Advanced Configuration and Power Interface Specification

Compaq Computer Corporation
Intel Corporation
Microsoft Corporation
Phoenix Technologies Ltd.
Toshiba Corporation

Revision 2.0b
October 11, 2002

INTELLECTUAL PROPERTY DISCLAIMER

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE.

NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED OR INTENDED HEREBY.

COMPAQ, INTEL, MICROSOFT, PHOENIX, AND TOSHIBA DISCLAIM ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF PROPRIETARY RIGHTS, RELATING TO IMPLEMENTATION OF INFORMATION IN THIS SPECIFICATION. COMPAQ, INTEL, MICROSOFT, PHOENIX, AND TOSHIBA DO NOT WARRANT OR REPRESENT THAT SUCH IMPLEMENTATION(S) WILL NOT INFRINGE SUCH RIGHTS.

Revision	Change Description	Affected Sections
Oct. 2002	Errata and clarifications added.	See below.
2.0b	Clarified use of double-backslash within ACPI strings that refer to ACPI names (Must use double backslash within quoted strings.)	4.7.4.2.1 5.2.13 6.3.2
	Clarified Global Lock description and corrected table name	5.2.9.1
	Clarification of description of _CID	6.1.2
	Clarification of rules for _HID strings.	6.1.4
	Corrected data type(s) for Source field in _PRT entries.	6.2.8
	Clarified _PRT example to show full namepaths in addition to namesegs.	6.2.8.1
	Defined OK status return for _WAK; return value is required	7.3.5
	Added _UIDs to fan control example.	12.5.2
	Changed DefinitionBlock grammar to disallow executable code outside of control methods. Fixed indenting and boldface as necessary.	16.1.3
	Added an overview of the explicit/implicit conversion rules for clarity	16.2.2.1
	Clarified action of the ToString operator	16.2.2.2
	Clarified description of Scope operator.	16.2.3.3.2.3
	Clarified NULL termination condition of ToString operator	16.2.3.4.2.44
	Added explanation of the use of the double-backslash within string literals.	16.2.3.6.3.2
	Define allowable format of the EISAID string.	16.2.3.6.4.1
	Define internal representation of Unicode strings.	16.2.3.6.4.3
	Corrected data type in DWORD address space descriptor.	16.2.4.13
	Corrected AML encoding of LGreaterEqualOp.	17.3
Mar. 2002	Errata and clarifications added. ACPI 2.0 Errata Document Revision 1.0 through 1.5 integrated.	See below.
2.0a	Clarifies that GAS fields other than Address_Space_ID may be used as indicated by the CPU manufacturer. Updated table 5-2.	5.2.3.1
	Split DESCRIPTION HEADER signature table into two tables. One for signatures of ACPI defined tables and one for signatures reserved by ACPI.	5.2.5
	Added “CPEP”, “HPET, and “TCPA” to ACPI reserved signature table 5-5a.	
	Update references to “DBGP”, “SPMI”, “SLIT”, and “SPCR” reserved signatures in table 5-5a.	
	Added signature reservation email address.	
	Clarified description/use of Firmware_Waking_Vector fields in table 5-11.	5.2.9
	Clarified Local APIC Address Override Structure - Local APIC Address field value for Itanium^{TM}-based systems.	5.2.10.11
	Clarified handling of GPEs and calling of _TXX and _QXX methods and added description of _QXX methods for handling SMBus alarms.	5.6.2.2.2

	Described SMBus alarms and added example of _QXX method for SMBus alarm.	5.6.2.2.3
	Added SMBus Alarms to _Qxx description in Table 5.49.	5.6.5
	Clarified _DDN description.	6.1.3
	Corrected _TRA field description (swapped primary and secondary)	6.4.3.5.1, 6.4.3.5.2, 6.4.3.5.3
	Clarfied firmware responsibilities during initialization and wake.	9.3
	Clarified operation of bi-directional data transfer for BufferAcc AccessType.	14.5
	Corrected examples to clarify that Store cannot change the value of the Source argument, which should is read only. Corrected value of BitIndex for CreateField in examples. Added examples of using BufferAcc fields with opcodes other than Store.	14.6,14.7
	Added description of arrow (=>) notation in table 16-1.	16.1.1
	Clarification of the purpose and use of the Type 3, 4, and 5 opcodes. Added AccessAttribKeyword SMBBlockProcessCall.	16.1.3
	Added Nothing term to definition of Return statement.	16.1.3 16.2.3.4.1.12
	Added return values to Package and Buffer definitions.	16.1.3 16.2.3.6.1 16.2.3.6.2
	Noted the exception in the handling of execution result for BufferAcc fields.	16.2.3.4
	Added Processor object to description of Notify.	16.2.3.4.1.9
	Clarified Buffer-to-String conversion in ToDecimalString operator description.	16.2.3.4.1.10
	Corrected Switch statement definition by adding TermArg.	16.2.3.4.1.16
	Corrected names of conversion operators in Switch operator description.	16.2.3.4.1.16
	Corrected ToBuffer description for the Integer source case from 4 to 8 bytes.	16.2.3.4.2.4
	Added case where the source operand is an Integer to the ToInteger description.	16.2.3.4.2.19
	Noted the exception in the handling of execution result for BufferAcc fields.	16.2.3.4.2.43
	Added non-null ASCII restriction on hex and octal escape sequence values within ASL String Literals.	16.2.3.6.3.2
	Corrected Resource Descriptor Field example.	16.2.4
	Added text to describe the type of offset (bit or byte) returned by a reference to a named Resource Descriptor Field.	16.2.4
	Add AccessAttrib SMBBlockProcessCall encoding information.	17.2.4.2
	Corrected names for operators and added definition of LoadTable.	17.2.4.4
	Table 17.2: Corrected names for various operators, corrected definition of LoadTable.	17.3
	Corrected _BCL example code to add return statement.	B.5.2
ACPI 2.0	Clarified Power / Sleep button override action will cause system to enter soft-	1.5

Errata Doc. Rev. 1.5	off state. Override action will not cause the system to reset.	
	Updated “SRAT” DESCRIPTION_HEADER signature reference.	5.2.5
	Replaced ASL Data Type section with a new section that clarifies ASL Data Type conversions.	16.2.2
ACPI 2.0 Errata Doc. Rev. 1.4	Corrected Figure 5-1 location of system description tables. Removed redundant description of finding the RSDP on IA-PC systems – added reference to other sections.	5.1
	Corrected FADT Boot Architecture Flags Reserved field bit offset from 3 to 2.	5.2.8.3
	Clarified _INI object evaluation – OSPM evaluates _SB._INI	6.5.1
	Corrected ElseTerm definition. Changed CMOS RegionSpaceKeyword to SystemCMOS to avoid collisions with existing ASL.	16.1.3
	Corrected description of Mutex object.	16.2.3.3.1.14
	Changed ASL CopyTerm to CopyObjectTerm to avoid collision with existing ASL.	16.2.3.4.2.8
ACPI 2.0 Errata Doc. Rev. 1.3	Corrected location of Firmware ACPI Control Structure may exist anywhere in the system’s memory address map.	5.2.9
	Corrected description of the Local APIC Address Override Structure.	5.2.10.11
	Corrected Local SAPIC Structure’s ACPI Processor ID field length from two bytes to one byte to enable a correct comparison with processor term’s ProcessorID field. Rearranged field ordering to more closely match the Local APIC Structure.	5.2.10.13
	Corrected _SCP reference section.	5.6.5
	Corrected TermArg and NameTerm to reference DataObject rather than DataRefObject. Added NameString to TermArg. Added missing DDBHandle and ObjectReference to ASL type definitions.	16.1.3
	Corrected Load and Unload operator descriptions – does not apply to Differentiated Definition Block	16.2.3.4.1.7, 16.2.3.4.1.17
	Corrected table reference.	16.2.3.4.2.37
ACPI 2.0 Errata Doc. Rev. 1.2	Clarified that OSPM is only required to write non-zero values of FADT fields PSTATE_CNT and CST_CNT to the SMI Command Port. Corrected PM1_CNT_LEN value is ≥ 2.	5.2.8
	Changed ASL type conversion function names to avoid collision with existing ASL (Buff >ToBuffer, DecStr>ToDecimalString, HexStr>ToHexString, Int>ToInteger, String>ToString).	16.1.3, 16.2.3.4.2, 16.2.3.4.2.4, 16.2.3.4.2.10, 16.2.3.4.2.16, 16.2.3.4.2.19, 16.2.3.4.2.44
ACPI 2.0 Errata Doc. Rev. 1.1	Clarified hardware interfaces may be defined as Functional Fixed Hardware **only when directed by the CPU manufacturer** as proprietary OS support is required that must be coordinated with the OS vendor.	4.1.1
	Clarified Definition Block support expanding from 32-bit to 64-bit integers.	5.2.10, 5.2.10.1, 5.2.10.2
	Local SAPIC Structure length corrected to 8 from 10 bytes.	5.2.10.13

	Updated DSDT DefinitionBlock example compliance revision.	5.5
	End value correction of event values for status bits in GPE0_BLK.	5.6.2.2
	Corrected Defined Generic Object and Control Method section references.	5.6.5
	Corrected Generic Register Descriptor Definition to include GAS reserved field.	6.4.3.7
	Corrected memory term's type field from TranslationType to Type	16.1.3
	Corrected Switch ACPI 1.0 translation	16.2.3.4.1.16
ACPI 2.0 Errata Doc. Rev. 1.0	Re-inserted mistakenly deleted sentence fragment.	5.2
	FADT SCI_INT field - clarified to be the SCI interrupts's Global System Interrupt number when no 8259 exists in the system.	5.2.8
	Incorrect reference to Processor declaration section.	5.2.10.5
	Local APIC Address Override Structure length field corrected.	5.2.10.11
	I/O SAPIC Strucure - length field corrected, Global System Interrupt Base and I/O SAPIC Address field descriptions expanded/clarified.	5.2.10.12
	Local SAPIC Structure flags length corrected to 4 from 2. Other offsets adjusted accordingly. Incorrect reference to Processor declaration section.	5.2.10.13
	_CS4 critical thermal trip point renamed to _HOT	12.4, 12.5
	Corrected Embedded Controller method name - removed trailing numbers	14.2
	LNOT(Logical Not) evaluation result correction.	16.2.3.4.2.26
	ASL macro for fixed I/O port descriptor listed incorrectly in previous section.	16.2.4.5, 16.2.4.6
	AML Root-Path only encoding for NamePath was missing as was NullName	17.2.1
Aug. 2000 2.0	Major specification revision. 64-bit addressing support added. Processor and device performance state support added. Numerous multiprocessor workstation and server-related enhancements. Consistency and readability enhancements throughout.	
Feb. 1999 1.0b	Fixed previous errata that deleted wrong paragraph in the RTC_EN description.	4.7.3.1.2
	Clarified P_BLK requirements on multiprocessor systems.	4.7.2.6.3
	Changed definition of SCI_INT pin in Table 5-5.	5.2.5
	Replaced section 5.2.8, adding new structures and clarifications to support multiprocessor configurations.	5.2.8
	Expanded Name Space description—clarified the name search rules, added Parent operator to operator list, described name padding.	5.3
	Expanded ASL definition—defined global objects, clarification that OpRegion accesses may block, added description of the scope and life of variables in control methods.	5.5.3
	Changed notify values.	5.6.3
	Added _PIC method to Table 5-33 and new section 5.8.	5.6.5 & 5.8
	Added USB _ADR values to Table 6-1.	6.1.1

	ACPI Control Method added for floppy enumeration (_FDE).	10.8
	ASL Grammar clarifications—initial and default SyncLevel values, ObjectType behavior for specific objects, usage of the RefOf operator and behavior of non-package method evaluation.	15.2.3
	Added top-level AML definition.	16.2
	Changed concat arguments to be TermArgs resolving to data.	16.2.4.4
	Added the _GLK object and referenced it in the Smart Battery and the Control Method Battery sections.	6.5.6 & 11.1.4 & 11.2.2 & 13.8 & 13.9 &13.12
	Added Video Extensions as an Appendix.	Appendix A
1.0a	Added _PRT requirement for PCI root bridges.	1.7
	Clarification H/W behavior—PM timer may be stopped when not in the G0/S0 state, Lid Switch behavior and correction of the RTC_EN bit in Table 4-10.	4.7.2.1
	Clarification of tables—trailing blank required in signature in Table 5-1, FLUSH_SIZE and FLUSH_STRIDE clarification Table 5-5.	5.2.x
	Clarified placement of APIC-related structures and general clean up, added Interrupt source overrides.	5.2.8
	Various removals—Figure 5-4, DCK_CAP flag from Table 5-6, _SBC and _SBS methods from Table 5-33.	

(continued)

Revision	Change Description	Affected Sections
	Various additions—AC device PnP ID to Table 5-32, _DDN (logical name association) to Table 6-1, _ADR values for floppy, _FDI–floppy configuration info, requirements for _CRS used with bus devices, battery presence bit to _STA definition, QWORD to Large Resource data type, _INI Method.	5.6.4
	Wake/Sleep clarifications—_PTS not executed for S5 and SCI cannot occur before enabled.	9.1 & 9.3
	Rewrote the IDE Controller Device section.	10.8
	Corrected the passive cooling equation for TC1 and TC2.	12.3.7 (&8)
	Removed requirement that PRx contain numeric lowest state.	7.2.x (0-2)
	Removed Duplicate Section “General-Purpose Register Blocks.”	4.7.4.3
	Clarified that C1 is required and C2 & C3 are optional and reiterate requirement for C1 processor state in Table 5-6.	4.7.2.6 & 5.2.5
	Clarified the Passive Cooling Equation.	12.1.5
	Numerous grammar updates and corrections.	15 & 16
	Added SxD objects.	7.2 &7.2.x
1.0	Original Release.	

Contents

1 Introduction

The Advanced Configuration and Power Interface (ACPI) specification was developed to establish industry common interfaces enabling robust operating system (OS)-directed motherboard device configuration and power management of both devices and entire systems. ACPI is the key element in Operating System-directed configuration and Power Management (OSPM).

ACPI evolves the existing collection of power management BIOS code, Advanced Power Management (APM) application programming interfaces (APIs, PNPBIOS APIs, Multiprocessor Specification (MPS) tables and so on into a well-defined power management and configuration interface specification. ACPI provides the means for an orderly transition from existing (legacy) hardware to ACPI hardware, and it allows for both ACPI and legacy mechanisms to exist in a single machine and to be used as needed.

Further, new system architectures are being built that stretch the limits of current Plug and Play interfaces. ACPI evolves the existing motherboard configuration interfaces to support these advanced architectures in a more robust, and potentially more efficient manner.

The interfaces and OSPM concepts defined within this specification are suitable to all classes of computers including (but not limited to) desktop, mobile, workstation, and server machines. From a power management perspective, OSPM/ACPI promotes the concept that systems should conserve energy by transitioning unused devices into lower power states including placing the entire system in a low-power state (sleeping state) when possible.

This document describes ACPI hardware interfaces, ACPI software interfaces and ACPI data structures that, when implemented, enable support for robust OS-directed configuration and power management (OSPM).

1.1 Principal Goals

ACPI is the key element in implementing OSPM. ACPI-defined interfaces are intended for wide adoption to encourage hardware and software vendors to build ACPI-compatible (and, thus, OSPM-compatible) implementations.

The principal goals of ACPI and OSPM are to:

1. Enable all computer systems to implement motherboard configuration and power management functions, using appropriate cost/function tradeoffs.
    - Computer systems include (but are not limited to) desktop, mobile, workstation, and server machines.
    - Machine implementers have the freedom to implement a wide range of solutions, from the very simple to the very aggressive, while still maintaining full OS support.
    - Wide implementation of power management will make it practical and compelling for applications to support and exploit it. It will make new uses of PCs practical and existing uses of PCs more economical.
2. Enhance power management functionality and robustness.
    - Power management policies too complicated to implement in a ROM BIOS can be implemented and supported in the OS, allowing inexpensive power managed hardware to support very elaborate power management policies.
    - Gathering power management information from users, applications, and the hardware together into the OS will enable better power management decisions and execution.
    - Unification of power management algorithms in the OS will reduce conflicts between the firmware and OS and will enhance reliability.

3. Facilitate and accelerate industry-wide implementation of power management.
    - OSPM and ACPI will reduce the amount of redundant investment in power management throughout the industry, as this investment and function will be gathered into the OS. This will allow industry participants to focus their efforts and investments on innovation rather than simple parity.
    - The OS can evolve independently of the hardware, allowing all ACPI-compatible machines to gain the benefits of OS improvements and innovations.
4. Create a robust interface for configuring motherboard devices.
    - Enable new advanced designs not possible with existing interfaces.

1.2 Power Management Rationale

It is necessary to move power management into the OS and to use an abstract interface (ACPI) between the OS and the hardware to achieve the principal goals set forth above.

- Minimal support for power management inhibits application vendors from supporting or exploiting it.
    - Moving power management functionality into the OS makes it available on every machine on which the OS is installed. The level of functionality (power savings, and so on) varies from machine to machine, but users and applications will see the same power interfaces and semantics on all OSPM machines.
    - This will enable application vendors to invest in adding power management functionality to their products.
- Legacy power management algorithms were restricted by the information available to the BIOS that implemented them. This limited the functionality that could be implemented.
    - Centralizing power management information and directives from the user, applications, and hardware in the OS allows the implementation of more powerful functionality. For example, an OS can have a policy of dividing I/O operations into normal and lazy. Lazy I/O operations (such as a word processor saving files in the background) would be gathered up into clumps and done only when the required I/O device is powered up for some other reason. A non-lazy I/O request made when the required device was powered down would cause the device to be powered up immediately, the non-lazy I/O request to be carried out, and any pending lazy I/O operations to be done. Such a policy requires knowing when I/O devices are powered up, knowing which application I/O requests are lazy, and being able to assure that such lazy I/O operations do not starve.
    - Appliance functions, such as answering machines, require globally coherent power decisions. For example, a telephone-answering application could call the OS and assert, “I am waiting for incoming phone calls; any sleep state the system enters must allow me to wake and answer the telephone in 1 second.” Then, when the user presses the “off” button, the system would pick the deepest sleep state consistent with the needs of the phone answering service.
- BIOS code has become very complex to deal with power management. It is difficult to make work with an OS and is limited to static configurations of the hardware.
    - There is much less state information for the BIOS to retain and manage (because the OS manages it).
    - Power management algorithms are unified in the OS, yielding much better integration between the OS and the hardware.
    - Because additional ACPI tables (Definition Blocks) can be loaded, for example, when a mobile system docks, the OS can deal with dynamic machine configurations.
    - Because the BIOS has fewer functions and they are simpler, it is much easier (and therefore cheaper) to implement and support.

- The existing structure of the PC platform constrains OS and hardware designs.
  - Because ACPI is abstract, the OS can evolve separately from the hardware and, likewise, the hardware from the OS.
  - ACPI is by nature more portable across operating systems and processors. ACPI control methods allow for very flexible implementations of particular features.

1.3 Legacy Support

ACPI provides support for an orderly transition from legacy hardware to ACPI hardware, and allows for both mechanisms to exist in a single machine and be used as needed.

Table 1-1 Hardware Type vs. OS Type Interaction

Hardware\OS	Legacy OS	OSPM/ACPI OS
Legacy hardware	A legacy OS on legacy hardware does what it always did.	If the OS lacks legacy support, legacy support is completely contained within the hardware functions.
Legacy and ACPI hardware support in machine	It works just like a legacy OS on legacy hardware.	During boot, the OS tells the hardware to switch from legacy to OSPM/ACPI mode and from then on, the system has full OSPM/ACPI support.
ACPI-only hardware	There is no power management.	There is full OSPM/ACPI support.

1.4 OEM Implementation Strategy

Any OEM is, as always, free to build hardware as they see fit. Given the existence of the ACPI specification, two general implementation strategies are possible:

- An original equipment manufacturer (OEM) can adopt the OS vendor-provided ACPI OSPM software and implement the hardware part of the ACPI specification (for a given platform) in one of many possible ways.
- An OEM can develop a driver and hardware that are not ACPI-compatible. This strategy opens up even more hardware implementation possibilities. However, OEMs who implement hardware that is OSPM-compatible but *not* ACPI-compatible will bear the cost of developing, testing, and distributing drivers for their implementation.

1.5 Power and Sleep Buttons

OSPM provides a new appliance interface to consumers. In particular, it provides for a sleep button that is a “soft” button that does *not* turn the machine physically off but signals the OS to put the machine in a soft off or sleeping state. ACPI defines two types of these “soft” buttons: one for putting the machine to sleep and one for putting the machine in soft off.

This gives the OEM two different ways to implement machines: A one-button model or a two-button model. The one-button model has a single button that can be used as a power button or a sleep button as determined by user settings. The two-button model has an easily accessible sleep button and a separate power button. In either model, an override feature that forces the machine to the soft-off state without OSPM interaction is also needed to deal with various rare, but problematic, situations.

1.6 ACPI Specification and the Structure Of ACPI

This specification defines ACPI hardware interfaces, ACPI software interfaces and ACPI data structures. This specification also defines the semantics of these interfaces.

Figure 1-1 lays out the software and hardware components relevant to OSPM/ACPI and how they relate to each other. This specification describes the *interfaces between* components, the contents of the ACPI System Description Tables, and the related semantics of the other ACPI components. Notice that the ACPI System Description Tables, which describe a particular platform's hardware, are at heart of the ACPI implementation and the role of the ACPI System Firmware is primarily to supply the ACPI Tables (rather than a native instruction API).

ACPI is *not* a software specification; it is *not* a hardware specification, although it addresses both software and hardware and how they must behave. ACPI is, instead, an interface specification comprised of both software and hardware elements.

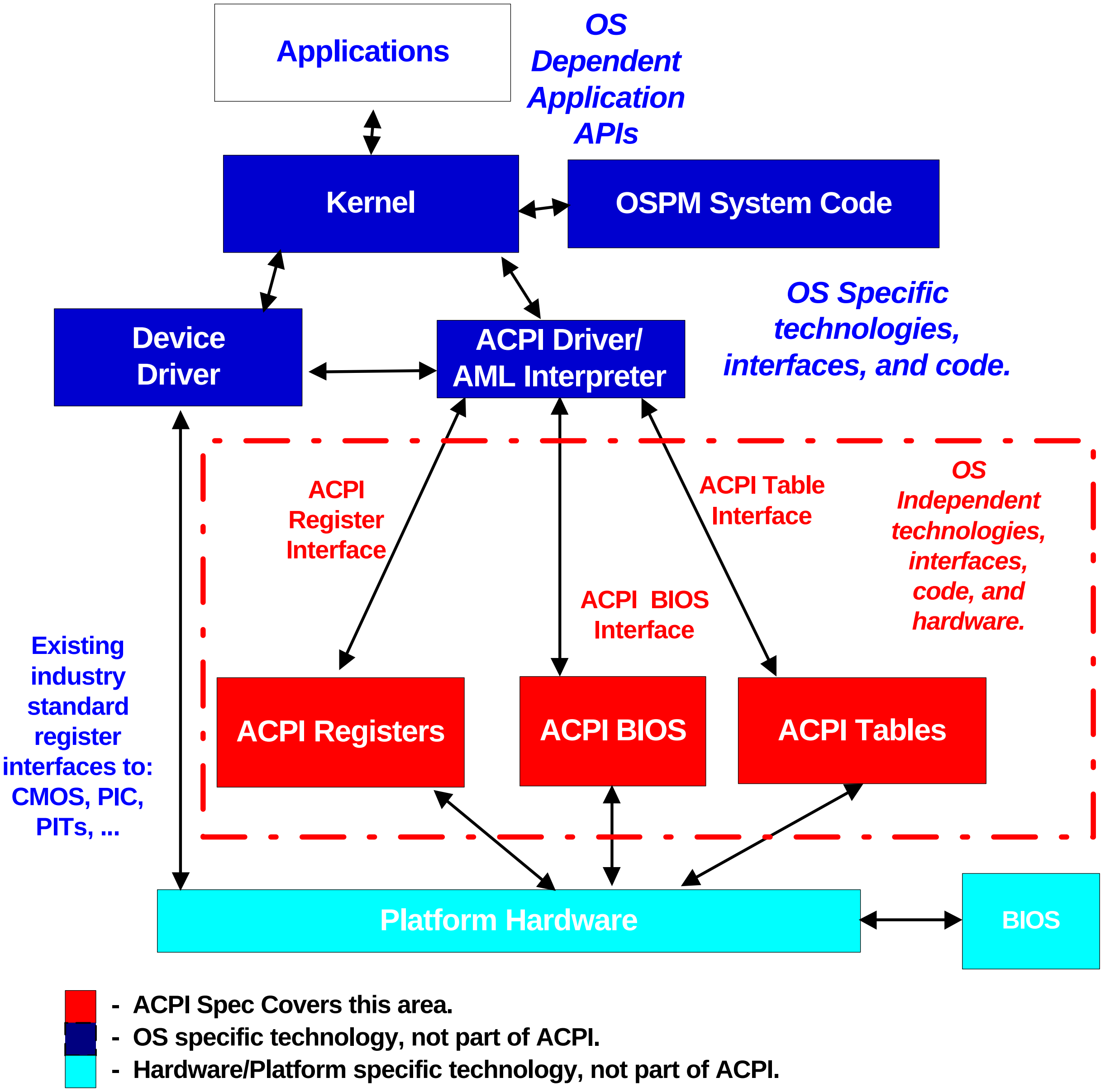
Applications
OS Dependent Application APIs
Kernel
OSPM System Code
Device Driver
ACPI Driver/ AML Interpreter
OS Specific technologies, interfaces, and code.
ACPI Register Interface
ACPI Table Interface
OS Independent technologies, interfaces, code, and hardware.
ACPI BIOS Interface
Existing industry standard register interfaces to: CMOS, PIC, PITs, ...
ACPI Registers
ACPI BIOS
ACPI Tables
Platform Hardware
BIOS
🟥 - ACPI Spec Covers this area.
🟦 - OS specific technology, not part of ACPI.
🟦 - Hardware/Platform specific technology, not part of ACPI.

Figure 1-1 OSPM/ACPI Global System

There are three run-time components to ACPI:

- **ACPI System Description Tables.** Describe the interfaces to the hardware. Some descriptions limit what can be built (for example, some controls are embedded in fixed blocks of registers and the table specifies the address of the register block). Most descriptions allow the hardware to be built in arbitrary ways and can describe arbitrary operation sequences needed to make the hardware function. ACPI Tables containing “Definition Blocks” can make use of a pseudocode type of language, the interpretation of which is performed by the OS. That is, OSPM contains and uses an interpreter that executes procedures encoded in the pseudocode language and stored in the ACPI tables containing “Definition Blocks.” The pseudocode language, known as ACPI Machine Language (AML), is a compact, tokenized, abstract type of machine language.
- **ACPI Registers.** The constrained part of the hardware interface, described (at least in location) by the ACPI System Description Tables.
- **ACPI System Firmware.** Refers to the portion of the firmware that is compatible with the ACPI specifications. Typically, this is the code that boots the machine (as legacy BIOSs have done) and implements interfaces for sleep, wake, and some restart operations. It is called rarely, compared to a legacy BIOS. The ACPI Description Tables are also provided by the ACPI System Firmware.

1.7 OS and Platform Compliance

The ACPI 2.0 specification contains only interface specifications. ACPI 2.0 does not contain any platform compliance requirements. The following sections provide guidelines for class specific platform implementations that reference ACPI-defined interfaces and guidelines for enhancements that operating systems may require to completely support OSPM/ACPI. The minimum feature implementation requirements of an ACPI-compatible OS are also provided.

1.7.1 Platform Implementations of ACPI-defined Interfaces

System platforms implement ACPI-defined hardware interfaces via the platform hardware and ACPI-defined software interfaces and system description tables via the ACPI system firmware. Specific ACPI-defined interfaces and OSPM concepts while appropriate for one class of machine (for example, a mobile system), may not be appropriate for another class of machine (for example, a multi-domain enterprise server). It is beyond the capability and scope of this specification to specify all platform classes and the appropriate ACPI-defined interfaces that should be required for the platform class.

Platform design guide authors are encouraged to require the appropriate ACPI-defined interfaces and hardware requirements suitable to the particular system platform class addressed in a particular design guide. Platform design guides should not define alternative interfaces that provide similar functionality to those defined in the ACPI specification.

1.7.1.1 Recommended Features and Interface Descriptions for Design Guides

Common description text and category names should be used in design guides to describe all features, concepts, and interfaces defined by the ACPI 2.0 specification as requirements for a platform class. Listed below is the recommended set of high-level text and category names to be used to describe the features, concepts, and interfaces defined by ACPI 2.0.

Note: Where definitions or relational requirements of interfaces are localized to a specific section, the section number is provided. The interface definitions and relational requirements of the interfaces specified below are generally spread throughout the ACPI specification. The ACPI specification defines:

System address map reporting interfaces (Section 15)

ACPI System Description Tables (Section 5.2):

Root System Description Pointer (RSDP)

System Description Table Header

Root System Description Table (RSDT)

Fixed ACPI Description Table (FADT)

Firmware ACPI Control Structure (FACS)

Differentiated System Description Table (DSDT)

Secondary System Description Table (SSDT)

Multiple APIC Description Table (MADT)

Smart Battery Table (SBST)

Extended System Description Table (XSDT)

Embedded Controller Boot Resources Table

ACPI-defined Fixed Registers Interfaces (Section 4, Section 5.2.8):

Power management timer control/status

Power or sleep button with S5 override (also possible in generic space)

Real time clock wakeup alarm control/status

SCI /SMI routing control/status for Power Management and General-purpose events

System power state controls (sleeping/wake control) (Section 9)

Processor power state control (c states) (Section 8)

Processor throttling control/status (Section 8)

Processor performance state control/status (Section 8)

General-purpose event control/status

Global Lock control/status

System Reset control (Section 4.7.3.6)

Embedded Controller control/status (Section 13)

SMBus Host Controller (HC) control/status (Section 14)

Smart Battery Subsystem (Section 11)

ACPI-defined Generic Register Interfaces and object definitions in the ACPI Namespace (Section 4.2, Section 5.6.5):

- *General-purpose event processing*
- *Motherboard device identification, configuration, and insertion/removal (Section 6)*
- *Thermal zones (Section 12)*
- *Power resource control (Section 7.1)*
- *Device power state control (Section 7.2)*
- *System power state control (Section 7.3)*
- *System indicators (Section 10.1)*
- *Devices and device controls (Section 10):*
  - *Processor (Section 8)*
  - *Control Method Battery (Section 11)*
  - *Smart Battery Subsystem (Section 11)*
  - *Mobile Lid*
  - *Power or sleep button with S5 override (also possible in fixed space)*
  - *Embedded controller (Section 13)*
  - *Fan*
  - *Generic Bus Bridge*
  - *IDE Controller*
  - *Floppy Controller*
  - *GPE Block*
  - *Module*
  - *Memory*
- *Global Lock related interfaces*

ACPI Event programming model (Section 5.6)

ACPI-defined System BIOS Responsibilities (Section 9)

ACPI-defined State Definitions (Section 2):

- *Global system power states (G-states, S0, S5)*
- *System sleeping states (S-states S1-S4) (Section 9)*
- *Device power states (D-states (Appendix B))*
- *Processor power states (C-states) (Section 8)*
- *Device and processor performance states (P-states) (Section 3, Section 8)*

1.7.1.2 Terminology Examples for Design Guides

The following provides an example of how a client platform design guide, whose goal is to require robust configuration and power management for the system class, could use the recommended terminology to define ACPI requirements.

Important: This example is provided as a guideline for how ACPI terminology can be used. It should not be interpreted as a statement of ACPI requirements.

Platforms compliant with this platform design guide must implement the following ACPI 2.0 defined system features, concepts, and interfaces, along with their associated event models:

System address map reporting interfaces

ACPI System Description Tables provided in the system firmware

ACPI-defined Fixed Registers Interfaces:

Power management timer control/status

Power or sleep button with S5 override (may also be implemented in generic register space)

Real time clock wakeup alarm control/status

General-purpose event control/status

SCI /SMI routing control/status for Power Management and General-purpose events

(control required only if system supports legacy mode)

System power state controls (sleeping/wake control)

Processor power state control (for C1)

Global Lock control/status (if Global Lock interfaces are required by the system)

- *ACPI-defined Generic Register Interfaces and object definitions in the ACPI Namespace:*
  General-purpose event processing
  Motherboard device identification, configuration, and insertion/removal (ACPI 2.0 Section 6)
  System power state control (ACPI 2.0 Section 7.3)
  Devices and device controls:
    Processor
    Control Method Battery (or Smart Battery Subsystem on a mobile system)
    Smart Battery Subsystem (or Control Method Battery on a mobile system)
    Power or sleep button with S5 override (may also be implemented in fixed register space)
  Global Lock related interfaces when a logical register in the hardware is shared between OS and firmware environments
- *ACPI Event programming model (ACPI 2.0 Section 5.6)*
- *ACPI-defined System BIOS Responsibilities (ACPI 2.0 Section 9)*
- *ACPI-defined State Definitions:*
  System sleeping states (At least one system sleeping state, S1-S4, must be implemented)
  Device power states (D-states must be implemented in accordance with device class specifications)
  Processor power states (All processors must support the C1 Power State)

The following provides an example of how a design guide for systems that execute multiple OS instances, whose goal is to require robust configuration and continuous availability for the system class, could use the recommended terminology to define ACPI related requirements.

Important: This example is provided as a guideline for how ACPI terminology can be used. It should not be interpreted as a statement of ACPI requirements.

Platforms compliant with this platform design guide must implement the following ACPI 2.0 defined system features and interfaces, along with their associated event models:

System address map reporting interfaces

ACPI System Description Tables provided in the system firmware

ACPI-defined Fixed Registers Interfaces:

Power management timer control/status

General-purpose event control/status

SCI /SMI routing control/status for Power Management and General-purpose events

(control required only if system supports legacy mode)

System power state controls (sleeping/wake control)

Processor power state control (for C1)

Global Lock control/status (if Global Lock interfaces are required by the system)

- *ACPI-defined Generic Register Interfaces and object definitions in the ACPI Namespace:*
  General-purpose event processing
  Motherboard device identification, configuration, and insertion/removal (ACPI 2.0 Section 6)
  System power state control (ACPI 2.0 Section 7.3)
  System indicators
  Devices and device controls:
    Processor
  Global Lock related interfaces when a logical register in the hardware is shared between OS and firmware environments
  - *ACPI Event programming model (ACPI 2.0 Section 5.6)*
  - *ACPI-defined System BIOS Responsibilities (ACPI 2.0 Section 9)*
  - *ACPI-defined State Definitions:*
    Processor power states (All processors must support the C1 Power State)

1.7.2 OSPM Implementations

OS enhancements are needed to support ACPI-defined features, concepts, and interfaces, along with their associated event models appropriate to the system platform class upon which the OS executes. This is the implementation of OSPM. The following outlines the OS enhancements and elements necessary to support all ACPI-defined interfaces. To support ACPI through the implementation of OSPM, the OS needs to be modified to:

- Use system address map reporting interfaces.
- Find and consume the ACPI System Description Tables.
- Interpret ACPI machine language (AML).
- Enumerate and configure motherboard devices described in the ACPI Namespace.
- Interface with the power management timer.
- Interface with the real-time clock wake alarm.
- Enter ACPI mode (on legacy hardware systems).
- Implement device power management policy.
- Implement power resource management.
- Implement processor power states in the scheduler idle handlers.
- Control processor and device performance states.
- Implement the ACPI thermal model.
- Support the ACPI Event programming model including handling SCI interrupts, managing fixed events, general-purpose events, embedded controller interrupts, and dynamic device support.
- Support acquisition and release of the Global Lock.
- Use the reset register to reset the system.
- Provide APIs to influence power management policy.
- Implement driver support for ACPI-defined devices.
- Implement APIs supporting the system indicators.
- Support all system states S1–S5.

1.7.3 OS Requirements

The following list describes the minimum requirements for an OSPM/ACPI-compatible OS:

- Use system address map reporting interfaces to get the system address map on Intel Architecture (IA) platforms:
  INT 15H, E820H - Query System Address Map interface (see section 15, "System Address Map Interfaces")
  EFI GetMemoryMap() Boot Services Function (see section 15, "System Address Map Interfaces")
- Find and consume the ACPI System Description Tables (see section 5, "ACPI Software Programming Model").
- Implementation of an AML interpreter supporting all defined AML grammar elements (see section 17, ACPI Machine Language Specification").
- Support for the ACPI Event programming model including handling SCI interrupts, managing fixed events, general-purpose events, embedded controller interrupts, and dynamic device support.
- Enumerate and configure motherboard devices described in the ACPI Namespace.
- Implement support for the following ACPI devices defined within this specification:
  Embedded Controller Device (see section 13, "ACPI Embedded Controller Interface Specification")
  GPE Block Device (see section 10.10, "GPE Block Device")
  Module Device (see section 10.11, "Module Device")
- Implementation of the ACPI thermal model (see section 12, "Thermal Management").
- Support acquisition and release of the Global Lock.
- OS-directed power management support (device drivers are responsible for maintaining device context as described by the Device Power Management Class Specifications described in Appendix A).

1.8 Target Audience

This specification is intended for the following users:

- OEMs building hardware containing ACPI-compatible interfaces
- Operating system and device driver developers
- BIOS and ACPI system firmware developers
- CPU and chip set vendors
- Peripheral vendors

1.9 Document Organization

The ACPI specification document is organized into the following four parts:

- The first part of the specification (sections 1 through 3) introduces ACPI and provides an executive overview.
- The second part (sections 4 and 5) defines the ACPI hardware and software programming models.
- The third part (sections 6 through 15) specifies the ACPI implementation details; this part of the specification is primarily for developers.
- The fourth part (sections 16 and 17) is technical reference material; section 16 is the ACPI Source Language (ASL) reference, parts of which are referred to by most of the other sections in the document.
- Appendices contain device class specifications, describing power management characteristics of specific classes of devices, and device class-specific ACPI interfaces.

1.9.1 ACPI Overview

The first three sections of the specification provide an executive overview of ACPI.

- **Section 1: Introduction.** Discusses the purpose and goals of the specification, presents an overview of the ACPI-compatible system architecture, specifies the minimum requirements for an ACPI-compatible system, and provides references to related specifications.
- **Section 2: Definition of Terms.** Defines the key terminology used in this specification. In particular, the global system states (Mechanical Off, Soft Off, Sleeping, Working, and Non-Volatile Sleep) are defined in this section, along with the device power state definitions: Off (D3), D2, D1, and Fully-On (D0). Device and processor performance states (P0, P1, …Pn) are also discussed.
- **Section 3: Overview.** Gives an overview of the ACPI specification in terms of the functional areas covered by the specification: system power management, device power management, processor power management, Plug and Play, handling of system events, battery management, and thermal management.

