愿意开发与ACPI以及OSPM兼容的产品。

ACPI和OSPM的首要目标如下。

（1）对成本和功能进行适当的权衡，让所有计算机系统实现主板配置和电源管理功能。

- 计算机系统包括但不限于桌面、移动平台、工作站和服务器。
- 机器设计者可以自由地实现各种解决方案，从简单的台式机到非常庞大、复杂的服务器，总能维护完整的OS支持。
- 广泛地使用电源管理，会使它更加实用，更能吸引应用程序来支持和利用它。

（2）增强电源管理的功能和健壮性。

- 因复杂性而无法在BIOS中实现的电源管理策略，完全可以在OS中实现。这样，通过廉价的电源管理硬件，就可以实现非常详尽的电源管理策略。
- 通过将从用户、应用程序和硬件收集的电源管理信息集中放在OS里，可以更好地确定电源管理策略和执行电源管理。
- OS中电源管理算法的一致性将减少固件和OS之间的冲突并且提升可靠性。

（3）促进和加快业界范围内的电源管理实现。

- OSPM和ACPI减少了业界在电源管理方面的冗余投入。通过将这些投入和功能聚集到OS中，可使业界参与者聚焦于自身在创新方面的努力和投入，而不是简单地实现等价/对等功能。
- OS可以独立于硬件而发展。所有与ACPI兼容的机器都能获得OS在功能提升和技术创新方面所取得的益处。

（4）为配置主板设备创建健壮的接口。

- 通过已存在遗留接口不可能实现新的高级设计。

1.2 电源管理

将电源管理移动到OS中并在OS和硬件之间使用一个抽象的ACPI接口来达到上述四个首要目标是必要的。理由包括下面6个方面。

（1）对电源管理最小能力的支持限制了应用程序供应商来支持和使用它。

- 将电源管理功能移到OS中，使得安装了OS的每台机器都能使用电源管理功能。虽然在不同的机器上实现了不同的功能，但在所有OSPM机器上，用户和应用程序看到的都是相同的电源接口和语义。
- 将使应用程序供应商致力于把电源管理功能增加到他们的产品中。

（2）遗留电源管理算法受BIOS可用信息的约束，这限制了它可以实现的功能。

- OS中汇集的电源管理信息加上从用户、应用和硬件获取的信息，可以实现更强大的功能。
- 一些装置的功能要求使用全局一致的电源策略。

（3）为了处理电源管理，BIOS代码已变得非常庞杂。让BIOS与OS一起协调工作非常困难，而且它们的功能被限制为仅能对硬件进行静态配置。

- BIOS仅需保留和管理少量的状态信息（因为OS将管理它们）。
- OS中电源管理算法是统一的，使得OS和硬件之间能更好地进行整合。
- 因为可以加载额外的ACPI表（定义块），OS可以处理动态的机器配置。
- BIOS可以实现更少、更简单的功能。这使得实现和支持BIOS更容易。

（4）PC平台中已存在的结构束缚了OS和硬件设计。

（5）ACPI是抽象接口。OS可以独立于硬件向前发展。同样地，硬件也可以独立于OS向前发展。

（6）可以在不同操作系统和处理器之间方便地移植ACPI。通过ACPI控制方法，可以更加灵活地实现特定特征。

1.3 遗留支持

为了有序地将遗留硬件转换到ACPI硬件，ACPI提供了相应的支持，允许两种机制同时存在于单个机器中并在需要时使用其中之一。硬件类型和OS类型之间的交互关系如表1-1所示。

表1-1 硬件类型和OS类型交互关系

硬件	遗留OS	具有OSPM的ACPI OS
遗留硬件	运行在遗留硬件上的遗留OS继续做之前所做的工作	如果OS缺少遗留支持，那么遗留支持完全包含在硬件功能中
同时支持遗留硬件和ACPI硬件	遗留OS就像在遗留硬件上工作	在启动期间，OS让硬件从遗留模式切换到OSPM/ACPI模式。切换后，系统有完整的OSPM/ACPI支持
仅支持ACPI硬件	不存在电源管理	有完整的OSPM/ACPI支持

1.4 OEM实现策略

任何OEM（Original Equipment Manufacturer）都可以自由地修建他们觉得合适的硬件。考虑到ACPI规范的存在，通常有如下两种可行的实现策略。

（1）OEM可以采用OS供应商提供的ACPI OSPM软件，并且采用多种可能方式之一为一个给定的平台实现符合ACPI规范的硬件部分。

（2）OEM可以开发一个驱动以及与ACPI不兼容的硬件。此策略可能会产生更多的硬件实现。然而，当实现与OSPM兼容而与ACPI不兼容硬件时，OEM将承担开发、测试和单独为他们的硬件实现分发驱动所产生的成本。

1.5 电源和睡眠按钮

在遗留系统中，用户通常可以使用电源按钮强制将机器关闭。在笔记本电脑上，也可以强制将机器转换到某个睡眠状态。在此情况下，不需要考虑用户策略（例如同用户将机器关闭时机器所执行的操作一样，用户希望机器能在1秒以内恢复所有上下文）、系统警告功能（例如系统被用作应答机或传真机）或应用功能（例如存储一个用户文件）。

在一个OSPM系统中存在两个开关。一个开关用来将系统转换到关机状态，为合理理由提供一个关闭电源的机制；另一个开关是一个“软”按钮，它在某个显而易见的位置（例如，位于笔记本电脑键盘的旁边）。与遗留系统中电源按钮不同，它不能直接将机器关闭，而是向OS发送信号来将机器置于关机或睡眠状态。系统依据此请求所完成的工作依赖

于从用户优选项获取的策略、用户功能请求和应用数据。ACPI定义了两类这样的“软”按钮：一个被用于将机器置于睡眠状态；另一个被用于将机器置于软关机状态。

这使得OEM可以按照两种不同的方式来实现机器：一个按钮模式和两个按钮模式。一个按钮模式只有单个按钮，根据用户的设置来确定此按钮是被用作电源按钮还是睡眠按钮。两个按钮模式有一个便于访问的睡眠按钮和一个独立的电源按钮。在每种模式中，都需要一个复写特征来处理各种少见的情况。此特征无须OSPM干涉就可强制将机器转换到关机状态。

1.6 ACPI规范与结构

ACPI规范定义了ACPI硬件接口、ACPI软件接口、ACPI数据结构以及这些接口的语义。

图1-1展示了与OSPM/ACPI相关的软、硬件组件以及它们彼此之间的关联。ACPI规范（图1-1中虚线框包围的部分）描述了组件之间的接口、ACPI系统描述表的内容和其他ACPI组件的相关语义。注意，在实现ACPI时，描述特定平台硬件的ACPI系统描述表处于中心位置。ACPI系统固件的主要功能之一就是提供这些ACPI系统描述表。

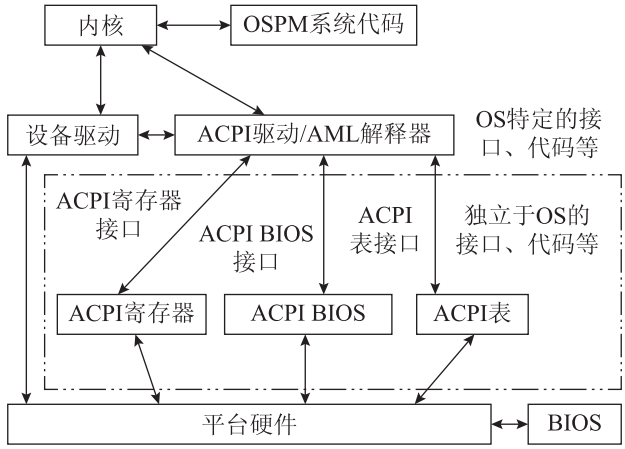

图1-1 OSPM/ACPI全局系统

尽管ACPI访问了软、硬件并且阐述了它们必须如何工作，但是ACPI既不是一个软件规范，也不是一个硬件规范，而是一个由软、硬件元素组成的接口规范。

ACPI包含三个运行组件。

- ACPI系统描述表——描述硬件的接口。表的格式限制了在表中可以创建的内容（例如，一些控制内嵌在固定寄存器块中，而此表指定了寄存器块所在的地址）。然而，大多数系统描述表允许按照任何方式来创建硬件并且可以描述硬件工作时所需的任意操作序列。包含定义块的ACPI表可以使用伪代码类型的语句。OS负责对这些代码进行解释和执行。也就是说，OSPM包含和使用了一个解释器，它执行采用伪代码语言编码并保存在ACPI表中的程序。被称为AML（ACPI Machine Language）的伪代码语言是一个紧凑、符号化、抽象类型的机器语言。
- ACPI寄存器——硬件接口中受限制的部分，通过ACPI系统描述表进行描述（至少描述ACPI寄存器的位置）。

- ACPI系统固件——此代码启动机器（与遗留BIOS所实现的功能相同）以及实现睡眠、唤醒和重启等操作所需的接口。与遗留BIOS相比，它很少被调用。ACPI系统固件也提供了ACPI系统描述表。

1.7 OS、平台兼容性

ACPI规范仅包含接口规范，不包含任何平台兼容性要求。本节为实现特定类型的平台提供向导，同时也提供了ACPI兼容OS所需的最小特征。

1.7.1 平台实现ACPI定义的接口

系统平台通过平台硬件、ACPI定义的软件接口以及ACPI系统固件中的系统描述表来实现ACPI定义的硬件接口。当ACPI定义的特定接口和OSPM概念适用于一类机器（例如移动系统）时，它们可能并不适用于另一类机器（例如多领域的企业服务器）。指定所有平台类型和这些平台类型所需的合适接口（由ACPI定义）已经超出了ACPI规范的能力和范围。

鼓励平台设计者采用满足这样三条原则的适当接口和硬件需求：出现在特定设计向导中；对特定系统平台合适；由ACPI定义。如果ACPI规范中定义的接口已经提供了相似的功能，那么平台设计者不应该定义替换的接口。

在设计向导中描述平台类型所需的由ACPI定义的特性、概念和接口时，应该使用通用的描述文本和类型名字。

1.7.2 OSPM实现

为了支持ACPI定义的特性、概念和接口以及OS执行时适用于系统平台类型的关联事件模型，需要对OS进行改进，这就是OSPM的实现。为了能通过OSPM来支持ACPI，需要修改OS来实现如下功能。

- 使用系统地址映射报告接口。
- 查找和使用ACPI系统描述表。
- 解释ACPI机器语言（AML）。
- 枚举和配置在ACPI名字空间中描述的主板设备。
- 与电源管理定时器进行配合。
- 与实时时钟唤醒警告进行配合。
- 进入ACPI模式（在遗留硬件系统中）。
- 实现设备电源管理策略。
- 实现电源资源管理。
- 在调度程序的空载处理例程中实现处理器电源状态。
- 控制处理器和设备的性能状态。
- 实现ACPI散热模型。
- 支持ACPI事件编程模型，包括动态设备支持、处理SCI中断、管理固定事件、通用

目的事件以及嵌入式控制器三类中断。

- 支持全局锁的获取和释放。
- 使用重置寄存器来重启系统。
- 提供APIs来影响电源管理策略。
- 为ACPI定义的设备实现驱动支持。
- 实现支持系统指示符的APIs。
- 支持所有系统状态S1～S5。

OSPM的使命是最优地配置平台和最优地管理系统的电源、性能和散热状态。为了达到这些目标，ACPI要求一旦遵从ACPI的平台处于ACPI模式，平台的硬件、固件或其他非OS软件不能独立于OSPM之外独自处理平台的配置、电源、性能和散热控制接口。OSPM独自负责协调系统的配置、电源管理、性能管理和散热控制策略。独立于OSPM之外处理这些接口会削弱OSPM/ACPI的用途，反过来可能会影响系统的配置、电源、性能和散热策略目标。对此要求有两个例外。第一个例外是当存在一个ACPI兼容的OS并且OSPM已经无法补救一个不利的散热条件时，为了避免过热条件对系统造成的可能破坏，平台可以使用一个安全的散热控制机制，降低系统组件的性能来避免对系统的破坏。发生此情况时，如果会一直持续地降低性能，那么平台必须将性能的降低通知给OSPM。第二个例外是平台包含了主动冷却设备却不包含被动冷却温度触发点或控制。此种情况下，在不影响OSPM目标的前提下，可以实现基于硬件的主动冷却机制。任何需要主动、被动冷却的平台要允许OSPM通过ACPI定义的主动和被动冷却接口来管理平台的散热。

1.7.3 OS要求

一个OSPM/ACPI兼容OS所需的最小功能集如下。

（1）在IA（Intel Architecture）平台中使用系统地址映射报告接口来获取系统地址映射。

- INT 15H，E820H——查询系统地址映射接口。
- EFI GetMemoryMap()启动服务功能。

（2）查找和利用ACPI系统描述表。

（3）实现支持所有已定义AML语法元素的AML解释器。

（4）支持ACPI事件编程模型，包括处理SCI中断、管理固定事件中断、通用目的事件中断以及嵌入式控制器中断和动态设备支持。

（5）枚举和配置在ACPI名字空间中描述的主板设备。

（6）实现对ACPI规范中定义的如下ACPI设备的支持。

- 嵌入式控制器设备。
- GPE块设备。
- 模块设备。

（7）实现ACPI散热模型。

（8）支持全局锁的获取和释放。

（9）OS主导的电源管理支持（设备驱动负责维护设备的上下文）。

第2章 ACPI基本术语及概念

为了能更好地理解ACPI规范，首先需要了解ACPI规范中涉及到的各种术语以及后续章节将会频繁使用的概念。

2.1 通用ACPI术语

Advanced Configuration and Power Interface (ACPI)——高级配置和电源接口

ACPI是一种方法，采用抽象、具体的术语描述了硬件接口来实现灵活、新颖的硬件和可扩展的OS代码。

ACPI Hardware——ACPI硬件

拥有OSPM所需的特征并且通过使用系统描述表来描述这些特征接口的计算机硬件。

ACPI Namespace——ACPI名字空间

这个分层的树形结构位于OS控制的内存中。在ACPI名字空间中包含了命名对象。这些对象可以是数据对象、控制方法对象、总线/设备对象数组等。ACPI系统描述表位于ACPI BIOS中。OS通过在运行时加载/卸载这些表中的定义块，可以动态地改变ACPI名字空间的内容。在ACPI名字空间中的所有信息来自于DSDT。DSDT包含了差异定义块和一个或多个其他定义块。

ACPI Machine Language (AML)——ACPI机器语言

此伪代码针对ACPI兼容OS所支持的虚拟机。采用它可以编写ACPI控制方法和对象。

Advanced Programmable Interrupt Controller (APIC)——高级可编程中断控制器

通常会在IA-32 PC系统中使用的一种中断控制器架构。APIC架构支持多处理器中断管理（通过所有处理器之间的对称中断分发）、多个I/O子系统、8259A兼容性和处理器间中断。此架构由直接连接到处理器的本地ACPI和集成在芯片中的I/O APIC共同组成。

ACPI Source Language (ASL)——ACPI描述语言

与AML等价的编程语言。ASL被编译成AML映像。

Control Method——控制方法

一个控制方法是对OS如何执行一个简单硬件任务进行的定义。例如，OS调用控制方法用来读取散热区的温度。控制方法采用AML来编写。ACPI兼容OS可以解释并执行AML。ACPI兼容系统必须在ACPI表中提供最少的控制方法集。OS提供了明确定义的控制方法集，这样，开发者在定义他们自己的控制方法时可以参考这些控制方法集。

Device——设备

平台中核心芯片集之外的硬件组件。这样的设备包含显卡适配器、IDE CD-ROM、IDE硬盘控制器、COM端口等。在ACPI电源管理模式中，把总线也当作一个设备。

Device Context——设备上下文

设备保留的可变数据。当设备进入或离开某个状态时，这些数据通常会丢失。在此情况下，OS软件负责保存和恢复这些信息。设备上下文参考是保存在设备外设中的少量信息。

Differentiated System Description Table (DSDT)——差异系统描述表

OEM必须为ACPI兼容OS提供一个DSDT。DSDT包含差异定义块，提供有关基础系统的实现和配置信息。在系统启动时，OS总是将DSDT信息加载到ACPI名字空间中，并且不再卸载它。

Unified Extensible Firmware Interface (UEFI)——统一的可扩展固件接口

UEFI是OS和平台固件之间的一个接口。此接口采用数据表格式，包含了平台相关的信息、OS和加载器可用的启动和运行时服务调用。这些为启动OS提供了一个标准的环境。

Embedded Controller——嵌入式控制器

一种通用类型的微控制器，被用来支持OEM特定的实现。它主要用在移动平台中。只要微控制器遵守ACPI规范中描述的模型之一，就可以将嵌入式控制器应用到任何平台设计中。嵌入式控制器通过一个简单的接口执行复杂的低级功能。

Embedded Controller Interface——嵌入式控制器接口

OS驱动和嵌入式控制器之间的一个标准软、硬件通讯接口。有了这个接口，对于任何OS，只需提供一个可以直接和系统中的嵌入式控制器进行通讯的标准驱动，就可使系统中的其他驱动能与系统嵌入式控制器进行通讯并且使用嵌入式控制器的资源（例如智能电池和AML代码）。这使得OEM可以提供OS和应用程序能使用的平台特征。

Firmware ACPI Control Structure (FACS)——固件ACPI控制结构

在可读、可写内存中的一个数据结构。BIOS使用此结构来实现固件和OS之间的握手。FACS通过FADT传递给ACPI兼容OS。FACS包含了上次启动时系统的硬件签名、固件唤醒向量和全局锁。

Fixed ACPI Description Table (FADT) ——固件ACPI描述表

包含ACPI硬件寄存器块实现和配置细节的一个表，使得OS可以直接管理ACPI硬件寄存器块和DSDT的物理地址。OEM必须在RSDT/XSDT中为ACPI兼容OS提供一个FADT。

Fixed Features——固定特征

由ACPI接口提供的特征集。ACPI规范对硬件编程模型产生的环境和方式施加了限制。不管使用了哪个固定特征，都要按照ACPI规范中所描述的方法来实现，以便OSPM能直接

访问这些固定特征寄存器。

Fixed Feature Events——固定特征事件

当固定特征寄存器中的一对状态和使能位都被设置时，在ACPI接口上发生的事件集合。当一个固定特征事件发生时，会产生一个系统控制中断（SCI）。针对ACPI固定特征事件，OSPM或者一个ACPI驱动会成为事件的处理例程。

Fixed Feature Registers——固定特征寄存器

系统I/O地址空间中特定地址位置上的固定特征寄存器里的硬件寄存器集合。ACPI为固定特征定义了寄存器块（针对每个寄存器块，都能从FADT中获取一个独立的指针）。

General–Purpose Event Registers——通用目的事件寄存器

通用目的事件寄存器包含了针对通用特征的事件编程模型。所有通用目的事件都会产生SCI。

Generic Feature——通用特征

平台的通用特征是通过控制方法和通用目的事件实现的增值硬件。

Global System Status——全局系统状态

全局系统状态应用到整个系统，对用户来说是可见的状态。在ACPI规范中，各种全局系统状态被分别标记为G0～G3。

Ignored Bits——忽略位

在ACPI规范中，ACPI硬件寄存器里一些未使用的位被定义为忽略位。忽略位未被定义，可以返回0或1（保留位总是返回0）。在读操作中，软件忽略ACPI硬件寄存器中的忽略位；在写操作中，软件保留ACPI硬件寄存器中的忽略位。

Input/Output Advanced Programmable Interrupt Controller (I/O APIC)——高级输入/输出可编程中断控制器

I/O APIC将从设备接收的中断路由到处理器的本地APIC。

Input/Output Streamlined Advanced Programmable Interrupt Controller (I/O SAPIC)——新型高级输入/输出可编程中断控制器

I/O SAPIC将从设备接收的中断路由到处理器的本地SAPIC。

Legacy——遗留

一种计算机状态。在此状态下，由平台硬件/固件进行电源管理策略决定。在当前系统中所发现的遗留电源管理特征被用来支持遗留OS系统中的电源管理。遗留OS不支持OS主导的电源管理架构。

Legacy Hardware——遗留硬件

不支持ACPI或OSPM电源管理功能的计算机系统。

Legacy OS——遗留OS

不能主导系统电源管理功能的OS。支持APM 1.x的操作系统包括在此类OS中。

Local APIC——本地APIC

本地APCI从I/O APIC接收中断。

Local SAPIC——本地SAPIC

本地SAPIC从I/O SAPIC接收中断。

Multiple APIC Description Table (MADT)——多种APIC描述表

在支持APIC和SAPIC的系统中，此表被用来描述APIC的实现。MADT表头之后的数据是一系列APIC/SAPIC结构，声明机器的APIC/SAPIC特性。

Object——对象

ACPI名字空间中的节点是由OS根据系统描述表中的信息添加到树形结构中的对象。这些对象可以是数据对象、控制方法对象、对象数组对象等。对象数组对象参考了其他对象。对象是有类型、大小和名字的。

Object Name——对象名

ACPI名字空间的一部分。在命名对象时需要遵守一系列规则。

Operating System–directed Power Management (OSPM)——操作系统主导的电源管理

一种电源（和系统）管理模型。在此模型中，OS起着核心作用并且使用全局信息来优化所执行任务的系统行为。

Package——对象数组

一组各种类型的对象。

Power Button——电源按钮

一个用户按钮或其他开关接触装置，可以将系统从睡眠/软关闭状态切换到工作状态，也可以向OS发送信号使其从工作状态转换到睡眠/软关闭状态。

Power Management——电源管理

通过软、硬件实现的一种机制，用来最小化系统的电量消耗、管理系统的散热限制以及最大化系统电池寿命。电源管理需要在系统性能、噪声、电池寿命、处理速度和交流电（AC）电量消耗之间进行折衷。

Power Resources——电源资源

设备操作在给定电源状态时所需的资源（例如电源板和时钟源）。

Power Sources——电源

电池（包含UPS电池）、交流电源适配器或者向平台提供电能的提供者。

Register Grouping——寄存器组

由两个寄存器块组成（它有两个指向不同寄存器块的指针）。在一个寄存器组中固定位置的位可以划分到两个寄存器块中。这允许一个寄存器组中的位被划分到两个芯片中。

Reserved Bits——保留位

在ACPI规范中，ACPI硬件寄存器里一些未使用的位被定义为保留位。为了将来的可扩展性，读取硬件寄存器里的保留位总是返回0b；对保留位进行写操作时，保留位不会受影响。OSPM必须将0b写到使能和状态寄存器里的所有保留位并且将控制寄存器里的保留位保持不变。

Root System Description Pointer (RSDP)——根级系统描述指针

一个ACPI兼容系统必须在系统的低端地址空间提供一个RSDP。此结构的唯一目的是提

供RSDT/XSDT的物理地址。

Root System Description Table (RSDT)——根级系统描述表

一个签名为“RSDT”的表。此表中包含了指向其他系统描述表的物理指针。OS通过RSDP结构中的指针来定位RSDT。

Secondary System Description Table (SSDT)——次级系统描述表

SSDT是DSDT的附加部分。SSDT可被用来作为平台描述的一部分。在DSDT被加载到ACPI名字空间后，RSDT/XSDT中具有唯一OEM Table ID字段值的每个二级描述表被加载。这允许OEM在一个表中提供基础支持，而在其他表中添加少量的系统选项。

Sleep Button——睡眠按钮

一个用户按钮，可以将系统从睡眠/软关闭状态切换到工作状态，也可以向OS发送信号使其从工作状态转换到睡眠状态。

Smart Battery Subsystem——智能电池子系统

符合如下规范的电池子系统：智能电池、智能电池系统管理者、智能电池充电器/转换器以及其他ACPI要求。

Smart Battery Table（SBST）——智能电池表

在具有智能电池子系统的平台中所使用的一个ACPI系统描述表。此表包含了电量级别的触发点。在此触发点，平台需要将系统置于不同的睡眠状态。另外，此表也包含了建议电量级别。在此电量下将警告用户并将平台转换到睡眠状态。

System Management Bus (SMBus)——系统管理总线

基于I2C协议的双线接口。SMBus是一个低速总线，为设备及总线仲裁提供正向寻址。

SMBus Interface——SMBus接口

在OS总线驱动和SMBus控制器之间的一个标准软、硬件通信接口。

Streamlined Advanced Programmable Interrupt Controller (SAPIC)——新型高级可编程中断控制器

基于Intel Itanium处理器家族的64位系统中经常使用的一个新型APIC。

System Context——系统上下文

系统中设备驱动不进行保存的易失数据。

System Control Interrupt (SCI)——系统控制中断

硬件使用此系统中断将ACPI事件通知给OS。SCI是一个低有效、电平类型的共享中断。

System Management Interrupt (SMI)——系统管理中断

在遗留系统中由中断事件产生的对OS透明的中断。相比之下，在ACPI系统中，中断事件会产生一个共享的、OS可见的中断（边沿类型的中断将不会工作）。如果硬件平台想要同时支持遗留操作系统和ACPI系统，那么在ACPI和遗留模式之间切换时，硬件平台必须重新映射SMI和SCI中断事件。

Thermal States——散热状态

散热状态表示系统散热区中不同的操作环境温度。一个系统可以有一个或多个散热区。散热区是某个温度传感器设备周围的空间范围，通过触发点来标记从一个散热状态到

另一个散热状态的转换。当散热区中的温度跨越触发点温度时，会产生一个SCI。

Extended Root System Description Table (XSDT)——扩展根级系统描述表

XSDT提供了与RSDT相同的功能，但XSDT中包含的描述表头部物理地址可以大于32位。注意，XSDT和RSDT的物理地址都包含在RSDP结构中。

2.2 各种状态定义

本节定义了ACPI规范中涉及到的各种状态信息，例如ACPI全局系统状态（工作、睡眠、软关机和机械关机）、ACPI设备电源状态（D0、D1、D2和D3）等。

2.2.1 全局系统状态定义

全局系统状态（Gx状态）应用到整个系统，是用户可以看见的状态。在定义全局系统状态时，主要依据如下六个准则。

1）应用软件在运行吗？

2）发生外部事件后，应用程序响应此事件的延时是多少？

3）电量消耗是多少？

4）返回到工作状态，需要重启OS吗？

5）拆卸计算机安全吗？

6）能通过电子装置进入和退出此状态吗？

定义的全局系统状态如表2-1所示。

表2-1 全局系统状态概要

全局系统状态	软件在运行吗	延时	电量消耗	需要重启OS吗	拆卸机器安全吗	电子装置能进入/退出此状态吗
G0工作状态	是	0	多	否	否	是
G1睡眠状态	否	>0	少	否	否	是
G2/S5软关机状态	否	长	接近于零	是	否	是
G3机械关机状态	否	长	RTC电池	是	是	否

- G3机械关机状态——通过机械方式进入或离开的一种计算机状态（例如关闭系统的电源）。此状态意味着主板电路上没有电流通过并且计算机上电工作时不会损坏硬件或伤害到使用者。为了返回到工作状态，必须重新启动OS。在此状态下，不会保留硬件的上下文。除了实时时钟外，电量消耗为零。
- G2/S5软关机状态——消耗最低电量的一种计算机状态。在此状态下，处理器不运行用户模式代码及系统模式代码，硬件不保存系统的上下文。要返回工作状态，必须重新启动系统并且需要很长的延时。在此状态下拆卸机器并不安全。
- G1睡眠状态——计算机消耗很少电量的一种计算机状态。在此状态下，处理器不执行用户模式代码，系统看上去就像关机一样（从用户的角度看，表现为显示器被关闭等）。返回到工作状态的延时需要根据进入此状态前选择的唤醒环境而变。不需要重启OS就可以恢复到工作状态，因为系统的上下文已经分别由软、硬件进行了保

存。在此状态下拆卸机器并不安全。

- G0工作状态——系统分发用户模式代码（应用程序）并使其得以执行的一种计算机状态。在此状态下，外部设备动态地改变自身的电源状态。用户可以通过用户接口程序选择系统的各种性能/电源特征，使得软件可以根据用户的选择来优化性能或者电池寿命。系统可以实时地对外部事件进行响应。在此状态下拆卸机器并不安全。

2.2.2 设备电源状态定义

设备电源状态是特定设备的状态。它们通常对用户并不可见。例如，尽管系统作为一个整体仍然处于工作状态，但是有一些设备可能已经处于关闭状态。

设备状态应用到任何总线上的任何设备。在定义设备电源状态时，通常会依据如下四个主要准则。

- 电量消耗——设备使用了多少电量。
- 设备上下文——硬件保存了多少设备的上下文。OS负责恢复任何未保存的设备上下文（通过重置设备来实现）。
- 设备驱动——设备驱动必须做些什么，才能使设备完全恢复到打开状态。
- 恢复时间——将设备完全恢复到打开状态需要多长时间。

设备电源状态（Dx状态）的定义如表2-2所示。

表2-2 设备电源状态概要

设备状态	电量消耗	保留设备上下文	驱动恢复
D0-打开	操作所需的电量	所有	否
D1	D0>D1>D2>$D3_{hot}$>D3	>D2	<D2
D2	D0>D1>D2>$D3_{hot}$>D3	<D1	>D1
$D3_{hot}$	D0>D1>D2>$D3_{hot}$>D3	可选的	否、重新加载和初始化
D3-关闭	0	没有	重新加载和初始化

- D3-关闭——设备不消耗任何电量。当进入此状态时，不会保存设备的上下文，因此，当重新为设备供电时，OS软件将重新初始化该设备。在此状态下，因为不会保存设备的上下文并且没有供应电源，所以设备不会解码地址线。相比其他设备电源状态来说，设备进行恢复时要花最长的时间。所有类型的设备都支持这个状态。
- $D3_{hot}$——$D3_{hot}$状态的含义由设备类自行定义。软件可以对$D3_{hot}$状态下的设备进行枚举。通常，期望$D3_{hot}$状态下的设备能节省更多的电量并可选地保存设备上下文。如果进入此状态时未保存设备上下文，那么在将设备状态转换到D0状态时，OS软件会重新初始化该设备。在此状态下，设备进行恢复时要花很长的时间。所有类型的设备都支持这种状态。
- D2——D2设备状态的含义由设备类自行定义。很多设备类并没有定义D2状态。通常期望设备在D2状态下能比在D1、D0状态下节省更多的电量并保存更少的设备上下文。D2状态下的总线会导致设备无法保存某些上下文（例如，通过减少总线上的电量，强迫设备关闭某些功能）。

- D1——D1设备状态的含义由设备类自行定义。很多设备类并没有定义D1状态。通常期望设备在D1状态下能比在D2状态下节省更少的电量并保存更多的设备上下文。
- D0-打开——此状态被认为是最高级别的电量消耗。设备能完全地操作和响应。期望设备能连续地记住所有相关的上下文。