1.9.2 Programming Models

Sections 4 and 5 define the ACPI hardware and software programming models. This part of the specification is primarily for system designers, developers, and project managers.

All of the implementation-oriented, reference, and platform example sections of the specification that follow (all the rest of the sections of the specification) are based on the models defined in sections 4 and 5. These sections are the heart of the ACPI specification. There are extensive cross-references between the two sections.

- **Section 4: ACPI Hardware Specification.** Defines a set of hardware interfaces that meet the goals of this specification.
- **Section 5: ACPI Software Programming Model.** Defines a set of software interfaces that meet the goals of this specification.

1.9.3 Implementation Details

The third part of the specification defines the implementation details necessary to actually build components that work on an ACPI-compatible platform. This part of the specification is primarily for developers.

- **Section 6: Configuration.** Defines the reserved Plug and Play objects used to configure and assign resources to devices, and share resources and the reserved objects used to track device insertion and removal. Also defines the format of ACPI-compatible resource descriptors.
- **Section 7: Power and Performance Management.** Defines the reserved device power-management objects and the reserved-system power-management objects.
- **Section 8: Processor Control.** Defines how the OS manages the processors’ power consumption and other controls while the system is in the working state.
- **Section 9: Waking and Sleeping.** Defines in detail the transitions between system working and sleeping states and their relationship to wake events. Refers to the reserved objects defined in sections 6, 7, and 8.
- **Section 10: ACPI-Specific Device Objects.** Lists the integrated devices that need support for some device-specific ACPI controls, along with the device-specific ACPI controls that can be provided. Most device objects are controlled through generic objects and control methods and have generic device IDs; this section discusses the exceptions.
- **Section 11: Power Source Devices.** Defines the reserved battery device and AC adapter objects.
- **Section 12: Thermal Management.** Defines the reserved thermal management objects.
- **Section 13: ACPI Embedded Controller Interface Specification.** Defines the interfaces between an ACPI-compatible OS and an embedded controller.
- **Section 14: ACPI System Management Bus Interface Specification.** Defines the interfaces between an ACPI-compatible OS and a System Management Bus (SMBus host controller.

1.9.4 Technical Reference

The fourth part of the specification contains reference material for developers.

- **Section 15: System Address Map Interfaces.** Explains the special INT 15 call for use in ISA/EISA/PCI bus-based systems. This call supplies the OS with a clean memory map indicating address ranges that are reserved and ranges that are available on the motherboard. Also describes memory devices.
- **Section 16: ACPI Source Language Reference.** Defines the syntax of all the ASL statements that can be used to write ACPI control methods, along with example syntax usage.
- **Section 17: ACPI Machine Language Specification.** Defines the grammar of the language of the ACPI virtual machine language. An ASL translator (compiler) outputs AML.
- **Appendix A: Device class specifications.** Describes device-specific power management behavior on a per device-class basis.
- **Appendix B: Video Extensions.** Contains video device class-specific ACPI interfaces.

1.10 Related Documents

Power management and Plug and Play specifications for legacy hardware platforms are the following, available from http://www.microsoft.com/hwdev/specs/:

- *Advanced Power Management (APM) BIOS Specification*, Revision 1.2.
- *Plug and Play BIOS Specification*, Version 1.0a.

Intel Architecture specifications are available from http://developer.intel.com:

Intel® Itanium^{TM} Architecture Software Developer's Manual, Volumes 1–4, Revision 1.0, Intel Corporation, January 2000.

Itanium^{TM} Processor Family System Abstraction Layer Specification, Intel Corporation, July 2001.

Extensible Firmware Interface Specification, Version 1.10, March 2002.

Documentation and specifications for the Smart Battery System components and the SMBus are available from http://www.sbs-forum.org:

- *Smart Battery Charger Specification*, Revision 1.1, Smart Battery System Implementers Forum, December, 1998.
- *Smart Battery Data Specification*, Revision 1.1, Smart Battery System Implementers Forum, December, 1998.
- *Smart Battery Selector Specification*, Revision 1.1, Smart Battery System Implementers Forum, December, 1998.
- *Smart Battery System Manager Specification*, Revision 1.0, Smart Battery System Implementers Forum, December, 1998.
- *System Management Bus Specification*, Revision 1.1, Smart Battery System Implementers Forum, December, 1998.

2 Definition of Terms

This specification uses a particular set of terminology, defined in this section. This section has three parts:

General ACPI terms are defined and presented alphabetically.

The ACPI global system states (working, sleeping, soft off, and mechanical off) are defined. Global system states apply to the entire system, and *are* visible to the user.

The ACPI device power states are defined. Device power states are states of particular devices; as such, they are generally *not* visible to the user. For example, some devices may be in the off state even though the system as a whole is in the working state. Device states apply to any device on any bus.

2.1 General ACPI Terminology

Advanced Configuration and Power Interface (***ACPI)***

As defined in this document, ACPI is a method for describing hardware interfaces in terms abstract enough to allow flexible and innovative hardware implementations and concrete enough to allow shrink-wrap OS code to use such hardware interfaces.

ACPI Hardware

Computer hardware with the features necessary to support OSPM and with the interfaces to those features described using the Description Tables as specified by this document.

ACPI Namespace

A hierarchical tree structure in OS-controlled memory that contains named objects. These objects may be data objects, control method objects, bus/device package objects, and so on. The OS dynamically changes the contents of the namespace at run-time by loading and/or unloading definition blocks from the ACPI Tables that reside in the ACPI BIOS. All the information in the ACPI Namespace comes from the Differentiated System Description Table (DSDT), which contains the Differentiated Definition Block, and one or more other definition blocks.

ACPI Machine Language (AML)

Pseudocode for a virtual machine supported by an ACPI-compatible OS and in which ACPI control methods and objects are written. The AML encoding definition is provided in section 17, “ACPI Machine Language (AML) Specification.”

Advanced Programmable Interrupt Controller (APIC)

An interrupt controller architecture commonly found on Intel Architecture-based 32-bit PC systems. The APIC architecture supports multiprocessor interrupt management (with symmetric interrupt distribution across all processors), multiple I/O subsystem support, 8259A compatibility, and inter-processor interrupt support. The architecture consists of local APICs commonly attached directly to processors and I/O APICs commonly in chip sets.

ACPI Source Language (ASL)

The programming language equivalent for AML. ASL is compiled into AML images. The ASL statements are defined in section 16, “ACPI Source Language (ASL) Reference.”

Control Method

A control method is a definition of how the OS can perform a simple hardware task. For example, the OS invokes control methods to read the temperature of a thermal zone. Control methods are written in an encoded language called AML that can be interpreted and executed by the ACPI-compatible OS. An ACPI-compatible system must provide a minimal set of control methods in the ACPI tables. The OS provides a set of well-defined control methods that ACPI table developers can reference in their control methods. OEMs can support different revisions of chip sets with one BIOS by either including control methods in the BIOS that test configurations and respond as needed or including a different set of control methods for each chip set revision.

Central Processing Unit (CPU) or processor

The part of a platform that executes the instructions that do the work. An ACPI-compatible OS can balance processor performance against power consumption and thermal states by manipulating the processor performance controls. The ACPI specification defines a working state, labeled G0 (S0), in which the processor executes instructions. Processor sleeping states, labeled C1 through C3, are also defined. In the sleeping states, the processor executes no instructions, thus reducing power consumption and, potentially, operating temperatures. For more information, see section 8, “Processor Control.”

Definition Block

A definition block contains information about hardware implementation and configuration details in the form of data and control methods, encoded in AML. An OEM can provide one or more definition blocks in the ACPI Tables. One definition block must be provided: the Differentiated Definition Block, which describes the base system. Upon loading the Differentiated Definition Block, the OS inserts the contents of the Differentiated Definition Block into the ACPI Namespace. Other definition blocks, which the OS can dynamically insert and remove from the active ACPI Namespace, can contain references to the Differentiated Definition Block. For more information, see section 5.2.10, “Device Power States.”

Device

Hardware component outside the core chip set of a platform. Examples of devices are liquid crystal display (LCD) panels, video adapters, Intergrated Drive Electronics (IDE) CD-ROM and hard disk controllers, COM ports, and so on. In the ACPI scheme of power management, buses are devices. For more information, see section 3.3.2, “Device Power States.”

Device Context

The variable data held by the device; it is usually volatile. The device might forget this information when entering or leaving certain states (for more information, see section 2.3, “Device Power State Definitions.”), in which case the OS software is responsible for saving and restoring the information. Device Context refers to small amounts of information held in device peripherals. See *System Context.*

Differentiated System Description Table (DSDT)

An OEM must supply a DSDT to an ACPI-compatible OS. The DSDT contains the Differentiated Definition Block, which supplies the implementation and configuration information about the base system. The OS always inserts the DSDT information into the ACPI Namespace at system boot time and never removes it.

Extensible Firmware Interface (EFI)

An interface between the OS and the platform firmware. The interface is in the form of data tables that contain platform related information, and boot and run-time service calls that are available to the OS and loader. Together, these provide a standard environment for booting an OS.

Embedded Controller

The general class of microcontrollers used to support OEM-specific implementations, mainly in mobile environments. The ACPI specification supports embedded controllers in any platform design, as long as the microcontroller conforms to one of the models described in this section. The embedded controller performs complex low-level functions through a simple interface to the host microprocessor(s).

Embedded Controller Interface

A standard hardware and software communications interface between an OS driver and an embedded controller. This allows any OS to provide a standard driver that can directly communicate with an embedded controller in the system, thus allowing other drivers within the system to communicate with and use the resources of system embedded controllers (for example, Smart Battery and AML code). This in turn enables the OEM to provide platform features that the OS and applications can use.

Firmware ACPI Control Structure (FACS)

A structure in read/write memory that the BIOS uses for handshaking between the firmware and the OS. FACS is passed to an ACPI-compatible OS via the Fixed ACPI Description Table (FADT). The FACS contains the system’s hardware signature at last boot, the firmware waking vector, and the Global Lock.

Fixed ACPI Description Table (FADT)

A table that contains the ACPI Hardware Register Block implementation and configuration details the OS needs to direct management of the ACPI Hardware Register Blocks, as well as the physical address of the DSDT that contains other platform implementation and configuration details. An OEM must provide an FADT to an ACPI-compatible OS in the RSDT/XSDT. The OS always inserts the namespace information defined in the Differentiated Definition Block in the DSDT into the ACPI Namespace at system boot time, and the OS never removes it.

Fixed Features

A set of features offered by an ACPI interface. The ACPI specification places restrictions on where and how the hardware programming model is generated. All fixed features, if used, are implemented as described in this specification so that OSPM can directly access the fixed feature registers.

Fixed Feature Events

A set of events that occur at the ACPI interface when a paired set of status and event bits in the fixed feature registers are set at the same time. When a fixed feature event occurs, a system control interrupt (SCI is raised. For ACPI fixed feature events, OSPM (or an ACPI-aware driver) acts as the event handler.

Fixed Feature Registers

A set of hardware registers in fixed feature register space at specific address locations in system I/O address space. ACPI defines register blocks for fixed features (each register block gets a separate pointer from the FADT). For more information, see section 4.6, “ACPI Hardware Features.”

General-Purpose Event Registers

The general-purpose event registers contain the event programming model for generic features. All generic events generate SCIs.

Generic Feature

A generic feature of a platform is value-added hardware implemented through control methods and general-purpose events.

Global System States

Global system states apply to the entire system, and are visible to the user. The various global system states are labeled G0 through G3 in the ACPI specification. For more information, see section 2.2, “Global System State Definitions.”

Ignored Bits

Some unused bits in ACPI hardware registers are designated as “ignored” in the ACPI specification. Ignored bits are undefined and can return zero or one (in contrast to reserved bits, which always return zero). Software ignores ignored bits in ACPI hardware registers on reads and preserves ignored bits on writes.

Intel Architecture-Personal Computer (IA-PC)

A general descriptive term for computers built with processors conforming to the architecture defined by the Intel processor family based on the Intel Architecture instruction set and having an industry-standard PC architecture.

I/O APIC

An Input/Output Advanced Programmable Interrupt Controller routes interrupts from devices to the processor’s local APIC.

I/O SAPIC

An Input/Output Streamlined Advanced Programmable Interrupt Controller routes interrupts from devices to the processor’s local APIC.

Legacy

A computer state where power management policy decisions are made by the platform hardware/firmware shipped with the system. The legacy power management features found in today’s systems are used to support power management in a system that uses a legacy OS that does not support the OS-directed power management architecture.

Legacy Hardware

A computer system that has no ACPI or OSPM power management support.

Legacy OS

An OS that is not aware of and does not direct the power management functions of the system. Included in this category are operating systems with APM 1.*x* support.

Local APIC

A local Advanced Programmable Interrupt Controller receives interrupts from the I/O APIC.

Local SAPIC

A local Streamlined Advanced Programmable Interrupt Controller receives interrupts from the I/O SAPIC.

Multiple APIC Description Table (MADT)

The Multiple APIC Description Table (MADT) is used on systems supporting the APIC and SAPIC to describe the APIC implementation. Following the MADT is a list of APIC/SAPIC structures that declare the APIC/SAPIC features of the machine.

Object

The nodes of the ACPI Namespace are objects inserted in the tree by the OS using the information in the system definition tables. These objects can be data objects, package objects, control method objects, and so on. Package objects refer to other objects. Objects also have type, size, and relative name.

Object name

Part of the ACPI Namespace. There is a set of rules for naming objects.

Operating System-directed Power Management (OSPM)
A model of power (and system) management in which the OS plays a central role and uses global information to optimize system behavior for the task at hand.

Package
A set of objects.

Power Button
A user push button or other switch contact device that switches the system from the sleeping/soft off state to the working state, and signals the OS to transition to a sleeping/soft off state from the working state.

Power Management
Mechanisms in software and hardware to minimize system power consumption, manage system thermal limits, and maximize system battery life. Power management involves trade-offs among system speed, noise, battery life, processing speed, and alternating current (AC) power consumption. Power management is required for some system functions, such as appliance (for example, answering machine, furnace control) operations.

Power Resources
Resources (for example, power planes and clock sources) that a device requires to operate in a given power state.

Power Sources
The battery (including a UPS battery) and AC line powered adapters or power supplies that supply power to a platform.

Register Grouping
Consists of two register blocks (it has two pointers to two different blocks of registers). The fixed-position bits within a register grouping can be split between the two register blocks. This allows the bits within a register grouping to be split between two chips.

Reserved Bits
Some unused bits in ACPI hardware registers are designated as “Reserved” in the ACPI specification. For future extensibility, hardware-register reserved bits always return zero, and data writes to them have no side effects. OSPM implementations must write zeros to all reserved bits in enable and status registers and preserve bits in control registers.

Root System Description Pointer (RSDP)
An ACPI-compatible system must provide an RSDP in the system’s low address space. This structure’s only purpose is to provide the physical address of the RSDT.

Root System Description Table (RSDT)
A table with the signature ‘RSDT,’ followed by an array of physical pointers to other system description tables. The OS locates that RSDT by following the pointer in the RSDP structure.

Secondary System Description Table (SSDT)
SSDTs are a continuation of the DSDT. Multiple SSDTs can be used as part of a platform description. After the DSDT is loaded into the ACPI Namespace, each secondary description table with a unique OEM Table ID is loaded. This allows the OEM to provide the base support in one table, while adding smaller system options in other tables.
Note: Additional tables can only add data; they cannot overwrite data from previous tables.

Sleep Button
A user push button that switches the system from the sleeping/soft off state to the working state, and signals the OS to transition to a sleeping state from the working state.

Smart Battery Subsystem

A battery subsystem that conforms to the following specifications: Smart Battery and either Smart Battery System Manager or Smart Battery Charger and Selector—and the additional ACPI requirements.

Smart Battery Table

An ACPI table used on platforms that have a Smart Battery subsystem. This table indicates the energy-level trip points that the platform requires for placing the system into different sleeping states and suggested energy levels for warning the user to transition the platform into a sleeping state.

System Management Bus (SMBus)

A two-wire interface based upon the I²C protocol. The SMBus is a low-speed bus that provides positive addressing for devices, as well as bus arbitration.

SMBus Interface

A standard hardware and software communications interface between an OS bus driver and an SMBus controller.

Streamlined Advanced Programmable Interrupt Controller (SAPIC)

An advanced APIC commonly found on Intel Architecture-based 64-bit systems.

System Context

The volatile data in the system that is not saved by a device driver.

System Control Interrupt (SCI)

A system interrupt used by hardware to notify the OS of ACPI events. The SCI is an active, low, shareable, level interrupt.

System Management Interrupt (SMI)

An OS-transparent interrupt generated by interrupt events on legacy systems. By contrast, on ACPI systems, interrupt events generate an OS-visible interrupt that is shareable (edge-style interrupts will not work). Hardware platforms that want to support both legacy operating systems and ACPI systems must support a way of re-mapping the interrupt events between SMIs and SCIs when switching between ACPI and legacy models.

Thermal States

Thermal states represent different operating environment temperatures within thermal zones of a system. A system can have one or more thermal zones; each thermal zone is the volume of space around a particular temperature-sensing device. The transitions from one thermal state to another are marked by trip points, which are implemented to generate an SCI when the temperature in a thermal zone moves above or below the trip point temperature.

Extended Root System Description Table (XSDT)

The XSDT provides identical functionality to the RSDT but accommodates physical addresses of DESCRIPTION HEADERs that are larger than 32-bits. Notice that both the XSDT and the RSDT can be pointed to by the RSDP structure.

2.2 Global System State Definitions

Global system states (G*x* states) apply to the entire system and are visible to the user.

Global system states are defined by six principal criteria:

- Does application software run?
- What is the latency from external events to application response?
- What is the power consumption?
- Is an OS reboot required to return to a working state?
- Is it safe to disassemble the computer?
- Can the state be entered and exited electronically?

Following is a list of the system states:

G3 Mechanical Off

A computer state that is entered and left by a mechanical means (for example, turning off the system's power through the movement of a large red switch). Various government agencies and countries require this operating mode. It is implied by the entry of this off state through a mechanical means that no electrical current is running through the circuitry and that it can be worked on without damaging the hardware or endangering service personnel. The OS must be restarted to return to the Working state. No hardware context is retained. Except for the real-time clock, power consumption is zero.

G2/S5 Soft Off

A computer state where the computer consumes a minimal amount of power. No user mode or system mode code is run. This state requires a large latency in order to return to the Working state. The system's context will not be preserved by the hardware. The system must be restarted to return to the Working state. It is not safe to disassemble the machine in this state.

G1 Sleeping

A computer state where the computer consumes a small amount of power, user mode threads are not being executed, and the system "appears" to be off (from an end user's perspective, the display is off, and so on). Latency for returning to the Working state varies on the wake environment selected prior to entry of this state (for example, whether the system should answer phone calls). Work can be resumed without rebooting the OS because large elements of system context are saved by the hardware and the rest by system software. It is not safe to disassemble the machine in this state.

G0 Working

A computer state where the system dispatches user mode (application) threads and they execute. In this state, peripheral devices (peripherals) are having their power state changed dynamically. The user can select, through some UI, various performance/power characteristics of the system to have the software optimize for performance or battery life. The system responds to external events in real time. It is not safe to disassemble the machine in this state.

S4 Non-Volatile Sleep

A special global system state that allows system context to be saved and restored (relatively slowly) when power is lost to the motherboard. If the system has been commanded to enter S4, the OS will write all system context to a file on non-volatile storage media and leave appropriate context markers. The machine will then enter the S4 state. When the system leaves the Soft Off or Mechanical Off state, transitioning to Working (G0) and restarting the OS, a restore from a NVS file can occur. This will only happen if a valid non-volatile sleep data set is found, certain aspects of the configuration of the machine have not changed, and the user has not manually aborted the restore. If all these conditions are met, as part of the OS restarting, it will reload the system context and activate it. The net effect for the user is what looks like a resume from a Sleeping (G1) state (albeit slower). The aspects of the machine configuration that must not change include, but are not limited to, disk layout and memory size. It might be possible for the user to swap a PC Card or a Device Bay device, however.

Notice that for the machine to transition directly from the Soft Off or Sleeping states to S4, the system context must be written to non-volatile storage by the hardware; entering the Working state first so that the OS or BIOS can save the system context takes too long from the user’s point of view. The transition from Mechanical Off to S4 is likely to be done when the user is not there to see it.

Because the S4 state relies only on non-volatile storage, a machine can save its system context for an arbitrary period of time (on the order of many years).

Table 2-1 Summary of Global Power States

Global system state	Software runs	Latency	Power consumption	OS restart required	Safe to disassemble computer	Exit state electronically
G0 Working	Yes	0	Large	No	No	Yes
G1 Sleeping	No	>0, varies with sleep state	Smaller	No	No	Yes
G2/S5 Soft Off	No	Long	Very near 0	Yes	No	Yes
G3 Mechanical Off	No	Long	RTC battery	Yes	Yes	No

Notice that the entries for G2/S5 and G3 in the Latency column of the above table are “Long.” This implies that a platform designed to give the user the appearance of “instant-on,” similar to a home appliance device, will use the G0 and G1 states almost exclusively (the G3 state may be used for moving the machine or repairing it).

2.3 Device Power State Definitions

Device power states are states of particular devices; as such, they are generally not visible to the user. For example, some devices may be in the Off state even though the system as a whole is in the Working state.

Device states apply to any device on any bus. They are generally defined in terms of four principal criteria:

- **Power consumption.** How much power the device uses.
- **Device context.** How much of the context of the device is retained by the hardware. The OS is responsible for restoring any lost device context (this may be done by resetting the device).
- **Device driver.** What the device driver must do to restore the device to full on.
- **Restore time.** How long it takes to restore the device to full on.

The device power states are defined below, although very generically. Many devices do not have all four power states defined. Devices may be capable of several different low-power modes, but if there is no user-perceptible difference between the modes, only the lowest power mode will be used. The Device Class Power Management Specifications, included in Appendix A of this specification, describe which of these power states are defined for a given type (class) of device and define the specific details of each power state for that device class. For a list of the available *Device Class Power Management Specifications*, see “Appendix A: Device Class Specifications.”

D3 Off

Power has been fully removed from the device. The device context is lost when this state is entered, so the OS software will reinitialize the device when powering it back on. Since device context and power are lost, devices in this state do not decode their address lines. Devices in this state have the longest restore times. All classes of devices define this state.

D2

The meaning of the D2 Device State is defined by each device class. Many device classes may not define D2. In general, D2 is expected to save more power and preserve less device context than D1 or D0. Buses in D2 may cause the device to lose some context (for example, by reducing power on the bus, thus forcing the device to turn off some of its functions).

D1

The meaning of the D1 Device State is defined by each device class. Many device classes may not define D1. In general, D1 is expected to save less power and preserve more device context than D2.

D0 Fully-On

This state is assumed to be the highest level of power consumption. The device is completely active and responsive, and is expected to remember all relevant context continuously.

Table 2-2 Summary of Device Power States

Device State	Power Consumption	Device Context Retained	Driver Restoration
D0 - Fully-On	As needed for operation	All	None
D1	D0>D1>D2>D3	>D2	<D2
D2	D0>D1>D2>D3	<D1	>D1
D3 - Off	0	None	Full initialization and load

Note: Devices often have different power modes within a given state. Devices can use these modes as long as they can automatically transparently switch between these modes from the software, without violating the rules for the current D*x* state the device is in. Low-power modes that adversely affect performance (in other words, low speed modes) or that are not transparent to software cannot be done automatically in hardware; the device driver must issue commands to use these modes.

2.4 Sleeping State Definitions

Sleeping states (S*x* states) are types of sleeping states within the global sleeping state, G1. The S*x* states are briefly defined below. For a detailed definition of the system behavior within each S*x* state, see section 7.3.4, "System _S*x* States." For a detailed definition of the transitions between each of the S*x* states, see section 9.1, "Sleeping States."

S1 Sleeping State

The S1 sleeping state is a low wake latency sleeping state. In this state, no system context is lost (CPU or chip set) and hardware maintains all system context.

S2 Sleeping State

The S2 sleeping state is a low wake latency sleeping state. This state is similar to the S1 sleeping state except that the CPU and system cache context is lost (the OS is responsible for maintaining the caches and CPU context). Control starts from the processor's reset vector after the wake event.

S3 Sleeping State

The S3 sleeping state is a low wake latency sleeping state where all system context is lost except system memory. CPU, cache, and chip set context are lost in this state. Hardware maintains memory context and restores some CPU and L2 configuration context. Control starts from the processor's reset vector after the wake event.

S4 Sleeping State

The S4 sleeping state is the lowest power, longest wake latency sleeping state supported by ACPI. In order to reduce power to a minimum, it is assumed that the hardware platform has powered off all devices. Platform context is maintained.

S5 Soft Off State

The S5 state is similar to the S4 state except that the OS does not save any context. The system is in the "soft" off state and requires a complete boot when it wakes. Software uses a different state value to distinguish between the S5 state and the S4 state to allow for initial boot operations within the BIOS to distinguish whether or not the boot is going to wake from a saved memory image.

2.5 Processor Power State Definitions

Processor power states (C*x* states) are processor power consumption and thermal management states within the global working state, G0. The C*x* states possess specific entry and exist semantics and are briefly defined below. For a more detailed definition of each C*x* state, see section 8.1, “Processor Power States.”

C0 Processor Power State

While the processor is in this state, it executes instructions.

C1 Processor Power State

This processor power state has the lowest latency. The hardware latency in this state must be low enough that the operating software does not consider the latency aspect of the state when deciding whether to use it. Aside from putting the processor in a non-executing power state, this state has no other software-visible effects.

C2 Processor Power State

The C2 state offers improved power savings over the C1 state. The worst-case hardware latency for this state is provided via the ACPI system firmware and the operating software can use this information to determine when the C1 state should be used instead of the C2 state. Aside from putting the processor in a non-executing power state, this state has no other software-visible effects.

C3 Processor Power State

The C3 state offers improved power savings over the C1 and C2 states. The worst-case hardware latency for this state is provided via the ACPI system firmware and the operating software can use this information to determine when the C2 state should be used instead of the C3 state. While in the C3 state, the processor’s caches maintain state but ignore any snoops. The operating software is responsible for ensuring that the caches maintain coherency.

2.6 Device and Processor Performance State Definitions

Device and Processor performance states (P*x* states) are power consumption and capability states within the active/executing states, C0 for processors and D0 for devices. The Px states are briefly defined below. For a more detailed definition of each Px state from a processor perspective, see section 8.3.3, “Declaring a Processor Object.” For a more detailed definition of each Px state from a device perspective see section 3.6, “Device and Processor Performance States,” and the device class specifications in Appendix A.

P0 Performance State

While a device or processor is in this state, it uses its maximum performance capability and may consume maximum power.

P1 Performance State

In this performance power state, the performance capability of a device or processor is limited below its maximum and consumes less than maximum power.

Pn Performance State

In this performance state, the performance capability of a device or processor is at its minimum level and consumes minimal power while remaining in an active state. State *n* is a maximum number and is processor or device dependent. Processors and devices may define support for an arbitrary number of performance states not to exceed 16.

3 Overview

ACPI provides OSPM with direct and exclusive control over the power management and motherboard device configuration functions of a computer. When it starts, OSPM takes over these functions from legacy BIOS interfaces such as the APM BIOS and the PNPBIOS. Having done this, OSPM is responsible for handling motherboard device configuration events as well as controlling the power, performance, and thermal status of the system based on user preference and application requests. ACPI provides low-level interfaces that allow OSPM to perform these functions. The functional areas covered by the ACPI specification are:

- **System power management.** ACPI defines mechanisms for putting the computer as a whole in and out of system sleeping states. It also provides a general mechanism for any device to wake the computer.
- **Device power management.** ACPI tables describe motherboard devices, their power states, the power planes the devices are connected to, and controls for putting devices into different power states. This enables the OS to put devices into low-power states based on application usage.
- **Processor power management.** While the OS is idle but not sleeping, it will use commands described by ACPI to put processors in low-power states.
- **Device and processor performance management.** While the system is active, OSPM will transition devices and processors into different performance states, defined by ACPI, to achieve a desirable balance between performance and energy conservation goals as well as other environmental requirements (for example, visibility and acoustics).
- **Plug and Play.** ACPI specifies information used to enumerate and configure motherboard devices. This information is arranged hierarchically so when events such as docking and undocking take place, the OS has precise, *a priori* knowledge of which devices are affected by the event.
- **System Events.** ACPI provides a general event mechanism that can be used for system events such as thermal events, power management events, docking, device insertion and removal, and so on. This mechanism is very flexible in that it does not define specifically how events are routed to the core logic chip set.
- **Battery management.** Battery management policy moves from the APM BIOS to the ACPI OS. An ACPI-compatible battery device needs either a Smart Battery subsystem interface, which is controlled by the OS directly through the embedded controller interface, or a Control Method Battery interface. A Control Method Battery interface is completely defined by AML control methods, allowing an OEM to choose any type of the battery and any kind of communication interface supported by ACPI. The battery must comply with the requirements of its interface, as described either herein or in other applicable standards. The OS may choose to alter the behavior of the battery, for example, by adjusting the Low Battery or Battery Warning trip point. When there are multiple batteries present, the battery subsystem is not required to perform any synthesis of a “composite battery” from the data of the separate batteries. In cases where the battery subsystem does not synthesize a “composite battery” from the separate battery’s data, the OS must provide that synthesis.
- **Thermal management.** Since the OS controls the power states of devices and processors, ACPI also addresses system thermal management. It provides a simple, scaleable model that allows OEMs to define thermal zones, thermal indicators, and methods for cooling thermal zones.

- **Embedded Controller.** ACPI defines a standard hardware and software communications interface between an OS bus enumerator and an embedded controller. This allows any OS to provide a standard bus enumerator that can directly communicate with an embedded controller in the system, thus allowing other drivers within the system to communicate with and use the resources of system embedded controllers. This in turn enables the OEM to provide platform features that the OS and applications can use.
- **SMBus Controller.** ACPI defines a standard hardware and software communications interface between an OS bus driver and an SMBus Controller. This allows any OS to provide a standard bus driver that can directly communicate with SMBus devices in the system. This in turn enables the OEM to provide platform features that the OS and applications can use.

Once in ACPI mode, system firmware or other software must not manipulate the platform's configuration, power, performance, and thermal control interfaces (if implemented) independently of OSPM. OSPM alone is responsible for coordinating the configuration, power management, performance management, and thermal control policy of the system. Manipulation of these interfaces independently of OSPM undermines the purpose of OSPM/ACPI and may adversely impact the system's configuration, power, performance, and thermal policy goals. However, in the case of the possibility of damage to system from excessive thermal conditions where OSPM latency is insufficient to remedy an adverse thermal condition, the platform may exercise a failsafe thermal control mechanism that reduces the performance of a system component to avoid damage. In this case, the platform should notify OSPM of the performance reduction if the reduction is of significant duration (in other words, if the duration of reduced performance could adversely impact OSPM's power or performance control policy).

3.1 System Power Management

Under OSPM, the OS directs all system and device power state transitions. Employing user preferences and knowledge of how devices are being used by applications, the OS puts devices in and out of low-power states. Devices that are not being used can be turned off. Similarly, the OS uses information from applications and user settings to put the system as a whole into a low- power state. The OS uses ACPI to control power state transitions in hardware.

3.2 Power States

From a user-visible level, the system can be thought of as being in one of the states in the following diagram:

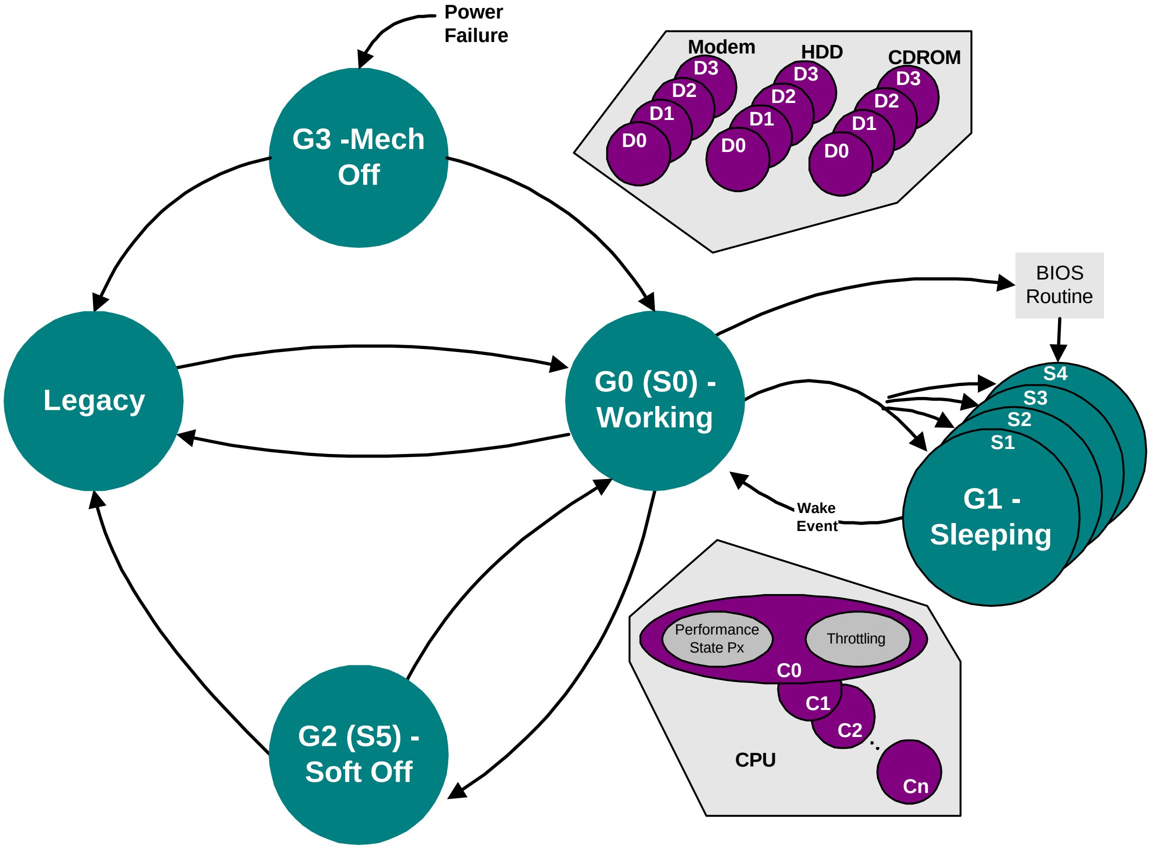

Figure 3-1 Global System Power States and Transitions

See section 2.2, “Global System State Definitions,” for detailed definitions of these states.

In general use, computers alternate between the Working and Sleeping states. In the Working state, the computer is used to doing work. User-mode application threads are dispatched and running. Individual devices can be in low-power (D*x*) states and processors can be in low-power (C*x*) states if they are not being used. Any device the system turns off because it is not actively in use can be turned on with short latency. (What “short” means depends on the device. An LCD display needs to come on in sub-second times, while it is generally acceptable to wait a few seconds for a printer to wake.)

The net effect of this is that the entire machine is functional in the Working state. Various Working sub-states differ in speed of computation, power used, heat produced, and noise produced. Tuning within the Working state is largely about trade-offs among speed, power, heat, and noise.

When the computer is idle or the user has pressed the power button, the OS will put the computer into one of the sleeping (S*x*) states. No user-visible computation occurs in a sleeping state. The sleeping sub-states differ in what events can arouse the system to a Working state, and how long this takes. When the machine must awaken to all possible events or do so very quickly, it can enter only the sub-states that achieve a partial reduction of system power consumption. However, if the only event of interest is a user pushing on a switch and a latency of minutes is allowed, the OS could save all system context into an NVS file and transition the hardware into the S4 sleeping state. In this state, the machine draws almost zero power and retains system context for an arbitrary period of time (years or decades if needed).

The other states are used less often. Computers that support legacy BIOS power management interfaces boot in the Legacy state and transition to the Working state when an ACPI OS loads. A system without legacy support (for example, a RISC system) transitions directly from the Mechanical Off state to the Working state. Users typically put computers into the Mechanical Off state by flipping the computer’s mechanical switch or by unplugging the computer.

3.2.1 New Meanings for the Power Button

In legacy systems, the power button typically either forces the machine into Soft Off or Mechanical Off or, on a laptop, forces it to some sleeping state. No allowance is made for user policy (such as the user wants the machine to “come on” in less than 1 second with all context as it was when the user turned the machine “off”), system alert functions (such as the system being used as an answering machine or fax machine), or application function (such as saving a user file).

In an OSPM system, there are two switches. One is to transition the system to the Mechanical Off state. A mechanism to stop current flow is required for legal reasons in some jurisdictions (for example, in some European countries). The other is the “main” power button. This is in some obvious place (for example, beside the keyboard on a laptop). Unlike legacy on/off buttons, all it does is send a request to the system. What the system does with this request depends on policy issues derived from user preferences, user function requests, and application data.

3.2.2 Platform Power Management Characteristics

3.2.2.1 Mobile PC

Mobile PCs will continue to have aggressive power management functionality. Going to OSPM/ACPI will allow enhanced power savings techniques and more refined user policies.

Aspects of mobile PC power management in the ACPI specification are thermal management (see section 12, “Thermal Management”) and the embedded controller interface (see section 13, “ACPI Embedded Controller Interface Specification”).

3.2.2.2 Desktop PCs

Power-managed desktops will be of two types, though the first type will migrate to the second over time.

- **Ordinary "Green PC**." Here, new appliance functions are not the issue. The machine is really only used for productivity computations. At least initially, such machines can get by with very minimal function. In particular, they need the normal ACPI timers and controls, but don't need to support elaborate sleeping states, and so on. They, however, do need to allow the OS to put as many of their devices/resources as possible into device standby and device off states, as independently as possible (to allow for maximum compute speed with minimum power wasted on unused devices). Such PCs will also need to support wake from the sleeping state by means of a timer, because this allows administrators to force them to turn on just before people are to show up for work.
- **Home PC**. Computers are moving into home environments where they are used in entertainment centers and to perform tasks like answering the phone. A home PC needs all of the functionality of the ordinary green PC. In fact, it has all of the ACPI power functionality of a laptop except for docking and lid events (and need not have any legacy power management).

3.2.2.3 Multiprocessor and Server PCs

Perhaps surprisingly, server machines often get the largest absolute power savings. Why? Because they have the largest hardware configurations and because it's not practical for somebody to hit the off switch when they leave at night.