许多设备并没有定义所有这四种电源状态。设备可以支持多个不同的低电量状态，但是如果这些状态之间并不存在用户可见的区别，那么将仅使用最低电量状态。

2.2.3 睡眠状态定义

睡眠状态（Sx状态）中的S1～S3是在全局睡眠状态G1下的睡眠状态类型。五种睡眠状态定义如下。

- S1睡眠状态——S1睡眠状态是低唤醒延时的睡眠状态。在此状态下，硬件维护所有的系统上下文。
- S2睡眠状态——S2睡眠状态是低唤醒延时的睡眠状态。除了未保存CPU和系统缓存上下文外，此状态类似于S1睡眠状态。唤醒事件发生后，控制从处理器的重置矢量处开始。
- S3睡眠状态——S3睡眠状态是低唤醒延时的睡眠状态。除了系统内存外，未保存其他所有系统上下文，如CPU、缓存和芯片集等。硬件维护着内存上下文并能恢复一些CPU和L2配置上下文。唤醒事件发生后，控制从处理器的重置矢量处开始。
- S4睡眠状态——S4睡眠状态是ACPI支持的最低电量消耗、最长唤醒延时的睡眠状态。为了将电量减少到最低程度，假设硬件平台关闭了所有设备，但仍维护着平台的上下文。
- S5软关机状态——除了OS不保存任何上下文外，S5状态类似于S4状态。当系统在软关机状态下被唤醒时，需要一个完整的启动过程。软件使用不同的状态值来区分S5状态和S4状态之间的差别。这样，BIOS的初始启动操作就可以确定启动是否需要从保存的内存映像中唤醒。

S4非易失性睡眠状态是一个特殊的全局系统状态。处于此状态时，保存和恢复系统上下文相对来说会慢些。如果系统接收到转换到S4指令，那么OS会将所有系统上下文保存到非易失存储（NVS）介质中的文件里并且记录下适当的上下文标记，然后将机器转换到S4状态。当系统离开软关机或者机械关机状态而转换到工作状态（G0）并重新启动OS时，就会恢复之前保存在NVS介质中的文件。只有在同时满足这三个条件时才会恢复NVS介质中的文件：发现了一个有效的非易失睡眠数据文件；机器配置的某些方面并没有改变；用户没有手动废弃此恢复。如果满足所有这些条件，作为OS重启的一部分，OS会重新加载系统上下文并且激活它，给用户的感觉就像是从睡眠状态（G1）中恢复一样（尽管要慢些）。不允许改变的机器配置包含但不限于磁盘布局和内存大小。注意，对直接从软关机或睡眠状态转换到S4的机器来说，必须由硬件负责将系统上下文保存到非易失存储介质中。从用户的角度来说，首先进入工作状态再由OS或BIOS负责存储系统上下文，将会花费太长的时间。如果用户不在现场，从机械关机状态转换到S4状态也是可能的，因为S4状态仅依赖非

易失存储介质，机器可以使用任意长的时间来保存其系统上下文。

在每个Sx状态下，有关系统行为的详细信息，请参考7.4.2节 “系统_Sx状态”。有关Sx状态之间转换的详细信息，请参考15.1节 “睡眠状态”。

2.2.4 处理器电源状态定义

处理器电源状态（Cx状态）是处理器在全局工作状态G0下的电量消耗和散热管理状态。为了在工作状态下进一步实现节省电量的目的，OS在空闲时会将CPU置于更低的电量状态（C1、C2和C3）。在这些Cx状态下，CPU不执行任何指令。当发生一个中断时，例如OS的调度定时器中断，CPU将会被唤醒。

空闲循环中的OS会通过读取ACPI电量管理接口来确定处在空闲循环中的时间。此定时器按照一个已知的、固定的频率运行并允许OS精确地获取空闲时间。依赖于这个空闲时间，OS会将CPU置于不同程度的低电量状态。

处理器电源状态定义如下。

- C0处理器电源状态——处理器在此状态时执行指令。
- C1处理器电源状态——此处理器电源状态有最短的延时。在此状态下的硬件延时必须足够短，使得操作软件在决定是否使用它时，不会考虑此状态的延时因素。除了将处理器置于不执行指令条件外，此状态没有其他软件可见的效果。
- C2处理器电源状态——处理器在C2状态比在C1状态节省了更多的电量。通过ACPI系统固件提供了比较长的硬件延时。操作软件可以使用此信息来确定什么时候应该使用C1状态来代替C2状态。除了将处理器置于不执行指令条件外，此状态没有其他软件可见的效果。
- C3处理器电源状态——处理器在C3状态比在C1、C2状态节省了更多的电量。通过ACPI系统固件提供了最坏的硬件延时。操作软件可以使用此信息来确定什么时候应该使用C2状态来代替C3状态。当处理器在C3状态时，处理器缓存维护着状态，但是忽略任何窥探。操作软件负责确保缓存的一致性。

关于每个Cx状态更详细的信息，请参考10.1节“处理器电源状态”。

2.2.5 设备和处理器性能状态定义

设备和处理器性能状态（Px状态）是设备或者处理器在执行状态下（处理器在C0状态，设备在D0状态）的电量消耗和能力状态。性能状态允许OSPM在性能和节省电量之间进行权衡。当设备和处理器性能状态调用了不同的设备和处理器功率级别时，相比对性能和节省电量进行线性伸缩，设备和处理器性能状态会有更大的影响。因为性能状态转换发生在设备执行状态下，所以必须加以小心，确保性能状态转换不会影响到整个系统。Px状态定义如下。

- P0性能状态——当设备或者处理器在此状态时，会达到最高的性能，消耗最大的电量。
- P1性能状态——在此状态下，设备或者处理器的性能能力被限制在最高能力之下，

设备或者处理器会消耗比最大电量少的电量。

- Pn性能状态——在此状态下，设备或者处理器的性能能力处于最低程度，消耗维持工作状态所需的最少电量。n是最大的性能状态数，其值依赖于处理器或设备。处理器和设备可以支持的性能状态不能超过16个。

2.3 电源状态

在OSPM中，OS会主导所有系统和设备的电源状态转换。OS利用用户优选项和应用程序使用设备的方式等先验知识，将设备转换到或者转换出低电量状态。不再使用的设备可以被关闭。相似地，OS利用从应用程序和用户优选项获取的信息将系统作为一个整体转换到或者转换出低电量状态。OS使用ACPI来控制硬件的电源状态转换。

以用户可见的角度，可以认为系统处于如图2-1所示的状态之一。

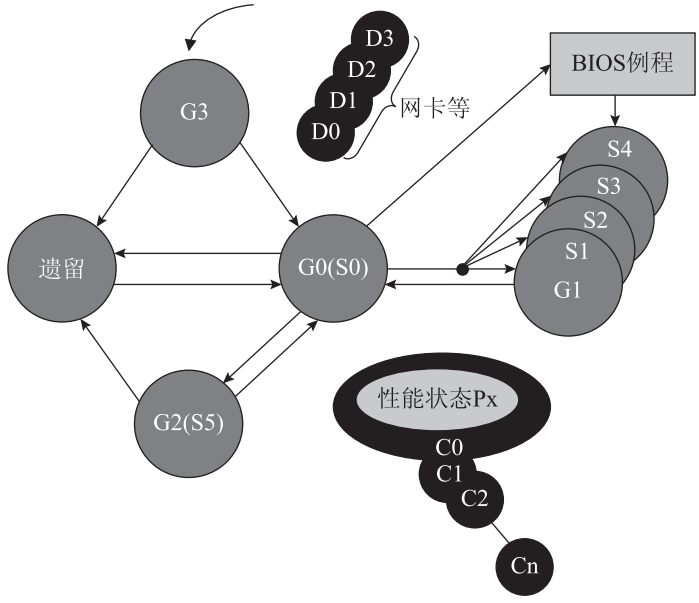

图2-1　全局系统电源状态和转换

平常使用时，计算机会在工作状态和睡眠状态之间来回切换。当处于工作状态时，计算机可以执行任务、运行和分发用户态的应用线程。如果不再使用某个个体设备和处理器时，可将此设备和处理器置于低电量状态（Dx和Cx状态）。系统关闭的任何设备都可以被再次打开，只需很短的延时。（延时的长短依赖于设备。只需要毫秒级延时就可以恢复一个LCD显示，而等待数秒来唤醒打印机通常也是可接受的。）

在工作状态的实际效果就是整个机器在运转。各种工作子状态的区别主要表现在计算速度、使用的电量、产生的热量和噪声等方面。在工作状态中进行的调整主要是在速度、电量、热量和噪声之间进行权衡。

当计算机处于空闲或者用户按下了电源按钮时，OS将计算机置于睡眠状态（Sx）中的一个状态。当处于睡眠状态时，不会发生用户可见的计算。睡眠状态之间的区别表现在

哪些事件能将系统唤醒到工作状态以及将花费多长时间。如果要求所有可能的事件都必须能唤醒机器或者要求非常快地唤醒机器时，那么它只能进入减少一部分系统电量消耗的睡眠状态。然而，如果唯一感兴趣的唤醒事件是用户按下一个开关并且允许花费数分钟的延时，那么OS可以将所有系统上下文保存到一个NVS文件中并将硬件转换到S4睡眠状态。在此状态下，机器只消耗了接近于零的电量并且可以将系统的上下文保存任意长的时间。

其他电源状态使用很少。支持遗留BIOS电源管理接口的计算机在遗留状态下启动并且在加载了ACPI OS后转换到工作状态。不支持遗留接口的系统直接从机械关机状态转换到工作状态。用户通过按下计算机的机械开关或者拔出计算模块来将计算机转换到机械关机状态。

2.3.1 平台电源管理特征

针对移动PC机、台式PC机以及多路服务器这三类平台，本节将详细介绍它们各自的电源管理特征。

2.3.1.1 移动PC机

移动PC机将继续使用新式先进的电源管理功能。不断发展的OSPM/ACPI将会提供增强的电源节能技术和更加精炼的用户策略。

在ACPI规范中，与移动PC机电源管理相关的功能是散热管理和嵌入式控制器接口。

2.3.1.2 台式PC机

由电源管理的台式机分为两类，尽管第一类台式PC机正随着时间的流逝而转为第二类。

- 普通“绿色PC机”——此处并不讨论新的装置功能。此机器实际上仅用于生产中的计算。至少在最初，这样的机器只具有非常少的功能。它们实际上只需要正常的ACPI定时器和控制功能、不需要支持详尽的睡眠状态等。然而，它们应允许OS尽可能独立地将尽可能多的设备/资源放置在挂起和关闭状态（在未使用设备上消耗最小的电量而实现最大的计算速度）。这样的PC机也需要支持通过一个定时器来将系统从睡眠状态中唤醒的能力，这将允许管理员恰好在用户出现在机器前工作时才强制打开此机器。
- 家庭PC机——当计算机被用在娱乐或者执行如应答电话这样的任务时，它们正式进入到家庭环境中。一个家庭PC机需要普通绿色PC机的所有功能。事实上，除了外盖事件外，它也具有移动PC机的所有ACPI电源功能（不需要任何遗留电源管理）。注意，针对家庭PC机，还存在散热管理要求。因为家庭PC机用户通常在散热受限环境下想让系统尽可能地安静运行。

2.3.1.3 多路服务器

服务器机器通常能节省最大的绝对电量，这听起来可能非常出人意料。为什么会这样呢？因为服务器使用了最大的硬件配置并且当使用者在晚间离开时，让他们按下关闭开关也不现实。

- 日间模式——在此模式下，使用电源管理的服务器机器与普通的绿色PC机非常像，所有时间都处于工作状态，但会将不使用的设备尽可能地置于低电量状态。因为服务器可以非常大，比如可以有很多的硬盘，所以电源管理可以节省非常多的电量。

允许OSPM在必要时进行适当地调整，以使服务器仍可正常工作。

- 夜间模式——在此模式下，服务器看起来像家庭PC机。它们会尽其所能地进入深睡眠，但是仍然能在指定的延时内苏醒和应答从网络、电话线等发过来的服务请求。例如，打印服务器可以进入深睡眠，直到它在凌晨3点接收到一个打印任务。此时，它可能用了不到30秒的时间就完成了苏醒、打印所接收的任务、再一次返回到睡眠状态等一系列操作。如果从网络发过来打印请求，那么此情景将依赖于一个智能的LAN适配器，此适配器可以对一个感兴趣的接收报文进行响应并唤醒系统。

2.4 电源管理标准

为了管理系统中所有设备的电源，OS需要使用标准的方法将命令发送到设备中。为了管理连接在特定I/O互连上的设备，这些I/O互连标准定义了电源操作和设备支持的电源状态。针对每个I/O互连类型定义的这些标准可以为OS使用的电源管理建立支持基线。这样，独立硬件供应商（IHVs）不需要花费太多的时间来编写软件就可以对其实现的设备所使用的电源进行管理，因为简单地遵守相应的I/O互连标准就可使它们获得OS的直接支持。对OS供应商来说，I/O互连标准允许将电源管理代码集中在每个I/O互连的驱动中。此外，由I/O互连实现的电源管理允许OS跟踪给定I/O互连中所有设备的状态。当所有设备都在一个给定的状态时（例如D3），OS可以将整个I/O互连置于与此状态匹配的电源模式（例如D3）。

在I/O互连级别，应为如下的总线编写电源管理规范。

- PCI
- PCI Express
- CardBus
- USB
- IEEE 1394

2.5 跨设备依赖

跨设备依赖被用来描述一种环境。在此环境下，对一个设备的操作干涉了其他不相干设备的操作，或者其他不相干设备干扰了它的行为。因为跨设备依赖可能会造成平台无法正常工作，所以ACPI未对跨设备依赖提供支持。在设计产品时，要尽量避免将设备包含在跨设备依赖中。

2.6 硬件编程模型

ACPI根据编程模型和硬件的行为来定义硬件。ACPI尽量将大多数已存在的遗留编程模型保持不变，然而，ACPI在实现某些特征时，要求这些特征能遵循特定的寻址和编程模型。此类硬件被称为“固定硬件”。软、硬件工程师应该仔细地阅读本节，理解硬件从仅支持遗

留硬件模型到同时支持ACPI/遗留硬件模型或仅支持ACPI硬件模型时所需进行的改变。

ACPI将硬件分为两类：固定类和通用类。固定类中的硬件满足ACPI的编程和行为规范。通用类中的硬件在实现上可以有更大的灵活性。

2.6.1 固定硬件编程模型

因为将遗留硬件迁移到固定硬件需要进行改变，所以ACPI限制了通过固定硬件支持的特征。在定义固定硬件特征时，主要依据如下几个准则。

- 性能敏感的特征。
- 在唤醒期间驱动所需的特征。
- 对OS软件灾难错误进行恢复的特征。

ACPI定义了通过寄存器访问固定硬件的接口。CPU时钟控制和电源管理定时器被定义为固定硬件，以便减少访问这些硬件时对性能所产生的影响，实现更快地降低热量或延长电池寿命。如果将这些功能逻辑放在PCI配置空间，那么将需要调用好几层驱动才能访问到这个地址空间。这将花费更长的时间，最终会影响到系统的电量（如尝试进入更低电源状态）或者事件的精度（如尝试获取时间戳值）。

通过OSPM访问固定硬件，可使OSPM在对唤醒过程进行控制时无需加载整个OS。例如，如果需要访问PCI配置空间，那么总线枚举时需要加载枚举时所需的所有驱动。将固定硬件中的这些接口定义在OSPM无需任何其他驱动协助就能通讯的地址上，可使OSPM在对继续加载整个OS还是将其返回到睡眠状态这类问题下决定前就可获得所需的信息。