- **Day Mode**. In day mode, servers are power-managed much like a corporate ordinary green PC, staying in the Working state all the time, but putting unused devices into low-power states whenever possible. Because servers can be very large and have, for example, many disk spindles, power management can result in large savings. OSPM allows careful tuning of when to do this, thus making it workable.
- **Night Mode**. In night mode, servers look like home PCs. They sleep as deeply as they can and are still able to wake and answer service requests coming in over the network, phone links, and so on, within specified latencies. So, for example, a print server might go into deep sleep until it receives a print job at 3 A.M., at which point it wakes in perhaps less than 30 seconds, prints the job, and then goes back to sleep. If the print request comes over the LAN, then this scenario depends on an intelligent LAN adapter that can wake the system in response to an interesting received packet.

3.3 Device Power Management

This section describes ACPI-compatible device power management. The ACPI device power states are introduced, the controls and information an ACPI-compatible OS needs to perform device power management are discussed, the wake operation devices use to wake the computer from a sleeping state is described, and an example of ACPI-compatible device management using a modem is given.

3.3.1 Power Management Standards

To manage power of all the devices in the system, the OS needs standard methods for sending commands to a device. These standards define the operations used to manage power of devices on a particular bus and the power states that devices can be put into. Defining these standards for each bus creates a baseline level of power management support the OS can utilize. Independent Hardware Vendors (IHVs) do not have to spend extra time writing software to manage power of their hardware, because simply adhering to the standard gains them direct OS support. For OS vendors, the bus standards allow the power management code to be centralized in each bus driver. Finally, bus-driven power management allows the OS to track the states of all devices on a given bus. When all the devices are in a given state (or example, D3 - off), the OS can put the entire bus into the power supply mode appropriate for that state (for example, D3 - off).

Bus-level power management specifications are written for the following buses:

- PCI
- CardBus
- USB
- IEEE 1394

3.3.2 Device Power States

To unify nomenclature and provide consistent behavior across devices, standard definitions are used for the power states of devices. Generally, these states are defined in terms of the following criteria:

- **Power consumption**. How much power the device uses.
- **Device context**. How much of the context of the device is retained by the hardware.
- **Device driver**. What the device driver must do to restore the device to fully on.
- **Restore latency**. How long it takes to restore the device to fully on.

More specifically, power management specifications for each class of device (for example, modem, network adapter, hard disk, and so on) more precisely define the power states and power policy for the class. See section 2.3, “Device Power State Definitions,” for the detailed description of the four general device power states (D0-D3).

3.3.3 Device Power State Definitions

The device power state definitions are device-independent, but classes of devices on a bus must support some consistent set of power-related characteristics. For example, when the bus-specific mechanism to set the device power state to a given level is invoked, the actions a device might take and the specific sorts of behaviors the OS can assume while the device is in that state will vary from device type to device type. For a fully integrated device power management system, these class-specific power characteristics must also be standardized:

- **Device Power State Characteristics.** Each class of device has a standard definition of target power consumption levels, state-change latencies, and context loss.
- **Minimum Device Power Capabilities.** Each class of device has a minimum standard set of power capabilities.
- **Device Functional Characteristics.** Each class of device has a standard definition of what subset of device functionality or features is available in each power state (for example, the net card can receive, but cannot transmit; the sound card is fully functional except that the power amps are off, and so on).
- **Device Wakeup Characteristics.** Each class of device has a standard definition of its wake policy.

The Microsoft Device Class Power Management specifications define these power state characteristics for each class of device.

3.4 Controlling Device Power

ACPI provides the OS the controls and information needed to perform device power management. ACPI describes to the OS the capabilities of all the devices it controls. It also gives the OS the control methods used to set the power state or get the power status for each device. Finally, it has a general scheme for devices to wake the machine.

Note: Other buses enumerate some devices on the main board. For example, PCI devices are reported through the standard PCI enumeration mechanisms. The ACPI table lists legacy devices that cannot be reported through their own bus specification, the root of each bus in the system, and devices that have additional power management or configuration options not covered by their own bus specification. Power management of these devices is handled through their own bus specification (in this case, PCI). All other devices are handled through ACPI.

For more detailed information see section 7, “Power and Performance Management.”

3.4.1 Getting Device Power Capabilities

As the OS enumerates devices in the system, it gets information about the power management features that the device supports. The Differentiated Definition Block given to the OS by the BIOS describes every device handled by ACPI. This description contains the following information:

- A description of what power resources (power planes and clock sources) the device needs in each power state that the device supports. For example, a device might need a high power bus and a clock in the D0 state but only a low-power bus and no clock in the D2 state.
- A description of what power resources a device needs in order to wake the machine (or none to indicate that the device does not support wake). The OS can use this information to infer what device and system power states from which the device can support wake.
- The optional control method the OS can use to set the power state of the device and to get and set resources.

In addition to describing the devices handled by ACPI, the table lists the power planes and clock sources themselves and the control methods for turning them on and off. For detailed information, see section 7, “Power and Performance Management.”

3.4.2 Setting Device Power States

OSPM uses the Set Power State operation to put a device into one of the four power states.

When a device is put in a lower power state, it configures itself to draw as little power from the bus as possible. The OS tracks the state of all devices on the bus, and will put the bus in the best power state based on the current device requirements on that bus. For example, if all devices on a bus are in the D3 state, the OS will send a command to the bus control chip set to remove power from the bus (thus putting the bus in the D3 state). If a particular bus supports a low-power supply state, the OS puts the bus in that state if all devices are in the D1 or D2 state. Whatever power state a device is in, the OS must be able to issue a Set Power State command to can resume the device.

Note: The device does not need to have power to do this. The OS must turn on power to the device before it can send commands to the device.

OSPM also uses the Set Power State operation to enable power management features such as wake (described in section 7, “Power and Performance Management.”).

When a device is to be set in a particular power state using the ACPI interface, the OS first decides which power resources will be used and which can be turned off. The OS tracks all the devices on a given power resource. When all the devices on a resource have been turned off, the OS turns off that power resource by running a control method. If a power resource is turned off and one of the devices on that resource needs to be turned on, the OS first turns on the power resource using a control method and then signals the device to turn on. The time that the OS must wait for the power resource to stabilize after turning it on or off is described in the description table. The OS uses the time base provided by the Power Management Timer to measure these time intervals.

Once the power resources have been switched, the OS executes the appropriate control method to put the device in that power state. Notice that this might not mean that power is removed from the device. If other active devices are sharing a power resource, the power resources will remain on.

3.4.3 Getting Device Power Status

OSPM uses the Get Power Status operation to determine the current power configuration (states and features), as well as the status of any batteries supported by the device. The device can signal an SCI to inform the OS of changes in power status. For example, a device can trigger an interrupt to inform the OS that the battery has reached low power level.

Devices use the ACPI event model (see below) to signal power status changes (battery status changes, for example), the ACPI chip set signals the OS via the SCI interrupt. An SCI interrupt status bit is set to indicate the event to the OS. The OS runs the control method associated with the event. This control method signals to the OS which device has changed.

ACPI supports two types of batteries: batteries that report only basic battery status information and batteries that support the Smart Battery System Implementers Forum Smart Battery Specification. For batteries that report only basic battery status information (such as total capacity and remaining capacity), the OS uses control methods from the battery’s description table to read this information. To read status information for Smart Batteries, the OS can use a standard Smart Battery driver that directly interfaces to Smart Batteries through the appropriate bus enumerator.

3.4.4 Waking the Computer

The wake operation enables devices to wake the computer from a sleeping power state. This operation must not depend on the CPU because the CPU will not be executing instructions.

The OS ensures any bridges between the device and the core logic are in the lowest power state in which they can still forward the wake signal. When a device with wake enabled decides to wake the machine, it sends the defined signal on its bus. Bus bridges must forward this signal to upstream bridges using the appropriate signal for that bus. Thus, the signal eventually reaches the core chip set (for example, an ACPI chip set), which in turn wakes the machine.

Before putting the machine in a sleeping power state, the OS determines which devices are needed to wake the machine based on application requests, and then enables wake on those devices.

The OS enables the wake feature on devices by setting that device’s SCI Enable bit. The location of this bit is listed in the device’s entry in the description table. Only devices that have their wake feature enabled can wake the machine. The OS keeps track of the power states that the wake devices support, and keeps the machine in a power state in which the wake can still wake the machine[1] (based on capabilities reported in the description table).

[1] Some OS policies may require the OS to put the machine into a global system state for which the device can no longer wake the system. Such as when a system has very low battery power.

When the computer is in the Sleeping state and a wake device decides to wake the machine, it signals to the ACPI chip set. The SCI status bit corresponding to the device waking the machine is set, and the ACPI chip set resumes the machine. After the OS is running again, it clears the bit and handles the event that caused the wake. The control method for this event then uses the Notify command to tell the OS which device caused the wake.

3.4.5 Example: Modem Device Power Management

To illustrate how these power management methods function in ACPI, consider an integrated modem. (This example is greatly simplified for the purposes of this discussion.) The power states of a modem are defined as follows (this is an excerpt from the Modem Device Class Power Management Specification):

D0 Modem controller on
Phone interface on
Speaker on
Can be on hook or off hook
Can be waiting for answer

D1 Modem controller in low-power mode (context retained by device)
Phone interface powered by phone line or in low-power mode
Speaker off
Must be on hook

D2 Same as D3

D3 Modem controller off (context lost)
Phone interface powered by phone line or off
Speaker off
On hook

The power policy for the modem is defined as follows:

D3 → D0	COM port opened
D0, D1 → D3	COM port closed
D0 → D1	Modem put in answer mode
D1 → D0	Application requests dial or the phone rings while the modem is in answer mode

The wake policy for the modem is very simple: When the phone rings and wake is enabled, wake the machine.

Based on that policy, the modem and the COM port to which it is attached can be implemented in hardware as shown in Figure 3-2. This is just an example for illustrating features of ACPI. This example is not intended to describe how OEMs should build hardware.

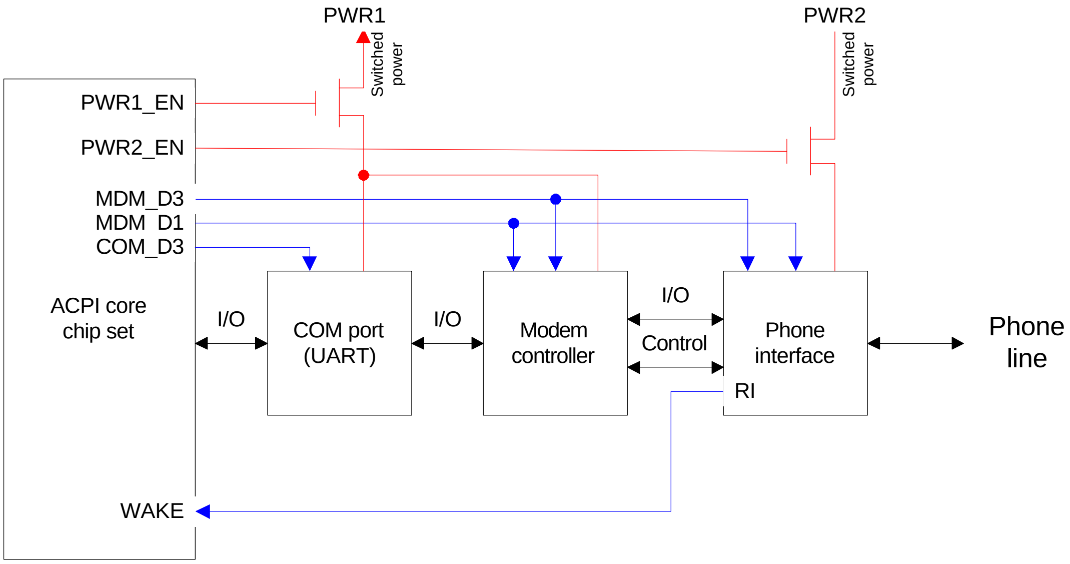

Figure 3-2 Example Modem and COM Port Hardware

Note: Although not shown above, each discrete part has some isolation logic so that the part is isolated when power is removed from it. Isolation logic controls are implemented as power resources in the ACPI Differentiated Description Block so that devices are isolated as power planes are sequenced off.

3.4.5.1 Getting the Modem's Capabilities

The OS determines the capabilities of this modem when it enumerates the modem by reading the modem's entry in the Differentiated Definition Block. In this case, the entry for the modem would report:

The device supports D0, D1, and D3:

D0 requires PWR1 and PWR2 as power resources
D1 requires PWR1 as a power resource
(D3 implicitly requires no power resources)

To wake the machine, the modem needs no power resources (implying it can wake the machine from D0, D1, and D3)

Control methods for setting power state and resources

3.4.5.2 Setting the Modem's Power State

While the OS is running (G0 state), it switches the modem to different power states according to the power policy defined for modems.

When an application opens the COM port, the OS turns on the modem by putting it in the D0 state. Then if the application puts the modem in answer mode, the OS puts the modem in the D1 state to wait for the call. To make this state transition, the ACPI first checks to see what power resources are no longer needed. In this case, PWR2 is not needed. Then it checks to make sure no other device in the system requires the use of the PWR2 power resource. If the resource is no longer needed, the OSPM uses the _OFF control method associated with that power resource in the Differentiated Definition Block to turn off the PWR2 power plane. This control method sends the appropriate commands to the core chip set to stop asserting the PWR2_EN line. Then, OSPM runs a control method (_PS1) provided in the modem's entry to put the device in the D1 state. This control method asserts the MDM_D1 signal that tells the modem controller to go into a low-power mode.

OSPM does not always turn off power resources when a given device is put in a lower power state. For example, assume that the PWR1 power plane also powers an active line printer (LPT) port. Suppose the user terminates the modem application, causing the COM port to be closed, and therefore causing the modem to be shut off (state D3). As always, OSPM checks to see which power resources are no longer needed. Because the LPT port is still active, PWR1 is in use. OSPM does not turn off the PWR1 resource. It continues the state transition process by running the modem's control method to switch the device to the D3 power state. The control method causes the MDM_D3 line to be asserted. The modem controller now turns off all its major functions so that it draws little power, if any, from the PWR1 line. Because the COM port is closed, the same sequence of events will take place to put it in the D3 state. Notice that these registers might not be in the device itself. For example, the control method could read the register that controls MDM_D3.

3.4.5.3 Getting the Modem's Power Status

Integrated modems have no batteries; the only power status information for the device is the power state of the modem. To determine the modem's current power state (D0-D3), OSPM runs a control method (_PSC) supplied in the modem's entry in the Differentiated Definition Block. This control method reads from the necessary registersto determine the modem's power state.

3.4.5.4 Waking the Computer

As indicated in the modem capabilities, this modem can wake the machine from any device power state. Before putting the computer in a sleep state, the OS enables wake on any devices that applications have requested to be able to wake the machine. Then, it chooses the lowest sleeping state that can still provide the power resources necessary to allow all enabled wake devices to wake the machine. Next, the OS puts each of those devices in the appropriate power state, and puts all other devices in the D3 state. In this case, the OS puts the modem in the D3 state because it supports wake from that state. Finally, the OS saves a resume vector and puts the machine into a sleep state through an ACPI register.

Waking the computer via modem starts with the modem's phone interface asserting its ring indicate (RI) line when it detects a ring on the phone line. This line is routed to the core chip set to generate a wake event. The chip set then wakes the system and the hardware will eventually passes control back to the OS (the wake mechanism differs depending on the sleeping state). After the OS is running, it puts the device in the D0 state and begins handling interrupts from the modem to process the event.

3.5 Processor Power Management

To further save power in the Working state, the OS puts the CPU into low-power states (C1, C2, and C3) when the OS is idle. In these low-power states, the CPU does not run any instructions, and wakes when an interrupt, such as the OS scheduler's timer interrupt, occurs.

The OS determines how much time is being spent in its idle loop by reading the ACPI Power Management Timer. This timer runs at a known, fixed frequency and allows the OS to precisely determine idle time. Depending on this idle time estimate, the OS will put the CPU into different quality low-power states (which vary in power and latency) when it enters its idle loop.

The CPU states are defined in detail in section 8, "Processor Control."

3.6 Device and Processor Performance States

This section describes the concept of device and processor performance states. Device and processor performance states (Px states) are power consumption and capability states within the active/executing states, C0 for processors and D0 for devices. Performance states allow OSPM to make tradeoffs between performance and energy conservation. Device and processor performance states have the greatest impact when the states invoke different device and processor efficiency levels as opposed to a linear scaling of performance and energy consumption. Since performance state transitions occur in the active/executing device states, care must be taken to ensure that performance state transitions do not adversely impact the system.

Examples of device performance states include:

- A hard drive that provides levels of maximum throughput that correspond to levels of power consumption.
- An LCD panel that supports multiple brightness levels that correspond to levels of power consumption.
- A graphics component that scales performance between 2D and 3D drawing modes that corresponds to levels of power consumption.
- An audio subsystem that provides multiple levels of maximum volume that correspond to levels of maximum power consumption.
- A Direct-RDRAM^{TM} controller that provides multiple levels of memory throughput performance, corresponding to multiple levels of power consumption, by adjusting the maximum bandwidth throttles.

Processor performance states are described in Section 8, "Processor Control."

3.7 Plug and Play

In addition to power management, ACPI provides controls and information so that the OS can direct Plug and Play on the motherboard. The Differentiated Description Table describes the motherboard devices. The OS enumerates motherboard devices simply by reading through the Differentiated Description Table looking for devices with hardware IDs.

Each device enumerated by ACPI includes control methods that report the hardware resources the device could occupy and those that are currently used, and a control method for configuring those resources. The information is used by the Plug and Play system to configure the devices.

ACPI is used only to enumerate and configure motherboard devices that do not have other hardware standards for enumeration and configuration. For example, PCI devices on the motherboard must not be enumerated by ACPI; therefore Plug and Play information for these devices is not included in the Differentiated Description Table. However, power management information for these devices can still appear in the table if the devices' power management is to be controlled through ACPI.

Note: When preparing to boot a computer, the BIOS only needs to configure boot devices. This includes boot devices described in the ACPI system description tables as well as devices that are controlled through other standards.

3.7.1 Example: Configuring the Modem

Returning to the modem device example above, the OS will find the modem and load a driver for it when the OS finds it in the DSDT. This table will have control methods that give the OS the following information:

- The device can use IRQ 3, I/O 3F8-3FF or IRQ 4, I/O 2E8-2EF
- The device is currently using IRQ 3, I/O 3F8-3FF

The OS configures the modem's hardware resources using Plug and Play algorithms. It chooses one of the supported configurations that does not conflict with any other devices. Then, OSPM configures the device for those resources by running a control method supplied in the modem's section of the Differentiated Definition Block. This control method will write to any I/O ports or memory addresses necessary to configure the device to the given resources.

3.8 System Events

ACPI includes a general event model used for Plug and Play, Thermal, and Power Management events. There are two registers that make up the event model: an event status register and an event enable register.

When an event occurs, the core logic sets a bit in the status register to indicate the event. If the corresponding bit in the enable register is set, the core logic will assert the SCI to signal the OS. When the OS receives this interrupt, it will run the control methods corresponding to any bits set in the event status register. These control methods use AML commands to tell the OS what event occurred.

For example, assume a machine has all of its Plug and Play, Thermal, and Power Management events connected to the same pin in the core logic. The event status and event enable registers would only have one bit each: the bit corresponding to the event pin.

When the computer is docked, the core logic sets the status bit and signals the SCI. The OS, seeing the status bit set, runs the control method for that bit. The control method checks the hardware and determines the event was a docking event (for example). It then signals to the OS that a docking event has occurred, and can tell the OS specifically where in the device hierarchy the new devices will appear.

Since the event model registers are generalized, they can describe many different platform implementations. The single pin model above is just one example. Another design might have Plug and Play, Thermal, and Power Management events wired to three different pins so there would be three status bits (and three enable bits). Yet another design might have every individual event wired to its own pin and status bit. This design, at the opposite extreme from the single pin design, allows very complex hardware, yet very simple control methods. Countless variations in wiring up events are possible.

3.9 Battery Management

Battery management policy moves from the APM BIOS to the ACPI-compatible OS. Batteries must comply with the requirements of their associated interfaces, as described either herein or in other applicable standards. The OS may choose to alter the behavior of the battery, for example, by adjusting the Low Battery or Battery Warning trip point. When there are multiple batteries present, the battery subsystem is not required to perform any synthesis of a “composite battery” from the data of the separate batteries. In cases where the battery subsystem does not synthesize a “composite battery” from the separate battery's data, the OS must provide that synthesis.

An ACPI-compatible battery device needs either a Smart Battery subsystem interface or a Control Method Battery interface.

- *Smart Battery* is controlled by the OS directly through the embedded controller (EC). For more information about the ACPI Embedded Controller SMBus interface, see section 13.9, “SMBus Host Controller Interface via Embedded Controller.” For additional information about the Smart Battery subsystem interface, see section 11.1, “Smart Battery Subsystems.”
- Control Method Battery is completely accessed by AML code control methods, allowing the OEM to choose any type of battery and any kind of communication interface supported by ACPI. For more information about the Control Method Battery Interface, see section 11.2, “Control Method Batteries.”

This section describes concepts common to all battery types.

3.9.1 Battery Communications

Both the Smart Battery and Control Method Battery interfaces provide a mechanism for the OS to query information from the platform’s battery system. This information may include full charged capacity, present battery capacity, rate of discharge, and other measures of the battery’s condition. All battery system types must provide notification to the OS when there is a change such as inserting or removing a battery, or when a battery starts or stops discharging. Smart Batteries and some Control Method Batteries are also able to give notifications based on changes in capacity. Smart batteries provide extra information such as estimated run-time, information about how much power the battery is able to provide, and what the run-time would be at a predetermined rate of consumption.

3.9.2 Battery Capacity

Each battery must report its designed capacity, latest full-charged capacity, and present remaining capacity. Remaining capacity decreases during usage, and it also changes depending on the environment. Therefore, the OS must use latest full-charged capacity to calculate the battery percentage. In addition the battery system must report warning and low battery levels at which the user must be notified and the system transitioned to a sleeping state. See Figure 3-3 for the relation of these five values.

A system may use either rate and capacity [mA/mAh] or power and energy [mW/mWh] for the unit of battery information calculation and reporting. Mixing [mA] and [mW] is not allowed on a system.

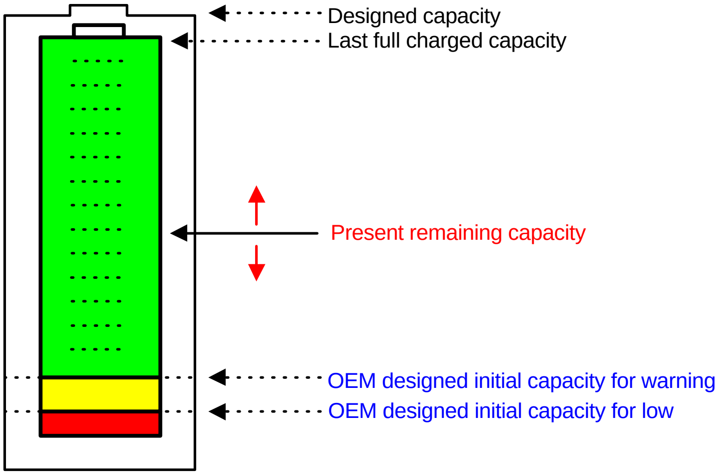
Designed capacity
Last full charged capacity
Present remaining capacity
OEM designed initial capacity for warning
OEM designed initial capacity for low

Figure 3-3 Reporting Battery Capacity

3.9.3 Battery Gas Gauge

At the most basic level, the OS calculates Remaining Battery Percentage [%] using the following formula:

$$\textbf{Remaining Battery Percentage[\%]} = \frac{\textbf{Battery Remaining Capacity [mAh/mWh]}}{\textbf{Last Full Charged Capacity [mAh/mWh]}} * \textbf{100}$$

Control Method Battery also reports the Present Drain Rate [mA or mW] for calculating the remaining battery life. At the most basic level, Remaining Battery life is calculated by following formula:

$$\textbf{Remaining Battery Life [h]} = \frac{\textbf{Battery Remaining Capacity [mAh/mWh]}}{\textbf{Battery Present Rate [mA/mW]}}$$

Smart Batteries also report the present rate of drain, but since they can directly report the estimated run-time, this function should be used instead as it can more accurately account for variations specific to the battery.

3.9.4 Low Battery Levels

A system has an OEM-designed initial capacity for warning, initial capacity for low, and a critical battery level or flag. The values for warning and low represent the amount of energy or battery capacity needed by the system to take certain actions. The critical battery level or flag is used to indicate when the batteries in the system are completely drained. OSPM can determine independent warning and low battery capacity values based on the OEM-designed levels, but cannot set these values lower than the OEM-designed values, as shown in Figure 3-4.

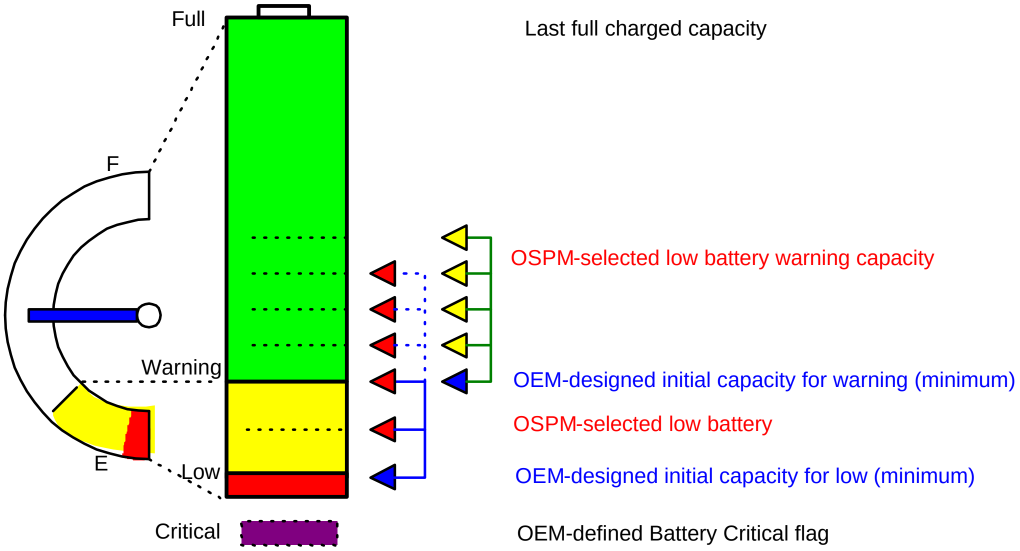
Full
Last full charged capacity
F
OSPM-selected low battery warning capacity
Warning
OEM-designed initial capacity for warning (minimum)
OSPM-selected low battery
E
Low
OEM-designed initial capacity for low (minimum)
Critical
OEM-defined Battery Critical flag

Figure 3-4 Low Battery and Warning

Each Control Method Battery in a system reports the OEM-designed initial warning capacity and OEM-designed initial low capacity as well as a flag to report when that battery has reached or is below its critical energy level. Unlike Control Method Batteries, Smart Batteries are not necessarily specific to one particular machine type, so the OEM-designed warning, low, and critical levels are reported separately in a Smart Battery Table described in section 5.2.12.

Table 3-1 described how these values should be set by the OEM and interpreted by the OS.

Table 3-1 Low Battery Levels

Level	Description
Warning	When the total available energy (mWh) or capacity (mAh) in the batteries falls below this level, the OS will notify the user through the UI. This value should allow for a few minutes of run-time before the “Low” level is encountered so the user has time to wrap up any important work, change the battery, or find a power outlet to plug the system in.
Low	This value is an estimation of the amount of energy or battery capacity required by the system to transition to any supported sleeping state. When the OS detects that the total available battery capacity is less than this value, it will transition the system to a user defined system state (S1-S5). In most situations this should be S4 so that system state is not lost if the battery eventually becomes completely empty. The design of the OS should consider that users of a multiple battery system may remove one or more of the batteries in an attempt replace or charge it. This might result in the remaining capacity falling below the “Low” level not leaving sufficient battery capacity for the OS to safely transition the system into the sleeping state. Therefore, if the batteries are discharging simultaneously, the action might need to be initiated at the point when both batteries reach this level.
Critical	The Critical battery state indicates that all available batteries are discharged and do not appear to be able to supply power to run the system any longer. When this occurs, the OS must attempt to perform an emergency shutdown as described below. For a smart battery system, this would typically occur when all batteries reach a capacity of 0, but an OEM may choose to put a larger value in the Smart Battery Table to provide an extra margin of safely. For a Control Method Battery system with multiple batteries, the flag is reported per battery. If any battery in the system is in a critically low state and is still providing power to the system (in other words, the battery is discharging), the system is considered to be in a critical energy state. The _BST control method is required to return the Critical flag on a discharging battery only when all batteries have reached a critical state; the ACPI BIOS is otherwise required to switch to a non-critical battery.

3.9.4.1 Emergency Shutdown

Running until all batteries in a system are critical is not a situation that should be encountered normally, since the system should be put into a sleeping state when the battery becomes low. In the case that this does occur, the OS should take steps to minimize any damage to system integrity. The emergency shutdown procedure should be designed to minimize bad effects based on the assumption that power may be lost at any time. For example, if a hard disk is spun down, the OS should not try to spin it up to write any data, since spinning up the disk and attempting to write data could potentially corrupt files if the write were not completed. Even if a disk is spun up, the decision to attempt to save even system settings data before shutting down would have to be evaluated since reverting to previous settings might be less harmful than having the potential to corrupt the settings if power was lost halfway through the write operation.

3.10 Thermal Management

ACPI allows the OS to play a role in the thermal management of the system while maintaining the platform's ability to mandate cooling actions as necessary. In the passive cooling mode, OSPM can make cooling decisions based on application load on the CPU as well as the thermal heuristics of the system. OSPM can also gracefully shutdown the computer in case of high temperature emergencies.

The ACPI thermal design is based around regions called thermal zones. Generally, the entire PC is one large thermal zone, but an OEM can partition the system into several logical thermal zones if necessary. Figure 3-5 is an example mobile PC diagram that depicts a single thermal zone with a central processor as the thermal-coupled device. In this example, the whole notebook is covered as one large thermal zone. This notebook uses one fan for active cooling and the CPU for passive cooling.

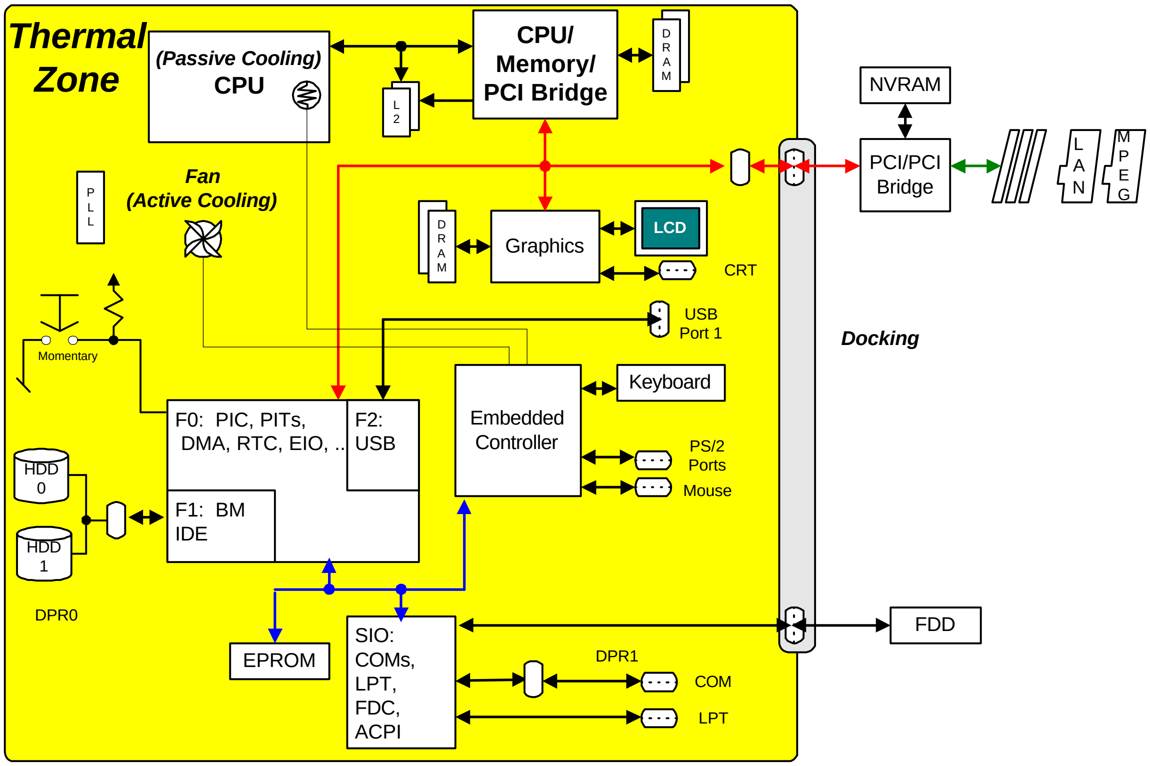

Figure 3-5 Thermal Zone

The following sections are an overview of the thermal control and cooling characteristics of a computer. For some thermal implementation examples on an ACPI platform, see section 12.4, "Thermal Zone Object Requirements."

3.10.1 Active and Passive Cooling Modes

ACPI defines two cooling modes, Active and Passive:

- **Passive cooling**. OS reduces the power consumption of devices at the cost of system performance to reduce the temperature of the machine.
- **Active cooling**. OS increases the power consumption of the system (for example, by turning on a fan) to reduce the temperature of the machine.

These two cooling modes are inversely related to each other. Active cooling requires increased power to reduce the heat within the system while Passive cooling requires reduced power to decrease the temperature. The effect of this relationship is that Active cooling allows maximum system performance, but it may create undesirable fan noise, while Passive cooling reduces system performance, but is inherently quiet.

3.10.2 Performance vs. Energy Conservation

A robust OSPM implementation provides the means for the end user to convey to OSPM a preference (or a level of preference) for either performance or energy conservation. Allowing the end user to choose this preference is most critical to mobile system users where maximizing system run-time on a battery charge often has higher priority over realizing maximum system performance.

A user's preference for performance corresponds to the Active cooling mode while a user's preference for energy conservation corresponds to the Passive cooling mode. ACPI defines an interface to convey the cooling mode to the platform. Active cooling can be performed with minimal OSPM thermal policy intervention. For example, the platform indicates through thermal zone parameters that crossing a thermal trip point requires a fan to be turned on. Passive cooling requires OSPM thermal policy to manipulate device interfaces that reduce performance to reduce thermal zone temperature.

3.10.3 Acoustics

Active cooling mode generally implies that fans will be used to cool the system and fans vary in their audible output. Fan noise can be quite undesirable given the loudness of the fan and the ambient noise environment. In this case, the end user's physical requirement for fan silence may override the preference for either performance or energy conservation.

A user's desire for fan silence corresponds to the Passive cooling mode. Accordingly, a user's desire for fan silence also means a preference for energy conservation.

For more information on thermal management and examples of platform settings for active and passive cooling, see section 12, "Thermal Management."

3.10.4 Multiple Thermal Zones

The basic thermal management model defines one thermal zone, but in order to provide extended thermal control in a complex system, ACPI specifies a multiple thermal zone implementation. Under a multiple thermal zone model, OSPM will independently manage several thermal-coupled devices and a designated thermal zone for each thermal-coupled device, using Active and/or Passive cooling methods available to each thermal zone. Each thermal zone can have more than one Passive and Active cooling device. Furthermore, each zone might have unique or shared cooling resources. In a multiple thermal zone configuration, if one zone reaches a critical state then OSPM must shut down the entire system.

4 ACPI Hardware Specification

ACPI defines standard interface mechanisms that allow an ACPI-compatible OS to control and communicate with an ACPI-compatible hardware platform. This section describes the hardware aspects of ACPI.

ACPI defines “hardware” as a programming model and its behavior. ACPI strives to keep much of the existing legacy programming model the same; however, to meet certain feature goals, designated features conform to a specific addressing and programming scheme. Hardware that falls within this category is referred to as “fixed.”

Although ACPI strives to minimize these changes, hardware engineers should read this section carefully to understand the changes needed to convert a legacy-only hardware model to an ACPI/Legacy hardware model or an ACPI-only hardware model.

ACPI classifies hardware into two categories: Fixed or Generic. Hardware that falls within the fixed category meets the programming and behavior specifications of ACPI. Hardware that falls within the generic category has a wide degree of flexibility in its implementation.

4.1 Fixed Hardware Programming Model

Because of the changes needed for migrating legacy hardware to the fixed category, ACPI limits the features specified by fixed hardware. Fixed hardware features are defined by the following criteria:

- Performance sensitive features
- Features that drivers require during wake
- Features that enable catastrophic OS software failure recovery

ACPI defines register-based interfaces to fixed hardware. CPU clock control and the power management timer are defined as fixed hardware to reduce the performance impact of accessing this hardware, which will result in more quickly reducing a thermal condition or extending battery life. If this logic were allowed to reside in PCI configuration space, for example, several layers of drivers would be called to access this address space. This takes a long time and will either adversely affect the power of the system (when trying to enter a low-power state) or the accuracy of the event (when trying to get a time stamp value).

Access to fixed hardware by OSPM allows OSPM to control the wake process without having to load the entire OS. For example, if PCI configuration space access is needed, the bus enumerator is loaded with all drivers used by the enumerator. Defining these interfaces in fixed hardware at addresses with which OSPM can communicate without any other driver’s assistance, allows OSPM to gather information prior to making a decision as to whether it continues loading the entire OS or puts it back to sleep.

If elements of the OS fail, it may be possible for OSPM to access address spaces that need no driver support. In such a situation, OSPM will attempt to honor fixed power button requests to transition the system to the G2 state. In the case where OSPM event handler is no longer able to respond to power button events, the power button override feature provides a back-up mechanism to unconditionally transition the system to the soft-off state.

4.1.1 Functional Fixed Hardware

ACPI defines the fixed hardware low-level interfaces as a means to convey to the system OEM the minimum interfaces necessary to achieve a level of capability and quality for motherboard configuration and system power management. Additionally, the definition of these interfaces, as well as others defined in this specification, conveys to OS Vendors (OSVs) developing ACPI-compatible operating systems, the necessary interfaces that operating systems must manipulate to provide robust support for system configuration and power management.

While the definition of low-level hardware interfaces defined by ACPI 1.0 afforded OSPM implementations a certain level of stability, controls for existing and emerging diverse CPU architectures cannot be accommodated by this model as they can require a sequence of hardware manipulations intermixed with native CPU instructions to provide the ACPI-defined interface function. In this case, an ACPI-defined fixed hardware interface can be functionally implemented by the CPU manufacturer through an equivalent combination of both hardware and software and is defined by ACPI 2.0 as Functional Fixed Hardware.

In IA-32-based systems, functional fixed hardware can be accommodated in an OS independent manner by using System Management Mode (SMM) based system firmware. Unfortunately, the nature of SMM-based code makes this type of OS independent implementation difficult if not impossible to debug. As such, this implementation approach is **not** recommended. In some cases, Functional Fixed Hardware implementations may require coordination with other OS components. As such, an OS independent implementation may not be viable.

OS-specific implementations of functional fixed hardware can be implemented using technical information supplied by the CPU manufacturer. The downside of this approach is that functional fixed hardware support must be developed for each OS. In some cases, the CPU manufacturer may provide a software component providing this support. In other cases support for the functional fixed hardware may be developed directly by the OS vendor.

In ACPI 2.0, the hardware register definition has been expanded to allow registers to exist in address spaces other than the System I/O address space. This is accomplished through the specification of an address space ID in the register definition (see section 5.2.3.1, “Generic Address Structure,” for more information). **When specifically directed by the CPU manufacturer,** the system firmware may define an interface as functional fixed hardware by supplying a special address space identifier, ***FfixedHW (0x7F)***, in the address space ID field for register definitions. It is emphasized that functional fixed hardware definitions may be declared in the ACPI system firmware **only as indicated by the CPU Manufacturer** for specific interfaces as the use of functional fixed hardware requires specific coordination with the OS vendor.

Only certain ACPI-defined interfaces may be implemented using functional fixed hardware and only when the interfaces are common across machine designs for example, systems sharing a common CPU architecture that does not support fixed hardware implementation of an ACPI-defined interface. OEMs are cautioned *not* to anticipate that functional fixed hardware support will be provided by OSPM differently on a system-by-system basis. The use of functional fixed hardware carries with it a reliance on OS specific software that must be considered. OEMs should consult OS vendors to ensure that specific functional fixed hardware interfaces are supported by specific operating systems.

4.2 Generic Hardware Programming Model

Although the fixed hardware programming model requires hardware registers to be defined at specific address locations, the generic hardware programming model allows hardware registers to reside in most address spaces and provide system OEMs with a wide degree of flexibility in the implementation of specific functions in hardware. OSPM directly accesses the fixed hardware registers, but relies on OEM-provided ACPI Machine Language (AML) code to access generic hardware registers.

AML code allows the OEM to provide the means for OSPM to control a generic hardware feature's control and event logic.

Section 16, "ACPI Source Language Reference," describes the ACPI Source Language (ASL)—a programming language that OEMs use to create AML. The ASL language provides many of the operators found in common object-oriented programming languages, but it has been optimized to enable the description of platform power management and configuration hardware. An ASL compiler converts ASL source code to AML, which is a very compact machine language that the ACPI AML code interpreter executes.

AML does two things:

- Abstracts the hardware from OSPM
- Buffers OEM code from the different OS implementations

One goal of ACPI is to allow the OEM "value added" hardware to remain basically unchanged in an ACPI configuration. One attribute of value-added hardware is that it is all implemented differently. To enable OSPM to execute properly on different types of value added hardware, ACPI defines higher level "control methods" that it calls to perform an action. The OEM provides AML code, which is associated with control methods, to be executed by OSPM. By providing AML code, generic hardware can take on almost any form.

Another important goal of ACPI is to provide OS independence. To do this, the OEM AML code has to execute the same under any ACPI-compatible OS. ACPI allows for this by making the AML code interpreter part of OSPM. This allows OSPM to take care of synchronizing and blocking issues specific to each particular OS.

The generic feature model is represented in the following block diagram. In this model the generic feature is described to OSPM through AML code. This description takes the form of an object that sits in the ACPI Namespace associated with the hardware to which it is adding value.

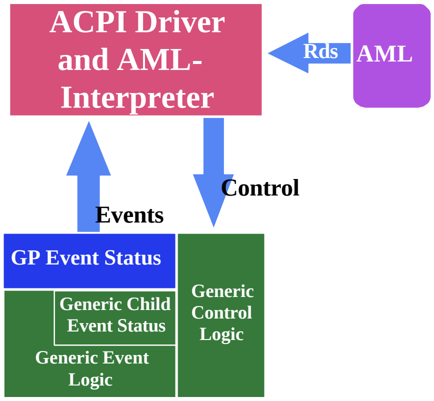

Figure 4-1 Generic Hardware Feature Model

As an example of a generic hardware control feature, a platform might be designed such that the IDE HDD's D3 state has value-added hardware to remove power from the drive. The IDE drive would then have a reference to the AML **PowerResource** object (which controls the value added power plane) in its namespace, and associated with that object would be control methods that OSPM invokes to control the D3 state of the drive:

- _PS0. A control method to sequence the IDE drive to the D0 state.
- _PS3. A control method to sequence the IDE drive to the D3 state.
- _PSC. A control method that returns the status of the IDE drive (on or off).

The control methods under this object provide an abstraction layer between OSPM and the hardware. OSPM understands how to control power planes (turn them on or off or to get their status) through its defined **PowerResource** object, while the hardware has platform-specific AML code (contained in the appropriate control methods) to perform the desired function. In this example, the platform would describe its hardware to the ACPI OS by writing and placing the AML code to turn the hardware off within the _PS3 control method. This enables the following sequence:

When OSPM decides to place the IDE drive in the D3 state, it calls the IDE driver and tells it to place the drive into the D3 state (at which point the driver saves the device's context).

When the IDE driver returns control, OSPM places the drive in the D3 state.

OSPM finds the object associated with the HDD and then finds within that object any AML code associated with the D3 state.

OSPM executes the appropriate _PS3 control method to control the value-added "generic" hardware to place the HDD into an even lower power state.

As an example of a generic event feature, a platform might have a docking capability. In this case, it will want to generate an event. Notice that all ACPI events generate an SCI, which can be mapped to any shareable system interrupt. In the case of docking, the event is generated when a docking has been detected or when the user requests to undock the system. This enables the following sequence:

OSPM responds to the SCI and calls the AML code event handler associated with that generic event. The ACPI table associates the hardware event with the AML code event handler.

The AML-code event handler collects the appropriate information and then executes an AML Notify command to indicate to OSPM that a particular bus needs re-enumeration.

The following sections describe the fixed and generic hardware feature set of ACPI. These sections enable a reader to understand the following:

- Which hardware registers are required or optional when an ACPI feature, concept or interface is required by a design guide for a platform class
- How to design fixed hardware features
- How to design generic hardware features
- The ACPI Event Model

4.3 Diagram Legends

The hardware section uses simplified logic diagrams to represent how certain aspects of the hardware are implemented. The following symbols are used in the logic diagrams to represent programming bits.

⊽	Write-only control bit
⊗	Enable, control or status bit
☒	Sticky status bit
##	Query value

The half round symbol with an inverted “V” represents a write-only control bit. This bit has the behavior that it generates its control function when it is set. Reads to write-only bits are treated as ignore by software (the bit position is masked off and ignored).

The round symbol with an “X” represents a programming bit. As an enable or control bit, software setting or clearing this bit will result in the bit being read as set or clear (unless otherwise noted). As a status bit it directly represents the value of the signal.

The square symbol represents a sticky status bit. A sticky status bit is set by the level (not edge) of a hardware signal (active high or active low). The bit is only cleared by software writing a “1” to its bit position.

The rectangular symbol represents a query value from the embedded controller. This is the value the embedded controller returns to the system software upon a query command in response to an SCI event. The query value is associated with the event control method that is scheduled to execute upon an embedded controller event.

4.4 Register Bit Notation

Throughout this section there are logic diagrams that reference bits within registers. These diagrams use a notation that easily references the register name and bit position. The notation is as follows:

Registername.Bit

Registername contains the name of the register as it appears in this specification

Bit contains a zero-based decimal value of the bit position.

For example, the SLP_EN bit resides in the PM1x_CNT register bit 13 and would be represented in diagram notation as:

```
SLP_EN
PM1x_CNT.13
```

4.5 The ACPI Hardware Model

The ACPI hardware model is defined to allow OSPM to sequence the platform between the various global system states (G0-G3) as illustrated in the following figure by manipulating the defined interfaces. When first powered on, the platform finds itself in the global system state G3 or "Mechanical Off." This state is defined as one where power consumption is very close to zero—the power plug has been removed; however, the real-time clock device still runs off a battery. The G3 state is entered by any power failure, defined as accidental or user-initiated power loss.

The G3 state transitions into either the G0 working state or the Legacy state depending on what the platform supports. If the platform is an ACPI-only platform, then it allows a direct boot into the G0 working state by always returning the status bit SCI_EN set (1) (for more information, see section 4.7.2.5, "Legacy/ACPI Select and the SCI Interrupt"). If the platform supports both legacy and ACPI operations (which is necessary for supporting a non-ACPI OS), then it would always boot into the Legacy state (illustrated by returning the SCI_EN clear (0)). In either case, a transition out of the G3 state requires a total boot of OSPM.

The Legacy system state is the global state where a non-ACPI OS executes. This state can be entered from either the G3 "Mechanical Off," the G2 "Soft Off," or the G0 "Working" states only if the hardware supports both Legacy and ACPI modes. In the Legacy state, the ACPI event model is disabled (no SCIs are generated) and the hardware uses legacy power management and configuration mechanisms. While in the Legacy state, an ACPI-compliant OS can request a transition into the G0 working state by performing an ACPI mode request. OSPM performs this transition by writing the ACPI_ENABLE value to the SMI_CMD, which generates an event to the hardware to transition the platform into ACPI mode. When hardware has finished the transition, it sets the SCI_EN bit and returns control back to OSPM. While in the G0 "working state," OSPM can request a transition to Legacy mode by writing the ACPI_DISABLE value to the SMI_CMD register, which results in the hardware going into legacy mode and resetting the SCI_EN bit LOW (for more information, see section 4.7.2.5, "Legacy/ACPI Select and the SCI Interrupt").

The G0 "Working" state is the normal operating environment of an ACPI machine. In this state different devices are dynamically transitioning between their respective power states (D0, D1, D2 or D3) and processors are dynamically transitioning between their respective power states (C0, C1, C2 or C3). In this state, OSPM can make a policy decision to place the platform into the system G1 "sleeping" state. The platform can only enter a single sleeping state at a time (referred to as the global G1 state); however, the hardware can provide up to four system sleeping states that have different power and exit latencies represented by the S1, S2, S3, or S4 states. When OSPM decides to enter a sleeping state it picks the most appropriate sleeping state supported by the hardware (OS policy examines what devices have enabled wake events and what sleeping state these support). OSPM initiates the sleeping transition by enabling the appropriate wake events and then programming the SLP_TYPx field with the desired sleeping state and then setting the SLP_ENx bit. The system will then enter a sleeping state; when one of the enabled wake events occurs, it will transition the system back to the working state (for more information, see section 9, "Waking and Sleeping").

Another global state transition option while in the G0 "working" state is to enter the G2 "soft off" or the G3 "mechanical off" state. These transitions represent a controlled transition that allows OSPM to bring the system down in an orderly fashion (unloading applications, closing files, and so on). The policy for these types of transitions can be associated with the ACPI power button, which when pressed generates an event to the power button driver. When OSPM is finished preparing the operating environment for a power loss, it will either generate a pop-up message to indicate to the user to remove power, in order to enter the G3 "Mechanical Off" state, or it will initiate a G2 "soft-off" transition by writing the value of the S5 "soft off" system state to the SLP_TYPx register and setting the SLP_EN bit.

The G1 sleeping state is represented by five possible sleeping states that the hardware can support. Each sleeping state has different power and wake latency characteristics. The sleeping state differs from the working state in that the user's operating environment is frozen in a low-power state until awakened by an enabled wake event. No work is performed in this state, that is, the processors are not executing instructions. Each system sleeping state has requirements about who is responsible for system context and wake sequences (for more information, see section 9, Waking and Sleeping").

The G2 "soft off" state is an OS initiated system shutdown. This state is initiated similar to the sleeping state transition (SLP_TYPx is set to the S5 value and setting the SLP_EN bit initiates the sequence). Exiting the G2 soft-off state requires rebooting the system. In this case, an ACPI-only machine will re-enter the G0 state directly (hardware returns the SCI_EN bit set), while an ACPI/Legacy machine transitions to the Legacy state (SCI_EN bit is clear).

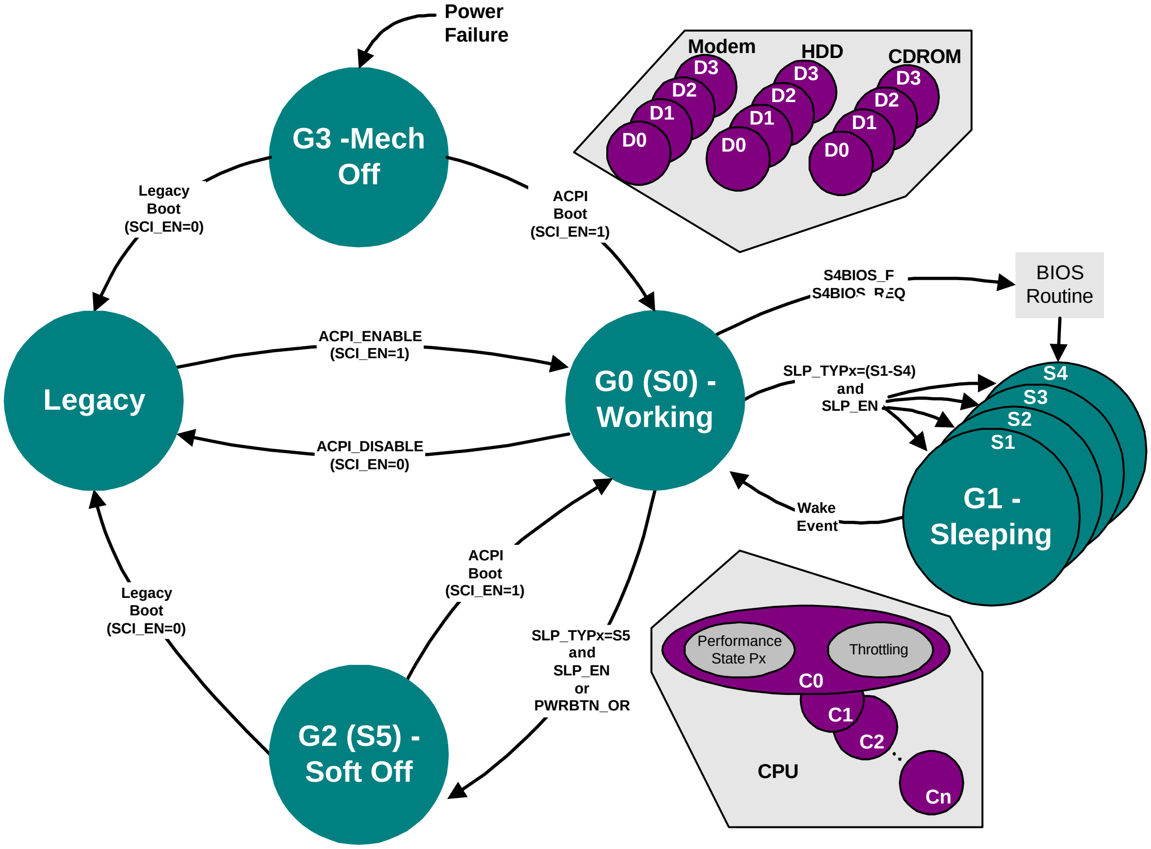

Figure 4-2 Global States and Their Transitions

The ACPI architecture defines mechanisms for hardware to generate events and control logic to implement this behavior model. Events are used to notify OSPM that some action is needed, and control logic is used by OSPM to cause some state transition. ACPI-defined events are "hardware" or "interrupt" events. A hardware event is one that causes the hardware to unconditionally perform some operation. For example, any wake event will sequence the system from a sleeping state (S1, S2, S3, and S4 in the global G1 state) to the G0 working state (see Figure 9-1).

An interrupt event causes the execution of an event handler (AML code or an ACPI-aware driver), which allows the software to make a policy decision based on the event. For ACPI fixed-feature events, OSPM or an ACPI-aware driver acts as the event handler. For generic logic events OSPM will schedule the execution of an OEM-supplied AML control method associated with the event.

For legacy systems, an event normally generates an OS-transparent interrupt, such as a System Management Interrupt, or SMI. For ACPI systems the interrupt events need to generate an OS-visible interrupt that is shareable; edge-style interrupts will not work. Hardware platforms that want to support both legacy operating systems and ACPI systems support a way of re-mapping the interrupt events between SMIs and SCIs when switching between ACPI and legacy models. This is illustrated in the following block diagram.

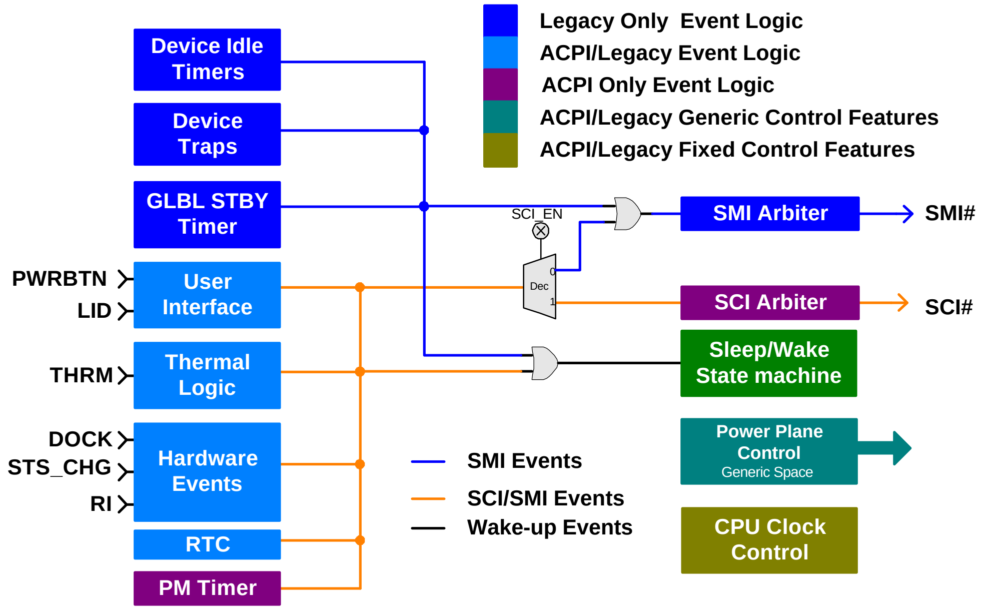

Figure 4-3 Example Event Structure for a Legacy/ACPI Compatible Event Model

This example logic illustrates the event model for a sample platform that supports both legacy and ACPI event models. This example platform supports a number of external events that are power-related (power button, LID open/close, thermal, ring indicate) or Plug and Play-related (dock, status change). The logic represents the three different types of events:

- **OS Transparent Events**. These events represent OEM-specific functions that have no OS support and use software that can be operated in an OS-transparent fashion (that is, SMIs).
- **Interrupt Events**. These events represent features supported by ACPI-compatible operating systems, but are not supported by legacy operating systems. When a legacy OS is loaded, these events are mapped to the transparent interrupt (SMI# in this example), and when in ACPI mode they are mapped to an OS-visible shareable interrupt (SCI#). This logic is represented by routing the event logic through the decoder that routes the events to the SMI# arbiter when the SCI_EN bit is cleared, or to the SCI# arbiter when the SCI_EN bit is set.
- **Hardware events**. These events are used to trigger the hardware to initiate some hardware sequence such as waking, resetting, or putting the machine to sleep unconditionally.

In this example, the legacy power management event logic is used to determine device/system activity or idleness based on device idle timers, device traps, and the global standby timer. Legacy power management models use the idle timers to determine when a device should be placed in a low-power state because it is idle—that is, the device has not been accessed for the programmed amount of time. The device traps are used to indicate when a device in a low-power state is being accessed by OSPM. The global standby timer is used to determine when the system should be allowed to go into a sleeping state because it is idle—that is, the user interface has not been used for the programmed amount of time.

These legacy idle timers, trap monitors, and global standby timer are not used by OSPM in the ACPI mode. This work is now handled by different software structures in an ACPI-compatible OS. For example, the driver model of an ACPI-compatible OS is responsible for placing its device into a low-power state (D1, D2, or D3) and transitioning it back to the On state (D0) when needed. And OSPM is responsible for determining when the system is idle by profiling the system (using the PM Timer) and other knowledge it gains through its operating structure environment (which will vary from OS to OS). When the system is placed into the ACPI mode, these events no longer generate SMIs, as drivers now handle this function. These events are disabled through some OEM-proprietary method.

On the other hand, many of the hardware events are shared between the ACPI and legacy models (docking, the power button, and so on) and this type of interrupt event changes to an SCI event when enabled for ACPI. The ACPI OS will generate a request to the platform's hardware (BIOS) to enter into the ACPI mode. The BIOS sets the SCI_EN bit to indicate that the system has successfully entered into the ACPI mode, so this is a convenient mechanism to map the desired interrupt (SMI or SCI) for these events (as shown in Figure 4-3).

The ACPI architecture specifies some dedicated hardware not found in the legacy hardware model: the power management timer (PM Timer). This is a free running timer that the ACPI OS uses to profile system activity. The frequency of this timer is explicitly defined in this specification and must be implemented as described.

Although the ACPI architecture reuses most legacy hardware as is, it does place restrictions on where and how the programming model is generated. If used, all fixed hardware features are implemented as described in this specification so that OSPM can directly access the fixed hardware feature registers.

Generic hardware features are manipulated by ACPI control methods residing in the ACPI Namespace. These interfaces can be very flexible; however, their use is limited by the defined ACPI control methods (for more information, see section 10, "ACPI-Specific Device Objects"). Generic hardware usually controls power planes, buffer isolation, and device reset resources. Additionally, "child" interrupt status bits can be accessed via generic hardware interfaces; however, they have a "parent" interrupt status bit in the GP_STS register. ACPI defines five address spaces where generic hardware may exist. These include:

- System I/O space
- System memory space
- PCI configuration space
- Embedded controller space
- System Management Bus (SMBus) space

Generic hardware power management features can be implemented using spare I/O ports residing in any of these I/O spaces. The ACPI specification defines an optional embedded controller and SMBus interfaces needed to communicate with these associated I/O spaces.

4.5.1 Hardware Reserved Bits

ACPI hardware registers are designed such that reserved bits always return zero, and data writes to them have no side affects. OSPM implementations must write zeros to reserved bits in enable and status registers and preserve bits in control registers, and they will treat these bits as ignored.

4.5.2 Hardware Ignored Bits

ACPI hardware registers are designed such that ignored bits are undefined and are ignored by software. Hardware-ignored bits can return zero or one. When software reads a register with ignored bits, it masks off ignored bits prior to operating on the result. When software writes to a register with ignored bit fields, it preserves the ignored bit fields.

4.5.3 Hardware Write-Only Bits

ACPI hardware defines a number of write-only control bits. These bits are activated by software writing a 1 to their bit position. Reads to write-only bit positions generate undefined results. Upon reads to registers with write-only bits, software masks out all write-only bits.

4.5.4 Cross Device Dependencies

Cross Device Dependency is a condition in which an operation to a device interferes with the operation of other unrelated devices, or allows other unrelated devices to interfere with its behavior. This condition is not supportable and can cause platform failures. ACPI provides no support for cross device dependencies and suggests that devices be designed to not exhibit this behavior. The following two examples describe cross device dependencies:

4.5.4.1 Example 1: Related Device Interference

This example illustrates a cross device dependency where a device interferes with the proper operation of other unrelated devices. Device A has a dependency that when it is being configured it blocks all accesses that would normally be targeted for Device B. Thus, the device driver for Device B cannot access Device B while Device A is being configured; therefore, it would need to synchronize access with the driver for Device A. High performance, multithreaded operating systems cannot perform this kind of synchronization without seriously impacting performance.

To further illustrate the point, assume that device A is a serial port and device B is a hard drive controller. If these devices demonstrate this behavior, then when a software driver configures the serial port, accesses to the hard drive need to block. This can only be done if the hard disk driver synchronizes access to the disk controller with the serial driver. Without this synchronization, hard drive data will be lost when the serial port is being configured.

4.5.4.2 Example 2: Unrelated Device Interference

This example illustrates a cross-device dependency where a device demonstrates a behavior that allows other unrelated devices to interfere with its proper operation. Device A exhibits a programming behavior that requires atomic back-to-back write accesses to successfully write to its registers; if any other platform access is able to break between the back-to-back accesses, then the write to device A is unsuccessful. If the device A driver is unable to generate atomic back-to-back accesses to its device, then it relies on software to synchronize accesses to its device with every other driver in the system; then a device cross dependency is created and the platform is prone to device A failure.

4.6 ACPI Hardware Features

This section describes the different hardware features defined by the ACPI interface. These features are categorized as the following:

- Fixed Hardware Features
- Generic Hardware Features

Fixed hardware features reside in a number of the ACPI-defined address spaces at the locations described by the ACPI programming model. Generic hardware features reside in one of five address spaces (system I/O, system memory, PCI configuration, embedded controller, or serial device I/O space) and are described by the ACPI Namespace through the declaration of AML control methods.

Fixed hardware features have exact definitions for their implementation. Although many fixed hardware features are optional, if implemented they must be implemented as described. This is necessary because a standard OS driver is talking to these registers and expects the defined behavior. Fixed functional hardware provides functional equivalents of the fixed hardware feature interfaces as described in section 4.1.1, "Functional Fixed Hardware."

Generic hardware feature implementation is flexible. This logic is controlled by OEM-supplied AML code (for more information, see section 5, "ACPI Software Programming Model"), which can be written to support a wide variety of hardware. Also, ACPI provides specialized control methods that provide capabilities for specialized devices. For example, the Notify command can be used to notify OSPM from a generic hardware event handler (control method) that a docking or thermal event has taken place. A good understanding of this section and section 5 of this specification will give designers a good understanding of how to design hardware to take full advantage of an ACPI-compatible OS.

Notice that the generic features are listed for illustration only, the ACPI specification can support many types of hardware not listed.

Table 4-1 Feature/Programming Model Summary

Feature Name	Description	Programming Model
Power Management Timer	24-bit/32-bit free running timer.	Fixed Hardware Feature Control Logic
Power Button	User pushes button to switch the system between the working and sleeping states.	Fixed Hardware Event and Control Logic or Generic Hardware Event and Logic
Sleep Button	User pushes button to switch the system between the working and sleeping state.	Fixed Hardware Event and Control Logic or Generic Hardware Event and Logic
Power Button Override	User sequence (press the power button for 4 seconds) to turn off a hung system.	
Real Time Clock Alarm	Programmed time to wake the system.	Optional Fixed Hardware Event[2]
Sleep/Wake Control Logic	Logic used to transition the system between the sleeping and working.states.	Fixed Hardware Control and Event Logic

[2] RTC wakeup alarm is required, the fixed hardware feature status bit is optional.

Table 4-1 Feature/Programming Model Summary *(continued)*

Feature Name	Description	Programming Model
Embedded Controller Interface	ACPI Embedded Controller protocol and interface, as described in section 13, “ACPI Embedded Controller Interface Specification.”	Generic Hardware Event Logic, must reside in the general-purpose register block
Legacy/ACPI Select	Status bit that indicates the system is using the legacy or ACPI power management model (SCI_EN).	Fixed Hardware Control Logic
Lid switch	Button used to indicate whether the system’s lid is open or closed (mobile systems only).	Generic Hardware Event Feature
C1 Power State	Processor instruction to place the processor into a low-power state.	Processor ISA
C2 Power Control	Logic to place the processor into a C2 power state.	Fixed Hardware Control Logic
C3 Power Control	Logic to place the processor into a C3 power state.	Fixed Hardware Control Logic
Thermal Control	Logic to generate thermal events at specified trip points.	Generic Hardware Event and Control Logic (See description of thermal logic in section 3.9, “Battery Management.”)
Device Power Management	Control logic for switching between different device power states.	Generic Hardware control logic
AC Adapter	Logic to detect the insertion and removal of the AC adapter.	Generic Hardware event logic
Docking/device insertion and removal	Logic to detect device insertion and removal events.	Generic Hardware event logic

4.7 ACPI Register Model

ACPI hardware resides in one of six address spaces:

- System I/O
- System memory
- PCI configuration
- SMBus
- Embedded controller
- Functional Fixed Hardware

Different implementations will result in different address spaces being used for different functions. The ACPI specification consists of fixed hardware registers and generic hardware registers. Fixed hardware registers are required to implement ACPI-defined interfaces. The generic hardware registers are needed for any events generated by value-added hardware.

ACPI defines register blocks. An ACPI-compatible system provides an ACPI table (the FADT, built in memory at boot-up) that contains a list of pointers to the different fixed hardware register blocks used by OSPM. The bits within these registers have attributes defined for the given register block. The types of registers that ACPI defines are:

- Status/Enable Registers (for events)
- Control Registers

If a register block is of the status/enable type, then it will contain a register with status bits, and a corresponding register with enable bits. The status and enable bits have an exact implementation definition that needs to be followed (unless otherwise noted), which is illustrated by the following diagram:

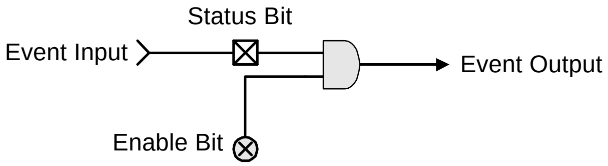

Figure 4-4 Block Diagram of a Status/Enable Cell

Notice that the status bit, which hardware sets by the Event Input being set in this example, can only be cleared by software writing a 1 to its bit position. Also, the enable bit has no effect on the setting or resetting of the status bit; it only determines if the SET status bit will generate an "Event Output," which generates an SCI when set if its enable bit is set.

ACPI also defines register groupings. A register grouping consists of two register blocks, with two pointers to two different blocks of registers, where each bit location within a register grouping is fixed and cannot be changed. The bits within a register grouping, which have fixed bit positions, can be split between the two register blocks. This allows the bits within a register grouping to reside in either or both register blocks, facilitating the ability to map bits within several different chips to the same register thus providing the programming model with a single register grouping bit structure.

OSPM treats a register grouping as a single register; but located in multiple places. To read a register grouping, OSPM will read the "A" register block, followed by the "B" register block, and then will logically "OR" the two results together (the SLP_TYP field is an exception to this rule). Reserved bits, or unused bits within a register block always return zero for reads and have no side effects for writes (which is a requirement).

The SLP_TYPx field can be different for each register grouping. The respective sleeping object _S*x* contains a SLP_TYPa and a SLP_TYPb field. That is, the object returns a package with two integer values of 0-7 in it. OSPM will always write the SLP_TYPa value to the "A" register block followed by the SLP_TYPb value within the field to the "B" register block. All other bit locations will be written with the same value. Also, OSPM does not read the SLP_TYPx value but throws it away.

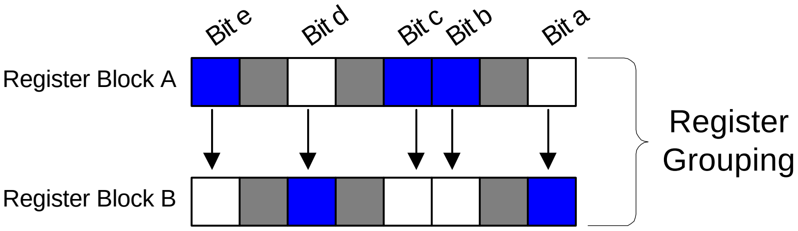

Figure 4-5 Example Fixed Hardware Feature Register Grouping

As an example, the above diagram represents a register grouping consisting of register block A and register block b. Bits "a" and "d" are implemented in register block B and register block A returns a zero for these bit positions. Bits "b", "c" and "e" are implemented in register block A and register block B returns a zero for these bit positions. All reserved or ignored bits return their defined ACPI values.

When accessing this register grouping, OSPM must read register block a, followed by reading register block b. OSPM then does a logical OR of the two registers and then operates on the results.

When writing to this register grouping, OSPM will write the desired value to register group A followed by writing the same value to register group B.

ACPI defines the following fixed hardware register blocks. Each register block gets a separate pointer from the FADT. These addresses are set by the OEM as static resources, so they are never changed—the Plug and Play driver cannot re-map ACPI resources. The following register blocks are defined:

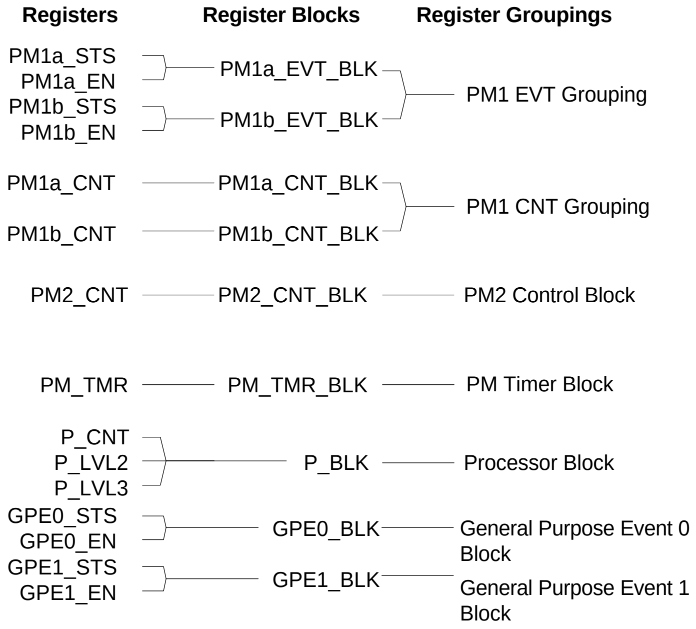

Figure 4-6 Register Blocks versus Register Groupings

The PM1 EVT grouping consists of the PM1a_EVT and PM1b_EVT register blocks, which contain the fixed hardware feature event bits. Each event register block (if implemented) contains two registers: a status register and an enable register. Each register grouping has a defined bit position that cannot be changed; however, the bit can be implemented in either register block (A or B). The A and B register blocks for the events allow chipsets to vary the partitioning of events into two or more chips. For read operations, OSPM will generate a read to the associated A and B registers, OR the two values together, and then operate on this result. For write operations, OSPM will write the value to the associated register in both register blocks. Therefore, there are a number of rules to follow when implementing event registers:

- Reserved or unimplemented bits always return zero (control or enable).
- Writes to reserved or unimplemented bits have no affect.

The PM1 CNT grouping contains the fixed hardware feature control bits and consists of the PM1a_CNT_BLK and PM1b_CNT_BLK register blocks. Each register block is associated with a single control register. Each register grouping has a defined bit position that cannot be changed; however, the bit can be implemented in either register block (A or B). There are a number of rules to follow when implementing CNT registers:

- Reserved or unimplemented bits always return zero (control or enable).
- Writes to reserved or unimplemented bits have no affect.

The PM2_CNT_BLK register block currently contains a single bit for the arbiter disable function. The general-purpose event register contains the event programming model for generic features. All generic events, just as fixed events, generate SCIs. Generic event status bits can reside anywhere; however, the top-level generic event resides in one of the general-purpose register blocks. Any generic feature event status not in the general-purpose register space is considered a child or sibling status bit, whose parent status bit is in the general-purpose event register space. Notice that it is possible to have N levels of general-purpose events prior to hitting the GPE event status.

General-purpose event registers are described by two register blocks: The GPE0_BLK or the GPE1_BLK. Each register block is pointed to separately from within the FADT. Each register block is further broken into two registers: GPEx_STS and GPEx_EN. The status and enable registers in the general-purpose event registers follow the event model for the fixed hardware event registers.

4.7.1 ACPI Register Summary

The following tables summarize the ACPI registers:

Table 4-2 PM1 Event Registers

Register	Size (Bytes)	Address (relative to register block)
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

Table 4-3 PM1 Control Registers

Register	Size (Bytes)	Address (relative to register block)
PM1_CNTa	PM1_CNT_LEN	<PM1a_CNT_BLK >
PM1_CNTb	PM1_CNT_LEN	<PM1b_CNT_BLK >

Table 4-4 PM2 Control Register

Register	Size (Bytes)	Address (relative to register block)
PM2_CNT	PM2_CNT_LEN	<PM2_CNT_BLK >

Table 4-5 PM Timer Register

Register	Size (Bytes)	Address (relative to register block)
PM_TMR	PM_TMR_LEN	<PM_TMR_BLK >

Table 4-6 Processor Control Registers

Register	Size (Bytes)	Address (relative to register block)
P_CNT	4	Either <P_BLK> or specified by the PTC object (See section 8.3.1, “PTC [Processor Throttling Control].”)
P_LVL2	1	<P_BLK>+4h
P_LVL3	1	<P_BLK>+5h

Table 4-7 General-Purpose Event Registers

Register	Size (Bytes)	Address (relative to register block)
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

4.7.1.1 PM1 Event Registers

The PM1 event register grouping contains two register blocks: the PM1a_EVT_BLK is a required register block when the following ACPI interface categories are required by a class specific platform design guide:

- Power management timer control/status
- Processor power state control/status
- Global Lock related interfaces
- Power or Sleep button (fixed register interfaces)
- System power state controls (sleeping/wake control)

The PM1b_EVT_BLK is an optional register block. Each register block has a unique 32-bit pointer in the Fixed ACPI Table (FADT) to allow the PM1 event bits to be partitioned between two chips. If the PM1b_EVT_BLK is not supported, its pointer contains a value of zero in the FADT.

Each register block in the PM1 event grouping contains two registers that are required to be the same size: the PM1x_STS and PM1x_EN (where x can be “a” or “b”). The length of the registers is variable and is described by the PM1_EVT_LEN field in the FADT, which indicates the total length of the register block in bytes. Hence if a length of “4” is given, this indicates that each register contains two bytes of I/O space. The PM1 event register block has a minimum size of 4 bytes.

4.7.1.2 PM1 Control Registers

The PM1 control register grouping contains two register blocks: the PM1a_CNT_BLK is a required register block when the following ACPI interface categories are required by a class specific platform design guide:

- SCI/SMI routing control/status for power management and general-purpose events
- Processor power state control/status
- Global Lock related interfaces
- System power state controls (sleeping/wake control)

The PM1b_CNT_BLK is an optional register block. Each register block has a unique 32-bit pointer in the Fixed ACPI Table (FADT) to allow the PM1 event bits to be partitioned between two chips. If the PM1b_CNT_BLK is not supported, its pointer contains a value of zero in the FADT.