如果OS出错了，那么对OSPM来说访问不需要驱动支持的地址空间也是可能的。在此情况下，OSPM将尝试使用固定电源按钮请求来将系统转换到G2状态。当OSPM事件处理例程不能再响应电源按钮事件时，电源按钮复写特征提供了备选机制，会无条件地将系统转换到软关机状态。

2.6.2 通用硬件编程模型

固定硬件编程模型要求在指定的地址位置上定义硬件寄存器，而通用硬件编程模型允许硬件寄存器位于更多的地址空间中。这使得OEM具有更大的灵活性来实现硬件特定的功能。OSPM可以直接访问固定硬件寄存器，但却需要依靠OEM提供的AML代码来访问通用硬件寄存器。

AML代码允许OEM为OSPM提供一种控制通用硬件特征和事件逻辑的方法。ASL是OEM用来创建AML的编程语言。此编程语言提供了很多在通用面向对象编程语言中使用的操作符。为了能对平台的电源管理进行描述并且能对硬件进行配置，对此编程语言进行了优化。ASL编译器将ASL源码转化成AML。AML是非常紧凑的机器语言，由ACPI AML代码解释器执行。

AML完成两件事：

- 将硬件从OSPM抽象出来；
- 从不同的OS实现中分离OEM代码。

ACPI的一个目标是允许OEM添加新硬件而仍然保持ACPI配置基本不变。为了使OSPM能在不同类型的新添加硬件上执行，ACPI定义了高层次的控制方法，它调用此方法来执行一个动作。OEM提供与OSPM执行的控制方法关联的AML代码，利用OEM提供的AML代码，通用硬件可以采取任何可行的形式。

ACPI的另一个重要目标是提供独立于OS的实现。为实现此目标，OEM AML代码在任何ACPI兼容OS下都必须相同地执行。通过将AML代码解释器作为OSPM的一部分可以达到此目标。

通用硬件特征模型如图2-2所示。在此模型下，通过AML代码向OSPM描述通用特征。这些AML代码采用对象的形式，位于与硬件关联的ACPI名字空间中。

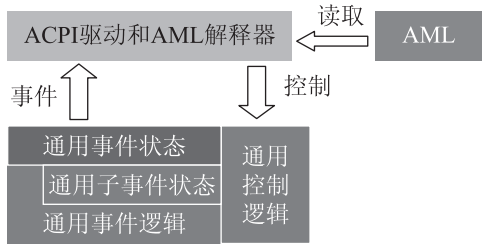

图2-2　通用硬件特征模型

例如，可以设计一个平台，使IDE HDD在D3状态下会有一个新添加的硬件将电源从驱动器中移走。IDE驱动器将在名字空间中有一个对AML PowerResource对象（控制新添加的电源板）的参考以及与此对象关联的控制方法。OSPM会调用控制方法来控制驱动器的D3状态。

- _PS0：用来将IDE驱动器放入D0状态的控制方法。
- _PS3：用来将IDE驱动器放入D3状态的控制方法。
- _PSC：返回IDE驱动器状态（开或者关）的控制方法。

此对象下的控制方法在OSPM和硬件之间构造了一个抽象层。OSPM知道如何通过定义的PowerResource对象来控制电源板，而硬件实现了平台特定的AML代码来执行期望的功能。在此例子中，平台通过编写和放置AML代码实现在_PS3控制方法中通过关闭硬件来向ACPI OS描述其硬件。这将使能如下过程：

当OSPM决定将IDE驱动器放入D3状态时，它调用IDE驱动并告诉其将驱动器置于D3状态（在此点驱动存储设备的上下文）。

- 当IDE驱动返回控制时，OSPM将驱动器置于D3状态。
- OSPM发现与HDD关联的对象并在对象中查找与D3状态关联的任何AML代码。
- OSPM执行适合的_PS3控制方法来控制新增加的通用硬件，将HDD置于更低的电量状态。

2.7　ACPI硬件特征

ACPI接口定义了如下两类不同的硬件特征。

- 固定硬件特征
- 通用硬件特征

固定硬件特征位于很多ACPI定义的地址空间中，这些地址空间位于ACPI编程模型所描述的位置。通用硬件特征位于四个地址空间之一（系统I/O、系统内存、PCI配置、嵌入式控制器或者串行设备I/O空间）并且通过在ACPI名字空间中声明AML控制方法进行描述。

固定硬件特征的实现需进行精确的定义。尽管很多固定硬件特征是可选的，但是如果要实现它们，那么必须按照所描述的方式来实现，因为OSPM期望以定义的行为来管理固件硬件的寄存器。

实现通用硬件特征时可以非常灵活。此逻辑由OEM提供的AML代码控制。ACPI也为专门的设备提供了专门的控制方法。例如，在一个通用硬件事件处理例程中，可以使用Notify命令通知OSPM已经发生散热事件。

表2-3仅列出了用于描述说明的通用特征。ACPI规范可以支持很多其他类型的硬件。

表2-3 特征/编程模型概要

特征名	描述	编程模型
电源管理定时器	24位或者32位自由运行的定时器	固定硬件特征控制逻辑
电源按钮	用户按下此按钮，系统将在工作状态和睡眠状态下进行切换	固定硬件事件和控制逻辑/ 通用硬件事件和逻辑
睡眠按钮	用户按下此按钮，系统将在工作状态和睡眠状态下进行切换	固定硬件事件和控制逻辑/ 通用硬件事件和逻辑
电源按钮覆盖	用户按下电源按钮后保持4秒以上，将关闭挂起的系统	
睡眠/唤醒控制逻辑	用来将系统在睡眠和工作状态之间进行转换的逻辑	固定硬件事件和控制逻辑
嵌入式控制器接口	ACPI嵌入式控制器协议和接口	通用硬件事件逻辑，必须位于通用目的寄存器块中
遗留/ACPI选择	表示系统使用遗留或者ACPI电源管理模型的状态位（SCI_EN）	固定硬件控制逻辑
外盖开关	用来表示系统外盖是打开还是关闭的按钮（仅用于移动系统）	通用硬件事件特征
C1电源状态	将处理器置于低电源状态的处理器指令	处理器ISA
C2电源控制	将处理器置于C2电源状态的逻辑	固定硬件控制逻辑
C3电源控制	将处理器置于C3电源状态的逻辑	固定硬件控制逻辑
散热控制	在指定触发点产生散热事件的逻辑	通用硬件事件和控制逻辑
设备电源管理	在不同设备电源状态间切换的控制逻辑	通用硬件控制逻辑
AC适配器	识别AC适配器插入和移走的逻辑	通用硬件事件逻辑
设备插入和移走	识别设备插入和移走的逻辑	通用硬件事件逻辑

2.8 ACPI寄存器模型

通过如下6个地址空间之一访问ACPI硬件。

- 系统I/O
- 系统内存

- PCI配置空间
- 嵌入式控制器
- 系统管理总线（SMBus）
- 固定硬件功能

不同的实现会导致不同的功能使用不同的地址空间。ACPI规范包含了固定硬件寄存器和通用硬件寄存器。固定硬件寄存器需要实现ACPI定义的接口，而新添加的硬件所产生的任何事件需要使用通用硬件寄存器。

ACPI定义了寄存器块。一个ACPI兼容系统在启动时会在内存中创建一个FADT系统描述表，此表包含了提供给OSPM使用的一系列指向不同固定硬件寄存器块的指针。这些寄存器中的位包含了为给定寄存器块定义的属性。ACPI定义的寄存器类型如下。

- 状态/使能寄存器（针对事件）
- 控制寄存器

如果一个寄存器块的类型是状态/使能类型，那么它将包含一个具有状态位的寄存器和一个对应的具有使能位的寄存器。状态位和使能位有一个需要遵循的准确实现定义（除非另外说明），如图2-3所示。

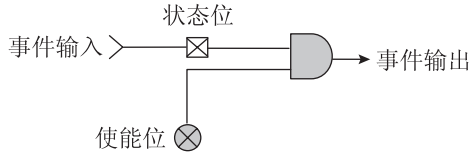

图2-3　状态/使能单元的方块图

注意，在此示例中，硬件根据正在设置的事件输入来设置状态位。仅能通过软件向此状态位所对应的位置写1来清除此状态位。使能位对状态位的设置和清除没有影响，它仅仅决定设置的状态位能否产生一个事件输出。如果使能位已被设置，那么状态位被设置时将产生一个SCI。

ACPI也定义了寄存器组。一个寄存器组包含两个不同的寄存器块。寄存器组中的每个位所在的位置都是固定的且不能被改变，位可以被划分到两个寄存器块中。这意味着允许寄存器组中的位可以存在于任意一个寄存器块中或者同时在两个寄存器块中，使得将多个不同芯片中的位映射到相同寄存器得到简化。

OSPM将一个寄存器组视为一个单独的寄存器，只是它们位于多个地方。为了读取一个寄存器组，OSPM先读取“A”寄存器块，接着读取“B”寄存器块，然后将这两个寄存器块的结果进行“逻辑或”运算（SLP_TYPx字段是一个例外）。读取寄存器块中保留位或未使用位时总是返回零。对寄存器块中保留位或未使用位进行写操作时，这些位不会受到任何影响。

在每个寄存器组中，SLP_TYPx字段值可以是不同的。各自的睡眠对象_Sx包含了一个SLP_TYPa字段和一个SLP_TYPb字段。也就是说，此对象返回一个数组，由两个范围在0～7的整数值构成。OSPM总是将SLP_TYPa值写到“A”寄存器块，接着将SLP_TYPb值写到“B”寄存器块，最后将相同的值写到所有其他位所对应的位置。OSPM不会读取SLP_TYPx值。

图2-4展示了一个寄存器组，此寄存器组包含寄存器块A和寄存器块B。在寄存器块B中实现了位“0”和“5”，寄存器块A针对这些位的位置返回零。在寄存器块A中实现了位“2”“3”和“7”，寄存器块B针对这些位的位置返回零。所有保留位、忽略位返回各自定义的ACPI值。

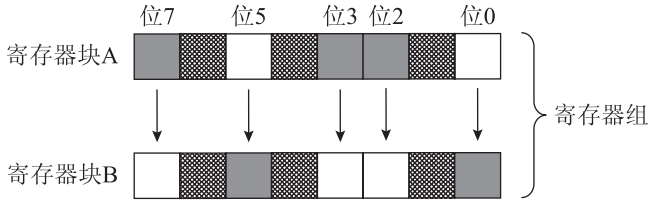

图2-4 固定硬件特征寄存器组示例

当访问这个寄存器组时，OSPM必须读取寄存器块A，接着读取寄存器块B。然后OSPM将这两个寄存器块中的值进行逻辑或运算并在此结果上进行操作。

当向此寄存器组进行写操作时，OPSM将期望的值写到寄存器块A，接着将相同的值写到寄存器块B。

ACPI定义了如图2-5所示的固定硬件寄存器块。每个寄存器块从FADT中获取一个独立的指针。OEM将这些地址作为静态资源进行设置，因此它们不会被改变——OSPM不能重新映射ACPI资源。

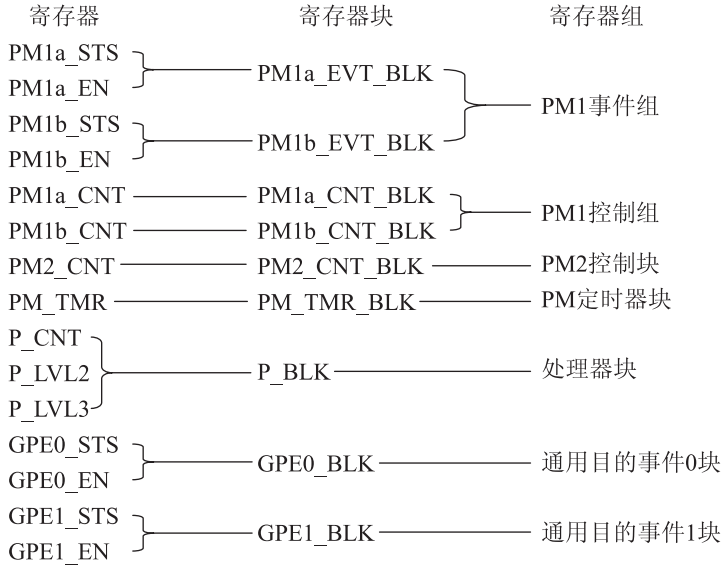

图2-5 寄存器块和寄存器组

表2-4～表2-9总结了ACPI寄存器。

表2-4 PM1事件寄存器

寄存器	大小（字节）	地址（相对于寄存器块）
PM1a_STS	PM1_EVT_LEN/2	<PM1a_EVT_BLK>
PM1a_EN	PM1_EVT_LEN/2	<PM1a_EVT_BLK> + PM1_EVT_LEN/2
PM1b_STS	PM1_EVT_LEN/2	<PM1b_EVT_BLK>
PM1b_EN	PM1_EVT_LEN/2	<PM1b_EVT_BLK> + PM1_EVT_LEN/2

表2-5　PM1控制寄存器

寄存器	大小（字节）	地址（相对于寄存器块）
PM1_CNTa	PM1_CNT_LEN	<PM1a_CNT_BLK>
PM1_CNTb	PM1_CNT_LEN	<PM1b_CNT_BLK>

表2-6　PM2控制寄存器

寄存器	大小（字节）	地址（相对于寄存器块）
PM2_CNT	PM2_CNT_LEN	<PM2_CNT_BLK>

表2-7　PM定时器寄存器

寄存器	大小（字节）	地址（相对于寄存器块）
PM_TMR	PM_TMR_LEN	<PM_TMR_BLK>

表2-8　处理器控制寄存器

寄存器	大小（字节）	地址（相对于寄存器块）
P_CNT	4	<P_BLK>或者PTC对象指定
P_LVL2	1	<P_BLK>+4h
P_LVL3	1	<P_BLK>+5h

表2-9　通用目的事件寄存器

寄存器	大小（字节）	地址（相对于寄存器块）
GPE0_STS	GPE0_LEN/2	<GPE0_BLK>
GPE0_EN	GPE0_LEN/2	< GPE0_BLK> + GPE0_LEN/2
GPE1_STS	GPE1_LEN/2	<GPE1_BLK>
GPE1_EN	GPE1_LEN/2	< GPE1_BLK> + GPE1_LEN/2

PM1事件组包含PM1a_EVT寄存器块和PM1b_EVT寄存器块，而这两个寄存器块包含了固定硬件特征事件位。每个实现的事件寄存器块包含两个寄存器：状态寄存器和使能寄存器。每个寄存器组中的位都有一个已定义的位置，不能被改变，然而却可以在任意寄存器块（A或B）中实现此位。A和B寄存器块允许芯片将事件分到两个或者更多的芯片中。

PM1控制组包含固定硬件特征控制位，由PM1a_CNT_BLK寄存器块和PM1b_CNT_BLK寄存器块组成。每个寄存器块与单个控制寄存器相关联。PM2_CNT_BLK寄存器块当前只包含一位，用于仲裁屏蔽功能。

通用目的事件寄存器包含针对通用特征的事件编程模型。所有通用事件与固定事件一样都会产生SCI。通用事件状态位可以位于任何地方，然而，顶级通用事件必须位于通用目的寄存器块之一。不在通用目的寄存器空间的任何通用特征事件状态被认为是次级状态位，它们的顶级状态位在通用目的事件寄存器空间。注意，在到达GPE事件状态前可能有N级的通用目的事件。通用目的事件寄存器通过两个寄存器块来描述：GPE0_BLK或GPE1_BLK。FADT中的指针独立地指向了每个寄存器块。每个寄存器块进一步划分成两个寄存器：GPEx_STS和GPEx_EN。通用目的事件寄存器中的状态和使能寄存器遵循固定硬件事件寄存器的事件模型。