Each register block in the PM1 control grouping contains a single register: the PM1x_CNT. The length of the register is variable and is described by the PM1_CNT_LEN field in the FADT, which indicates the total length of the register block in bytes. The PM1 control register block must have a minimum size of 2 bytes.

4.7.1.3 PM2 Control Register

The PM2 control register is contained in the PM2_CNT_BLK register block. The FADT contains a length variable for this register block (PM2_CNT_LEN) that is equal to the size in bytes of the PM2_CNT register (the only register in this register block). This register block is optional, if not supported its block pointer and length contains a value of zero.

4.7.1.4 PM Timer Register

The PM timer register is contained in the PM_TMR_BLK register block, which is a required register block when the power management timer control/status ACPI interface category is required by a class specific platform design guide.

This register block contains the register that returns the running value of the power management timer. The FADT also contains a length variable for this register block (PM_TMR_LEN) that is equal to the size in bytes of the PM_TMR register (the only register in this register block).

4.7.1.5 Processor Control Block (P_BLK)

There is an optional processor control register block for each processor in the system. As this is a homogeneous feature, all processors must have the same level of support. The ACPI OS will revert to the lowest common denominator of processor control block support. The processor control block contains the processor control register (P_CNT-a 32-bit performance control configuration register), and the P_LVL2 and P_LVL3 CPU sleep state control registers. The 32-bit P_CNT register controls the behavior of the processor clock logic for that processor, the P_LVL2 register is used to place the CPU into the C2 state, and the P_LVL3 register is used to place the processor into the C3 state.

4.7.1.6 General-Purpose Event Registers

The general-purpose event registers contain the root level events for all generic features. To facilitate the flexibility of partitioning the root events, ACPI provides for two different general-purpose event blocks: GPE0_BLK and GPE1_BLK. These are separate register blocks and are not a register grouping, because there is no need to maintain an orthogonal bit arrangement. Also, each register block contains its own length variable in the FADT, where GPE0_LEN and GPE1_LEN represent the length in bytes of each register block.

Each register block contains two registers of equal length: GPEx_STS and GPEx_EN (where x is 0 or 1). The length of the GPE0_STS and GPE0_EN registers is equal to half the GPE0_LEN. The length of the GPE1_STS and GPE1_EN registers is equal to half the GPE1_LEN. If a generic register block is not supported then its respective block pointer and block length values in the FADT table contain zeros. The GPE0_LEN and GPE1_LEN do not need to be the same size.

4.7.2 Fixed Hardware Features

This section describes the fixed hardware features defined by ACPI.

4.7.2.1 Power Management Timer

The ACPI specification requires a power management timer that provides an accurate time value used by system software to measure and profile system idleness (along with other tasks). The power management timer provides an accurate time function while the system is in the working (G0) state. To allow software to extend the number of bits in the timer, the power management timer generates an interrupt when the last bit of the timer changes (from 0 to 1 or 1 to 0). ACPI supports either a 24-bit or 32-bit power management timer. The PM Timer is accessed directly by OSPM, and its programming model is contained in fixed register space. The programming model can be partitioned in up to three different register blocks. The event bits are contained in the PM1_EVT register grouping, which has two register blocks, and the timer value can be accessed through the PM_TMR_BLK register block. A block diagram of the power management timer is illustrated in the following figure:

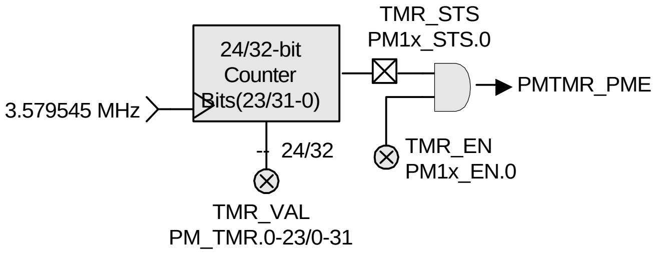

Figure 4-7 Power Management Timer

The power management timer is a 24-bit or 32-bit fixed rate free running count-up timer that runs off a 3.579545 MHz clock. The ACPI OS checks the FADT to determine whether the PM Timer is a 32-bit or 24-bit timer. The programming model for the PM Timer consists of event logic, and a read port to the counter value. The event logic consists of an event status and enable bit. The status bit is set any time the last bit of the timer (bit 23 or bit 31) goes from set to clear or clear to set. If the TMR_EN bit is set, then the setting of the TMR_STS will generate an ACPI event in the PM1_EVT register grouping (referred to as PMTMR_PME in the diagram). The event logic is only used to emulate a larger timer.

OSPM uses the read-only TMR_VAL field (in the PM TMR register grouping) to read the current value of the timer. OSPM never assumes an initial value of the TMR_VAL field; instead, it reads an initial TMR_VAL upon loading OSPM and assumes that the timer is counting. It is allowable to stop the Timer when the system transitions out of the working (G0/S0) state. The only timer reset requirement is that the timer functions while in the working state.

The PM Timer's programming model is implemented as a fixed hardware feature to increase the accuracy of reading the timer.

4.7.2.2 Buttons

ACPI defines user-initiated events to request OSPM to transition the platform between the G0 working state and the G1 sleeping, G2 soft off and G3 mechanical off states. ACPI also defines a recommended mechanism to unconditionally transition the platform from a hung G0 working state to the G2 soft-off state.

ACPI operating systems use power button events to determine when the user is present. As such, these ACPI events are associated with buttons in the ACPI specification.

The ACPI specification supports two button models:

- A single-button model that generates an event for both sleeping and entering the soft-off state. The function of the button can be configured using OSPM UI.
- A dual-button model where the power button generates a soft-off transition request and a sleeping button generates a sleeping transition request. The type of button implies the function of the button.

Control of these button events is either through the fixed hardware programming model or the generic hardware programming model (control method based). The fixed hardware programming model has the advantage that OSPM can access the button at any time, including when the system is crashed. In a crashed system with a fixed hardware power button, OSPM can make a “best” effort to determine whether the power button has been pressed to transition to the system to the soft-off state, because it doesn't require the AML interpreter to access the event bits.

4.7.2.2.1 Power Button

The power button logic can be used in one of two models: single button or dual button. In the single-button model, the user button acts as both a power button for transitioning the system between the G0 and G2 states and a sleeping button for transitioning the system between the G0 and G1 states. The action of the user pressing the button is determined by software policy or user settings. In the dual-button model, there are separate buttons for sleeping and power control. Although the buttons still generate events that cause software to take an action, the function of the button is now dedicated: the sleeping button generates a sleeping request to OSPM and the power button generates a waking request.

Support for a power button is indicated by a combination of the PWR_BUTTON flag and the power button device object, as shown in the following:

Table 4-8 Power Button Support

Indicated Support	PWR_BUTTON Flag	Power Button Device Object
Fixed hardware power button	Clear	Absent
Control method power button	Set	Present

The power button can also have an additional capability to unconditionally transition the system from a hung working state to the G2 soft-off state. In the case where OSPM event handler is no longer able to respond to power button events, the power button override feature provides a back-up mechanism to unconditionally transition the system to the soft-off state. This feature can be used when the platform doesn't have a mechanical off button, which can also provide this function. ACPI defines that holding the power button active for four seconds or longer will generate a power button override event.

4.7.2.2.1.1 Fixed Power Button

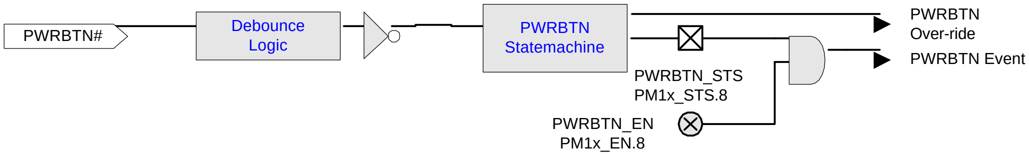

Figure 4-8 Fixed Power Button Logic

The fixed hardware power button has its event programming model in the PM1x_EVT_BLK. This logic consists of a single enable bit and sticky status bit. When the user presses the power button, the power button status bit (PWRBTN_STS) is unconditionally set. If the power button enable bit (PWRBTN_EN) is set and the power button status bit is set (PWRBTN_STS) due to a button press while the system is in the G0 state, then an SCI is generated. OSPM responds to the event by clearing the PWRBTN_STS bit. The power button logic provides debounce logic that sets the PWRBTN_STS bit on the button press "edge."

While the system is in the G1 or G2 global states (S1, S2, S3, S4 or S5 states), any further power button press after the button press that transitioned the system into the sleeping state unconditionally sets the power button status bit and wakes the system, regardless of the value of the power button enable bit. OSPM responds by clearing the power button status bit and waking the system.

4.7.2.2.1.2 Control Method Power Button

The power button programming model can also use the generic hardware programming model. This allows the power button to reside in any of the generic hardware address spaces (for example, the embedded controller) instead of fixed space. If the power button is implemented using generic hardware, then the OEM needs to define the power button as a device with an _HID object value of "PNP0C0C," which then identifies this device as the power button to OSPM. The AML event handler then generates a Notify command to notify OSPM that a power button event was generated. While the system is in the working state, a power button press is a user request to transition the system into either the sleeping (G1) or soft-off state (G2). In these cases, the power button event handler issues the Notify command with the device specific code of 0x80. This indicates to OSPM to pass control to the power button driver (PNP0C0C) with the knowledge that a transition out of the G0 state is being requested. Upon waking from a G1 sleeping state, the AML event handler generates a notify command with the code of 0x2 to indicate it was responsible for waking the system.

The power button device needs to be declared as a device within the ACPI Namespace for the platform and only requires an _HID. An example definition follows.

This example ASL code performs the following:

- Creates a device named “PWRB” and associates the Plug and Play identifier (through the _HID object) of “PNP0C0C.”
  The Plug and Play identifier associates this device object with the power button driver.
- Creates an operational region for the control method power button’s programming model:
  System I/O space at 0x200.
  Fields are not accessed are written as zeros. These status bits clear upon writing a 1 to their bit position, therefore preserved would fail in this case.
- Creates a field within the operational region for the power button status bit (called PBP). In this case the power button status bit is a child of the general-purpose event status bit 0. When this bit is set, it is the responsibility of the ASL-code to clear it (OSPM clears the general-purpose status bits). The address of the status bit is 0x200.0 (bit 0 at address 0x200).
- Creates an additional status bit called PBW for the power button wake event. This is the next bit and its physical address would be 0x200.1 (bit 1 at address 0x200).
- Generates an event handler for the power button that is connected to bit 0 of the general-purpose event status register 0. The event handler does the following:
  Clears the power button status bit in hardware (writes a one to it).
  Notifies OSPM of the event by calling the Notify command passing the power button object and the device specific event indicator 0x80.

```
// Define a control method power button
Device(\_SB.PWRB){
    Name(_HID, EISAID(“PNP0C0C”))
    Name(_PRW,Package(){0, 0x4})
    }

OperationRegion(\Pho, SystemIO, 0x200, 0x1)
Field(\Pho, ByteAcc, NoLock, WriteAsZeros){
    PBP, 1,    //  sleep/off request
    PBW, 1     //  wakeup request
    }          // end of power button device object

Scope(\_GPE){      // Root level event handlers
    Method(_L00){  // uses bit 0 of GP0_STS register
        If(PBP){
            Store(One, PBP)       // clear power button status
            Notify(\_SB.PWRB, 0x80)   // Notify OS of event
            }
        IF(PBW){
            Store(One, PBW)
            Notify(\_SB.PWRB, 0x2)
            }
        }   // end of _L00 handler
    }       // end of \_GPE scope
```

4.7.2.2.1.3 Power Button Override

The ACPI specification also allows that if the user presses the power button for more than four seconds while the system is in the working state, a hardware event is generated and the system will transition to the soft-off state. This hardware event is called a power button override. In reaction to the power button override event, the hardware clears the power button status bit (PWRBTN_STS).

4.7.2.2.2 Sleep Button

When using the two button model, ACPI supports a second button that when pressed will request OSPM to transition the platform between the G0 working and G1 sleeping states. Support for a sleep button is indicated by a combination of the SLEEP_BUTTON flag and the sleep button device object:

Table 4-9 Sleep Button Support

Indicated Support	SLEEP_BUTTON Flag	Sleep Button Device Object
No sleep button	Set	Absent
Fixed hardware sleep button	Clear	Absent
Control method sleep button	Set	Present

4.7.2.2.2.1 Fixed Hardware Sleeping Button

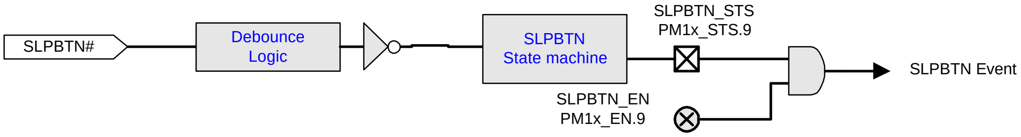

Figure 4-9 Fixed Hardware Sleep Button Logic

The fixed hardware sleep button has its event programming model in the PM1x_EVT_BLK. This logic consists of a single enable bit and sticky status bit. When the user presses the sleep button, the sleep button status bit (SLPBTN_STS) is unconditionally set. Additionally, if the sleep button enable bit (SLPBTN_EN) is set, and the sleep button status bit is set (SLPBTN_STS, due to a button press) while the system is in the G0 state, then an SCI is generated. OSPM responds to the event by clearing the SLPBTN_STS bit. The sleep button logic provides debounce logic that sets the SLPBTN_STS bit on the button press "edge."

While the system is sleeping (in either the S0, S1, S2, S3 or S4 states), any further sleep button press (after the button press that caused the system transition into the sleeping state) sets the sleep button status bit (SLPBTN_STS) and wakes the system if the SLP_EN bit is set. OSPM responds by clearing the sleep button status bit and waking the system.

4.7.2.2.2.2 Control Method Sleeping Button

The sleep button programming model can also use the generic hardware programming model. This allows the sleep button to reside in any of the generic hardware address spaces (for example, the embedded controller) instead of fixed space. If the sleep button is implemented via generic hardware, then the OEM needs to define the sleep button as a device with an _HID object value of "PNP0C0E", which then identifies this device as the sleep button to OSPM. The AML event handler then generates a Notify command to notify OSPM that a sleep button event was generated. While in the working state, a sleep button press is a user request to transition the system into the sleeping (G1) state. In these cases the sleep button event handler issues the Notify command with the device specific code of 0x80. This will indicate to OSPM to pass control to the sleep button driver (PNP0C0E) with the knowledge that the user is requesting a transition out of the G0 state. Upon waking-up from a G1 sleeping state, the AML event handler generates a Notify command with the code of 0x2 to indicate it was responsible for waking the system.

The sleep button device needs to be declared as a device within the ACPI Namespace for the platform and only requires an _HID. An example definition is shown below.

The AML code below does the following:

- Creates a device named “SLPB” and associates the Plug and Play identifier (through the _HID object) of “PNP0C0E.”
  The Plug and Play identifier associates this device object with the sleep button driver.
- Creates an operational region for the control method sleep button’s programming model:
  System I/O space at 0x201.
  Fields that are not accessed are written as “1s” (these status bits clear upon writing a “1” to their bit position, hence preserved would fail in this case).
- Creates a field within the operational region for the sleep button status bit (called PBP). In this case the sleep button status bit is a child of the general-purpose status bit 0. When this bit is set it is the responsibility of the AML code to clear it (OSPM clears the general-purpose status bits). The address of the status bit is 0x201.0 (bit 0 at address 0x201).
- Creates an additional status bit called PBW for the sleep button wake event. This is the next bit and its physical address would be 0x201.1 (bit 1 at address 0x201).
- Generates an event handler for the sleep button that is connected to bit 0 of the general-purpose status register 0. The event handler does the following:
  Clears the sleep button status bit in hardware (writes a “1” to it).
  Notifies OSPM of the event by calling the Notify command passing the sleep button object and the device specific event indicator 0x80.

```
// Define a control method sleep button
Device(\_SB.SLPB){
    Name(_HID, EISAID(“PNP0C0E”))
    Name(_PRW, Package(){0x01, 0x04})
    OperationRegion(\Boo, SystemIO, 0x201, 0x1)
    Field(\Boo, ByteAcc, NoLock, WriteAsZeros){
        SBP, 1,    //  sleep request
        SBW, 1     //  wakeup request
        }          // end of field definition
    }
Scope(\_GPE){      // Root level event handlers
    Method(_L01){  // uses bit 1 of GP0_STS register
        If(SBP){
            Store(One, SBP)       // clear sleep button status
            Notify(\_SB.SLPB, 0x80)       // Notify OS of event
            }
        IF(SBW){
            Store(One, SBW)
            Notify(\_SB.SLPB, 0x2)
            }
        }   // end of _L01 handler
    }       // end of \_GPE scope
```

4.7.2.3 Sleeping/Wake Control

The sleeping/wake logic consists of logic that will sequence the system into the defined low-power hardware sleeping state (S1-S4) or soft-off state (S5) and will wake the system back to the working state upon a wake event. Notice that the S4BIOS state is entered in a different manner (for more information, see section 9.1.4.2, “The S4BIOS Transistion”).

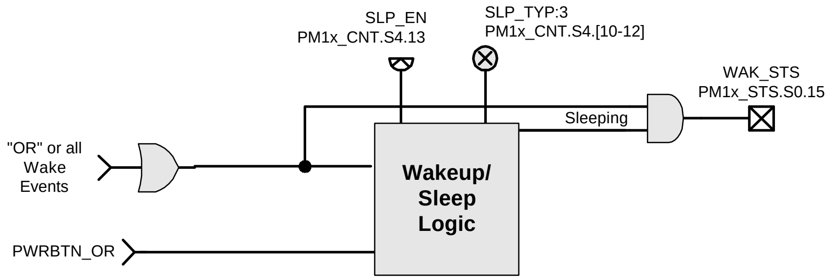

Figure 4-10 Sleeping/Wake Logic

The logic is controlled via two bit fields: Sleep Enable (SLP_EN) and Sleep Type (SLP_TYPx). The type of sleep state desired is programmed into the SLP_TYPx field and upon assertion of the SLP_EN the hardware will sequence the system into the defined sleeping state. OSPM gets values for the SLP_TYPx field from the _S*x* objects defined in the static definition block. If the object is missing OSPM assumes the hardware does not support that sleeping state. Prior to entering the desired sleeping state, OSPM will read the designated _S*x* object and place this value in the SLP_TYP field.

Additionally ACPI defines a fail-safe Off protocol called the “power button override,” which allows the user to initiate an Off sequence in the case where the system software is no longer able to recover the system (the system has hung). ACPI defines that this sequence be initiated by the user pressing the power button for over 4 seconds, at which point the hardware unconditionally sequences the system to the Off state. This logic is represented by the PWRBTN_OR signal coming into the sleep logic.

While in any of the sleeping states (G1), an enabled “Wake” event will cause the hardware to sequence the system back to the working state (G0). The “Wake Status” bit (WAK_STS) is provided for OSPM to “spin-on” after setting the SLP_EN/SLP_TYP bit fields. When waking from the S1 sleeping state, execution control is passed backed to OSPM immediately, whereas when waking from the S2-S5 states execution control is passed to the BIOS software (execution begins at the CPU’s reset vector). The WAK_STS bit provides a mechanism to separate OSPM’s sleeping and waking code during an S1 sequence. When the hardware has sequenced the system into the sleeping state (defined here as the processor is no longer able to execute instructions), any enabled wake event is allowed to set the WAK_STS bit and sequence the system back on (to the G0 state). If the system does not support the S1 sleeping state, the WAK_STS bit can always return zero.

-If more than a single sleeping state is supported, then the sleeping/wake logic is required to be able to dynamically sequence between the different sleeping states. This is accomplished by waking the system; OSPM programs the new sleep state into the SLP_TYP field, and then sets the SLP_EN bit–placing the system again in the sleeping state.

4.7.2.4 Real Time Clock Alarm

If implemented, the Real Time Clock (RTC) alarm must generate a hardware wake event when in the sleeping state. The RTC can be programmed to generate an alarm. An enabled RTC alarm can be used to generate a wake event when the system is in a sleeping state. ACPI provides for additional hardware to support OSPM in determining that the RTC was the source of the wake event: the RTC_STS and RTC_EN bits. Although these bits are optional, if supported they must be implemented as described here.

If the RTC_STS and RTC_EN bits are not supported, OSPM will attempt to identify the RTC as a possible wake source; however, it might miss certain wake events. If implemented the RTC wake feature is required to work in the following sleeping states: S1-S3. S4 wake is optional and supported through the RTC_S4 flag within the FADT (if set, then the platform supports RTC wake in the S4 state)[3].

When the RTC generates an alarm event the RTC_STS bit will be set. If the RTC_EN bit is set, an RTC hardware power management event will be generated (which will wake the system from a sleeping state, provided the battery low signal is not asserted).

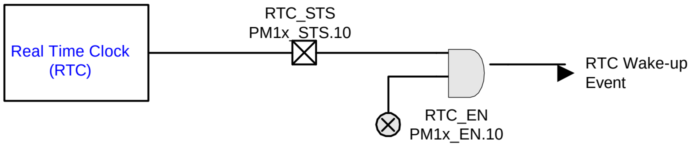

Figure 4-11 RTC Alarm

The RTC wake event status and enable bits are an optional fixed hardware feature and a flag within the FADT (FIXED_RTC) indicates if the register bits are to be used by OSPM. If the RTC wake event status and enable bits are implemented in fixed hardware, OSPM can determine if the RTC was the source of the wake event without loading the entire OS. If the fixed hardware feature event bits are not supported, then OSPM will attempt to determine this by reading the RTC's status field.

OSPM supports enhancements over the existing RTC device (which only supports a 99 year date and 24-hour alarm). Optional extensions are provided for the following features:

- **Day Alarm**. The DAY_ALRM field points to an optional CMOS RAM location that selects the day within the month to generate an RTC alarm.
- **Month Alarm**. The MON_ALRM field points to an optional CMOS RAM location that selects the month within the year to generate an RTC alarm.
- **Centenary Value**. The CENT field points to an optional CMOS RAM location that represents the centenary value of the date (thousands and hundreds of years).

[3] Notice that the G2/S5 “soft off” and the G3 “mechanical off” states are not sleeping states. The OS will disable the RTC_EN bit prior to entering the G2/S5 or G3 states regardless.

The RTC_STS bit is set through the RTC interrupt (IRQ8 in IA-PC architecture systems). OSPM will insure that the periodic and update interrupt sources are disabled prior to sleeping. This allows the RTC's interrupt pin to serve as the source for the RTC_STS bit generation.

Table 4-10 Alarm Field Decodings within the FADT

Field	**Value**	**Address (Location) in RTC CMOS RAM (Must be Bank 0)**
DAY_ALRM	Eight bit value that can represent 0x01-0x31 days in BCD or 0x01-0x1F days in binary. Bits 6 and 7 of this field are treated as Ignored by software. The RTC is initialized such that this field contains a "don't care" value when the BIOS switches from legacy to ACPI mode. A don't care value can be any unused value (not 0x1-0x31 BCD or 0x01-0x1F hex) that the RTC reverts back to a 24 hour alarm.	The DAY_ALRM field in the FADT will contain a non-zero value that represents an offset into the RTC's CMOS RAM area that contains the day alarm value. A value of zero in the DAY_ALRM field indicates that the day alarm feature is not supported.
MON_ALRM	Eight bit value that can represent 01-12 months in BCD or 0x01-0xC months in binary. The RTC is initialized such that this field contains a don't care value when the BIOS switches from legacy to ACPI mode. A "don't care" value can be any unused value (not 1-12 BCD or x01-xC hex) that the RTC reverts back to a 24 hour alarm and/or 31 day alarm).	The MON_ALRM field in the FADT will contain a non-zero value that represents an offset into the RTC's CMOS RAM area that contains the month alarm value. A value of zero in the MON_ALRM field indicates that the month alarm feature is not supported. If the month alarm is supported, the day alarm function must also be supported.
CENTURY	8-bit BCD or binary value. This value indicates the thousand year and hundred year (Centenary) variables of the date in BCD (19 for this century, 20 for the next) or binary (x13 for this century, x14 for the next).	The CENTURY field in the FADT will contain a non-zero value that represents an offset into the RTC's CMOS RAM area that contains the Centenary value for the date. A value of zero in the CENTURY field indicates that the Centenary value is not supported by this RTC.

4.7.2.5 Legacy/ACPI Select and the SCI Interrupt

As mentioned previously, power management events are generated to initiate an interrupt or hardware sequence. ACPI operating systems use the SCI interrupt handler to respond to events, while legacy systems use some type of transparent interrupt handler to respond to these events (that is, an SMI interrupt handler). ACPI-compatible hardware can choose to support both legacy and ACPI modes or just an ACPI mode. Legacy hardware is needed to support these features for non-ACPI-compatible operating systems. When the ACPI OS loads, it scans the BIOS tables to determine that the hardware supports ACPI, and then if the it finds the SCI_EN bit reset (indicating that ACPI is not enabled), issues an ACPI activate command to the SMI handler through the SMI command port. The BIOS acknowledges the switching to the ACPI model of power management by setting the SCI_EN bit (this bit can also be used to switch over the event mechanism as illustrated below):

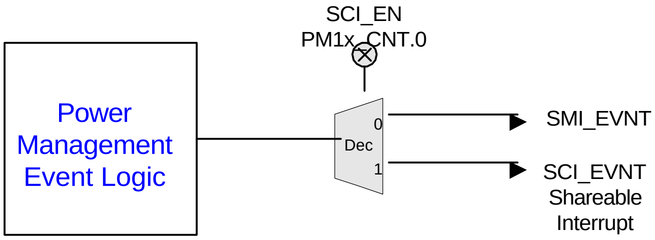

Figure 4-12 Power Management Events to SMI/SCI Control Logic

The interrupt events (those that generate SMIs in legacy mode and SCIs in ACPI mode) are sent through a decoder controlled by the SCI_EN bit. For legacy mode this bit is reset, which routes the interrupt events to the SMI interrupt logic. For ACPI mode this bit is set, which routes interrupt events to the SCI interrupt logic. This bit always returns set for ACPI-compatible hardware that does not support a legacy power management mode (in other words, the bit is wired to read as “1” and ignore writes).

The SCI interrupt is defined to be a shareable interrupt and is connected to an OS visible interrupt that uses a shareable protocol. The FADT has an entry that indicates what interrupt the SCI interrupt is mapped to (see section 5.2.5, “System Description Table Header”).

If the ACPI platform supports both legacy and ACPI modes, it has a register that generates a hardware event (for example, SMI for IA-PC processors). OSPM uses this register to make the hardware to switch in and out of ACPI mode. Within the FADT are three values that signify the address (SMI_CMD) of this port and the data value written to enable the ACPI state (ACPI_ENABLE), and to disable the ACPI state (ACPI_DISABLE).

To transition an ACPI/Legacy platform from the Legacy mode to the ACPI mode the following would occur:

ACPI driver checks that the SCI_EN bit is zero, and that it is in the Legacy mode.

OSPM does an OUT to the SMI_CMD port with the data in the ACPI_ENABLE field of the FADT.

OSPM polls the SCI_EN bit until it is sampled as SET.

To transition an ACPI/Legacy platform from the ACPI mode to the Legacy mode the following would occur:

ACPI driver checks that the SCI_EN bit is one, and that it is in the ACPI mode.

OSPM does an OUT to the SMI_CMD port with the data in the ACPI_DISABLE field of the FADT.

OSPM polls the SCI_EN bit until it is sampled as RESET.

Platforms that only support ACPI always return a 1 for the SCI_EN bit. In this case OSPM skips the Legacy to ACPI transition stated above.

4.7.2.6 Processor Control

The ACPI specification defines several processor controls including power state control, throttling control, and performance state control. See Section 8, “Processor Control,” for a complete description of the processor controls.

4.7.3 Fixed Hardware Registers

The fixed hardware registers are manipulated directly by OSPM. The following sections describe fixed hardware features under the programming model. OSPM owns all the fixed hardware resource registers; these registers cannot be manipulated by AML code. Registers are accessed with any width up to its register width (byte granular).

4.7.3.1 PM1 Event Grouping

The PM1 Event Grouping has a set of bits that can be distributed between two different register blocks. This allows these registers to be partitioned between two chips, or all placed in a single chip. Although the bits can be split between the two register blocks (each register blocks has a unique pointer within the FADT), the bit positions are maintained. The register block with unimplemented bits (that is, those implemented in the other register block) always returns zeros, and writes have no side effects.

4.7.3.1.1 PM1 Status Registers

```
Register Location: <PM1a_EVT_BLK/PM1b_EVT_BLK> System I/O or Memory Space
Default Value:     00h
Attribute:         Read/Write
Size:              PM1_EVT_LEN/2
```

The PM1 status registers contain the fixed hardware feature status bits. The bits can be split between two registers: PM1a_STS or PM1b_STS. Each register grouping can be at a different 32-bit aligned address and is pointed to by the PM1a_EVT_BLK or PM1b_EVT_BLK. The values for these pointers to the register space are found in the FADT. Accesses to the PM1 status registers are done through byte or word accesses.

For ACPI/legacy systems, when transitioning from the legacy to the G0 working state this register is cleared by BIOS prior to setting the SCI_EN bit (and thus passing control to OSPM). For ACPI only platforms (where SCI_EN is always set), when transitioning from either the mechanical off (G3) or soft-off state to the G0 working state this register is cleared prior to entering the G0 working state.

This register contains optional features enabled or disabled within the FADT. If the FADT indicates that the feature is not supported as a fixed hardware feature, then software treats these bits as ignored.

Table 4-11 PM1 Status Registers Fixed Hardware Feature Status Bits

Bit	Name	Description
0	TMR_STS	This is the timer carry status bit. This bit gets set any time the 23rd/31st bit of a 24/32-bit counter changes (whenever the MSB changes from clear to set or set to clear. While TMR_EN and TMR_STS are set, an interrupt event is raised.
1-3	Reserved	Reserved
4	BM_STS	This is the bus master status bit. This bit is set any time a system bus master requests the system bus, and can only be cleared by writing a “1” to this bit position. Notice that this bit reflects bus master activity, not CPU activity (this bit monitors any bus master that can cause an incoherent cache for a processor in the C3 state when the bus master performs a memory transaction).

Table 4-11 PM1 Status Registers Fixed Hardware Feature Status Bits *(continued)*

Bit	Name	Description
5	GBL_STS	This bit is set when an SCI is generated due to the BIOS wanting the attention of the SCI handler. BIOS will have a control bit (somewhere within its address space) that will raise an SCI and set this bit. This bit is set in response to the BIOS releasing control of the Global Lock and having seen the pending bit set.
6-7	Reserved	Reserved. These bits always return a value of zero.
8	PWRBTN_STS	This optional bit is set when the Power Button is pressed. In the system working state, while PWRBTN_EN and PWRBTN_STS are both set, an interrupt event is raised. In the sleeping or soft-off state, a wake event is generated when the power button is pressed (regardless of the PWRBTN_EN bit setting). This bit is only set by hardware and can only be reset by software writing a “1” to this bit position. ACPI defines an optional mechanism for unconditional transitioning a system that has stopped working from the G0 working state into the G2 soft-off state called the power button override. If the Power Button is held active for more than four seconds, this bit is cleared by hardware and the system transitions into the G2/S5 Soft Off state (unconditionally). Support for the power button is indicated by the PWR_BUTTON flag in the FADT being reset (zero). If the PWR_BUTTON flag is set or a power button device object is present in the ACPI Namespace, then this bit field is ignored by OSPM. If the power button was the cause of the wake (from an S1-S4 state), then this bit is set prior to returning control to OSPM.
9	SLPBTN_STS	This optional bit is set when the sleep button is pressed. In the system working state, while SLPBTN_EN and SLPBTN_STS are both set, an interrupt event is raised. In the sleeping or soft-off states a wake event is generated when the sleeping button is pressed and the SLPBTN_EN bit is set. This bit is only set by hardware and can only be reset by software writing a “1” to this bit position. Support for the sleep button is indicated by the SLP_BUTTON flag in the FADT being reset (zero). If the SLP_BUTTON flag is set or a sleep button device object is present in the ACPI Namespace, then this bit field is ignored by OSPM. If the sleep button was the cause of the wake (from an S1-S4 state), then this bit is set prior to returning control to OSPM.

Table 4-11 PM1 Status Registers Fixed Hardware Feature Status Bits *(continued)*

Bit	Name	Description
10	RTC_STS	This optional bit is set when the RTC generates an alarm (asserts the RTC IRQ signal). Additionally, if the RTC_EN bit is set then the setting of the RTC_STS bit will generate a power management event (an SCI, SMI, or resume event). This bit is only set by hardware and can only be reset by software writing a "1" to this bit position. If the RTC was the cause of the wake (from an S1-S3 state), then this bit is set prior to returning control to OSPM. If the RTC_S4 flag within the FADT is set, and the RTC was the cause of the wake from the S4 state), then this bit is set prior to returning control to OSPM.
11	Ignore	This bit field is ignored by software.
12-14	Reserved	Reserved. These bits always return a value of zero.
15	WAK_STS	This bit is set when the system is in the sleeping state and an enabled wake event occurs. Upon setting this bit system will transition to the working state. This bit is set by hardware and can only be cleared by software writing a "1" to this bit position.

4.7.3.1.2 PM1 Enable Registers

```
Register Location: <PM1a_EVT_BLK/PM1b_EVT_BLK>+PM1_EVT_LEN/2 System I/O or Memory Space
Default Value:     00h
Attribute:         Read/Write
Size:              PM1_EVT_LEN/2
```

The PM1 enable registers contain the fixed hardware feature enable bits. The bits can be split between two registers: PM1a_EN or PM1b_EN. Each register grouping can be at a different 32-bit aligned address and is pointed to by the PM1a_EVT_BLK or PM1b_EVT_BLK. The values for these pointers to the register space are found in the FADT. Accesses to the PM1 Enable registers are done through byte or word accesses.

For ACPI/legacy systems, when transitioning from the legacy to the G0 working state the enables are cleared by BIOS prior to setting the SCI_EN bit (and thus passing control to OSPM). For ACPI-only platforms (where SCI_EN is always set), when transitioning from either the mechanical off (G3) or soft-off state to the G0 working state this register is cleared prior to entering the G0 working state.

This register contains optional features enabled or disabled within the FADT. If the FADT indicates that the feature is not supported as a fixed hardware feature, then software treats the enable bits as write as zero.

Table 4-12 PM1 Enable Registers Fixed Hardware Feature Enable Bits

Bit	Name	Description
0	TMR_EN	This is the timer carry interrupt enable bit. When this bit is set then an SCI event is generated anytime the TMR_STS bit is set. When this bit is reset then no interrupt is generated when the TMR_STS bit is set.
1-4	Reserved	Reserved. These bits always return a value of zero.
5	GBL_EN	The global enable bit. When both the GBL_EN bit and the GBL_STS bit are set, an SCI is raised.
6-7	Reserved	Reserved
8	PWRBTN_EN	This optional bit is used to enable the setting of the PWRBTN_STS bit to generate a power management event (SCI or wake). The PWRBTN_STS bit is set anytime the power button is asserted. The enable bit does not have to be set to enable the setting of the PWRBTN_STS bit by the assertion of the power button (see description of the power button hardware). Support for the power button is indicated by the PWR_BUTTON flag in the FADT being reset (zero). If the PWR_BUTTON flag is set or a power button device object is present in the ACPI Namespace, then this bit field is ignored by OSPM.
9	SLPBTN_EN	This optional bit is used to enable the setting of the SLPBTN_STS bit to generate a power management event (SCI or wake). The SLPBTN_STS bit is set anytime the sleep button is asserted. The enable bit does not have to be set to enable the setting of the SLPBTN_STS bit by the active assertion of the sleep button (see description of the sleep button hardware). Support for the sleep button is indicated by the SLP_BUTTON flag in the FADT being reset (zero). If the SLP_BUTTON flag is set or a sleep button device object is present in the ACPI Namespace, then this bit field is ignored by OSPM.
10	RTC_EN	This optional bit is used to enable the setting of the RTC_STS bit to generate a wake event. The RTC_STS bit is set any time the RTC generates an alarm.
11-15	Reserved	Reserved. These bits always return a value of zero.

4.7.3.2 PM1 Control Grouping

The PM1 Control Grouping has a set of bits that can be distributed between two different registers. This allows these registers to be partitioned between two chips, or all placed in a single chip. Although the bits can be split between the two register blocks (each register block has a unique pointer within the FADT), the bit positions specified here are maintained. The register block with unimplemented bits (that is, those implemented in the other register block) returns zeros, and writes have no side effects.

4.7.3.2.1 PM1 Control Registers

```
Register Location: <PM1a_CNT_BLK/PM1b_CNT_BLK>   System I/O or Memory Space
Default Value:     00h
Attribute:         Read/Write
Size:              PM1_CNT_LEN
```

The PM1 control registers contain the fixed hardware feature control bits. These bits can be split between two registers: PM1a_CNT or PM1b_CNT. Each register grouping can be at a different 32-bit aligned address and is pointed to by the PM1a_CNT_BLK or PM1b_CNT_BLK. The values for these pointers to the register space are found in the FADT. Accesses to PM1 control registers are accessed through byte and word accesses.

This register contains optional features enabled or disabled within the FADT. If the FADT indicates that the feature is not supported as a fixed hardware feature, then software treats these bits as ignored.

Table 4-13 PM1 Control Registers Fixed Hardware Feature Control Bits

Bit	Name	Description
0	SCI_EN	Selects the power management event to be either an SCI or SMI interrupt for the following events. When this bit is set, then power management events will generate an SCI interrupt. When this bit is reset power management events will generate an SMI interrupt. It is the responsibility of the hardware to set or reset this bit. OSPM always preserves this bit position.
1	BM_RLD	When set, this bit allows the generation of a bus master request to cause any processor in the C3 state to transition to the C0 state. When this bit is reset, the generation of a bus master request does not affect any processor in the C3 state.
2	GBL_RLS	This write-only bit is used by the ACPI software to raise an event to the BIOS software, that is, generates an SMI to pass execution control to the BIOS for IA-PC platforms. BIOS software has a corresponding enable and status bit to control its ability to receive ACPI events (for example, BIOS_EN and BIOS_STS). The GBL_RLS bit is set by OSPM to indicate a release of the Global Lock and the setting of the pending bit in the FACS memory structure.
3-8	Reserved	Reserved. These bits are reserved by OSPM.
9	Ignore	Software ignores this bit field.
10-12	SLP_TYPx	Defines the type of sleeping state the system enters when the SLP_EN bit is set to one. This 3-bit field defines the type of hardware sleep state the system enters when the SLP_EN bit is set. The _S*x* object contains 3-bit binary values associated with the respective sleeping state (as described by the object). OSPM takes the two values from the _S*x* object and programs each value into the respective SLP_TYPx field.
13	SLP_EN	This is a write-only bit and reads to it always return a zero. Setting this bit causes the system to sequence into the sleeping state associated with the SLP_TYPx fields programmed with the values from the _S*x* object.
14-15	Reserved	Reserved. This field always returns zero.