2.9 系统描述表头部

为了使硬件厂商能更加灵活地选择产品实现的方式，ACPI使用表来描述系统的信息、特征以及控制这些特征的方法。这些表列出了系统主板上的设备、无法通过其他硬件标准识别的设备、进行电源管理的设备以及这些设备的能力。这些表也列出了系统的能力，例如系统支持的睡眠电源状态；系统中现有电源板、时钟源、电池和系统指示灯等。这样，OSPM可以控制这些系统设备，而无须知道如何来实现这些系统控制。

所有系统描述表都以如表2-10所示的结构开始。签名字段决定了系统描述表的内容。此规范定义的系统描述表签名如表2-11所示。

表2-10 描述表头部结构

字段	字节长度	字节偏移	描述
Signature	4	0	描述表签名的ASCII字符串
Length	4	4	描述表的字节长度，包含头部本身
Revision	1	8	描述表的版本号，高版本号的描述表要兼容于具有相同签名的低版本描述表
Checksum	1	9	包含此字段在内的整个表中的字节值相加必须等于零，才认为是有效的描述表
OEMID	6	10	OEM提供的标识OEM的字符串
OEM Table ID	8	16	OEM提供的字符串，OEM使用此字段标识特定的数据表。当定义一个定义块来区分定义块功能时此字段特别有用。OEM为每个不同的表分配一个新的OEM Table ID
OEM Revision	4	24	OEM提供的版本号，高版本号假设是更新的版本
Create ID	4	28	创建此表所用工具的厂商ID。对包含定义块的表，此字段值是ASL编译器的ID
Create Revision	4	32	创建此表所用工具的版本号。对包含定义块的表，此字段值是ASL编译器的版本号

对OEM来说，好的设计实践应确保在任何表中分配的OEMID和OEM Table ID字段的一致性。这样，工具就能通过编程来确定哪个表与之前具有相同OEMID和OEM Table ID字段的表相兼容并且是最近修订的版本。

表2-11和表2-12包含了由ACPI规范定义的系统描述符表签名。在ACPI规范中定义并进行说明的系统描述表签名如表2-11所示。简单地被ACPI保留并且由其他业界规范进行定义和说明的系统描述表签名如表2-12所示。这样，允许定义OS和平台特定的表并且在需要时将表的物理地址包含在RSDT/XSDT中。针对其他业界规范定义的表，ACPI规范被用来避免表签名之间的冲突。

表2-11 ACPI定义的部分描述表中头部签名

签名	描述
“APIC”	Multiple APIC Description Table
“DSDT”	Differentiated System Description Table
“ECDT”	Embedded Controller Boot Resources Table
“FACP”	Fixed ACPI Description Table (FADT)
“FACS”	Firmwre ACPI Control Structure

签名	描述
“RSDT”	Root System Description Table
“SBST”	Smart Battery Specification Table
“SLIT”	System Locality Distance Information Table
“SRAT”	System Resource Affinity Table
“SSDT”	Secondary System Description Table
“XSDT”	Extended System Description Table

表2-12　ACPI为表保留的部分签名

签名	描述
“DMAR”	DMA Remapping Table
“HPET”	IA-PC High Precision Event Timer Table
“IBFT”	iSCSI Boot Firmware Table
“IVRS”	I/O Virtualization Reporting Structure
“MCFG”	PCI Express memory mapped configuration space base address Description Table
“SLIC”	Microsoft Software Licensing Table Specification
“SPCR”	Serial Port Console Redirection Table
“SPMI”	Server Platform Management Interface Table
“UEFI”	UEFI ACPI Data Table

第3章 ASL编程语言

ASL是一种开发语言，用来定义ACPI对象以及编写ACPI控制方法。OEM厂商和BIOS开发者首先使用ASL定义对象和编写控制方法，然后再使用转换工具（编译器）产生AML格式的控制方法。尽管ASL与AML紧密相关，但它们是不同的语言。每个ACPI兼容OS必须支持AML。给定用户可以定义任意的语言来代替ASL并且编写相应的工具来将其转化成AML。

ASL语句示例如下。

```
DefinitionBlock("forbook.aml","DSDT",0x02,"OEM","forbook",0x1000)
{
    OperationRegion(\GIO, SystemIO, 0x125, 0x1)
    Field(\GIO, ByteAcc, NoLock, Preserve) {
          CT01, 1,
    }
    Scope(\_SB) {
          Device(PCI0) {
                PowerResource(FET0, 0, 0) {
                      Method (_ON) {
                            Store (Ones, CT01)
                            Sleep (30)
                      }
                      Method (_OFF) {
                            Store (Zero, CT01)
                      }
                      Method (_STA) {
                            Return (CT01)
                      }
                }
          }
    }
}
```

3.1 ASL语法

使用ASL语句来声明对象，每个对象包含三个部分，其中的两个部分可以不存在，例如。

```
Object := ObjectType FixedList VariableList
```

FixedList表示已知长度的列表，提供了给定对象类型的所有实例都必须具有的数据。固定列表写成(a, b, c, …)，其中参数的数量依赖于特定的ObjectType。固定列表中元素可以是嵌套对象，如(a, b, (q, r, s, t), d)。FixedList中的元素可以使用默认值，使用默认值的参数可以被跳过。例如，(a,,c)将使用第二个参数的默认值。

VariableList表示预先不能确定长度的子对象列表。此子对象用来帮助定义父对象。可变列表写成{x, y, z, aa, bb, cc}，其中的任何元素都可以是嵌套对象。ObjectType决定了哪些项是VariableList中的合法元素。

在编写ASL语句时，应该遵守的其他规则如下。

- 多个空格与一个空格是一样的。空格、（、）、‘、’和回车都是符号分隔符。
- //表示注释的开始。注释从//开始，到行尾结束。
- /*表示注释的开始。注释从/*开始，到*/结束。
- 一个ASCII字符串需要使用一对双引号“”括起来。
- 可采用三种方式书写数值常量：十进制、八进制（0ddd）或者十六进制（0xdd）。
- Nothing表示空项。例如{Nothing}等价于{}。

3.1.1 ASL语法表示法

用来表示ASL语法的注释如表3-1所示。

表3-1 ASL语法注释

注释惯例	描述	实例
Term := Term Term …	:=左侧的Term可以扩展成右侧的Term序列	aterm := btem ctem意味着aterm可以被扩展成btem ctem序列
=>	表示在运行时需要将ASL参数评估成指定的AML数据类型	“TermArg => Integer”意味着当AML解释器评估此语句时，必须将一个ASL TermArg参数评估成Integer数据类型
<>	用来组合条目	<a b> | <c d> 表示a b或者c d
|	分隔替代者	btem | <cterm dterm>意味着下面格式是可能的： ● bterm ● cterm dterm
Term Term	通过空格将Term彼此分隔，形成一个有序列表	无
粗体文本	操作符的名字。注意，操作符不区分大小写	ThermalZone (ZoneName) {ObjectList} ThermalZone是操作符的名字
斜体文本	参数的名字	ThermalZone (ZoneName) {ObjectList} ZoneName是一个参数
单引号(‘’)	表示字符常量	‘A’
0xdd	表示一个用两个十六进制数字表示的字节值	0x21表示十六进制数21，或十进制数33。注意，用十六进制表示的值必须以前导符0x开始
-	表示一个范围	“1-9”意味着在1和9之间（包含1和9）的一个数

3.2 ASL概念

本节介绍ASL中常用的基础概念，如ASL名字、ASL数据类型、ACPI名字空间等，主要面向编写ASL代码并且为平台开发定义块的开发者。

3.2.1 ASL名字

在ASL对象名字中任何位置都合法的字符包括‘A’～‘Z’、‘_’、‘a’～‘z’。在非起始位置合法的字符还包含‘0’～‘9’。ASL名字不对大小写字符进行区分。所有字符都将被转换成大写字符。可放置在ASL名字前的编辑符包括表示根的‘\’（0x5C）和表示父节点的‘^’（0x5E）。ACPI规范保留了以前缀_T_开头的对象名字，提供给ASL编译器内部使用。例如，ASL编译器在将复杂的控制结构转换到AML时，可以使用这些对象来存储临时的值。

3.2.2 ASL字面常量

本节描述如何使用ASL对整数和字符串常量进行编码。

3.2.2.1 整数

数值常量可以是十进制、八进制或者十六进制数值。八进制常量跟随一个前导符‘0’。十六进制常量跟随一个前导符‘0’以及一个大写的‘X’或小写的‘x’。在某些环境下，语法会指定数值必须被评估成有限范围内的整数，例如0x00-0xFF。

3.2.2.2 字符串

字符串常量由通过双引号 “” 括起的零个或多个ASCII字符组成。字符串常量表示一连串的字符。这些字符放在一起形成了一个以null结尾的字符串。源文件中的字符串可以使用UTF-8编码方案进行编码。UTF-8是面向字节的编码方案，其中一些字符占用一个字节，而另一些占用多个字节。ASCII字符值0x01-0x7F仅仅占用一个字节。然而，当前仅仅有一个操作符能支持UTF-8字符串：Unicode。因为定义的字符串常量仅包含非空（null）字符值，所以在ASCII 0x01-0x7F范围中的转义十六进制或八进制数必须是非空值。对ASCII值之外的任意字节数据，必须要使用字节数组对象。

因为反斜线被用来作为转义字符，同时也被用来作为名字空间的根前缀，所以当字符串常量包含一个从名字空间的根开始的完整名字路径时，必须使用两个反斜线来表示（注：仅在引号内的字符串常量需要两个反斜线），例如：

```
Name (_EJD, "\\_SB.PCI0.DOCK1")
```

因为双引号被用来括起一个字符串，所以在字符串中使用了一个特殊转义序列（\”）来表示双引号。其他转义序列如表3-2所示。

表3-2 ASL转义序列

转义序列	ASCII字符
\a	0x07(BEL)

转义序列	ASCII字符
\b	0x08(BS)
\f	0x0C(FF)
\n	0x0A(LF)
\r	0x0D(CR)
\t	0x09(TAB)
\v	0x0B(VT)
\”	0x22(“)
\’	0x27(‘)
\\	0x5C(\)

因为字面字符串是只读常量，所以不支持下面的ASL语句。

```
Store ("ABC", "DEF")
```

然而，可以支持下面的语句序列。

```
Name (STR, "DEF")
…
Store ("ABC", STR)
```

3.2.3 ASL资源模板

ASL包含一些用来创建资源描述符的宏。ResourceTemplate宏创建一个数组，资源描述符宏就被用在此数组中创建资源描述符。ResourceTemplate宏自动产生一个结束描述符并计算资源模板的校验和。ResourceTemplate宏的格式如下。

```
ResourceTemplate ()
{
    // List of resource macros
}
```

下面的ASL代码示例展示了资源描述符宏是如何被用来创建由_PRS控制方法返回的资源模板的。

```
Name (PRS0,
     ResourceTemplate ()
     {
            StartDependentFn (1, 1)
            {
                  IRQ (Level, ActiveLow, Shared) {10, 11}
                  DMA (TypeF, NotBusMaster, Transfer16) {4}
                  IO (Decode16, 0x1000, 0x2000, 0, 0x100)
                  IO (Decode16, 0x5000, 0x6000, 0, 0x100, IO1)
            }
            StartDependentFn (1, 1)
            {
```

```
                IRQ (Level, ActiveLow, Shared) {}
                DMA (TypeF, NotBusMaster, Transfer16){5}
                IO (Decode16, 0x3000, 0x4000, 0, 0x100)
                IO (Decode16, 0x5000, 0x6000, 0, 0x100, IO2)
            }
            EndDependentFn ()
        }
)
```

有时，在运行时（例如方法执行期间）需要改变已存在资源模板中描述符的参数，描述符宏可以包含一个名字声明。在创建此名字之后，可以用此名字来参考描述符。当在描述符中声明了一个名字后，ASL编译器将自动在给定名字之下创建字段名字以参考描述符中的个体字段。

例如，给定上面的资源模板，下面的ASL代码示例将改变名字为IO2的I/O描述符的最小和最大地址。

```
CreateWordField (PRS0, IO2._MIN, IMIN)
Store (0xA000, IMIN)
CreateWordField (PRS0, IO2._MAX, IMAX)
Store (0xB000, IMAX)
```

3.2.4 ASL宏

ASL编译器本身内嵌了一些宏来协助各种ASL编码操作。表3-3列出了一部分这样的宏并且对它们的功能进行了解释。

表3-3 ASL内嵌宏例子

ASL语句	描述
EISAID(TextID)	将7个字符的文本参数转化成对应的4字节数字EISA ID编码。在为设备声明ID时可以使用此宏
ResourceTemplate()	采用用户可读的方式来提供即插即用资源描述符信息，然后再转化成适合的二进制即插即用资源描述符编码
ToUUID(AsciiString)	将一个ASCII字符串转换成128位的字节数组
Unicode(StringData)	将一个ASCII字符串转换成包含在字节数组中的Unicode字符串

3.2.5 ASL数据类型

ASL提供了广泛的数据类型和处理数据的操作符。它也提供了在数据类型之间进行明确和隐含转换的机制。表3-4描述了现有的每种数据类型。

表3-4 ASL数据类型概要

ASL数据类型	描述
[Uninitialized]	未分配的类型或值。所有控制方法中LocalX变量、方法执行开始处未使用的ArgX变量以及未初始化的对象数组元素都是此类型的变量。对象在ASL表达式中当作源操作数之前，必须被初始化（使用Store或CopyObject）
Buffer	字节数组。未初始化的元素默认值是零
Buffer Field	使用CreateBitField、CreateByteField、CreateWordField、CreateQWordField、CreateField创建的字节数组字段或通过Index操作符返回的字节数组字段

ASL数据类型	描述
DDB Handle	Load操作符返回的定义块句柄
Debug Object	调试输出对象
Device	设备或总线对象
Event	事件同步对象
Field Unit（within an Operation Region）	使用Field、BankField或IndexField创建的字段单元，是地址空间的一部分，按位对齐并且粒度是一位
Integer	N位小头无符号整数。在ACPI 1.0是32位，在ACPI 2.0或之后版本是64位
Integer Constant	由ASL操作符“Zero”“One”“Ones”和“Revision”创建的整数常量
Method	控制方法（可执行的AML功能）
Mutex	同步互斥对象
Object Reference	参考一个使用RefOf、Index或者CondRefOf操作符创建的对象
Operation Region	操作区域（地址空间中的一个区域）
Package	具有固定元素数量（最多255）的ASL对象集合
Power Resource	电源资源描述对象
Processor	处理器描述对象
RawDataBuffer	字节数组。未初始化的元素默认值是零。RawDataBuffer不包含任何AML编码字节，仅包含原始字节
String	以Null结尾的ASCII字符串
Thermal Zone	散热区描述对象

3.2.5.1 数据类型转换概要

当ASL在运行时（AML解释器执行期间），会提供两种机制来将一种数据类型的对象转换成另一种数据类型。第一种机制是明确地进行数据类型转换。此机制允许使用明确的ASL操作符来将一个对象转换成另一种数据类型对象，参考3.3.12节中对ASL操作符的说明。第二种机制是隐含地进行数据类型转换。在使用或者存储数据对象前，如果需要将数据对象转换成期望的数据类型，那么AML解释器可以进行隐含的数据类型转换。

数据类型转换遵寻以下规则。

- 当输入参数的操作数类型与期望的输入类型不匹配时，输入参数总是要进行隐含的数据类型转换（也被称为隐含的源操作数转换）。
- 当输出的是一个已存在的命名对象或者命名字段并且与将要存储的对象类型不同时，除明确进行数据转换的操作符外，所有其他操作符的输出参数都要进行隐含的数据类型转换（也被称为隐含的结果对象转换）。
- 明确进行数据转换的操作符输出参数以及参考方法本地变量和参数（LocalX或者ArgX）的输出参数不需要进行隐含的类型转换。

3.2.5.2 隐含的源操作数转换

在ASL操作符执行期间，AML解释器按照如下规则处理每个源操作数。

- 如果操作数是操作符所期望的类型，那么不需要进行转换。

- 如果操作数类型不正确，那么尝试将其转换成正确的类型。
- 针对Concatenate操作符和逻辑操作符（LEqual、LGreater、LGreaterEqueal、LLess、LLessEqual和LNotEqual），第一个操作数的数据类型表明了所需的第二个操作数数据类型。针对Concatenate操作符，如果第二个操作数要隐含地转换成与第一个操作数类型相匹配，那么第一个操作数类型也表明了结果对象的类型。
- 如果转换不可行，那么终止运行控制方法并且产生一个致命错误。

当源操作数包含的数据类型不同于操作符所期望的类型时，可以尝试进行隐含的源操作数转换。例如：

```
Store ("5678", Local1)
Add (0x1234, Local1, BUF1)
```

在上面的Add语句中，Local1包含了一个字符串对象。在进行Add操作处理前，必须先将Local1转换成整数对象。

在某些情况下，操作符可以采用多种类型的操作数（例如整数和字符串）。在此情况下，依赖于操作数的类型，将应用最高优先级的转换。例如：

```
Store (Buffer (1) {}, Local0)
Name (ABCD, Buffer (10) {1, 2, 3, 4, 5, 6, 7, 8, 9, 0})
CreateDWordField (ABCD, 2, XYZ)
Name (MNOP, "1234")
Concatenate (XYZ, MNOP, Local0)
```

Concatenate操作符在前两个参数中可以使用Integer、Buffer或者String类型，并且第一个参数的类型决定了将如何转换第二个参数的类型。在此示例中，第一个参数的类型是Buffer Field。那么它将被转换成Integer、Buffer还是String类型呢？如表3-5所示，最高优先级转换是Integer。因此，如下两个对象将被转换成Integer。

```
XYZ (0x05040302)
MNOP (0x31, 0x32, 0x33, 0x34)
```

这样，组合在一起之后，结果类型和值如下。

```
Buffer (0x02, 0x03, 0x04, 0x05, 0x31, 0x32, 0x33, 0x34)
```

3.2.5.3 隐含的结果对象转换

针对所有产生和存储一个结果值的ASL操作符（包含Store操作符），AML解释器按照如下规则处理和存储结果对象。

- 如果ASL操作符是明确转换操作符之一（ToString、ToInteger等和CopyObject操作符），那么不执行任何转换（换句话说，结果对象被直接存储到目的对象并且完全覆盖任何已经存储在目的对象中的已存在数据）。
- 如果目的对象是一个方法的本地变量或者参数变量（LocalX或ArgX），那么不执行任何转换，结果被直接存储到目的对象。

- 如果目的对象是固定的类型，例如一个命名的对象或者Field对象，那么在存储前尝试将源操作数转换到已存在的目的对象类型。
- 如果转换不可行，那么终止运行控制方法并且产生一个致命错误。

当一个操作符的结果被存储到一个固定类型的对象时，可能会发生一个隐含的结果转换。例如：

```
Name (BUF1, Buffer (10))
Add (0x1234, 0x789A, BUF1)
```

因为BUF1是一个固定类型Buffer的命名对象，所以Add操作的整数结果在被存储到BUF1前必须被转换成一个字节数组。

3.2.5.4 数据类型和类型转换

表3-5列出了现有的ASL数据类型和针对每个数据类型现有的数据类型转换。允许的转换应用到明确和隐含两种转换中。

表3-5 数据类型和类型转换

ASL数据类型	可以隐含或者明确地转换成这些数据类型（按照优先级顺序）	可以隐含或者明确地对这些数据类型进行转换
[Uninitialized]	无。在任何ASL语句中被用作源操作数都会导致一个致命错误	Integer、String、Buffer、Package、DDB Handle、Object Reference
Buffer	Integer、String、Debug Object	Integer、String
Buffer Field	Integer、Buffer、String、Debug Object	Integer、Buffer、String
DDB Handle	Integer、Debug Object	Integer
Debug Object	无。在任何ASL语句中被用作源操作数都会导致一个致命错误	Integer、String、Buffer、Package、DDB Handle、Field Unit、Buffer Field
Device	无	无
Event	无	无
Field Unit（within an Operation Region）	Integer、Buffer、String、Debug Object	Integer、Buffer、String
Integer	Buffer、Buffer Field、DDB Handle、Field Unit、String、Debug Object	Buffer、String
Integer Constant	Integer、Debug Object	无。存储任何对象到一个常量都是一个空操作，而不是一个错误
Method	无	无
Mutex	无	无
Object Reference	无	无
Operation Region	无	无
Package	Debug Object	无
Power Resource	无	无
Processor	无	无
RawDataBuffer	无	无

ASL数据类型	可以隐含或者明确地转换成这些数据类型（按照优先级顺序）	可以隐含或者明确地对这些数据类型进行转换
String	Integer、Buffer、Debug Object	Integer、Buffer
Thermal Zone	无	无

3.2.5.5 数据类型转换规则

针对每个允许的数据类型转换，表3-6列出了详细的数据转换规则。这些转换规则由AML解释器来实现并且应用到所转换的类型——明确转换、隐含源操作数转换和隐含结果转换。

表3-6 对象转换规则

转换一个此种数据类型的对象	转换到一个此种数据类型的对象	AML解释器执行的动作
Buffer	Buffer Field	Buffer中的内容被复制到Buffer Field。如果Buffer的大小小于Buffer Field的大小，那么使用零进行填充。如果Buffer的大小大于Buffer Field的大小，那么高位被截断
	Debug Object	Buffer中的每个字节以十六进制整数显示
	Field Unit	Buffer中的整个内容被复制到Field Unit。如果Buffer中位的数量大于Field Unit的大小，那么它被划分成块并且完整地写到Field Unit，首先是低块。如果Buffer或者划分后的最后一块小于Field Unit的大小，那么在写之前使用零进行扩展
	Integer	如果没有整数对象存在，那么创建一个新的整数。Buffer中的内容被复制到整数中，从最低有效位开始，一直到Buffer已被完全复制或者达到整数的最大位数。整数的大小通过定义块表头中的Revision字段表示。Revision字段值小于2时表示整数的大小是32位；大于等于2时表示整数的大小是64位。如果Buffer的大小小于整数的大小，那么使用零进行扩展；如果Buffer的大小大于整数的大小，那么它被截断。不允许零长度的Buffer转换到一个整数
	String	如果没有字符串对象存在，那么创建一个新的字符串。如果字符串已经存在，那么它被完全覆盖并且被截断或者被扩展来容纳转换的Buffer。Buffer中的内容被转换成由两个字符构成的十六进制数表示的字符串，每个十六进制数之间用一个空格进行分割。零长度的Buffer将被转换成null(零长度)字符串
Buffer Field	[参考Integer和Buffer规则]	如果Buffer Field的长度小于或者等于整数的位数，那么它被视为一个整数；否则，被视为一个Buffer。整数的位数（32或者64）由定义块表头中的Revision字段来表示
DDB Handle	[参考Integer规则]	此对象视为一个整数
Field Unit	[参考Integer和Buffer规则]	如果Field Unit的长度小于或者等于整数的位数，那么它被视为一个整数；否则，它被视为一个Buffer。整数的位数（32或者64）由定义块表头中的Revision字段来表示
Integer	Buffer	如果不存在Buffer对象，那么基于整数的大小创建一个新的Buffer对象（针对32位整数是4字节，针对64位整数是8字节）。如果Buffer对象已经存在，那么整数覆盖整个Buffer对象。如果整数需要比Buffer大小更多的位，那么在复制到Buffer之前整数被截断。如果整数包含比Buffer大小更少的位，那么整数被按照零扩展方式填充整个Buffer
	Buffer Field	整数覆盖整个Buffer Field。如果整数位数小于Buffer Field的长度，那么进行零扩展。如果整数位数大于Buffer Field的长度，那么高位被截断

转换一个此种数据类型的对象	转换到一个此种数据类型的对象	AML解释器执行的动作
Integer	Debug Object	整数以十六进制数值显示
	Field Unit	整数覆盖整个Field Unit。如果整数位数小于Field Unit的长度，那么进行零扩展。如果整数位数大于Buffer Field的长度，那么高位被截断
	String	如果不存在字符串对象，那么基于整数的大小创建一个新的字符串对象（针对32位整数是8个字符，针对64位整数是16个字符）。如果字符串对象已经存在，那么整数覆盖整个字符串对象。如果整数需要比字符串大小更多的位，那么在复制到字符串之前整数被截断。如果整数包含比字符串大小更少的位，那么整数被按照零扩展方式填充整个字符串。在每种情况下，整个整数被转换成由十六进制ASCII字符表示的字符串
Package	Package	如果不存在Package对象，那么创建一个新的Package对象。如果Package对象已经存在，那么它被完全覆盖并且被截断或者被扩展来容纳源Package。目的Package中任何已存在的有效元素被删除，源Package中整个内容被复制到目的Package
	Debug Object	Package中的每个元素基于其类型进行显示
String	Buffer	如果不存在Buffer对象，那么创建一个新的Buffer对象。如果Buffer对象已经存在，那么字符串覆盖整个Buffer对象。如果字符串比Buffer的大小长，那么在复制到Buffer之前字符串被截断。如果字符串比Buffer的大小短，那么余下的Buffer字节被设置为零。在每种情况下，字符串都作为Buffer，每个ASCII字符串字符被复制到一个Buffer字节，包含null终止符。空字符串（零长度）将被转换成零长度的Buffer
	Buffer Field	字符串被作为一个Buffer。如果Buffer长度小于Buffer Field的长度，那么进行零扩展。如果Buffer长度大于Buffer Field的长度，那么高位被截断
	Debug Object	每个字符串字符按照一个ASCII字符进行显示
	Field Unit	从首个字符开始，字符串的每个字符被写到Field Unit。如果Field Unit小于8位，那么每个字符的高位将丢失。如果Field Unit大于8位，那么额外的位使用零进行填充
	Integer	如果整数对象不存在，那么创建一个新的整数。整数被初始化为零并且ASCII字符串被解释成十六进制的常数。每个字符串字符被解释成一个十六进制值（‘0’～‘9’，‘A’～‘F’、‘a’～‘f’），首个字符作为最高有效数字，以第一个非十六进制字符、字符串结束或者达到整数的大小时（针对32位整数是8个字符，针对64位整数是16个字符）结束。注意，首个非十六进制字符终止转换，不会产生错误。“0x”前缀不被允许。空字符串到整数的转换也不被允许

3.2.5.6 存储与复制对象规则

表3-7列出了将对象存储到不同类型的命名目的对象时执行的动作。ASL提供了如下类型的存储操作。

- Store操作符被用来明确地将一个对象存储到一个位置。在进行类型转换时，使用原始对象的隐含转换规则。
- 很多ASL操作符可以可选地将它们的结果存储到由最后一个参数指定的对象中。在这些操作符中，如果指定了目的地，那么动作就如Store操作符一样，将结果放置到目的地。
- CopyObject操作符被用来明确地存储一个对象的备份到一个位置，不使用隐含的类型转换。

表3-7 对象存储和复制规则

当存储任何数据类型的对象到此类型目的位置时	Store操作符或者具有目的操作数的任何ASL操作符执行的动作	CopyObject操作符执行的动作
方法ArgX变量	不需要任何转换，此对象被复制到目的地。存在一个例外。如果ArgX包含了一个对象参考，那么会自动进行参考，考并且对象被复制到对象参考所在的目的地，而不会覆盖ArgX的内容	
方法LocalX变量	不需要任何转换将此对象复制到目的地。即使LocalX包含了一个对象参考，它也会被覆盖	
Field Unit或者Buffer Field	应用隐含结果转换到此对象后，再将此对象复制到目的地	Fields永久地保持它们的类型并且不能被改变。因此，CopyObject仅能被用来复制类型为Integer或者Buffer的对象到Fields
命名数据对象	应用隐含结果转换到此对象来匹配命名位置已存在的类型，然后将此对象复制到目的地	此对象和类型被复制到命名位置

3.2.5.7 读、写对象规则

在下面的描述中，读操作总是返回实际的对象，而不是对象的备份。

```
Add (Local1, Local2, Local3)
```

上述语句不会为Local1或者Local2创建不必要的备份。也就是说，在控制方法调用中，参数要通过引用进行传递。

1. ArgX对象

从ArgX参数中读取：

- ObjectReference——自动进行参考，返回参考的结果。使用DeRefof返回相同的结果。
- Buffer——返回字节数组，可以创建索引、字段或者参考此字节数组。
- Package——返回对象数组，可以创建一个索引或者参考此对象数组。
- 其他对象类型——返回此对象。

针对如下ASL代码示例，读取参数的结果如表3-8所示。

MTHD (RefOf (Obj), Buf, Pkg, Obj)

表3-8 读取ArgX对象

参数	MTHD ArgX类型	在ArgX上进行的读操作	读的结果
RefOf(Obj)	对对象Obj的参考	Store(Arg0, ……) CopyObject(Arg0, ……) DeRefOf(Arg0)	Obj Obj Obj
Buf	字节数组	Store(Arg1, ……) CopyObject(Arg1, ……) Index(Arg1, ……) Field(Arg1, ……)	Buf Buf Index(Buf) Field(Buf)
Pkg	对象数组	Store(Arg2, ……) CopyObject(Arg2, ……) Index(Arg2, ……)	Pkg Pkg Index(Pkg)
Obj	所有其他对象类型	Store(Arg3, ……) CopyObject(Arg3, ……)	Obj Obj

如表3-9所示，向ArgX参数进行存储：

- ObjectReference对象——自动进行参考、复制对象并覆盖最后的目标。
- 所有其他类型对象——复制此对象并覆盖ArgX对象。

表3-9　写向ArgX对象

当前ArgX类型	所写的对象	在ArgX上进行的写操作	写的结果
RefOf(OldObj)	Obj	Store(……, ArgX) CopyObject(……, ArgX)	RefOf(Obj的备份) RefOf(Obj的备份)
所有其他对象类型	Obj	Store(……, ArgX) CopyObject(……, ArgX)	Obj的备份 Obj的备份

2. LocalX对象

如表3-10所示，从LocalX变量中读取：

- ObjectReference——如果执行一个DeRefOf，那么返回参考的结果，否则，返回此参考。
- 所有其他对象类型——返回此对象。

表3-10　读取LocalX对象

当前LocalX类型	在LocalX上进行的读操作	读的结果
RefOf(Obj)	Store(LocalX, ……) CopyObject(LocalX, ……) DeRefOf(LocalX)	RefOf（Obj） RefOf（Obj） Obj
Obj	Store(LocalX, ……) CopyObject(LocalX, ……)	Obj Obj

如表3-11所示，向LocalX变量存储所有对象类型时，首先删除LocalX中任何已存在的对象，然后存储此对象的备份。

表3-11　写向LocalX对象

当前ArgX类型	所写的对象	在LocalX上进行的写操作	写的结果
所有对象类型	Obj	Store(……, ArgX) CopyObject(……, ArgX)	Obj的备份 Obj的备份