4.7.3.3 Power Management Timer (PM_TMR)

```
Register Location: <PM_TMR_BLK>   System I/O or Memory Space
Default Value:     00h
Attribute:         Read-Only
Size:              32-bits
```

This read-only register returns the current value of the power management timer (PM timer). The FADT has a flag called TMR_VAL_EXT that an OEM sets to indicate a 32-bit PM timer or reset to indicate a 24-bit PM timer. When the last bit of the timer toggles the TMR_STS bit is set. This register is accessed as 32 bits.

This register contains optional features enabled or disabled within the FADT. If the FADT indicates that the feature is not supported as a fixed hardware feature, then software treats these bits as ignored.

Table 4-14 PM Timer Bits

Bit	Name	Description
0-23	TMR_VAL	This read-only field returns the running count of the power management timer. This is a 24-bit counter that runs off a 3.579545-MHz clock and counts while in the S0 working system state. The starting value of the timer is undefined, thus allowing the timer to be reset (or not) by any transition to the S0 state from any other state. The timer is reset (to any initial value), and then continues counting until the system's 14.31818 MHz clock is stopped upon entering its S*x* state. If the clock is restarted without a reset, then the counter will continue counting from where it stopped.
24-31	E_TMR_VAL	This read-only field returns the upper eight bits of a 32-bit power management timer. If the hardware supports a 32-bit timer, then this field will return the upper eight bits; if the hardware supports a 24-bit timer then this field returns all zeros.

4.7.3.4 PM2 Control (PM2_CNT)

```
Register Location: <PM2_CNT_BLK>  System I/O, System Memory, or
                                  Functional Fixed Hardware Space
Default Value:     00h
Attribute:         Read/Write
Size:              PM2_CNT_LEN
```

This register block is naturally aligned and accessed based on its length. For ACPI 1.0 this register is byte aligned and accessed as a byte.

This register contains optional features enabled or disabled within the FADT. If the FADT indicates that the feature is not supported as a fixed hardware feature, then software treats these bits as ignored.

Table 4-15 PM2 Control Register Bits

Bit	Name	Description
0	ARB_DIS	This bit is used to enable and disable the system arbiter. When this bit is CLEAR the system arbiter is enabled and the arbiter can grant the bus to other bus masters. When this bit is SET the system arbiter is disabled and the default CPU has ownership of the system. OSPM clears this bit when using the C0, C1 and C2 power states.
1-7	Reserved	Reserved

4.7.3.5 Processor Register Block (P_BLK)

This optional register block is used to control each processor in the system. There is one unique processor register block per processor in the system. For more information about controlling processors and control methods that can be used to control processors, see section 8, “Processor Control.” This register block is DWORD aligned and the context of this register block is not maintained across S3 or S4 sleeping states, or the S5 soft-off state.

4.7.3.5.1 Processor Control (P_CNT): 32

```
Register Location: Either <P_BLK>: System I/O Space or
                          Specified by _PTC Object: System I/O, System Memory, or
                          Functional Fixed Hardware Space
Default Value:     00h
Attribute:         Read/Write
Size:              32-bits
```

This register is accessed as a DWORD. The CLK_VAL field is where the duty setting of the throttling hardware is programmed as described by the DUTY_WIDTH and DUTY_OFFSET values in the FADT. Software treats all other CLK_VAL bits as ignored (those not used by the duty setting value).

Table 4-16 Processor Control Register Bits

Bit	Name	Description
0-3	CLK_VAL	Possible locations for the clock throttling value.
4	THT_EN	This bit enables clock throttling of the clock as set in the CLK_VAL field. THT_EN bit must be reset LOW when changing the CLK_VAL field (changing the duty setting).
5-31	CLK_VAL	Possible locations for the clock throttling value.

4.7.3.5.2 Processor LVL2 Register (P_LVL2): 8

```
Register Location: Either <P_BLK>+4: System I/O Space or
                          Specified by _CST Object: System I/O, System Memory, or
                          Functional Fixed Hardware Space
Default Value:     00h
Attribute:         Read-Only
Size:              8-bits
```

This register is accessed as a byte.

Table 4-17 Processor LVL2 Register Bits

Bit	Name	Description
0-7	P_LVL2	Reads to this register return all zeros; writes to this register have no effect. Reads to this register also generate an “enter a C2 power state” to the clock control logic.

4.7.3.5.3 Processor LVL3 Register (P_LVL3): 8

```
Register Location: Either <P_BLK>+5: System I/O Space or
                         Specified by _CST Object: System I/O, System Memory, or
                         Functional Fixed Hardware Space
Default Value:     00h
Attribute:         Read-Only
Size:              8-bits
```

This register is accessed as a byte.

Table 4-18 Processor LVL3 Register Bits

Bit	Name	Description
0-7	P_LVL3	Reads to this register return all zeros; writes to this register have no effect. Reads to this register also generate an “enter a C3 power state” to the clock control logic.

4.7.3.6 Reset Register

The optional ACPI reset mechanism specifies a standard mechanism that provides a complete system reset. When implemented, this mechanism must reset the entire system. This includes processors, core logic, all buses, and all peripherals. From an OSPM perspective, asserting the reset mechanism is the logical equivalent to power cycling the machine. Upon gaining control after a reset, OSPM will perform actions in like manner to a cold boot.

The reset mechanism is implemented via an 8-bit register described by RESET_REG in the FADT (always accessed via the natural alignment and size described in RESET_REG). To reset the machine, software will write a value (indicated in RESET_VALUE in FADT) to the reset register. The RESET_REG field in the FADT indicates the location of the reset register.

The reset register may exist only in I/O space, Memory space, or in PCI Configuration space on a function in bus 0. Therefore, the Address_Space_ID value in RESET_REG must be set to I/O space, Memory space, or PCI Configuration space (with a bus number of 0). As the register is only 8 bits, Register_Bit_Width must be 8 and Register_Bit_Offset must be 0.

The system must reset immediately following the write to this register. OSPM assumes that the processor will not execute beyond the write instruction. OSPM should execute spin loops on the CPUs in the system following a write to this register.

4.7.4 Generic Hardware Registers

ACPI provides a mechanism that allows a unique piece of “value added” hardware to be described to OSPM in the ACPI Namespace. There are a number of rules to be followed when designing ACPI-compatible hardware.

Programming bits can reside in any of the defined generic hardware address spaces (system I/O, system memory, PCI configuration, embedded controller, or SMBus), but the top-level event bits are contained in the general-purpose event registers. The general-purpose event registers are pointed to by the GPE0_BLK and GPE1_BLK register blocks, and the generic hardware registers can be in any of the defined ACPI address spaces. A device’s generic hardware programming model is described through an associated object in the ACPI Namespace, which specifies the bit’s function, location, address space, and address location.

The programming model for devices is normally broken into status and control functions. Status bits are used to generate an event that allows OSPM to call a control method associated with the pending status bit. The called control method can then control the hardware by manipulating the hardware control bits or by investigating child status bits and calling their respective control methods. ACPI requires that the top level “parent” event status and enable bits reside in either the GPE0_STS or GPE1_STS registers, and “child” event status bits can reside in generic address space.

The example below illustrates some of these concepts. The top diagram shows how the logic is partitioned into two chips: a chipset and an embedded controller.

- The chipset contains the interrupt logic, performs the power button (which is part of the fixed register space, and is not discussed here), the lid switch (used in portables to indicate when the clam shell lid is open or closed), and the RI# function (which can be used to wake a sleeping system).
- The embedded controller chip is used to perform the AC power detect and dock/undock event logic. Additionally, the embedded controller supports some system management functions using an OS-transparent interrupt in the embedded controller (represented by the EXTSMI# signal).

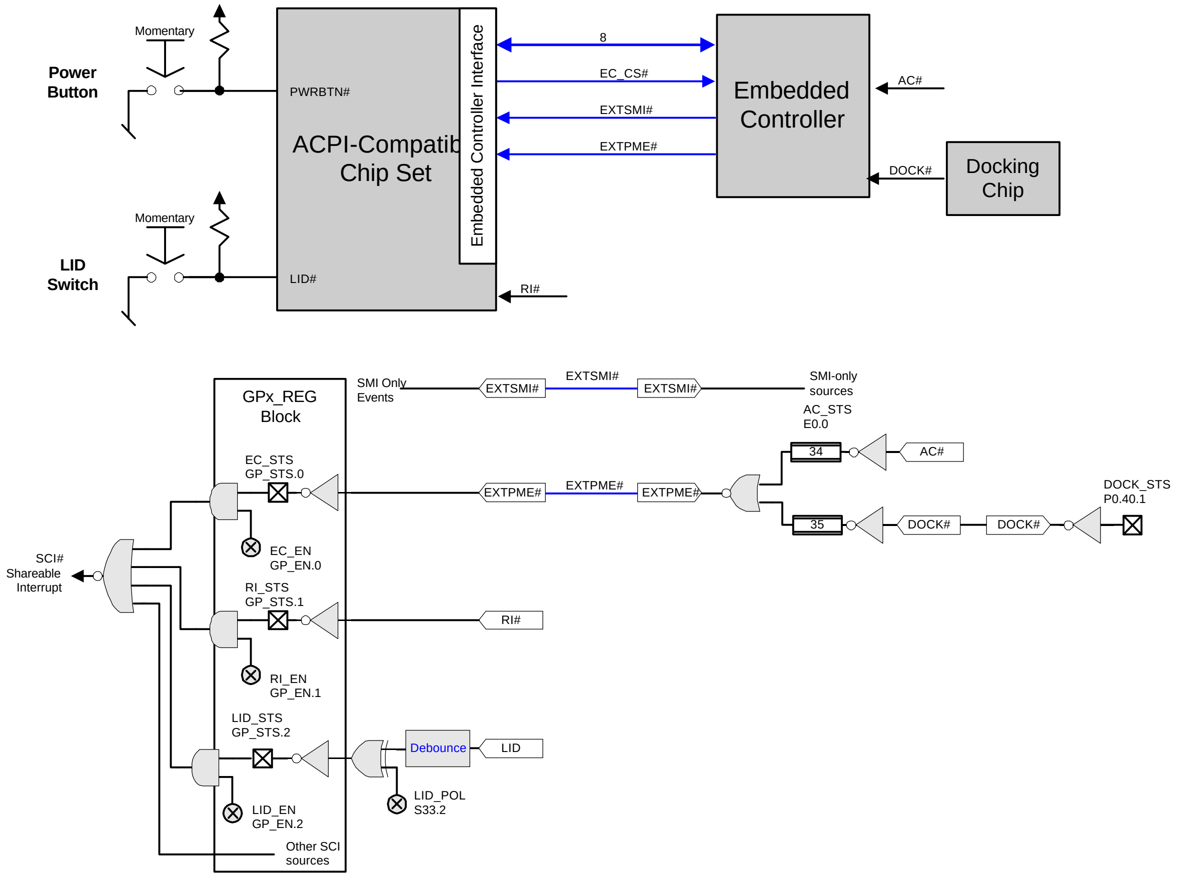

Figure 4-13 Example of General-Purpose vs. Generic Hardware Events

At the top level, the generic events in the GPE*x*_STS register are the:

- Embedded controller interrupt, which contains two query events: one for AC detection and one for docking (the docking query event has a child interrupt status bit in the docking chip).
- Ring indicate status (used for waking the system).
- Lid status.

The embedded controller event status bit (EC_STS) is used to indicate that one of two query events is active.

- A query event is generated when the AC# signal is asserted. The embedded controller returns a query value of 34 (any byte number can be used) upon a query command in response to this event; OSPM will then schedule for execution the control method associated with query value 34.
- Another query event is for the docking chip that generates a docking event. In this case, the embedded controller will return a query value of 35 upon a query command from system software responding to an SCI from the embedded controller. OSPM will then schedule the control method associated with the query value of 35 to be executed, which services the docking event.

For each of the status bits in the GPE*x*_STS register, there is a corresponding enable bit in the GPE*x*_EN register. Notice that the child status bits do not necessarily need enable bits (see the DOCK_STS bit).

The lid logic contains a control bit to determine if its status bit is set when the LID is open (LID_POL is and LID is) or closed (LID_POL is clear and LID is clear). This control bit resides in generic I/O space (in this case, bit 2 of system I/O space 33h) and would be manipulated with a control method associated with the lid object.

As with fixed hardware events, OSPM will clear the status bits in the GPE*x* register blocks. However, AML code clears all sibling status bits in the generic hardware.

Generic hardware features are controlled by OEM supplied control methods, encoded in AML. ACPI provides both an event and control model for development of these features. The ACPI specification also provides specific control methods for notifying OSPM of certain power management and Plug and Play events. Section 5, "ACPI Software Programming Model," provides information on the types of hardware functionality that support the different types of subsystems. The following is a list of features supported by ACPI. The list is not intended to be complete or comprehensive.

- Device insertion/ejection (for example, docking, device bay, A/C adapter)
- Batteries[4]
- Platform thermal subsystem
- Turning on/off power resources
- Mobile lid Interface
- Embedded controller
- System indicators
- OEM-specific wake events
- Plug and Play configuration

4.7.4.1 General-Purpose Event Register Blocks

ACPI supports up to two general-purpose register blocks as described in the FADT (see section 5, "ACPI Software Programming Model") and an arbitrary number of additional GPE blocks described as devices within the ACPI namespace. Each register block contains two registers: an enable and a status register. Each register block is 32-bit aligned. Each register in the block is accessed as a byte. It is up to the specific design to determine if these bits retain their context across sleeping or soft-off states. If they lose their context across a sleeping or soft-off state, then BIOS resets the respective enable bit prior to passing control to the OS upon waking.

4.7.4.1.1 General-Purpose Event 0 Register Block

This register block consists of two registers: The GPE0_STS and the GPE0_EN registers. Each register's length is defined to be half the length of the GPE0 register block, and is described in the ACPI FADT's GPE0_BLK and GPE0_BLK_LEN operators. OSPM owns the general-purpose event resources and these bits are only manipulated by OSPM; AML code cannot access the general-purpose event registers.

It is envisioned that chipsets will contain GPE event registers that provide GPE input pins for various events.

The platform designer would then wire the GPEs to the various value-added event hardware and the AML code would describe to OSPM how to utilize these events. As such, there will be the case where a platform has GPE events that are not wired to anything (they are present in the chip set), but are not utilized by the platform and have no associated AML code. In such, cases these event pins are to be tied inactive such that the corresponding SCI status bit in the GPE register is not set by a floating input pin.

[4] ACPI operating systems assume the use of the Smart Battery System Implementers Forum defined standard for batteries, called the "Smart Battery Specification" (SBS). ACPI provides a set of control methods for use by OEMs that use a proprietary "control method" battery interface.

4.7.4.1.1.1 General-Purpose Event 0 Status Register

```
Register Location: <GPE0_STS> System I/O or System Memory Space
Default Value:     00h
Attribute:         Read/Write
Size:              GPE0_BLK_LEN/2
```

The general-purpose event 0 status register contains the general-purpose event status bits in bank zero of the general-purpose registers. Each available status bit in this register corresponds to the bit with the same bit position in the GPE0_EN register. Each available status bit in this register is set when the event is active, and can only be cleared by software writing a “1” to its respective bit position. For the general-purpose event registers, unimplemented bits are ignored by OSPM.

Each status bit can optionally wake the system if asserted when the system is in a sleeping state with its respective enable bit set. OSPM accesses GPE registers through byte accesses (regardless of their length).

4.7.4.1.1.2 General-Purpose Event 0 Enable Register

```
Register Location: <GPE0_EN> System I/O or System Memory Space
Default Value:     00h
Attribute:         Read/Write
Size:              GPE0_BLK_LEN/2
```

The general-purpose event 0 enable register contains the general-purpose event enable bits. Each available enable bit in this register corresponds to the bit with the same bit position in the GPE0_STS register. The enable bits work similar to how the enable bits in the fixed-event registers are defined: When the enable bit is set, then a set status bit in the corresponding status bit will generate an SCI bit. OSPM accesses GPE registers through byte accesses (regardless of their length).

4.7.4.1.2 General-Purpose Event 1 Register Block

This register block consists of two registers: The GPE1_STS and the GPE1_EN registers. Each register’s length is defined to be half the length of the GPE1 register block, and is described in the ACPI FADT’s GPE1_BLK and GPE1_BLK_LEN operators.

4.7.4.1.2.1 General-Purpose Event 1 Status Register

```
Register Location: <GPE1_STS> System I/O or System Memory Space
Default Value:     00h
Attribute:         Read/Write
Size:              GPE1_BLK_LEN/2
```

The general -purpose event 1 status register contains the general-purpose event status bits. Each available status bit in this register corresponds to the bit with the same bit position in the GPE1_EN register. Each available status bit in this register is set when the event is active, and can only be cleared by software writing a “1” to its respective bit position. For the general-purpose event registers, unimplemented bits are ignored by the operating system.

Each status bit can optionally wake the system if asserted when the system is in a sleeping state with its respective enable bit set.

OSPM accesses GPE registers through byte accesses (regardless of their length).

4.7.4.1.2.2 General-Purpose Event 1 Enable Register

```
Register Location: <GPE1_EN> System I/O or System Memory Space
Default Value:     00h
Attribute:         Read/Write
Size:              GPE1_BLK_LEN/2
```

The general-purpose event 1 enable register contains the general-purpose event enable. Each available enable bit in this register corresponds to the bit with the same bit position in the GPE1_STS register. The enable bits work similar to how the enable bits in the fixed-event registers are defined: When the enable bit is set, a set status bit in the corresponding status bit will generate an SCI bit.

OSPM accesses GPE registers through byte accesses (regardless of their length).

4.7.4.2 Example Generic Devices

This section points out generic devices with specific ACPI driver support.

4.7.4.2.1 Lid Switch

The Lid switch is an optional feature present in most "clam shell" style mobile computers. It can be used by the OS as policy input for sleeping the system, or for waking the system from a sleeping state. If used, then the OEM needs to define the lid switch as a device with an _HID object value of "_PNP0C0D", which identifies this device as the lid switch to OSPM. The Lid device needs to contain a control method that returns its status. The Lid event handler AML code reconfigures the lid hardware (if it needs to) to generate an event in the other direction, clear the status, and then notify OSPM of the event.

Example hardware and ASL code is shown below for such a design.

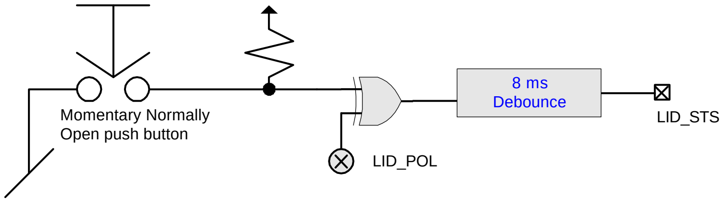

Figure 4-14 Example Generic Address Space Lid Switch Logic

This logic will set the Lid status bit when the button is pressed or released (depending on the LID_POL bit).

The ASL code defines the following:

- An operational region where the lid polarity resides in address space
  System address space in registers 0x201.
- A field operator to allow AML code to access this bit:
  Polarity control bit (LID_POL) is called LPOL and is accessed at 0x201.0.
- A device named _SB.LID with the following:
  A Plug and Play identifier "PNP0C0D" that associates OSPM with this object.
  Defines an object that specifies a change in the lid's status bit can wake the system from the S4 sleep state and from all higher sleep states (S1, S2, or S3).

- The lid switch event handler that does the following:
  - Defines the lid’s status bit (LID_STS) as a child of the general-purpose event 0 register bit 1.
  - Defines the event handler for the lid (only event handler on this status bit) that does the following:
    - Flips the polarity of the LPOL bit (to cause the event to be generated on the opposite condition).
    - Generates a notify to the OS that does the following:
      - Passes the _SB.LID object.
      - Indicates a device specific event (notify value 0x80).

```
// Define a Lid switch
OperationRegion(\Pho, SystemIO, 0x201, 0x1)
Field(\Pho, ByteAcc, NoLock, Preserve) {
    LPOL, 1        //  Lid polarity control bit
    }

Device(\_SB.LID){
    Name(_HID, EISAID(“PNP0C0D”))
    Method(_LID){Return(LPOL)}
    Name(_PRW, Package(2){
        1,         // bit 1 of GPE to enable Lid wakeup
        0x04}      // can wakeup from S4 state
        )
    }
Scope(\_GPE){              // Root level event handlers
    Method(_L01){          // uses bit 1 of GP0_STS register
        Not(LPOL, LPOL)    // Flip the lid polarity bit
        Notify(LID, 0x80)  // Notify OS of event
        }
    }
```

At the top level, the generic events in the GPE*x*_STS register are:

Embedded controller interrupt, which contains two query events: one for AC detection and one for docking (the docking query event has a child interrupt status bit in the docking chip)

Ring indicate status (used for waking the system)

Lid status

The embedded controller event status bit (EC_STS) is used to indicate that one of two query events is active.

A query event is generated when the AC# signal is asserted. The embedded controller returns a query value of 34 (any byte number can be used) upon a query command in response to this event; OSPM will then schedule for execution the control method associated with query value 34.

Another query event is for the docking chip that generates a docking event. In this case, the embedded controller will return a query value of 35 upon a query command from system software responding to an SCI from the embedded controller. OSPM will then schedule the control method associated with the query value of 35 to be executed, which services the docking event.

For each of the status bits in the GPE*x*_STS register, there is a corresponding enable bit in the GPE*x*_EN register. Notice that the child status bits do not necessarily need enable bits (see the DOCK_STS bit).

The lid logic contains a control bit to determine if its status bit is set when the LID is open (LID_POL is set and LID is set) or closed (LID_POL is clear and LID is clear). This control bit resides in generic I/O space (in this case, bit 2 of system I/O space 33h) and would be manipulated with a control method associated with the lid object.

As with fixed hardware events, OSPM will clear the status bits in the GPE*x* register blocks. However, AML code is required to clear all sibling status bits in generic space.

Generic hardware features are controlled by OEM supplied AML code. ACPI provides both an event and control model for development of these features. The ACPI specification also provides specific control methods for notifying OSPM of certain power management and Plug and Play events. Section 5, “ACPI Software Programming Model,” provides information on the types of hardware hooks required to support the different types of subsystems. The following is a list of features supported by ACPI, however the list is not intended to be complete or comprehensive

Device insertion/ejection (for example, docking, device bay, A/C adapter)

Batteries[5]

Platform thermal subsystem

Turning on/off power resources

Mobile lid interface

Embedded controller

System indicators

OEM-specific wake events

Plug and Play configuration

4.7.4.2.2 Embedded Controller

ACPI provides a standard interface that enables AML code to define and access generic logic in “embedded controller space.” This supports current computer models where much of the value added hardware is contained within the embedded controller while allowing the AML code to access this hardware in an abstracted fashion.

The embedded controller is defined as a device and must contain a set number of control methods:

- _HID with a value of PNP0C09 to associate this device with the ACPI’s embedded controller’s driver.
- _CRS to return the resources being consumed by the embedded controller.
- _GPE that returns the general-purpose event bit that this embedded controller is wired to.

Additionally the embedded controller can support up to 255 generic events per embedded controller, referred to as query events. These query event handles are defined within the embedded controller’s device as control methods. An example of defining an embedded controller device is shown below:

[5] ACPI OS’s assume the use of the Duracell/Intel defined standard for batteries, called the “Smart Battery Specification” (SBS). ACPI provides a set of control methods for use by OEMs that use a proprietary “control method” battery interface.

```
Device(EC0) {
// PnP ID
Name(_HID, EISAID(“PNP0C09”))
// Returns the “Current Resources” of EC
Name(_CRS,
ResourceTemplate(){
           IO(Decode16, 0x62, 0x62, 0, 1)
           IO(Decode16, 0x66, 0x66, 0, 1)
})
// Define that the EC SCI is bit 0 of the GP_STS register
Name(_GPE, 0)      // embedded controller is wired to bit 0 of GPE

OperationRegion(\EC0, EmbeddedControl, 0, 0xFF)
Field(EC0, ByteAcc, Lock, Preserve) {
// Field definitions
       }
Method(Q00){..}
Method(QFF){..}
}
```

For more information on the embedded controller, see section 13, “ACPI Embedded Controller Interface Specification.”

4.7.4.2.3 Fan

ACPI has a device driver to control fans (active cooling devices) in platforms. A fan is defined as a device with the Plug and Play ID of “PNP0C0B.” It should then contain a list power resources used to control the fan.

For more information, see section 10, “ACPI-Specific Device Objects.”

5 ACPI Software Programming Model

ACPI defines a hardware register interface that an ACPI-compatible OS uses to control core power management features of a machine, as described in section 4, “ACPI Hardware Specification.” ACPI also provides an abstract interface for controlling the power management and configuration of an ACPI system. Finally, ACPI defines an interface between an ACPI-compatible OS and the system BIOS.

To give hardware vendors flexibility in choosing their implementation, ACPI uses tables to describe system information, features, and methods for controlling those features. These tables list devices on the system board or devices that cannot be detected or power managed using some other hardware standard, plus their capabilities as described in section 3, “Overview.” They also list system capabilities such as the sleeping power states supported, a description of the power planes and clock sources available in the system, batteries, system indicator lights, and so on. This enables OSPM to control system devices without needing to know how the system controls are implemented.

Topics covered in this section are:

- The ACPI system description table architecture is defined, and the role of OEM-provided definition blocks in that architecture is discussed.
- The concept of the ACPI Namespace is discussed.

5.1 Overview of the System Description Table Architecture

The Root System Description Pointer (RSDP) structure is located in the system’s memory address space and is setup by the BIOS. This structure contains the address of the Root System Description Table (RSDT), which references other description tables that provide data to OSPM, supplying it with knowledge of the base system’s implementation and configuration (see Figure 5-1).

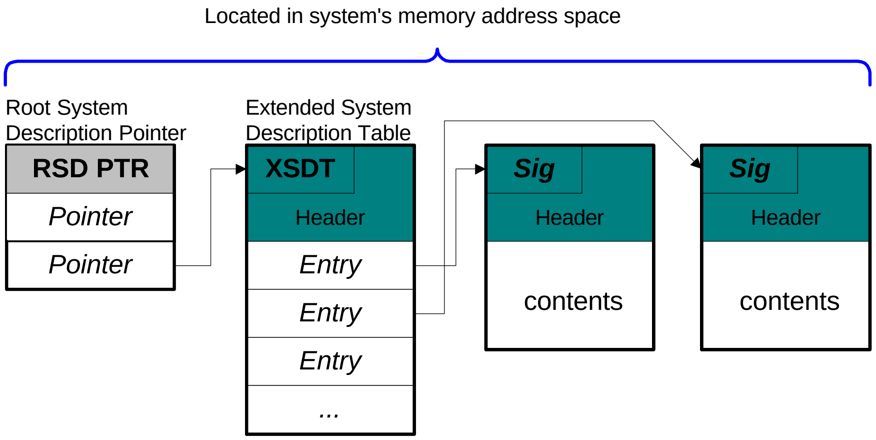
Located in system's memory address space
Root System Description Pointer
RSD PTR
Pointer
Pointer
Extended System Description Table
XSDT
Header
Entry
Entry
Entry
...
Sig
Header
contents
Sig
Header
contents

Figure 5-1 Root System Description Pointer and Table

All system description tables start with identical headers. The primary purpose of the system description tables is to define for OSPM various industry-standard implementation details. Such definitions enable various portions of these implementations to be flexible in hardware requirements and design, yet still provide OSPM with the knowledge it needs to control hardware directly.

The Root System Description Table (RSDT) points to other tables in memory. Always the first table, it points to the Fixed ACPI Description table (FADT). The data within this table includes various fixed-length entries that describe the fixed ACPI features of the hardware. The FADT table always refers to the Differentiated System Description Table (DSDT), which contains information and descriptions for various system features. The relationship between these tables is shown in Figure 5-2.

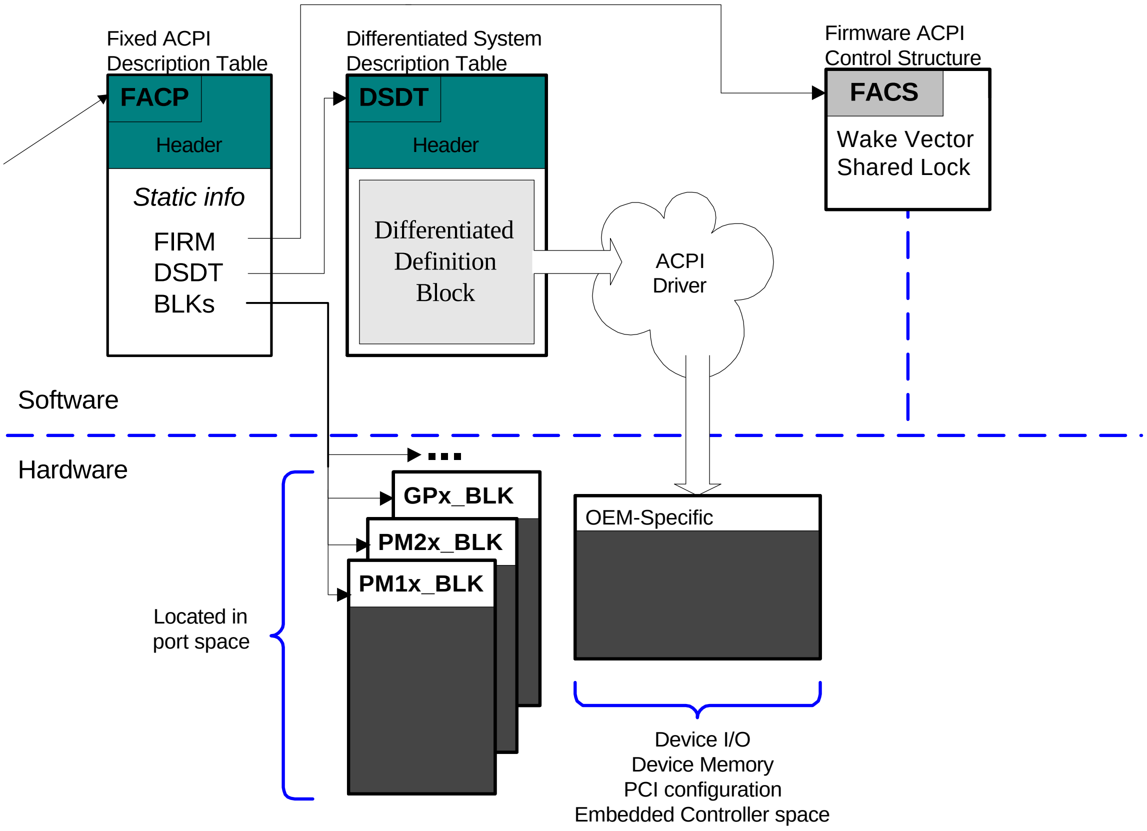

Figure 5-2 Description Table Structures

- OSPM finds the RSDP structure as described in section 5.2.4.1 (“Finding the RSDP on IA-PC Systems”) or section 5.2.4.2 (“Finding the RSDP on EFI Enabled Systems”).

When OSPM locates the structure, it looks at the physical address for the Root System Description Table. The Root System Description Table starts with the signature “RSDT” and contains one or more physical pointers to other system description tables that provide various information about the system. As shown in Figure 5-1, there is always a physical address in the Root System Description Table for the Fixed ACPI Description table (FADT).

When OSPM follows a physical pointer to another table, it examines each table for a known signature. Based on the signature, OSPM can then interpret the implementation-specific data within the description table.

The purpose of the FADT is to define various static system information related to configuration and power management. The Fixed ACPI Description Table starts with the “FACP” signature. The FADT describes the implementation and configuration details of the ACPI hardware registers on the platform.

For a specification of the ACPI Hardware Register Blocks (PM1a_EVT_BLK, PM1b_EVT_BLK, PM1a_CNT_BLK, PM1b_CNT_BLK, PM2_CNT_BLK, PM_TMR_BLK, GP0_BLK, GP1_BLK, and one or more P_BLKs), see section 4.7, “ACPI Register Model.” The PM1a_EVT_BLK, PM1b_EVT_BLK, PM1a_CNT_BLK, PM1b_CNT_BLK, PM2_CNT_BLK, and PM_TMR_BLK blocks are for controlling low-level ACPI system functions.

The GPE0_BLK and GPE1_BLK blocks provide the foundation for an interrupt-processing model for Control Methods. The P_BLK blocks are for controlling processor features.

Besides ACPI Hardware Register implementation information, the FADT also contains a physical pointer to the Differentiated System Description Table (DSDT). The DSDT contains a Definition Block named the Differentiated Definition Block for the DSDT that contains implementation and configuration information OSPM can use to perform power management, thermal management, or Plug and Play functionality that goes beyond the information described by the ACPI hardware registers.

A Definition Block contains information about hardware implementation details in the form of a hierarchical namespace, data, and control methods encoded in AML. OSPM “loads” or “unloads” an entire definition block as a logical unit. The Differentiated Definition Block is always loaded by OSPM at boot time and cannot be unloaded.

Definition Blocks can either define new system attributes or, in some cases, build on prior definitions. A Definition Block can be loaded from system memory address space. One use of a Definition Block is to describe and distribute platform version changes.

Definition blocks enable wide variations of hardware platform implementations to be described to the ACPI-compatible OS while confining the variations to reasonable boundaries. Definition blocks enable simple platform implementations to be expressed by using a few well-defined object names. In theory, it might be possible to define a PCI configuration space-like access method within a Definition Block, by building it from I/O space, but that is not the goal of the Definition Block specification. Such a space is usually defined as a “built in” operator.

Some operators perform simple functions and others encompass complex functions. The power of the Definition Block comes from its ability to allow these operations to be glued together in numerous ways, to provide functionality to OSPM. The operators present are intended to allow many useful hardware designs to be ACPI-expressed, not to allow all hardware designs to be expressed.

5.1.1 Address Space Translation

Some platforms may contain bridges that perform translations as I/O and/or Memory cycles pass through the bridges. This translation can take the form of the addition or subtraction of an offset. Or it can take the form of a conversion from I/O cycles into Memory cycles and back again. When translation takes place, the addresses placed on the processor bus by the processor during a read or write cycle are not the same addresses that are placed on the I/O bus by the I/O bus bridge. The address the processor places on the processor bus will be known here as the processor-relative address. And the address that the bridge places on the I/O bus will be known as the bus-relative address. Unless otherwise noted, all addresses used within this section are processor-relative addresses.

For example, consider a platform with two root PCI buses. The platform designer has several choices. One solution would be to split the 16-bit I/O space into two parts, assigning one part to the first root PCI bus and one part to the second root PCI bus. Another solution would be to make both root PCI buses decode the entire 16-bit I/O space, mapping the second root PCI bus’s I/O space into memory space. In this second scenario, when the processor needs to read from an I/O register of a device underneath the second root PCI bus, it would need to perform a memory read within the range that the root PCI bus bridge is using to map the I/O space.

Note: Industry standard PCs do not provide address space translations because of historical compatibility issues.

5.2 ACPI System Description Tables

This section specifies the structure of the system description tables:

- Root System Description Pointer (RSDP)
- System Description Table Header
- Root System Description Table (RSDT)
- Fixed ACPI Description Table (FADT)
- Firmware ACPI Control Structure (FACS)
- Differentiated System Description Table (DSDT)
- Secondary System Description Table (SSDT)
- Multiple APIC Description Table (MADT)
- Smart Battery Table (SBST)
- Extended System Description Table (XSDT)
- Embedded Controller Boot Resources Table (ECDT)

All numeric values from the above tables, blocks, and structures are always encoded in little endian format. Signature values are stored as fixed-length strings.

5.2.1 Reserved Bits and Fields

For future expansion, all data items marked as *reserved* in this specification have strict meanings. This section lists software requirements for *reserved* fields. Notice that the list contains terms such as ACPI tables and AML code defined later in this section of the specification.

5.2.1.1 Reserved Bits and Software Components

- OEM implementations of software and AML code return the bit value of 0 for all reserved bits in ACPI tables or in other software values, such as resource descriptors.
- For all reserved bits in ACPI tables and registers, OSPM implementations must:
    - Ignore all reserved bits that are read.
    - Preserve reserved bit values of read/write data items (for example, OSPM writes back reserved bit values it reads).
    - Write zeros to reserved bits in write-only data items.
    -

5.2.1.2 Reserved Values and Software Components

- OEM implementations of software and AML code return only defined values and do not return reserved values.
- OSPM implementations write only defined values and do not write reserved values.

5.2.1.3 Reserved Hardware Bits and Software Components

- Software ignores all reserved bits read from hardware enable or status registers.
- Software writes zero to all reserved bits in hardware enable registers.
- Software ignores all reserved bits read from hardware control and status registers.
- Software preserves the value of all reserved bits in hardware control registers by writing back read values.

5.2.1.4 Ignored Hardware Bits and Software Components

- Software handles ignored bits in ACPI hardware registers the same way it handles reserved bits in these same types of registers.

5.2.2 Compatibility

All versions of the ACPI tables must maintain backward compatibility. To accomplish this, modifications of the tables consist of redefinition of previously reserved fields and values plus appending data to the 1.0 tables. Modifications of the ACPI tables require that the version numbers of the modified tables be incremented. The length field in the tables includes all additions and the checksum is maintained for the entire length of the table.

5.2.3 Address Format

Addresses used in the ACPI 1.0 system description tables were expressed as either system memory or I/O space. This was targeted at the IA-32 environment. Newer architectures require addressing mechanisms beyond that defined in ACPI 1.0. To support these architectures ACPI must support 64-bit addressing and it must allow the placement of control registers in address spaces other than System I/O.

5.2.3.1 Generic Address Structure

In order to expand ACPI addressing capabilities, a Generic Address Structure (GAS) is defined that enables access to registers in ACPI-defined address spaces. This 12-byte structure, described below (Table 5-1), is used to express register addresses within the new tables defined by ACPI 2.0.

Table 5-1 Generic Address Structure (GAS)

Field	Byte Length	Byte Offset	Description
Address_Space_ID	1	0	The address space where the data structure or register exists. Defined values are: 0–System Memory 1–System I/O 2–PCI Configuration Space 3–Embedded Controller 4–SMBus 0x7F–Functional Fixed Hardware
Register_Bit_Width	1	1	The size in bits of the given register. When addressing a data structure, this field must be zero.
Register_Bit_Offset	1	2	The bit offset of the given register at the given address. When addressing a data structure, this field must be zero.
Reserved	1	3	Must be 0.
Address	8	4	The 64-bit address of the data structure or register in the given address space (relative to the processor). (See below for specific formats.)