3. 命名对象

如表3-12所示，从命名对象中读取：

- ObjectReference——如果执行一个DeRefOf，那么返回参考的结果，否则，返回此参考。
- 其他对象类型——返回此对象。

表3-12　读取命名对象

当前名字类型	在名字上进行的读操作	读的结果
RefOf(Obj)	Store(NAME, ……) CopyObject(NAME, ……) DeRefOf(NAME)	RefOf（Obj） RefOf（Obj） Obj
Obj	Store(NAME, ……) CopyObject(NAME, ……)	Obj Obj

如表3-13所示，向命名对象存储所有对象类型时，首先删除命名对象中任何已存在的对象，然后存储此对象的备份。Store操作符将执行一个到命名对象中已存在类型的隐含转换。CopyObject不执行隐含的存储。

表3-13 写向命名对象

当前NAME类型	所写的对象	在NAME上进行的写操作	写的结果
所有对象类型	Obj	Store(……, NAME) CopyObject(……, NAME)	Obj的备份（转换成与NAME已存在对象类型相匹配的结果） Obj的备份（不转换）

3.2.6 ACPI名字空间

系统为所有定义块维护着单个分级的树形名字空间并且使用此名字空间来参考各种对象。所有定义块被加载到相同的名字空间中。像这样允许一个定义块参考另一个定义块中的对象和数据，意味着OEM必须加以仔细来避免任何名字冲突。仅仅定义块的卸载操作就会从名字空间中移走名字，因此加载定义块时出现的名字冲突被认为是致命错误。名字空间中的内容仅会在加载或卸载操作中被改变。

下面的命名约定适用于所有名字。

- 所有名字都是固定的32位。
- 名字第一个字节的合法内容是：大写字母‘A’～‘Z’（0x41-0x5A）和下划线‘_’（0x5F）。
- 名字中其他三个字节的合法内容是：大写字母‘A’～‘Z’（0x41-0x5A）、数字‘0’～‘9’（0x30-0x39）和下划线‘_’（0x5F）。
- 对于一个长度小于4个字节的名字，ASL编译器使用下划线‘_’在尾部进行填充。
- 此规范保留了以下划线‘_’开头的名字。在定义块中，仅可以使用由此规范定义的以下划线‘_’开头的名字。
- 以‘\’开头的名字使得此名字参考名字空间的根（‘\’不是名字的一部分）。
- 以‘^’开头的名字使得此名字参考当前名字空间的上级名字（‘^’不是名字的一部分）。

除了以‘\’开头的名字外，当前名字空间决定了将要创建的名字在名字空间中所处的位置以及其所参考的名字所处的位置。通过在当前名字空间及上一级名字空间中查找匹配的名字来定位名字所在的位置。如果上一级名字空间也不包含此名字，那么继续递归地向上查找，直到发现此名字或者名字空间不再有上一级名字空间（根名字空间）。如果到达根名字空间，那么表示无法发现此名字。尝试访问根的上一级名字将导致无法查找到此名字的结果。

存在两类名字路径：绝对名字空间路径（也就是以‘\’开头的路径）和相对名字空间路径（也就是相对于当前名字空间的路径）。之前讨论的名字空间搜索规则仅应用于单个名字的路径，它是一个相对名字空间路径。对包含多个名字组合或者父前缀‘^’的这些相对名字空间路径，搜索规则不适用。如果搜索规则不适用一个相对名字空间路径，则在当前路径中查找名字空间对象。例如：

```
 ABCD                  //搜索规则适用
^ABCD                  //搜索规则不适用
 XYZ.ABCD              //搜索规则不适用
\XYZ.ABCD              //搜索规则不适用
```

所有名字参考使用32位固定长度的名字或者使用名字扩展前缀来将多个32位固定长度的名字连接在一起。这对参考一个对象的名字非常有用，例如一个不在当前名字空间范围内的控制方法。

图3-1展示了在加载差分定义块之后的ACPI名字空间。

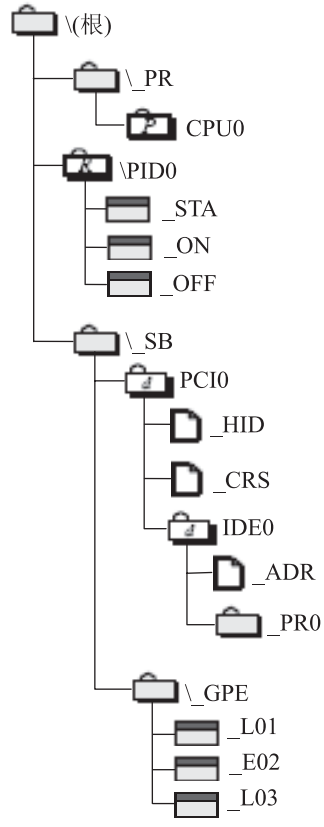

图3-1　ACPI名字空间例子

当使用单个相对路径名访问名字空间对象时必须要多加小心。尝试访问通过相对路径名表示的对象时会向根进行迭代，直到发现此对象或者遇到了根。这可能会产生不希望的结果。例如，使用图3-1所描述的名字空间，尝试从_SB_.PCI0.IDE0中访问_CRS命名对象将有不同的结果，区别在于使用了绝对路径名还是相当路径名。如果指定了绝对路径名（_SB_.PCI0.IDE0._CRS），那么将产生一个错误，因为此对象不存在。使用单个路径名（_CRS）进行访问将返回_SB_.PCI0._CRS对象。注意，此访问可以成功，不会产生错误。

3.2.6.1　预定义根名字空间

在根名字空间下定义的名字空间如表3-14所示。

表3-14　在根名字空间下定义的名字空间

名字	描述
_GPE	GPE寄存器块中通用事件
_PR	ACPI 1.0处理器名字空间。ACPI 1.0要求所有Processor对象都必须定义在此名字空间下。ACPI允许将Processor对象定义在_SB名字空间下。为了兼容于ACPI 1.0操作系统，平台可以包含_PR名字空间。ACPI兼容名字空间可以将Processor对象定义在_SB或_PR范围内，但不能同时定义在这两个范围内

名字	描述
_SB	所有设备/总线对象都定义在此名字空间下
_SI	系统指示对象都定义在此名字空间下
_TZ	ACPI 1.0散热区名字空间。ACPI 1.0要求所有散热区对象都必须定义在此名字空间下。ACPI允许将散热区对象定义在_SB名字空间下。为了兼容于ACPI 1.0操作系统，平台可以包含_TZ名字空间。ACPI兼容名字空间可以将散热区对象定义在_SB或_TZ范围内，但不能同时定义在这两个范围内

3.2.6.2 对象

除了本地数据对象之外，所有对象在范围上都是全局的。本地数据对象仅在每次调用范围和生命期中有效，被用来从头到尾处理当前的调用。

对象的内容变化很大。很多对象可以参考任何支持的数据类型、控制方法或系统软件提供的函数等数据变量。

对象可以包含一个版本字段。后续的ACPI规范定义新的对象版本，以致它们能向后兼容于之前规范/对象版本的OSPM实现。新对象字段添加在之前对象定义的末尾。OSPM根据它支持的版本号（包含更早的版本号）来解释对象。这样，OSPM期望一个对象的长度大于或等于已知对象版本的长度。当评估一个版本号大于OSPM已知版本号的对象时，OPSM忽略超出已知版本定义的对象字段范围之外的内部对象字段。

3.2.7 定义块

定义块由AML格式的数据组成，包含了采用AML对象表示的有关硬件实现细节的信息。AML对象包含数据、AML代码或其他AML对象。在加载了定义块之后，此信息的顶层组织是分级名字空间中的名字标签。

OSPM将整个定义块作为一个逻辑单位进行加载或者卸载。OSPM在执行AML Load()或LoadTable()操作符或者在初始化期间遇到一个表定义时，会加载一个定义块。OSPM使用从FADT中获取的DSDT指针加载DSDT，DSDT中包含了差异定义块。在初始化期间，OSPM遇到在RSDT/XSDT中定义的SSDT时会加载其他定义块。

AML Load()和LoadTable()操作符使得定义块可以静态或动态地加载其他定义块。这样，定义块就可以定义新的系统属性或者在某些情况下对之前定义的系统属性进行扩展。尽管这给了硬件在实现上进行修改的能力，但也将其限制在合理范围内。

一些AML操作符执行简单的功能，而另一些包含了复杂的功能。定义块的能量来自于它允许将这些操作按各种方式组合在一起向OSPM提供功能。

3.2.7.1 定义块编码

下面介绍在定义块中定义名字（仅加载时间）、对象和对象数组时使用的编码。定义块被编码成一个从头到尾的数据流。在数据流中的前导字节来自于AML编码表，说明了如何解释随后的一些字节，而随后的字节又能说明如何解释在它之后的一些字节。

在数据流中定义了两类数据。一类是对象数组和对象声明（加载时）；另一类是对象参考（对象数组内容/运行时）。

所有编码格式如图3-2所示。编码的前导字节说明了声明或参考的类型。类型在数据流中表示一个隐含的或明确的长度。所有明确的长度声明PkgLength是包含数据长度字节本身在内的数据长度。

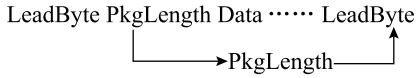

图3-2　AML编码

隐含的长度编码对象有固定的长度或者使用嵌套编码来表示另一个明确或隐含的长度。

PkgLength在数据流中编码成1到4个字节序列，字节0中的最高两位表示随后多少字节包含在PkgLength编码中，随后两位仅用在一个字节编码中，允许在一个字节编码中支持最大为0x3F的数据流长度。不使用这两位的更长编码支持的最大数据流长度为：两字节编码支持0xFFF，三字节编码支持0xFFFFF，四字节编码支持0xFFFFFFF。

注意，PkgLength不能落在逻辑边界之外。例如，如果一个对象数组包含在另一个对象数组中，那么根据定义它的长度必须包含在外部对象数组中。隐含长度的数据也有类似的要求。

在定义块中存在的首个对象必须是一个命名的控制方法。这是定义块的初始控制。

对象数组是对象，包含了对一个或多个对象的参考。对象数组也可被认为是一组数组，包含在对象数组中的任何对象可能也是一个对象数组。这样，可以实现固定或变化深度的多维数组。

未命名对象用来填充命名对象的内容。不能在根名字空间下创建未命名对象。未命名对象可用在控制方法的参数中。

控制方法在执行过程中创建对象时，可能会产生错误。如果创建命名对象块的方法阻塞后被重新执行，就可能发生此情况，这是因为所有命名对象有一个绝对的路径。将对象名指定为相对路径名，也会发生错误。例如，下面的ASL代码段在功能上是一致的。

（1）

```
Method (DEAD,) {
    Scope (\_SB_.FOO) {
           Name (BAR,) // Run time definition
}
}
```

（2）

```
Scope (\_SB_) {
    Name (\_SB_. FOO.BAR,) // Load time definition
}
```

注意，在上面的例子中，执行DEAD方法总会失败，因为对象_SB_.FOO.BAR在加载时已被创建。

3.2.8 控制方法执行

OSPM为了查询或调整系统级硬件状态，需要评估控制方法对象。这被称为一次调用。

一个控制方法可以使用内部或严格定义的其他控制方法来完成当前的任务。这些控制方法可以包含由操作系统软件提供的控制方法。方法可以参考名字空间中的任何对象。虽然对控制方法进行解释时并不进行抢占，但是它却可以阻塞。当一个控制方法阻塞时，OSPM可以重新或者继续执行一个不同的控制方法。一个控制方法仅可以认为在此控制方法未阻塞期间对全局对象的访问是独享的。

全局对象是在表加载时创建的名字空间对象。

3.2.8.1 参数

最多可以向一个控制方法传递7个参数。每个参数是一个对象。此对象也可以是参考其他对象的对象数组类型对象。ASL ArgX操作符提供了对参数对象的访问。传递到任何控制方法中的参数数量都是固定的，在创建这个控制方法时进行定义。

方法参数可以采用如下格式。

- 一个参考命名对象的ACPI名字或名字路径，包含LocalX和ArgX名字。在此情况下，与名字关联的对象作为参数进行传递。
- 一个参考另一个控制方法的ACPI名字或名字路径。在此情况下，方法被调用并且将方法的返回值作为参数进行传递。如果方法并没有返回一个对象，那么将产生致命错误。如果此对象在方法调用后不再使用，那么会自动删除。
- 一个有效的ASL表达式。在此情况下，评估ASL表达式并且将评估结果作为参数进行传递。如果此对象在方法调用后不再使用，那么会自动删除。

3.2.8.2 方法调用惯例

通过引用常量调用方式（call-by-reference-constant）来描述控制方法的调用非常合适。作为参数传递的对象会通过引用方式进行传递。这意味着将参数传递给被调用方法时，它们不会被复制到新对象中。

除了在特殊可控环境外，传递到控制方法的参数作为常量进行传递。它们不能被修改。

被调用方法通常不能对通过ArgX操作符传递到此控制方法的对象直接地进行写或者修改。换句话说，当ArgX操作符在一个ASL语句中作为目的操作数使用时，已存在的ArgX对象不被修改。相反，新对象代替已存在的对象，ArgX操作符有效地变成LocalX操作符。

只读参数规则的唯一例外是ArgX操作符包含了一个通过RefOf ASL操作符创建的对象参考。在此情况下，当ArgX操作符作为目的操作数使用时，将导致任何保存在由RefOf操作符参考的ACPI名字中的已存在对象被覆盖。

在某些受限情况下，可以创建一个新的、可写的对象，以允许一个控制方法来改变ArgX对象的值。这种情况仅针对字节数组对象和对象数组对象，并且这两种对象的值要通过间接方式表示。针对字节数组对象，可以创建一个可写的索引或字段来参考原始的字节数组数据并允许被调用方法读取或修改数据。针对对象数组对象，可以创建一个可写的索引，以允许被调用方法修改对象数组中个体元素的内容。

3.2.8.3 本地变量和本地建立的数据对象

控制方法可以访问最多八个本地数据对象。对本地数据对象的访问有简化的编码。在控制方法开始执行时，本地数据对象为NULL。可以通过ASL LocalX操作符来访问本地数据对象。

一旦控制方法执行结束，可以返回一个用来作为方法执行结果的对象。调用者可以直接使用此结果。如果它想保存此结果，那么需要将结果保存到一个不同的对象中。

在一个方法的范围内创建的名字空间对象是动态的。它们仅在方法执行期间存在。它们由控制方法中的代码创建，在方法退出时被销毁。方法也可以使用Scope操作符或完整路径名在名字空间的当前范围之外创建动态对象，但方法退出时它们仍被销毁。在方法的范围外加载时创建的对象是静态的。例如：

```
Scope (\XYZ) {
    Name (BAR, 5) // Creates \XYZ.BAR
    Method (FOO, 1) {
          Store (BAR, CREG)    // same effect as Store (\XYZ.BAR, CREG)
          Name (BAR, 7)        // Creates \XYZ.FOO.BAR
          Store (BAR, DREG)    // same effect as Store (\XYZ.FOO.BAR,
                               // DREG
          Name (\XYZ.FOOB, 3) // Creates \XYZ.FOOB
    }
}
```

当加载包含上述ASL语句的描述表时，创建的对象\XYZ.BAR是一个静态的对象。当执行FOO方法中的Name(BAR, 7)语句时，创建的对象\XYZ.FOO.BAR是一个动态的对象。当执行Name(\XYZ.FOOB, 3)语句时，由\XYZ.FOO方法创建的对象\XYZ.FOOB是一个动态的对象。注意，在退出\XYZ.FOO方法时，会销毁\XYZ.FOOB对象。

3.2.8.4 访问操作区域

控制方法使用Field操作符在操作区域中声明一个数据字段。当使用数据字段名字来执行访问时，可以对地址空间位置上的数据进行读取和写入。一个操作区域是地址空间中一个特殊的操作区间，通过使用开始地址（偏移）和长度来声明成整个地址空间的一个子集。通过在操作区域中声明的字段进行访问时，控制方法必须对任何地址有独享的访问权。控制方法不能直接访问任何其他硬件寄存器，包含ACPI定义的寄存器块。例如，控制方法不能直接访问GPEx_BLK。但是GPEx_BLK可用来为控制方法调用提供一个可扩展的中断处理模型。

ACPI指定的预定义操作区域类型如表3-15所示。

表3-15 操作区域地址空间标识

名字（RegionSpace参数）	值
SystemMemory	0
SystemIO	1
PCI_Config	2
EmbeddedConfig	3
SMBus	4

名字（RegionSpace参数）	值
SystemCMOS	5
PCIBARTarget	6
IPMI	7
GeneralPurposeIO	8
GenericSerialbus	9
Reserved	0x0A-0x7F
OEM定义的操作区域类型	0x80-0xFF

操作区域可以简单地、直接地访问SystemMemory、SystemIO和PCI_Config地址空间。操作区域对SystemCMOS、PCIBARTarget、IPMI地址空间的访问描述如下。

1）CMOS协议

大多数计算机都包含一个RTC/CMOS设备。可以将RTC/CMOS设备当作非易失内存中的一个线性字节数组。存在一种标准的机制来访问设备中非易失内存的前64字节，此设备与在原始IBM PC/AT上使用的Motorola RTC/CMOS设备兼容。目前存在的RTC/CMOS设备通常包含了比64字节更多的非易失内存，不存在标准的机制来访问这个额外的存储区域。为了在AML中提供对这些设备中所有非易失内存的访问，每类设备扩展都存在一个PNP ID，分别是“PNP0B00”“PNP0B01”和“PNP0B02”。在11.14节中对这些PNP ID所支持的特殊设备进行了描述。

各设备类型的驱动处理对SystemCMOS操作区域的访问，只允许对CMOS中与当前时间、周、日期、月、年和世纪相关的字节进行读操作。

2）PCI设备BAR目标协议

每个PCI设备都有一个称为配置空间的地址空间与之关联。在配置空间中，偏移0x10和偏移0x27之间存在最多6个基地址寄存器BAR（Base Address Register）。这些BAR包含了一系列控制寄存器的基地址（在I/O或内存空间）。因为即插即用OS可以在任何时间改变这些BAR的值，所以ASL不能使用I/O或内存操作区域对这些目标进行读取和写入。此外，即插即用OS将自动地将与这些BAR关联的I/O和内存区域的所有权分配给一个与此PCI设备关联的设备驱动。一个ACPI OS（必须也是一个即插即用操作系统）将不允许ASL读取和写入由设备驱动所拥有的区域。

如果一个平台使用了PCIBARTarget类型的操作区域，那么ACPI OS将不会加载与此PCI功能关联的设备驱动。例如，如果一个PCI功能中的任何BAR与PCIBARTarget类型的操作区域相关联，那么OS将假设此PCI功能完全在ACPI BIOS的控制下，不会加载设备驱动。这样，此PCI功能可以用作平台控制器来支持ACPI BIOS执行的某些任务（如热插拔PCI等）。

① 声明PCIBARTarget类型的操作区域

PCI基地址寄存器包含了PCI设备的控制寄存器所在的I/O或内存区域的起始地址。每个基地址寄存器实现了一个协议，用来确定这些控制寄存器在I/O空间还是内存空间，以及PCI设备能解码多大的地址空间。

通过提供基地址寄存器在PCI设备的PCI配置空间中的偏移来声明PCIBARTarget类型的操作区域。基地址寄存器决定了对设备产生的访问是通过I/O周期还是内存周期，而不是由

操作区域的声明来决定。区域的长度也同样如此。

在ASL语句OperationRegion(PBAR, PciBarTarget, 0x10, 0x4)中，第三个参数是基地址寄存器在设备配置空间中的偏移。0x10表示此操作区域使用了设备配置空间中的第一个基地址寄存器。

② PCI头部类型和PCIBARTarget类型的操作区域

PCIBARTarget类型的操作区域仅可以声明在PCI头部类型值为0的PCI设备范围中。具有其他头部类型值的PCI设备是桥控制器。对PCI桥的控制已超出ASL的范围。

3.2.9 资源描述符类型

ACPI中诸如_CRS、_PRS和_SRS这样的控制方法会使用包含资源描述符的对象数组来描述设备所需的资源。此外，某些设备配置对象也会使用设备资源描述符作为参数。为了简化操作，ASL中包含了一些用来创建资源描述符的宏。

3.2.9.1 小型资源数据类型

小型资源数据类型的长度在2～8个字节之间，格式如表3-16所示。

表3-16 小型资源数据类型字段定义

偏移	字段:位		
字节0	类型(0):7	小项名字:6～3	长度(n字节):2～0
字节1到n	数据字节（长度0～7）		

表3-17列出了当前为即插即用设备定义的小型资源数据类型中名字字段的取值。

表3-17 小型资源数据中小项名字定义

小项名字	值
保留	0x00-0x03、0x0B-0x0D
IRQ格式描述符	0x04
DMA格式描述符	0x05
开始依赖功能描述符	0x06
结束依赖功能描述符	0x07
I/O端口描述符	0x08
固定位置I/O端口描述符	0x09
固定DMA描述符	0x0A
厂商定义描述符	0x0E
结尾标签描述符	0x0F

1. IRQ描述符

IRQ数据结构表示设备使用的一个中断号，并且提供了一个可设置的位掩码来表示设备实现的中断号。标准的PC-AT实现，最多有16个可能的中断，因此使用了两个字节大小的掩码字段。设备所需的每个独立中断，都需要一个IRQ描述符。IRQ描述符定义如表3-18所示。

表3-18 IRQ描述符定义

偏移	字段名
字节0	值为0x22或0x23（0010001nB）——类型为0，小项名字为0x4，长度为2或者3
字节1	IRQ掩码位7:0，_INT。位0表示IRQ0，位1表示IRQ1，依次类推

偏移	字段名
字节2	IRQ掩码位15:8，_INT。位0表示IRQ8，位1表示IRQ9，依次类推
字节3	IRQ信息 ● 位7:6 保留（必须是零） ● 位5 唤醒能力，_WKC ◆ 0 无唤醒能力——中断不能唤醒系统 ◆ 1 有唤醒能力——中断能从低电量空闲状态或系统睡眠状态中唤醒系统 ● 位4 中断共享，_SHR ◆ 0 独有——中断不能与其他设备共享 ◆ 1 共享——中断可以与其他设备共享 ● 位3 中断极性，_LL ◆ 0 高有效——当信号为高或真时采样中断 ◆ 1 低有效——当信号为低或假时采样中断 ● 位2:1 忽略 ● 位0 中断模式，_HE ◆ 0 电平触发——在响应低状态的信号时触发中断 ◆ 1 边沿触发——在响应信号从低到高状态改变时触发中断

注：如果不包含第3个字节，那么假设是高有效、边沿触发的非共享中断。

2. DMA描述符

DMA数据结构表示设备使用的一个DMA通道，并且提供了一个可设置的位掩码来表示设备真实实现的通道。设备所需的每个独立通道，都需要一个DMA描述符。DMA描述符定义如表3-19所示。

表3-19 DMA描述符定义

偏移	字段名
字节0	值为0x2A（00101010B）——类型为0，小项名字为0x5，长度为2
字节1	DMA通道掩码位7:0（通道0～7），_DMA。位0表示通道0，位1表示通道1，依次类推
字节2	● DMA通道信息： ● 位7 保留（必须是零） ● 位6:5 支持的DMA通道速度，_TYP ◆ 00 表示兼容模式 ◆ 01 表示在EISA中描述的类型A DMA ◆ 10 表示类型B DMA ◆ 11 表示类型F ● 位4:3 忽略 ● 位2 逻辑设备总线主控状态，_BM ◆ 0 逻辑设备不是总线的主控 ◆ 1 逻辑设备是总线的主控 ● 位1:0 DMA传输类型偏好，_SIZ ◆ 00 仅8位 ◆ 01 8位和16位 ◆ 10 仅16位 ◆ 11 保留

3. 开始依赖功能描述符

每个逻辑设备需要一系列的资源。这些资源之间可能存在相互依赖性。为了使仲裁软件程序能为逻辑设备进行资源分配，需要将这种依赖性展示出来。依赖功能描述符被用来表示这种相互依赖性。依赖功能描述符的数据结构定义如表3-20所示。

表3-20　开始依赖功能描述符定义

偏移	字段名
字节0	值为0x30或0x31（0011000nB）——类型为0，小项名字为0x6，长度为0或者1

开始依赖功能描述符的长度是0或者1。此额外的字节可选地用来表示紧随此描述符之后的资源组的兼容性或者性能/健壮性优先级。兼容性优先级指与遗留操作系统兼容的配置等级。例如，为了兼容性原因，优选的COM1配置是IRQ4、I/O端口3F8-3FF。性能/健壮性优先级指满足性能和健壮性理由的配置等级。如果不包含优先级字节，那么缺省表示依赖功能优先级是可接受配置。此字节的定义如表3-21所示。

表3-21　开始依赖功能优先级字段定义

位	定义
1:0	兼容优先级。可接受值如下： ● 0　好的配置——最高优先级并且推荐的配置 ● 1　可接受配置——更低的优先级却是可接受的配置 ● 2　下一级最佳的配置——功能配置但不是最佳的 ● 3　保留
3:2	性能/健壮性。可接受值如下： ● 0　好的配置——最高优先级并且推荐的配置 ● 1　可接受配置——更低的优先级却是可接受的配置 ● 2　下一级最佳的配置——功能配置但不是最佳的 ● 3　保留
7:4	保留（必须是零）

当多个依赖功能有相同的优先级时，它们的优先级再根据资源数据结构中出现的顺序确定。在结构中最先出现的依赖功能有最高的优先级，依次类推。

4. 结束依赖功能描述符

只允许每个逻辑设备有一个结束依赖功能项，如表3-22所示，保证了依赖功能不会嵌套。

表3-22　结束依赖功能描述符定义

偏移	字段名
字节0	值为0x38（00111000B）——类型为0，小项名字为0x7，长度为0

5. I/O端口描述符

I/O端口描述符为可编程设备提供了完整的功能描述。I/O端口描述符的定义如表3-23所示。

表3-23　I/O端口描述符定义

偏移	字段名	定义
字节0	I/O Port Descriptor	值为0x47（01000111B）——类型为0，小项名字为0x8，长度为7
字节1	Information	I/O端口信息 ● 位7:1　保留，必须是零 ● 位0　（_DEC） ◆ 1　逻辑设备解码16位地址 ◆ 2　逻辑设备仅解码地址位9:0
字节2	Range minimum base address, _MIN bits[7:0]	设备配置的最小I/O基地址
字节3	Range minimum base address, _MIN bits[15:8]	
字节4	Range maximum base address, _MAX bits[7:0]	设备配置的最大I/O基地址
字节5	Range maximum base address, _MAX bits[15:8]	

偏移	字段名	定义
字节6	Base alignment, _ALN	最小基地址的对齐要求，按1个字节大小增加
字节7	Range length, _LEN	请求的连续I/O端口数

6. 固定位置I/O端口描述符

此描述符被用来描述10位的I/O位置，如表3-24所示。

表3-24 固定位置I/O端口描述符定义

偏移	字段名	定义
字节0	Fixed Location I/O Port Descriptor	值为0x4B（01001011B）——类型为0，小项名字为0x9，长度为3
字节1	Range base address, _BAS bits[7:0]	I/O基地址
字节2	Range base address, _BAS bits[9:8]	
字节3	Range length, _LEN	请求的连续I/O端口数

7. 固定DMA描述符

固定DMA描述符为平台提供了一种方式，可以将DMA请求线和通道静态地分配给连接到共享DMA控制器上的设备，如表3-25所示。此描述符不同于DMA描述符，就在于它支持更多的DMA请求线和DMA控制器通道以及两者之间的灵活映射。此外，它也提供了设备使用的传输总线宽度。针对请求的每个独立请求线和通道，都要重复使用此结构。此结构仅能用在_CRS对象中。

表3-25 固定DMA资源描述符

偏移	字段名
字节0	值为0x55（01010101B）——类型为0，小项名字为0xA，长度为5
字节1	DMA请求线，_DMA，被用来唯一地标识请求线。在控制器特定OS驱动中进行请求线到控制器的映射
字节2	
字节3	DMA通道，_TYP。被用来唯一地标识控制器逻辑通道。通道号可在多个请求线之间共享
字节4	
字节5	DMA传输宽度，_SIZ。连接到此请求线的设备所支持的总线宽度如下： ● 0x00 8位 ● 0x01 16位 ● 0x02 32位 ● 0x03 64位 ● 0x04 128位 ● 0x05 256位 ● 0x06-0xFF 保留

8. 厂商定义描述符

厂商定义的资源数据类型由厂商使用，如表3-26所示。

表3-26 厂商定义的资源描述符定义

偏移	字段名
字节0	值为0x71-0x77（01110nnnB）——类型为0，小项名字为0x0E，长度为1～7
字节1到字节7	厂商定义

9. 结尾标签

结尾标签表示资源数据的结束，如表3-27所示。ASL编译器在ResourceTemplate语句之

后自动产生结尾标签。

表3-27　结尾标签定义

偏移	字段名
字节0	值为0x79（01111001B）——类型为0，小项名字为0xF，长度为1
字节1	包含所有资源数据的校验值。产生此校验值的目的是将此校验值和所有数据类型数据相加后产生一个零值

3.2.9.2　大型资源数据类型

为了将更多的数据包含在配置数据结构中，要使用大型资源数据类型，如表3-28所示。

表3-28　大型资源数据类型字段定义

偏移	字段名
字节0	值等于1xxxxxxxb——类型为1（大项），大项名字如表3-29所示
字节1	数据项长度N的低字节，位0:7
字节2	数据项长度N的高字节，位8:15
字节3到(N+2)	实际的数据项

表3-29列出了当前为即插即用设备定义的大型资源数据类型中名字字段的取值。

表3-29　大型数据中名字字段定义

大项名字	值
保留	0x00
24位内存范围描述符	0x01
通用寄存器描述符	0x02
保留	0x03
厂商定义描述符	0x04
32位内存范围描述符	0x05
32位固定位置内存范围描述符	0x06
字地址空间描述符	0x08
双字地址空间描述符	0x07
扩展IRQ描述符	0x09
四字地址空间描述符	0x0A
扩展地址空间描述符	0x0B
GPIO连接描述符	0x0C
保留	0x0D
GenericSerialBus连接描述符	0x0E
保留	0x0F-0x7F

1. 24位内存范围描述符

24位内存范围描述符描述了设备在24位地址空间中的内存范围，其定义如表3-30所示。

表3-30　24位内存范围描述符定义

偏移	字段名	定义
字节0	24-bit Memory Range Descriptor	值为0x81（10000001B）——类型为1，大项名字为0x01
字节1	Length, bits[7:0]	值为0x09（9）
字节2	Length, bits[15:8]	值为0x00