Table 5-2 Address Space Format

<table>
<tr><th>Address Space</th><th colspan="2">Format</th></tr>
<tr><td>0–System Memory</td><td colspan="2">The 64-bit physical memory address (relative to the processor) of the register. 32-bit platforms must have the high DWORD set to 0.</td></tr>
<tr><td>1–System I/O</td><td colspan="2">The 64-bit I/O address (relative to the processor) of the register. 32-bit platforms must have the high DWORD set to 0.</td></tr>
<tr><td rowspan="7">2–PCI Configuration Space</td><td colspan="2">PCI Configuration space addresses must be confined to devices on PCI bus 0 segment 0. The format of addresses are defined as follows:</td></tr>
<tr><th>WORD Location</th><th>Description</th></tr>
<tr><td>Highest WORD</td><td>Reserved (must be 0)</td></tr>
<tr><td>…</td><td>PCI Device number on bus 0</td></tr>
<tr><td>…</td><td>PCI Function number</td></tr>
<tr><td>Lowest WORD</td><td>Offset in the configuration space header</td></tr>
<tr><td colspan="2">For example: Offset 23h of Function 2 on device 7 on bus 0 segment 0 would be represented as: 0x0000000700020023.</td></tr>
<tr><td>0x7F–Functional Fixed Hardware</td><td colspan="2">Use of GAS fields other than Address_Space_ID is specified by the CPU manufacturer. The use of functional fixed hardware carries with it a reliance on OS specific software that must be considered. OEMs should consult OS vendors to ensure that specific functional fixed hardware interfaces are supported by specific operating systems.</td></tr>
</table>

5.2.4 Root System Description Pointer (RSDP)

During OS initialization, OSPM must obtain the Root System Description Pointer (RSDP) structure from the platform. When OSPM locates the Root System Description Pointer (RSDP) structure, it then locates the Root System Description Table (RSDT) or the Extended Root System Description Table (XSDT) using the physical system address supplied in the RSDP.

5.2.4.1 Finding the RSDP on IA-PC Systems

OSPM finds the Root System Description Pointer (RSDP) structure by searching physical memory ranges on 16-byte boundaries for a valid Root System Description Pointer structure signature and checksum match as follows:

- The first 1 KB of the Extended BIOS Data Area (EBDA). For EISA or MCA systems, the EBDA can be found in the two-byte location 40:0Eh on the BIOS data area.
- The BIOS read-only memory space between 0E0000h and 0FFFFFh.

5.2.4.2 Finding the RSDP on EFI Enabled Systems

In Extensible Firmware Interface (EFI) enabled systems (for example, Itanium$^{\text{TM}}$-based platforms) a pointer to the RSDP structure exists within the EFI System Table. The OS loader's EFI image is provided a pointer to the EFI System Table at invocation. The OS loader must retrieve the pointer to the RSDP structure from the EFI System table and convey the pointer to OSPM, using an OS dependent data structure, as part of the hand off of control from the OS loader to the OS.

The OS loader locates the pointer to the RSDP structure by examining the EFI configuration table within the EFI system table. EFI configuration table entries consist of Globally Unique Identifier (GUID)/table pointer pairs. The EFI 1.0 specification defines a GUID for ACPI. An EFI configuration table entry that matches this GUID points to an ACPI 1.0-compatible RSDP structure (ACPI 1.0 GUID).

The EFI GUID for the ACPI 2.0 RSDP structure pointer is: 8868E871-E4F1-11d3-BC22-0080C73C8881.

The OS loader for an ACPI 2.0-compatible OS will search for an RSDP structure pointer using the ACPI 2.0 GUID first and if it finds one, will use the corresponding RSDP structure pointer. If the GUID is not found then the OS loader will search for the RSDP structure pointer using the ACPI 1.0 GUID.

The OS loader must retrieve the pointer to the RSDP structure from the EFI System Table **before** assuming platform control via the EFI ExitBootServices interface. See the EFI specification for more information.

5.2.4.3 RSDP Structure

The revision number contained within the structure indicates the size of the table structure.

Table 5-3 Root System Description Pointer Structure

Field	Byte Length	Byte Offset	Description
Signature	8	0	“RSD PTR ” (Notice that this signature must contain a trailing blank character.)
Checksum	1	8	This is the checksum of the fields defined in the ACPI 1.0 specification. This includes only the first 20 bytes of this table, bytes 0 to 19, including the checksum field. These bytes must sum to zero.
OEMID	6	9	An OEM-supplied string that identifies the OEM.
Revision	1	15	The revision of this structure. Larger revision numbers are backward compatible to lower revision numbers. The ACPI version 1.0 revision number of this table is zero. The ACPI 2.0 value for this field is 2.
RsdtAddress	4	16	32 bit physical address of the RSDT.
Length	4	20	The length of the table, in bytes, including the header, starting from offset 0. This field is used to record the size of the entire table.
XsdtAddress	8	24	64 bit physical address of the XSDT.
Extended Checksum	1	32	This is a checksum of the entire table, including both checksum fields.
Reserved	3	33	Reserved field

5.2.5 System Description Table Header

All system description tables begin with the structure shown in Table 5-4. The Signature field determines the content of the system description table. System description table signatures defined by this specification are listed in Table 5-5.

Table 5-4 DESCRIPTION_HEADER Fields

Field	Byte Length	Byte Offset	Description
Signature	4	0	The ASCII string representation of the table identifier. Notice that if OSPM finds a signature in a table that is not listed in Table 5-5, OSPM ignores the entire table (it is not loaded into ACPI namespace); OSPM ignores the table even though the values in the Length and Checksum fields are correct.
Length	4	4	The length of the table, in bytes, including the header, starting from offset 0. This field is used to record the size of the entire table.
Revision	1	8	The revision of the structure corresponding to the signature field for this table. Larger revision numbers are backward compatible to lower revision numbers with the same signature.
Checksum	1	9	The entire table, including the checksum field, must add to zero to be considered valid.
OEMID	6	10	An OEM-supplied string that identifies the OEM.
OEM Table ID	8	16	An OEM-supplied string that the OEM uses to identify the particular data table. This field is particularly useful when defining a definition block to distinguish definition block functions. The OEM assigns each dissimilar table a new OEM Table ID.
OEM Revision	4	24	An OEM-supplied revision number. Larger numbers are assumed to be newer revisions.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.

For OEMs, good design practices will ensure consistency when assigning OEMID and OEM Table ID fields in any table. The intent of these fields is to allow for a binary control system that support services can use. Because many support functions can be automated, it is useful when a tool can programmatically determine which table release is a compatible and more recent revision of a prior table on the same OEMID and OEM Table ID.

Tables 5-5 and 5-5a contain the system description table signatures defined by this specification. These system description tables may be defined by ACPI (Table 5-5) or reserved by ACPI and declared by other industry specifications (Table 5-5a). This allows OS and platform specific tables to be defined and pointed to by the RSDT/XSDT as needed. For tables defined by other industry specifications, the ACPI specification acts as gatekeeper to avoid collisions in table signatures. To help avoid signature collisions, table signatures will be reserved by the ACPI promoters and posted independently of this specification in ACPI errata and clarification documents on the ACPI Web site. Requests to reserve a 4-byte alphanumeric table signature should be sent to the email address **info@acpi.info** and should include the purpose of the table and reference url to a document that describes the table format.

Table 5-5 DESCRIPTION_HEADER Signatures for tables defined by ACPI

Signature	Description	Reference
“APIC”	Multiple APIC Description Table	Section 5.2.10.4, “Multiple APIC Description Table”
“DSDT”	Differentiated System Description Table	Section 5.2.10.1, “Differentiated System Description Table”
“ECDT”	Embedded Controller Boot Resources Table	Section 5.2.13, “Embedded Controller Boot Resources Table”
”FACP”	Fixed ACPI Description Table (FADT)	Section 5.2.8, “Fixed ACPI Description Table”
“FACS”	Firmware ACPI Control Structure	Section 5.2.9, “Firmware ACPI Control Structure”
“OEMx”	OEM Specific Information Tables	OEM Specific tables. All table signatures starting with “OEM” are reserved for OEM use.
“PSDT”	Persistent System Description Table	Section 5.2.10.3, “Persistent System Description Table”
“RSDT”	Root System Description Table	Section 5.2.6, “Root System Description Table”
“SBST”	Smart Battery Specification Table	Section 5.2 12, “Smart Battery Table”
“SSDT”	Secondary System Description Table	Section 5.2.10.2, “Secondary System Description Table”
“XSDT”	Extended System Description Table	Section 5.2.7, “Extended System Description Table”

Table 5-5a DESCRIPTION_HEADER Signatures for tables reserved by ACPI

Signature	Description	Reference
“BOOT”	Simple Boot Flag Table	Microsoft Simple Boot Flag Specification http://www.microsoft.com/HWDEV/desinit/simp_bios.htm
“CPEP”	Corrected Platform Error Polling Table	Corrected Platform Error Polling Table Specification http://h21007.www2.hp.com/dspp/files/unprotected/devresource/Docs/TechPapers/IA64/cpep.pdf
“DBGP”	Debug Port Table	Microsoft Debug Port Specification http://www.microsoft.com/HWDEV/PLATFORM/pcdesign/LR/debugspec.asp
“ETDT”	Event Timer Description Table	IA-PC Multimedia Timers Specification. This signature has been superceded by “HPET” and is now obsolete.
“HPET”	IA-PC High Precision Event Timer Table	IA-PC High Precision Event Timer Specification. http://developer.intel.com/ial/home/sp/pcmmspec.htm
“SLIT”	System Locality Information Table	http://h21007.www2.hp.com/dspp/files/unprotected/devresource/Docs/TechPapers/IA64/slit.pdf
“SPCR”	Serial Port Console Redirection Table	Microsoft Serial Port Console Redirection Table http://www.microsoft.com/HWDEV/PLATFORM/server/headless/SPCR.asp
“SRAT”	Static Resource Affinity Table	Interim processor-memory proximity table http://www.microsoft.com/HWDEV/design/SRAT.htm
“SPMI”	Server Platform Management Interface Table	http://h21007.www2.hp.com/dspp/files/unprotected/devresource/Docs/TechPapers/IA64/hpspmi.pdf
“TCPA”	Trusted Computing Platform Alliance Capabilities Table	http://www.trustedpc.org TCPA PC Specific Implementation Specification

5.2.6 Root System Description Table (RSDT)

OSPM locates that Root System Description Table by following the pointer in the RSDP structure. The RSDT, shown in Table 5-6, starts with the signature ‘RSDT’ followed by an array of physical pointers to other system description tables that provide various information on other standards defined on the current system. OSPM examines each table for a known signature. Based on the signature, OSPM can then interpret the implementation-specific data within the table.

Systems provide the RSDT to enable compatibility with ACPI 1.0 operating systems. The XSDT, described in the next section, supersedes RSDT functionality for ACPI 2.0.

Table 5-6 Root System Description Table Fields (RSDT)

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	'RSDT.' Signature for the Root System Description Table.
Length	4	4	Length, in bytes, of the entire RSDT. The length implies the number of Entry fields (*n*) at the end of the table.
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the RSDT, the table ID is the manufacture model ID. This field must match the OEM Table ID in the FADT.
OEM Revision	4	24	OEM revision of RSDT table for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.
Entry	4*n*	36	An array of 32-bit physical addresses that point to other DESCRIPTION_HEADERs. OSPM assumes at least the DESCRIPTION_HEADER is addressable, and then can further address the table based upon its Length field.

5.2.7 Extended System Description Table (XSDT)

The XSDT provides identical functionality to the RSDT but accommodates physical addresses of DESCRIPTION HEADERs that are larger than 32-bits. Notice that both the XSDT and the RSDT can be pointed to by the RSDP structure. An ACPI 2.0-compatible OS must use the XSDT if present.

Table 5-7 Extended System Description Table Fields (XSDT)

Field	**Byte Length**	**Byte Offset**	**Description**
Header			
Signature	4	0	‘XSDT’. Signature for the Extended System Description Table.
Length	4	4	Length, in bytes, of the entire table. The length implies the number of Entry fields (*n*) at the end of the table.
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the RSDTle, the table ID is the manufacture model ID. This field must match the OEM Table ID in the FADT.
OEM Revision	4	24	OEM revision of RSDT table for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.
Entry	8**n*	36	An array of 64-bit physical addresses that point to other DESCRIPTION_HEADERs. OSPM assumes at least the DESCRIPTION_HEADER is addressable, and then can further address the table based upon its Length field.

5.2.8 Fixed ACPI Description Table (FADT)

The Fixed ACPI Description Table (FADT) defines various fixed hardware ACPI information vital to an ACPI-compatible OS, such as the base address for the following hardware registers blocks: PM1a_EVT_BLK, PM1b_EVT_BLK, PM1a_CNT_BLK, PM1b_CNT_BLK, PM2_CNT_BLK, PM_TMR_BLK, GPE0_BLK, and GPE1_BLK.

The FADT also has a pointer to the DSDT that contains the Differentiated Definition Block, which in turn provides variable information to an ACPI-compatible OS concerning the base system design.

All fields in the FADT that provide hardware addresses provide processor-relative physical addresses.

Table 5-8 Fixed ACPI Description Table (FADT) Format

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	‘FACP’. Signature for the Fixed ACPI Description Table.
Length	4	4	Length, in bytes, of the entire FADT.
Revision	1	8	3
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the FADT, the table ID is the manufacture model ID. This field must match the OEM Table ID in the RSDT.
OEM Revision	4	24	OEM revision of FADT for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.
FIRMWARE_CTRL	4	36	Physical memory address (0-4 GB) of the FACS, where OSPM and Firmware exchange control information. See section 5.2.6, “Root System Description Table,” for a description of the FACS.
DSDT	4	40	Physical memory address (0-4 GB) of the DSDT.
Reserved	1	44	ACPI 1.0 defined this offset as a field named INT_MODEL, which has been eliminated in ACPI 2.0.as operating systems to date have had no use for this field. New systems should set this field to zero but field values of one are also allowed to maintain compatibility with ACPI 1.0.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
Preferred_PM_Profile	1	45	This field is set by the OEM to convey the preferred power management profile to OSPM. OSPM can use this field to set default power management policy parameters during OS installation. Field Values: 0–Unspecified 1–Desktop 2–Mobile 3–Workstation 4–Enterprise Server 5–SOHO Server 6–Appliance PC >6–Reserved
SCI_INT	2	46	System vector the SCI interrupt is wired to in 8259 mode. On systems that do not contain the 8259, this field contains the Global System interrupt number of the SCI interrupt. OSPM is required to treat the ACPI SCI interrupt as a sharable, level, active low interrupt.
SMI_CMD	4	48	System port address of the SMI Command Port. During ACPI OS initialization, OSPM can determine that the ACPI hardware registers are owned by SMI (by way of the SCI_EN bit), in which case the ACPI OS issues the ACPI_ENABLE command to the SMI_CMD port. The SCI_EN bit effectively tracks the ownership of the ACPI hardware registers. OSPM issues commands to the SMI_CMD port synchronously from the boot processor. This field is reserved and must be zero on system that does not support System Management mode.
ACPI_ENABLE	1	52	The value to write to SMI_CMD to disable SMI ownership of the ACPI hardware registers. The last action SMI does to relinquish ownership is to set the SCI_EN bit. During the OS initialization process, OSPM will synchronously wait for the transfer of SMI ownership to complete, so the ACPI system releases SMI ownership as quickly as possible. This field is reserved and must be zero on systems that do not support Legacy Mode.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
ACPI_DISABLE	1	53	The value to write to SMI_CMD to re-enable SMI ownership of the ACPI hardware registers. This can only be done when ownership was originally acquired from SMI by OSPM using ACPI_ENABLE. An OS can hand ownership back to SMI by relinquishing use to the ACPI hardware registers, masking off all SCI interrupts, clearing the SCI_EN bit and then writing ACPI_DISABLE to the SMI_CMD port from the boot processor. This field is reserved and must be zero on systems that do not support Legacy Mode.
S4BIOS_REQ	1	54	The value to write to SMI_CMD to enter the S4BIOS state. The S4BIOS state provides an alternate way to enter the S4 state where the firmware saves and restores the memory context. A value of zero in S4BIOS_F indicates S4BIOS_REQ is not supported. (See Table 5-12.)
PSTATE_CNT	1	55	If non-zero, this field contains the value OSPM writes to the SMI_CMD register to assume processor performance state control responsibility.
PM1a_EVT_BLK	4	56	System port address of the PM1a Event Register Block. See section 4.7.3.1, “PM1 Event Grouping,” for a hardware description layout of this register block. This is a required field. This field is superseded in ACPI 2.0 by the X_PM1a_EVT_BLK field.
PM1b_EVT_BLK	4	60	System port address of the PM1b Event Register Block. See section 4.7.3.1, “PM1 Event Grouping,” for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_PM1b_EVT_BLK field.
PM1a_CNT_BLK	4	64	System port address of the PM1a Control Register Block. See section 4.7.3.2, “PM1 Control Grouping,” for a hardware description layout of this register block. This is a required field. This field is superseded in ACPI 2.0 by the X_PM1a_CNT_BLK field.
PM1b_CNT_BLK	4	68	System port address of the PM1b Control Register Block. See section 4.7.3.2, “PM1 Control Grouping,” for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_PM1b_CNT_BLK field.
PM2_CNT_BLK	4	72	System port address of the PM2 Control Register Block. See section 4.7.3.4, “PM2 Control (PM2_CNT),” for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_PM2_CNT_BLK field.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
PM_TMR_BLK	4	76	System port address of the Power Management Timer Control Register Block. See section 4.7.3.3, “Power Management Timer (PM_TMR),” for a hardware description layout of this register block. This is a required field. This field is superseded in ACPI 2.0 by the X_PM_TMR_BLK field.
GPE0_BLK	4	80	System port address of General-Purpose Event 0 Register Block. See section 5.2.8, “Fixed ACPI Description Table,” for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_GPE0_BLK field.
GPE1_BLK	4	84	System port address of General-Purpose Event 1 Register Block. See section 5.2.8, “Fixed ACPI Description Table,” for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero. This field is superseded in ACPI 2.0 by the X_GPE1_BLK field.
PM1_EVT_LEN	1	88	Number of bytes decoded by PM1a_EVT_BLK and, if supported, PM1b_ EVT_BLK. This value is ≥ 4.
PM1_CNT_LEN	1	89	Number of bytes decoded by PM1a_CNT_BLK and, if supported, PM1b_CNT_BLK. This value is ≥ 2.
PM2_CNT_LEN	1	90	Number of bytes decoded by PM2_CNT_BLK. Support for the PM2 register block is optional. If supported, this value is ≥ 1. If not supported, this field contains zero.
PM_TMR_LEN	1	91	Number of bytes decoded by PM_TMR_BLK. This field’s value must be 4.
GPE0_BLK_LEN	1	92	Number of bytes decoded by GPE0_BLK. The value is a non-negative multiple of 2.
GPE1_BLK_LEN	1	93	Number of bytes decoded by GPE1_BLK. The value is a non-negative multiple of 2.
GPE1_BASE	1	94	Offset within the ACPI general-purpose event model where GPE1 based events start.
CST_CNT	1	95	If non-zero, this field contains the value OSPM writes to the SMI_CMD register to indicate OS support for the _CST object and C States Changed notification.
P_LVL2_LAT	2	96	The worst-case hardware latency, in microseconds, to enter and exit a C2 state. A value > 100 indicates the system does not support a C2 state.
P_LVL3_LAT	2	98	The worst-case hardware latency, in microseconds, to enter and exit a C3 state. A value > 1000 indicates the system does not support a C3 state.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
FLUSH_SIZE	2	100	If WBINVD=0, the value of this field is the number of flush strides that need to be read (using cacheable addresses) to completely flush dirty lines from any processor's memory caches. Notice that the value in FLUSH_STRIDE is typically the smallest cache line width on any of the processor's caches (for more information, see the FLUSH_STRIDE field definition). If the system does not support a method for flushing the processor's caches, then FLUSH_SIZE and WBINVD are set to zero. Notice that this method of flushing the processor caches has limitations, and WBINVD=1 is the preferred way to flush the processors caches. This value is typically at least 2 times the cache size. The maximum allowed value for FLUSH_SIZE multiplied by FLUSH_STRIDE is 2 MB for a typical maximum supported cache size of 1 MB. Larger cache sizes are supported using WBINVD=1. This value is ignored if WBINVD=1. This field is maintained for ACPI 1.0 processor compatibility on existing systems. Processors in new ACPI 2.0-compatible systems are required to support the WBINVD function and indicate this to OSPM by setting the WBINVD field = 1.
FLUSH_STRIDE	2	102	If WBINVD=0, the value of this field is the cache line width, in bytes, of the processor's memory caches. This value is typically the smallest cache line width on any of the processor's caches. For more information, see the description of the FLUSH_SIZE field. This value is ignored if WBINVD=1. This field is maintained for ACPI 1.0 processor compatibility on existing systems. Processors in new ACPI 2.0-compatible systems are required to support the WBINVD function and indicate this to OSPM by setting the WBINVD field = 1.
DUTY_OFFSET	1	104	The zero-based index of where the processor's duty cycle setting is within the processor's P_CNT register.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
DUTY_WIDTH	1	105	The bit width of the processor's duty cycle setting value in the P_CNT register. Each processor's duty cycle setting allows the software to select a nominal processor frequency below its absolute frequency as defined by: THTL_EN = 1 BF * DC/(2^{DUTY_WIDTH}) Where: BF–Base frequency DC–Duty cycle setting When THTL_EN is 0, the processor runs at its absolute BF. A DUTY_WIDTH value of 0 indicates that processor duty cycle is not supported and the processor continuously runs at its base frequency.
DAY_ALRM	1	106	The RTC CMOS RAM index to the day-of-month alarm value. If this field contains a zero, then the RTC day of the month alarm feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that OSPM can use to program the day of the month alarm. See section 4.7.2.4, "Real Time Clock Alarm," for a description of how the hardware works.
MON_ALRM	1	107	The RTC CMOS RAM index to the month of year alarm value. If this field contains a zero, then the RTC month of the year alarm feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that OSPM can use to program the month of the year alarm. If this feature is supported, then the DAY_ALRM feature must be supported also.
CENTURY	1	108	The RTC CMOS RAM index to the century of data value (hundred and thousand year decimals). If this field contains a zero, then the RTC centenary feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that OSPM can use to program the centenary field.
IAPC_BOOT_ARCH	2	109	IA-PC Boot Architecture Flags. See Table 5-10 for a description of this field.
Reserved	1	111	Must be 0.
Flags	4	112	Fixed feature flags. See Table 5-9 for a description of this field.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
RESET_REG	12	116	The address of the reset register represented in Generic Address Structure format (See section 4.7.3.6, “Reset Register,” for a description of the reset mechanism.) **Note**: Only System I/O space, System Memory space and PCI Configuration space (bus #0) are valid for values for Address_Space_ID. Also, Register_Bit_Width must be 8 and Register_Bit_Offset must be 0.
RESET_VALUE	1	128	Indicates the value to write to the RESET_REG port to reset the system. (See section 4.7.3.6, “Reset Register,” for a description of the reset mechanism.)
Reserved	3	129	Must be 0.
X_FIRMWARE_CTRL	8	132	64bit physical address of the FACS.
X_DSDT	8	140	64bit physical address of the DSDT.
X_PM1a_EVT_BLK	12	148	Extended address of the PM1a Event Register Block, represented in Generic Address Structure format. See section 4.7.3.1, “PM1 Event Grouping,” for a hardware description layout of this register block. This is a required field.
X_PM1b_EVT_BLK	12	160	Extended address of the PM1b Event Register Block, represented in Generic Address Structure format. See section 4.7.3.1, “PM1 Event Grouping,” for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero.
X_PM1a_CNT_BLK	12	172	Extended address of the PM1a Control Register Block, represented in Generic Address Structure format. See section 4.7.3.2, “PM1 Control Grouping,” for a hardware description layout of this register block. This is a required field.
X_PM1b_CNT_BLK	12	184	Extended address of the PM1b Control Register Block, represented in Generic Address Structure format. See section 4.7.3.2, “PM1 Control Grouping,” for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero.
X_PM2_CNT_BLK	12	196	Extended address of the Power Management 2 Control Register Block, represented in Generic Address Structure format. See section 4.7.3.4, “PM2 Control (PM2_CNT),” for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero.

Table 5-8 Fixed ACPI Description Table (FADT) Format *(continued)*

Field	Byte Length	Byte Offset	Description
X_PM_TMR_BLK	12	208	Extended address of the Power Management Timer Control Register Block, represented in Generic Address Structure format. See section 4.7.3.3, “Power Management Timer (PM_TMR),” for a hardware description layout of this register block. This is a required field.
X_GPE0_BLK	12	220	Extended address of the General-Purpose Event 0 Register Block, represented in Generic Address Structure format. See section 5.2.8, “Fixed ACPI Description Table,” for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero.
X_GPE1_BLK	12	232	Extended address of the General-Purpose Event 1 Register Block, represented in Generic Address Structure format. See section 5.2.8, “Fixed ACPI Description Table,” for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero.

Table 5-9 Fixed ACPI Description Table Fixed Feature Flags

FACP - Flag	Bit length	Bit offset	Description
WBINVD	1	0	Processor properly implements a functional equivalent to the WBINVD IA-32 instruction. If set, signifies that the WBINVD instruction correctly flushes the processor caches, maintains memory coherency, and upon completion of the instruction, all caches for the current processor contain no cached data other than what OSPM references and allows to be cached. If this flag is not set, the ACPI OS is responsible for disabling all ACPI features that need this function. This field is maintained for ACPI 1.0 processor compatibility on existing systems. Processors in new ACPI 2.0-compatible systems are required to support this function and indicate this to OSPM by setting this field.
WBINVD_FLUSH	1	1	If set, indicates that the hardware flushes all caches on the WBINVD instruction and maintains memory coherency, but does not guarantee the caches are invalidated. This provides the complete semantics of the WBINVD instruction, and provides enough to support the system sleeping states. If neither of the WBINVD flags is set, the system will require FLUSH_SIZE and FLUSH_STRIDE to support sleeping states. If the FLUSH parameters are also not supported, the machine cannot support sleeping states S1, S2, or S3.

Table 5-9 Fixed ACPI Description Table Fixed Feature Flags *(continued)*

FACP - Flag	Bit length	Bit offset	Description
PROC_C1	1	2	A one indicates that the C1 power state is supported on all processors.
P_LVL2_UP	1	3	A zero indicates that the C2 power state is configured to only work on a uniprocessor (UP) system. A one indicates that the C2 power state is configured to work on a UP or multiprocessor (MP) system.
PWR_BUTTON	1	4	A zero indicates the power button is handled as a fixed feature programming model; a one indicates the power button is handled as a control method device. If the system does not have a power button, this value would be “1” and no sleep button device would be present.
SLP_BUTTON	1	5	A zero indicates the sleep button is handled as a fixed feature programming model; a one indicates the sleep button is handled as a control method device. If the system does not have a sleep button, this value would be “1” and no sleep button device would be present.
FIX_RTC	1	6	A zero indicates the RTC wake status is supported in fixed register space; a one indicates the RTC wake status is not supported in fixed register space.
RTC_S4	1	7	Indicates whether the RTC alarm function can wake the system from the S4 state. The RTC must be able to wake the system from an S1, S2, or S3 sleep state. The RTC alarm can optionally support waking the system from the S4 state, as indicated by this value.
TMR_VAL_EXT	1	8	A zero indicates TMR_VAL is implemented as a 24-bit value. A one indicates TMR_VAL is implemented as a 32-bit value. The TMR_STS bit is set when the most significant bit of the TMR_VAL toggles.
DCK_CAP	1	9	A zero indicates that the system cannot support docking. A one indicates that the system can support docking. Notice that this flag does not indicate whether or not a docking station is currently present; it only indicates that the system is capable of docking.
RESET_REG_SUP	1	10	If set, indicates the system supports system reset via the FADT RESET_REG as described in section 4.7. 3.6, “Reset Register.”
SEALED_CASE	1	11	System Type Attribute. If set indicates that the system has no internal expansion capabilities and the case is sealed.
HEADLESS	1	12	System Type Attribute. If set indicates the system does not have local video capabilities or local input devices.
CPU_SW_SLP	1	13	If set, indicates to OSPM that a processor native instruction must be executed after writing the SLP_TYPx register.
Reserved	18	14	

5.2.8.1 Preferred PM Profile System Types

The following descriptions of preferred power management profile system types are to be used as a guide for setting the Preferred_PM_Profile field in the FADT. OSPM can use this field to set default power management policy parameters during OS installation.

Desktop. A single user, full featured, stationary computing device that resides on or near an individual's work area. Most often contains one processor. Must be connected to AC power to function. This device is used to perform work that is considered mainstream corporate or home computing (for example, word processing, Internet browsing, spreadsheets, and so on).

Mobile. A single-user, full-featured, portable computing device that is capable of running on batteries or other power storage devices to perform its normal functions. Most often contains one processor. This device performs the same task set as a desktop. However it may have limitations dues to its size, thermal requirements, and/or power source life.

Workstation. A single-user, full-featured, stationary computing device that resides on or near an individual's work area. Often contains more than one processor. Must be connected to AC power to function. This device is used to perform large quantities of computations in support of such work as CAD/CAM and other graphics-intensive applications.

Enterprise Server. A multiuser, stationary computing device that frequently resides in a separate, often specially designed, room. Will almost always contain more than one processor. Must be connected to AC power to function. This device is used to support large-scale networking, database, communications, or financial operations within a corporation or government.

SOHO Server. A multiuser, stationary computing device that frequently resides in a separate area or room in a small or home office. May contain more than one processor. Must be connected to AC power to function. This device is generally used to support all of the networking, database, communications, and financial operations of a small office or home office.

Appliance PC. A device specifically designed to operate in a low-noise, high-availability environment such as a consumer's living rooms orfamily room. Most often contains one processor. This category also includes home Internet gateways, Web pads, set top boxes and other devices that support ACPI. Must be connected to AC power to function. Normally they are sealed case style and may only perform a subset of the tasks normally associated with today's personal computers.

5.2.8.2 System Type Attributes

This set of flags is used by the OS to assist in determining assumptions about power and device management. These flags are read at boot time and are used to make decisions about power management and device settings. For example, a system that has the SEALED_CASE bit set may take a very aggressive low noise policy toward thermal management. In another example an OS might not load video, keyboard or mouse drivers on a HEADLESS system.

5.2.8.3 IA-PC Boot Architecture Flags

This set of flags is used by an OS to guide the assumptions it can make in initializing hardware on IA-PC platforms. These flags are used by an OS at boot time (before the OS is capable of providing an operating environment suitable for parsing the ACPI namespace) to determine the code paths to take during boot. In IA-PC platforms with reduced legacy hardware, the OS can skip code paths for legacy devices if none are present. For example, if there are no ISA devices, an OS could skip code that assumes the presence of these devices and their associated resources. These flags are used independently of the ACPI namespace. The presence of other devices must be described in the ACPI namespace as specified in section 6, "Configuration."

These flags pertain only to IA-PC platforms. On other system architectures, the entire field should be set to 0.

Table 5-10 Fixed ACPI Description Table Boot Architecture Flags

BOOT_ARCH	Bit length	Bit offset	Description
LEGACY_DEVICES	1	0	If set, indicates that the motherboard supports user-visible devices on the LPC or ISA bus. User-visible devices are devices that have end-user accessible connectors (for example, LPT port), or devices for which the OS must load a device driver so that an end-user application can use a device. If clear, the OS may assume there are no such devices and that all devices in the system can be detected exclusively via industry standard device enumeration mechanisms (including the ACPI namespace).
8042	1	1	If set, indicates that the motherboard contains support for a port 60 and 64 based keyboard controller, usually implemented as an 8042 or equivalent micro-controller.
Reserved	14	2	Must be 0.

5.2.9 Firmware ACPI Control Structure (FACS)

The Firmware ACPI Control Structure (FACS) is a structure in read/write memory that the BIOS reserves for ACPI usage. This structure is passed to an ACPI-compatible OS using the FADT. For more information about the FADT FIRMWARE_CTRL field, see section 5.2.8, “Fixed ACPI Description Table (FADT).”

The BIOS aligns the FACS on a 64-byte boundary anywhere within the system’s memory address space. The memory where the FACS structure resides must not be reported as system AddressRangeMemory in the system address map. For example, the E820 address map reporting interface would report the region as AddressRangeReserved. For more information about system address map reporting interfaces, see section 15, “System Address Map Interfaces.”

Table 5-11 Firmware ACPI Control Structure (FACS)

Field	Byte Length	Byte Offset	Description
Signature	4	0	‘FACS’
Length	4	4	Length, in bytes, of the entire Firmware ACPI Control Structure. This value is 64 bytes or larger.
Hardware Signature	4	8	The value of the system’s “hardware signature” at last boot. This value is calculated by the BIOS on a best effort basis to indicate the base hardware configuration of the system such that different base hardware configurations can have different hardware signature values. OSPM uses this information in waking from an S4 state, by comparing the current hardware signature to the signature values saved in the non-volatile sleep image. If the values are not the same, OSPM assumes that the saved non-volatile image is from a different hardware configuration and cannot be restored.

Table 5-11 Firmware ACPI Control Structure (FACS) ***(continued)***

Field	Byte Length	Byte Offset	Description
Firmware_Waking_ Vector	4	12	This field is superseded in ACPI 2.0 by the X_Firmware_Waking_Vector field. The 32-bit address field where OSPM puts its waking vector. Before transitioning the system into a global sleeping state, OSPM fills in this field with the physical memory address of an OS-specific wake function. During POST, the platform firmware first checks if the value of the X_Firmware_Waking_Vector field is non-zero and if so transfers control to OSPM as outlined in the X_Firmware_Waking_vector field description below. If the X_Firmware_Waking_Vector field is zero then the platform firmware checks the value of this field and if it is non-zero, transfers control to the specified address. On PCs, the wake function address is in memory below 1 MB and the control is transferred while in real mode. OSPM's wake function restores the processors' context. For IA-PC platforms, the following example shows the relationship between the physical address in the Firmware Waking Vector and the real mode address the BIOS jumps to. If, for example, the physical address is 0x12345, then the BIOS must jump to real mode address 0x1234:0x0005. In general this relationship is Real-mode address = Physical address>>4 : Physical address and 0x000F Notice that on IA-PC platforms, A20 must be enabled when the BIOS jumps to the real mode address derived from the physical address stored in the Firmware Waking Vector.
Global_Lock	4	16	This field contains the Global Lock used to synchronize access to shared hardware resources between the OSPM environment and an external controller environment (for example, the SMI environment). This lock is owned exclusively by either OSPM or the firmware at any one time. When ownership of the lock is attempted, it might be busy, in which case the requesting environment exits and waits for the signal that the lock has been released. For example, the Global Lock can be used to protect an embedded controller interface such that only OSPM or the firmware will access the embedded controller interface at any one time. See section 5.2.9.1, "Global Lock," for more information on acquiring and releasing the Global Lock.
Flags	4	20	Firmware control structure flags. See Table 5-12 for a description of this field.

Table 5-11 Firmware ACPI Control Structure (FACS) *(continued)*

Field	Byte Length	Byte Offset	Description
X_Firmware_Waking _Vector	8	24	64-bit physical address of OSPM's Waking Vector. Before transitioning the system into a global sleeping state, OSPM fills in this field with the physical memory address of an OS-specific wake function. During POST, the platform firmware checks if the value of this field is non-zero and if so transfers control to OSPM by jumping to this address. Prior to transferring control, the execution environment must be configured as follows: Memory address translation / paging and interrupts must be disabled. For IA 32-bit platforms, a 4GB flat address space for all segment registers and EFLAGS.IF set to 0. For 64-bit Itanium^{TM}-based platforms, the processor must have psr.i, psr.it, psr.dt, and psr.rt set to 0. See the *Intel® Itanium^{TM} Architecture Software Developer's Manual* for more information. If this field is zero then OSPM checks the Firmware_Waking_Vector field as outlined above.
Version	1	32	1–Version of this table
Reserved	31	33	This value is zero.

Table 5-12 Firmware Control Structure Feature Flags

FACS – Flag	Bit Length	Bit Offset	Description
S4BIOS_F	1	0	Indicates whether the platform supports S4BIOS_REQ. If S4BIOS_REQ is not supported, OSPM must be able to save and restore the memory state in order to use the S4 state.
Reserved	31	1	The value is zero.

5.2.9.1 Global Lock

The purpose of the ACPI Global Lock is to provide mutual exclusion between the host OS and the ROM BIOS. The Global Lock is a 32-bit (DWORD) value in read/write memory located within the FACS and is accessed and updated by both the OS environment and the SMI environment in a defined manner to provide an exclusive lock. Note: this is not a *pointer* to the Global Lock, it is the *actual* memory location of the lock. The FACS and Global Lock may be located anywhere in physical memory.

By convention, this lock is used to ensure that while one environment is accessing some hardware, the other environment is not. By this convention, when ownership of the lock fails because the other environment owns it, the requesting environment sets a "pending" state within the lock, exits its attempt to acquire the lock, and waits for the owning environment to signal that the lock has been released before attempting to acquire the lock again. When releasing the lock, if the pending bit in the lock is set after the lock is released, a signal is sent via an interrupt mechanism to the other environment to inform it that the lock has been released. During interrupt handling for the "lock released" event within the corresponding environment, if the lock ownership were still desired an attempt to acquire the lock would be made. If ownership is not acquired, then the environment must again set “pending” and wait for another “lock release” signal.

Table 5-13 shows the encoding of the Global Lock DWORD in memory.

Table 5-13 Global Lock Structure within the FACS

Field	Bit Length	Bit Offset	Description
Pending	1	0	Non-zero indicates that a request for ownership of the Global Lock is pending.
Owned	1	1	Non-zero indicates that the Global Lock is Owned.
Reserved	30	2	Reserved for future use.

The following code sequence is used by both OSPM and the firmware to acquire ownership of the Global Lock. If non-zero is returned by the function, the caller has been granted ownership of the Global Lock and can proceed. If zero is returned by the function, the caller has not been granted ownership of the Global Lock, the “pending” bit has been set, and the caller must wait until it is signaled by an interrupt event that the lock is available before attempting to acquire access again.

Note: In the examples that follow, the “GlobalLock” variable is a pointer that has been previously initialized to point to the 32-bit Global Lock location within the FACS.

```
AcquireGlobalLock:
           mov     ecx, GlobalLock            ; ecx = Address of Global Lock in FACS
acq10:     mov     eax, [ecx]                 ; Get current value of Global Lock


           mov     edx, eax
           and     edx, not 1                 ; Clear pending bit
           bts     edx, 1                     ; Check and set owner bit
           adc     edx, 0                     ; If owned, set pending bit

           lock cmpxchg dword ptr[ecx], edx  ; Attempt to set new value
           jnz short acq10                    ; If not set, try again

           cmp     dl, 3                      ; Was it acquired or marked pending?
           sbb     eax, eax                   ; acquired = -1, pending = 0

           ret
```

The following code sequence is used by OSPM and the firmware to release ownership of the Global Lock. If non-zero is returned, the caller must raise the appropriate event to the other environment to signal that the Global Lock is now free. Depending on the environment, this signaling is done by setting the either the GBL_RLS or BIOS_RLS within their respective hardware register spaces. This signal only occurs when the other environment attempted to acquire ownership while the lock was owned.

```
ReleaseGlobalLock:
           mov    ecx, GlobalLock           ; ecx = Address of Global Lock in FACS
rel10:     mov    eax, [ecx]                ; Get current value of Global Lock

           mov    edx, eax
           and    edx, not 03h              ; Clear owner and pending field

           lock cmpxchg dword ptr[ecx], edx ; Attempt to set it
           jnz short rel10                  ; If not set, try again

           and    eax, 1                    ; Was pending set?

           ; If one is returned (we were pending) the caller must signal that the
           ; lock has been released using either GBL_RLS or BIOS_RLS as appropriate

           ret
```

Although using the Global Lock allows various hardware resources to be shared, it is important to notice that its usage when there is ownership contention could entail a significant amount of system overhead as well as waits of an indeterminate amount of time to acquire ownership of the Global Lock. For this reason, implementations should try to design the hardware to keep the required usage of the Global Lock to a minimum.

The Global Lock is required whenever a logical register in the hardware is shared. For example, if bit 0 is used by ACPI (OSPM) and bit 1 of the same register is used by SMI, then access to that register needs to be protected under the Global Lock, ensuring that the register's contents do not change from underneath one environment while the other is making changes to it. Similarly if the entire register is shared, as the case might be for the embedded controller interface, access to the register needs to be protected under the Global Lock.

5.2.10 Definition Blocks

A Definition Block consists of data in AML format (see section 5.4 “Definition Block Encoding”) and contains information about hardware implementation details in the form of AML objects that contain data, AML code, or other AML objects. The top-level organization of this information after a definition block is loaded is name-tagged in a hierarchical namespace.

OSPM “loads” or “unloads” an entire definition block as a logical unit. OSPM will load a definition block either as a result of executing the AML **Load()** or **LoadTable()** operator or encountering a table definition during initialization. During initialization, OSPM loads the Differentiated System Description Table (DSDT), which contains the Differentiated Definition Block, using the DSDT pointer retrieved from the FADT. OSPM will load other definition blocks during initialization as a result of encountering Secondary System Description Table (SSDT) definitions in the RSDT/XSDT. The DSDT and SSDT are described in the following sections.

As mentioned, the AML **Load()** and **LoadTable()** operators make it possible for a Definition Block to load other Definition Blocks, either statically or dynamically, where they in turn can either define new system attributes or, in some cases, build on prior definitions. Although this gives the hardware the ability to vary widely in implementation, it also confines it to reasonable boundaries. In some cases, the Definition Block format can describe only specific and well-understood variances. In other cases, it permits implementations to be expressible only by means of a specified set of “built in” operators. For example, the Definition Block has built in operators for I/O space.

In theory, it might be possible to define something like PCI configuration space in a Definition Block by building it from I/O space, but that is not the goal of the definition block. Such a space is usually defined as a “built in” operator.

Some AML operators perform simple functions, and others encompass complex functions. The power of the Definition block comes from its ability to allow these operations to be glued together in numerous ways, to provide functionality to OSPM.

The AML operators defined in this specification are intended to allow many useful hardware designs to be easily expressed, not to allow all hardware designs to be expressed.

Note: To accommodate addressing beyond 32 bits, the integer type is expanded to 64 bits in ACPI 2.0, see section 16.2.2, “ASL Data Types”. Existing ACPI definition block implementations may contain an inherent assumption of a 32-bit integer width. Therefore, to maintain backwards compatibility, OSPM uses the Revision field, in the header portion of system description tables containing Definition Blocks, to determine whether integers declared within the Definition Block are to be evaluated as 32-bit or 64-bit values. A Revision field value greater than or equal to 2 signifies that integers declared within the Definition Block are to be evaluated as 64-bit values. The ASL writer specifies the value for the Definition Block table header’s Revision field via the ASL DefinitionBlockTerm’s *ComplianceRevision* field. See section 16.2.3.1, “Definition Block Term”, for more information. It is the responsibility of the ASL writer to ensure the Definition Block’s compatibility with the corresponding integer width when setting the *ComplianceRevision* field.

5.2.10.1 Differentiated System Description Table (DSDT)

The Differentiated System Description Table (DSDT) is part of the system fixed description. The DSDT is comprised of a system description table header followed by data in Definition Block format. This Definition Block is like all other Definition Blocks, with the exception that it cannot be unloaded. See section 5.2.10, “Definition Blocks,” for a description of Definition Blocks.

Table 5-13a Differentiated System Description Table Fields (DSDT)

Field	**Byte Length**	**Byte Offset**	**Description**
Header			
Signature	4	0	‘DSDT.’ Signature for the Differentiated System Description Table.
Length	4	4	Length, in bytes, of the entire DSDT (including the header).
Revision	1	8	2
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	The manufacture model ID.
OEM Revision	4	24	OEM revision of DSDT for supplied OEM Table ID.
Creator ID	4	28	Vendor ID for the ASL Compiler.
Creator Revision	4	32	Revision number of the ASL Compiler.
Definition Block	*n*	36	*n* bytes of AML code (see section 5.4, “Definition Block Encoding”)

5.2.10.2 Secondary System Description Table (SSDT)

Secondary System Description Tables (SSDT) are a continuation of the DSDT. The SSDT is comprised of a system description table header followed by data in Definition Block format. There can be multiple SSDTs present. OSPM first loads the DSDT and then loads each SSDT. This allows the OEM to provide the base support in one table and add smaller system options in other tables. For example, the OEM might put dynamic object definitions into a secondary table such that the firmware can construct the dynamic information at boot without needing to edit the static DSDT. A SSDT can only rely on the DSDT being loaded prior to it.

Table 5-13b Secondary System Description Table Fields (SSDT)

Field	**Byte Length**	**Byte Offset**	**Description**
Header			
Signature	4	0	‘SSDT.’ Signature for the Secondary System Description Table.
Length	4	4	Length, in bytes, of the entire SSDT (including the header).
Revision	1	8	2
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	The manufacture model ID.
OEM Revision	4	24	OEM revision of DSDT for supplied OEM Table ID.
Creator ID	4	28	Vendor ID for the ASL Compiler.
Creator Revision	4	32	Revision number of the ASL Compiler.
Definition Block	*n*	36	*n* bytes of AML code (see section 5.4 , “Definition Block Encoding”)

5.2.10.3 Persistent System Description Table (PSDT)

The table signature, “PSDT” refers to the Persistent System Description Table (PSDT) defined in the ACPI 1.0 specification. The PSDT was judged to provide no specific benefit and as such has been deleted from this version of the ACPI specification. OSPM will evaluate a table with the “PSDT” signature in like manner to the evaluation of an SSDT as described in section 5.2.10.2, “Secondary System Description Table.”

5.2.10.4 Multiple APIC Description Table (MADT)

The ACPI interrupt model describes all interrupts for the entire system in a uniform interrupt model implementation. Supported interrupt models include the PC-AT–compatible dual 8259 interrupt controller and, for Intel processor-based systems, the Intel Advanced Programmable Interrupt Controller (APIC) and Intel Streamlined Advanced Programmable Interrupt Controller (SAPIC). The choice of the interrupt model(s) to support is up to the platform designer. The interrupt model cannot be dynamically changed by the system firmware; OSPM will choose which model to use and install support for that model at the time of installation. If a platform supports both models, an OS will install support for one model or the other; it will not mix models. Multi-boot capability is a feature in many modern operating systems. This means that a system may have multiple operating systems or multiple instances of an OS installed at any one time. Platform designers must allow for this.

This section describes the format of the Multiple APIC Description Table (MADT), which provides OSPM with information necessary for operation on systems with APIC or SAPIC implementations.

ACPI represents all interrupts as “flat” values known as global system interrupts. Therefore to support APICs or SAPICs on an ACPI-enabled system, each used APIC or SAPIC interrupt input must be mapped to the global system interrupt value used by ACPI. See Section 5.2.11. Global System Interrupts,” for a description of Global System Interrupts.

Additional support is required to handle various multi-processor functions that APIC or SAPIC implementations might support (for example, identifying each processor’s local APIC ID).

All addresses in the MADT are processor-relative physical addresses.

Table 5-14 Multiple APIC Description Table (MADT) Format

Field	**Byte Length**	**Byte Offset**	**Description**
Header			
Signature	4	0	‘APIC.’ Signature for the Multiple APIC Description Table.
Length	4	4	Length, in bytes, of the entire MADT.
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the MADT, the table ID is the manufacturer model ID.
OEM Revision	4	24	OEM revision of MADTfor supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.
Local APIC Address	4	36	The 32-bit physical address at which each processor can access its local APIC.
Flags	4	40	Multiple APIC flags. See Table 5-15 for a description of this field.
APIC Structure[*n*]	—	44	A list of APIC structures for this implementation. This list will contain all of the I/O APIC, I/O SAPIC, Local APIC, Local SAPIC, Interrupt Source Override, Non-maskable Interrupt Source, Local APIC NMI Source, Local APIC Address Override, and Platform Interrupt Sources structures needed to support this platform. These structures are described in the following sections.

Table 5-15 Multiple APIC Flags

Multiple APIC Flags	**Bit Length**	**Bit Offset**	**Description**
PCAT_COMPAT	1	0	A one indicates that the system also has a PC-AT-compatible dual-8259 setup. The 8259 vectors must be disabled (that is, masked) when enabling the ACPI APIC operation.

Multiple APIC Flags	Bit Length	Bit Offset	Description
Reserved	31	1	This value is zero.

Immediately after the Flags value in the MADTis a list of APIC structures that declare the APIC features of the machine. The first byte of each structure declares the type of that structure and the second byte declares the length of that structure.

Table 5-16 APIC Structure Types

Value	Description
0	Processor Local APIC
1	I/O APIC
2	Interrupt Source Override
3	Non-maskable Interrupt Source (NMI)
4	Local APIC NMI Structure
5	Local APIC Address Override Structure
6	I/O SAPIC
7	Local SAPIC
8	Platform Interrupt Sources
>8	Reserved. OSPM skips structures of the reserved type.

5.2.10.5 Processor Local APIC

When using the APIC interrupt model, each processor in the system is required to have a Processor Local APIC record and an ACPI Processor object. OSPM does not expect the information provided in this table to be updated if the processor information changes during the lifespan of an OS boot. While in the sleeping state, processors are not allowed to be added, removed, nor can their APIC ID or Flags change. When a processor is not present, the Processor Local APIC information is either not reported or flagged as disabled.

Table 5-17 Processor Local APIC Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	0–Processor Local APIC structure
Length	1	1	8
ACPI Processor ID	1	2	The ProcessorId for which this processor is listed in the ACPI Processor declaration operator. For a definition of the Processor operator, see section 16.2.3.3.1.17, “Processor (Declare Processor).”
APIC ID	1	3	The processor’s local APIC ID.
Flags	4	4	Local APIC flags. See Table 5-18 for a description of this field.

Table 5-18 Local APIC Flags

Local APIC - Flags	Bit Length	Bit Offset	Description
Enabled	1	0	If zero, this processor is unusable, and the operating system support will not attempt to use it.
Reserved	31	1	Must be zero.

5.2.10.6 I/O APIC

In an APIC implementation, there are one or more I/O APICs. Each I/O APIC has a series of interrupt inputs, referred to as INTI*n*, where the value of *n* is from 0 to the number of the last interrupt input on the I/O APIC. The I/O APIC structure declares which global system interrupts are uniquely associated with the I/O APIC interrupt inputs. There is one I/O APIC structure for each I/O APIC in the system. For more information on global system interrupts see Section 5.2.11, “Global System Interrupts.”

Table 5-19 I/O APIC Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	1–I/O APIC structure
Length	1	1	12
I/O APIC ID	1	2	The I/O APIC’s ID.
Reserved	1	3	0
I/O APIC Address	4	4	The 32-bit physical address to access this I/O APIC. Each I/O APIC resides at a unique address.
Global System Interrupt Base	4	8	The global system interrupt number where this I/O APIC’s interrupt inputs start. The number of interrupt inputs is determined by the I/O APIC’s *Max Redir Entry* register.

5.2.10.7 Platforms with APIC and Dual 8259 Support

Systems that support both APIC and dual 8259 interrupt models must map global system interrupts 0-15 to the 8259 IRQs 0-15, except where Interrupt Source Overrides are provided (see section 5.2.10.8, “Interrupt Source Overrides”). This means that I/O APIC interrupt inputs 0-15 must be mapped to global system interrupts 0-15 and have identical sources as the 8259 IRQs 0-15 unless overrides are used. This allows a platform to support OSPM implementations that use the APIC model as well as OSPM implementations that use the 8259 model (OSPM will only use one model; it will not mix models).

When OSPM supports the 8259 model, it will assume that all interrupt descriptors reporting global system interrupts 0-15 correspond to 8259 IRQs. In the 8259 model all global system interrupts greater than 15 are ignored. If OSPM implements APIC support, it will enable the APIC as described by the APIC specification and will use all reported global system interrupts that fall within the limits of the interrupt inputs defined by the I/O APIC structures. For more information on hardware resource configuration see section 6, “Configuration.”

5.2.10.8 Interrupt Source Overrides

Interrupt Source Overrides are necessary to describe variances between the IA-PC standard dual 8259 interrupt definition and the platform's implementation.

It is assumed that the ISA interrupts will be identity-mapped into the first I/O APIC sources. Most existing APIC designs, however, will contain at least one exception to this assumption. The Interrupt Source Override Structure is provided in order to describe these exceptions. It is not necessary to provide an Interrupt Source Override for every ISA interrupt. Only those that are not identity-mapped onto the APIC interrupt inputs need be described.

Note: This specification only supports overriding ISA interrupt sources.

For example, if your machine has the ISA Programmable Interrupt Timer (PIT) connected to ISA IRQ 0, but in APIC mode, it is connected to I/O APIC interrupt input 2, then you would need an Interrupt Source Override where the source entry is '0' and the Global System Interrupt is '2.'

Table 5-20 Interrupt Source Override Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	2–Interrupt Source Override
Length	1	1	10
Bus	1	2	0–Constant, meaning ISA
Source	1	3	Bus-relative interrupt source (IRQ)
Global System Interrupt	4	4	The Global System Interrupt that this bus-relative interrupt source will signal.
Flags	2	8	MPS INTI flags. See Table 5-21 for a description of this field.

The MPS INTI flags listed in Table 5-21 are identical to the flags used in Table 4-10 of the MPS version 1.4 specifications. The Polarity flags are the PO bits and the Trigger Mode flags are the EL bits.

Table 5-21 MPS INTI Flags

Local APIC - Flags	Bit Length	Bit Offset	Description
Polarity	2	0	Polarity of the APIC I/O input signals: 00–Conforms to the specifications of the bus (For example, EISA is active-low for level-triggered interrupts) 01–Active high 10–Reserved 11–Active low

Local APIC - Flags	**Bit Length**	**Bit Offset**	**Description**
Trigger Mode	2	2	Trigger mode of the APIC I/O Input signals: 00–Conforms to specifications of the bus (For example, ISA is edge-triggered) 01–Edge-triggered 10–Reserved 11–Level-triggered
Reserved	12	4	Must be zero.

Interrupt Source Overrides are also necessary when an identity mapped interrupt input has a non-standard polarity.

Note: You must have an interrupt source override entry for the IRQ mapped to the SCI interrupt if this IRQ is not identity mapped. This entry will override the value in SCI_INT in FADT. For example, if SCI is connected to IRQ 9 in PIC mode and IRQ 9 is connected to INTIN11 in APIC mode, you should have 9 in SCI_INT in the FADT and an interrupt source override entry mapping IRQ 9 to INTIN11.

5.2.10.9 Non-Maskable Interrupt Sources (NMIs)

This structure allows a platform designer to specify which I/O (S)APIC interrupt inputs should be enabled as non-maskable. Any source that is non-maskable will not be available for use by devices.

Table 5-22 Non-maskable Source Structure

Field	**Byte Length**	**Byte Offset**	**Description**
Type	1	0	3–NMI
Length	1	1	8
Flags	2	2	Same as MPS INTI flags
Global System Interrupt	4	4	The Global System Interrupt that this NMI will signal.

5.2.10.10 Local APIC NMI

This structure describes the Local APIC interrupt input (LINT*n*) that NMI is connected to for each of the processors in the system where such a connection exists. This information is needed by OSPM to enable the appropriate local APIC entry.

Each Local APIC NMI connection requires a separate Local APIC NMI structure. For example, if the platform has 4 processors with ID 0-3 and NMI is connected LINT1 for processor 3 and 2, two Local APIC NMI entries would be needed in the MADT.

Table 5-23 Local APIC NMI Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	4–Local APIC NMI Structure
Length	1	1	6
ACPI Processor ID	1	2	Processor ID corresponding to the ID listed in the processor object. A value of 0xff signifies that this applies to all processors in the machine.
Flags	2	3	MPS INTI flags. See Table 5-21 for a description of this field.
Local APIC LINT#	1	5	Local APIC interrupt input LINTn to which NMI is connected.

5.2.10.11 Local APIC Address Override Structure

This optional structure supports 64-bit systems by providing an override of the physical address of the local APIC in the MADT'stable header, which is defined as a 32-bit field.

If defined, OSPM must use the address specified in this structure for all local APICs (and local SAPICs), rather than the address contained in the MADT's table header. Only one Local APIC Address Override Structure may be defined.

Table 5-24 Local APIC Address Override Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	5–Local APIC Address Override Structure
Length	1	1	12
Reserved	2	2	Reserved (must be set to zero)
Local APIC Address	8	4	Physical address of Local APIC. For Itanium™-based systems, this field contains the starting address of the Processor Interrupt Block. See the *Intel® Itanium™ Architecture Software Developer's Manual* for more information.

5.2.10.12 I/O SAPIC Structure

The I/O SAPIC structure is very similar to the I/O APIC structure. If both I/O APIC and I/O SAPIC structures exist for a specific APIC ID, the information in the I/O SAPIC structure must be used.

The I/O SAPIC structure uses the I/O_APIC_ID field as defined in the I/O APIC table. The Vector_Base field remains unchanged but has been moved. The I/O APIC address has been deleted. A new address and reserved field have been added.

Table 5-25 I/O SAPIC Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	6–I/O SAPIC Structure
Length	1	1	16
I/O APIC ID	1	2	I/O SAPIC ID
Reserved	1	3	Reserved (must be zero)
Global System Interrupt Base	4	4	The global system interrupt number where this I/O SAPIC's interrupt inputs start. The number of interrupt inputs is determined by the I/O SAPIC's *Max Redir Entry* register.
I/O SAPIC Address	8	8	The 64-bit physical address to access this I/O SAPIC. Each I/O SAPIC resides at a unique address.

If defined, OSPM must use the information contained in the I/O SAPIC structure instead of the information from the I/O APIC structure.

If both I/O APIC and an I/O SAPIC structures exist in an MADT, the OEM/BIOS writer must prevent "mixing" I/O APIC and I/O SAPIC addresses. This is done by ensuring that there are at least as many I/O SAPIC structures as I/O APIC structures and that every I/O APIC structure has a corresponding I/O SAPIC structure (same APIC ID).

5.2.10.13 Local SAPIC Structure

The Processor local SAPIC structure is very similar to the processor local APIC structure. When using the SAPIC interrupt model, each processor in the system is required to have a Processor Local SAPIC record and an ACPI Processor object. OSPM does not expect the information provided in this table to be updated if the processor information changes during the lifespan of an OS boot. While in the sleeping state, processors are not allowed to be added, removed, nor can their SAPIC ID or Flags change. When a processor is not present, the Processor Local SAPIC information is either not reported or flagged as disabled.

Table 5-26 Processor Local SAPIC Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	7–Processor Local SAPIC structure
Length	1	1	12
ACPI Processor ID	1	2	The Processor Id listed in the processor object. For a definition of the Processor object, see section 16.2.3.3.1.17, "Processor (Declare Processor)."
Local SAPIC ID	1	3	The processor's local SAPIC ID
Local SAPIC EID	1	4	The processor's local SAPIC EID
Reserved	3	5	Reserved (must be set to zero)
Flags	4	8	Local SAPIC flags. See Table 5-18 for a description of this field.

5.2.10.14 Platform Interrupt Source Structure

The Platform Interrupt Source structure is used to communicate which I/O SAPIC interrupt inputs are connected to the platform interrupt sources.

Platform Management Interrupts (PMIs) are used to invoke platform firmware to handle various events (similar to SMI in IA-32). The Intel® Itanium^{TM} architecture permits the I/O SAPIC to send a vector value in the interrupt message of the PMI type. This value is specified in the I/O SAPIC Vector field of the Platform Interrupt Sources Structure.

INIT messages cause processors to soft reset.

If a platform can generate an interrupt after correcting platform errors (e.g., single bit error correction), the interrupt input line used to signal such corrected errors is specified by the Global System Interrupt field in the following table. The firmware indicates the processor that can retrieve the corrected platform error information through the Processor ID and EID fields in the structure below. In some systems, retrieval of the error information may not be possible from other processors. OSPM is required to program the I/O SAPIC redirection table entries with the Processor ID, EID values specified by the ACPI system firmware. Refer to the Itanium^{TM} Processor Family System Abstraction Layer (SAL) Specification for details on handling the Corrected Platform Error Interrupt.

Table 5-27 Platform Interrupt Sources Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	8–Platform Interrupt Source structure
Length	1	1	16
Flags	2	2	MPS INTI flags. See Table 5-21 for a description of this field.
Interrupt Type	1	4	1–PMI 2–INIT 3–Corrected Platform Error Interrupt All other values are reserved.
Processor ID	1	5	Processor ID of destination.
Processor EID	1	6	Processor EID of destination.
I/O SAPIC Vector	1	7	Value that OSPM must use to program the vector field of the I/O SAPIC redirection table entry for entries with the PMI interrupt type.
Global System Interrupt	4	8	The Global System Interrupt that this platform interrupt will signal.
Reserved	4	12	Reserved, must be zero.

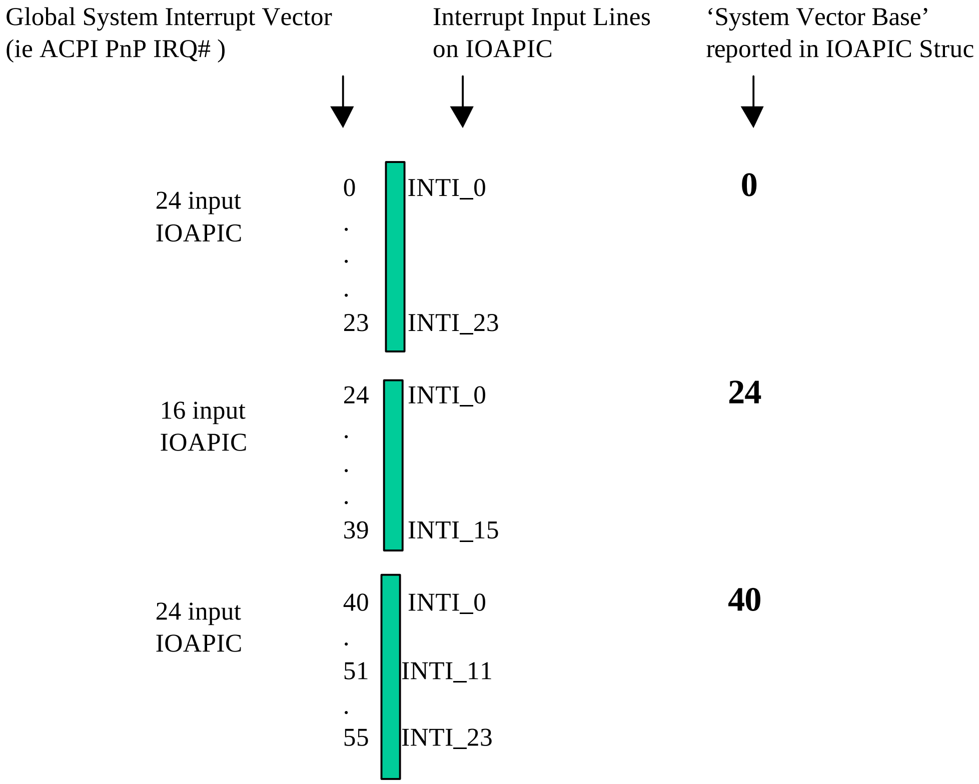

Figure 5-3 APIC–Global System Interrupts

5.2.11 Global System Interrupts

Global System Interrupts can be thought of as ACPI Plug and Play IRQ numbers. They are used to virtualize interrupts in tables and in ASL methods that perform resource allocation of interrupts. Do not confuse global system interrupts with ISA IRQs although in the case of the IA-PC 8259 interrupts they correspond in a one-to-one fashion.

There are two interrupt models used in ACPI-enabled systems.

The first model is the APIC model. In the APIC model, the number of interrupt inputs supported by each I/O APIC can vary. OSPM determines the mapping of the Global System Interrupts by determining how many interrupt inputs each I/O APIC supports and by determining the global system interrupt base for each I/O APIC as specified by the I/O APIC Structure. OSPM determines the number of interrupt inputs by reading the Max Redirection register from the I/O APIC. The global system interrupts mapped to that I/O APIC begin at the global system interrupt base and extending through the number of interrupts specified in the Max Redirection register. This mapping is depicted in Figure 5-3.

There is exactly one I/O APIC structure per I/O APIC in the system.

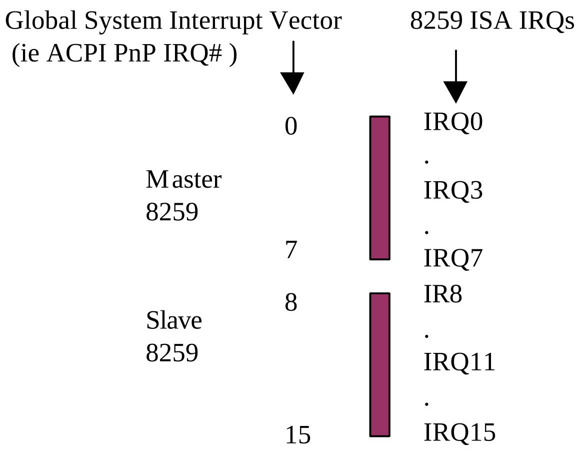

Figure 5-4 8259–Global System Interrupts

The other interrupt model is the standard AT style mentioned above which uses ISA IRQs attached to a master slave pair of 8259 PICs. The system vectors correspond to the ISA IRQs. The ISA IRQs and their mappings to the 8259 pair are part of the AT standard and are well defined. This mapping is depicted in Figure 5-4.

5.2.12 Smart Battery Table (SBST)

If the platform supports batteries as defined by the Smart Battery Specification 1.0 or 1.1, then an Smart Battery Table (SBST) is present. This table indicates the energy level trip points that the platform requires for placing the system into the specified sleeping state and the suggested energy levels for warning the user to transition the platform into a sleeping state. Notice that while Smart Batteries can report either in current (mA/mAh) or in energy (mW/mWh), OSPM must set them to operate in energy (mW/mWh) mode so that the energy levels specified in the SBST can be used. OSPM uses these tables with the capabilities of the batteries to determine the different trip points. For more precise definitions of these levels, see section 3.9.3, “Battery Gas Gauge.”

Table 5-28 Smart Battery Description Table (SBST) Format

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	‘SBST.’ Signature for the Smart Battery Description Table (SBST).
Length	4	4	Length, in bytes, of the entire SBST
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the SBST, the table ID is the manufacturer model ID.
OEM Revision	4	24	OEM revision of SBST for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.
Warning Energy Level	4	36	OEM suggested energy level in milliWatt-hours (mWh) at which OSPM warns the user.
Low Energy Level	4	40	OEM suggested platform energy level in mWh at which OSPM will transition the system to a sleeping state.
Critical Energy Level	4	44	OEM suggested platform energy level in mWh at which OSPM performs an emergency shutdown.

5.2.13 Embedded Controller Boot Resources Table

This optional table provides the processor-relative, translated resources of an Embedded Controller. The presence of this table allows OSPM to provide Embedded Controller operation region space access before the namespace has been evaluated. If this table is not provided, the Embedded Controller region space will not be available until the Embedded Controller device in the AML namespace has been discovered and enumerated. The availability of the region space can be detected by providing a _REG method object underneath the Embedded Controller device.

Table 5-29 Embedded Controller Boot Resources Table Format

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	'ECDT.' Signature for the Embedded Controller Table.
Length	4	4	Length, in bytes, of the entire Embedded Controller Table
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the Embedded Controller Table, the table ID is the manufacturer model ID.
OEM Revision	4	24	OEM revision of Embedded Controller Table for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler.
EC_CONTROL	12	36	Contains the processor relative address, represented in Generic Address Structure format, of the Embedded Controller Command/Status register. **Note**: Only System I/O space and System Memory space are valid for values for Address_Space_ID.
EC_DATA	12	48	Contains the processor-relative address, represented in Generic Address Structure format, of the Embedded Controller Data register. **Note**: Only System I/O space and System Memory space are valid for values for Address_Space_ID.
UID	4	60	Unique ID–Same as the value returned by the _UID under the device in the namespace that represents this embedded controller.
GPE_BIT	1	64	The bit assignment of the SCI interrupt within the GPEx_STS register of a GPE block described in the FADT that the embedded controller triggers.
EC_ID	Variable	65	ASCII, null terminated, string that contains a fully qualified reference to the name space object that is this embedded controller device (for example, "\\_SB.PCI0.ISA.EC"). Quotes are omitted in the data field.

ACPI 2.0 OSPM implementations supporting Embedded Controller devices must also support the ECDT. ACPI 1.0 OSPM implementation will not recognize or make use of the ECDT. The following example code shows how to detect whether the Embedded Controller operation regions are available in a manner that is backward compatible with prior versions of ACPI/OSPM.

```
Device(EC0) {
        Name(REGC,Ones)
        Method(_REG,2) {
                If(Lequal(Arg0, 3)) {
                        Store(Arg1, REGC)
        }
    }
        Method(ECAV,0) {
            If(Lequal(REGC,Ones)) {
                If(LgreaterEqual(_REV,2)) {
                                Return(One)
                        }
                Else {
                                Return(Zero)
                        }
            }
            Return(REGC)
        }
}
```

To detect the availability of the region, call the ECAV method. For example:

```
If (\_SB.PCI0.EC0.ECAV()) {
    ...regions are available...
}
else {
    ...regions are not available...
}
```

5.3 ACPI NameSpace

For all Definition Blocks, the system maintains a single hierarchical namespace that it uses to refer to objects. All Definition Blocks load into the same namespace. Although this allows one Definition Block to reference objects and data from another (thus enabling interaction), it also means that OEMs must take care to avoid any naming collisions[6]. Only an unload operation of a Definition Block can remove names from the namespace, so a name collision in an attempt to load a Definition Block is considered fatal. The contents of the namespace changes only on a load or unload operation.

The namespace is hierarchical in nature, with each name allowing a collection of names “below” it. The following naming conventions apply to all names:

- All names are a fixed 32 bits.
- The first byte of a name is inclusive of: ‘A’–‘Z’, ‘_’, (0x41–0x5A, 0x5F).
- The remaining three bytes of a name are inclusive of: ‘A’–‘Z’, ‘0’–‘9’, ‘_’, (0x41–0x5A, 0x30–0x39, 0x5F).
- By convention, when an ASL compiler pads a name shorter than 4 characters, it is done so with trailing underscores (‘_’). See the language definition for AML NameSeg in Section 16, “ACPI Source Language Reference.”
- Names beginning with ‘_’ are reserved by this specification. Definition Blocks can only use names beginning with ‘_’ as defined by this specification.
- A name proceeded with ‘\’ causes the name to refer to the root of the namespace (‘\’ is not part of the 32-bit fixed-length name).
- A name proceeded with ‘^’ causes the name to refer to the parent of the current namespace (‘^’ is not part of the 32-bit fixed-length name).

[6] For the most part, since the name space is hierarchical, typically the bulk of a dynamic definition file will load into a different part of the hierarchy. The root of the name space and certain locations where interaction is being designed are the areas in which extra care must be taken.

Except for names preceded with a '\', the current namespace determines where in the namespace hierarchy a name being created goes and where a name being referenced is found. A name is located by finding the matching name in the current namespace, and then in the parent namespace. If the parent namespace does not contain the name, the search continues recursively upwards until either the name is found or the namespace does not have a parent (the root of the namespace). This indicates that the name is not found[7]. An attempt to access names in the parent of the root will result in the name not being found.

There are two types of namespace paths: an absolute namespace path (that is, one that starts with a '\' prefix), and a relative namespace path (that is, one that is relative to the current namespace). The namespace search rules discussed above, only apply to single NameSeg paths, which is a relative namespace path. For those relative name paths that contain multiple NameSegs or Parent Prefixes, '^', the search rules do not apply. If the search rules do not apply to a relative namespace path, the namespace object is looked up relative to the current namespace. For example:

ABCD	//search rules apply
^ABCD	//search rules don't apply
XYZ.ABCD	//search rules don't apply
\XYZ.ABCD	//search rules don't apply

All name references use a 32-bit fixed-length name or use a Name Extension prefix to concatenate multiple 32-bit fixed-length name components together. This is useful for referring to the name of an object, such as a control method, that is not in the scope of the current namespace.

[7] Unless the operation being performed is explicitly prepared for failure in name resolution, this is considered an error and may cause the system to stop working.

Figure 5-5 shows a sample of the ACPI namespace after a Differentiated Definition Block has been loaded.

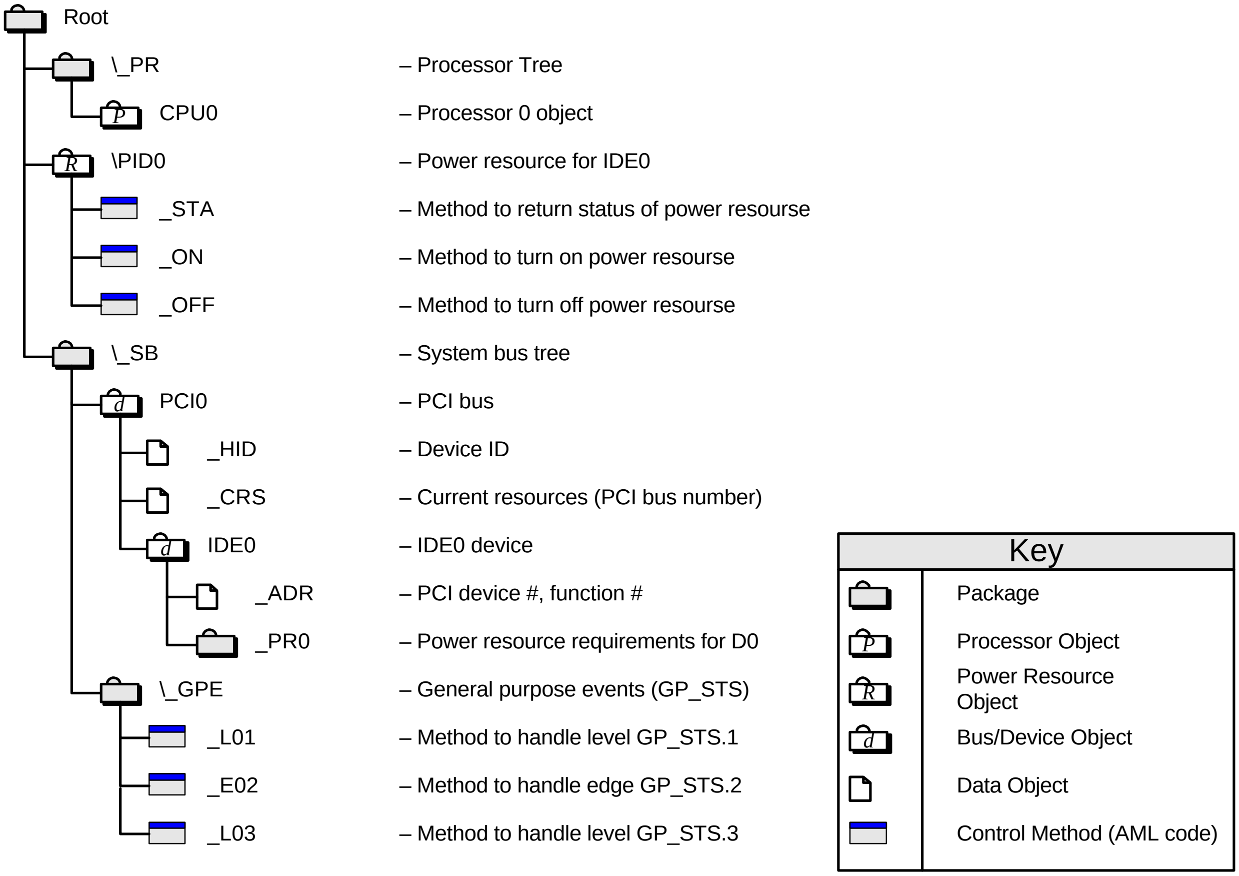
Root
_PR – Processor Tree
P CPU0 – Processor 0 object
R \PID0 – Power resource for IDE0
_STA – Method to return status of power resourse
_ON – Method to turn on power resourse
_OFF – Method to turn off power resourse
_SB – System bus tree
d PCI0 – PCI bus
_HID – Device ID
_CRS – Current resources (PCI bus number)
d IDE0 – IDE0 device
_ADR – PCI device #, function #
_PR0 – Power resource requirements for D0
_GPE – General purpose events (GP_STS)
_L01 – Method to handle level GP_STS.1
_E02 – Method to handle edge GP_STS.2
_L03 – Method to handle level GP_STS.3
Key
Package
P Processor Object
R Power Resource Object
d Bus/Device Object
Data Object
Control Method (AML code)

Figure 5-5 Example ACPI NameSpace

Care must be taken when accessing namespace objects using a relative single segment name because of the namespace search rules. An attempt to access a relative object recurses toward the root until the object is found or the root is encountered. This can cause unintentional results. For example, using the namespace described in Figure 5.5, attempting to access a _CRS named object from within the _SB_.PCI0.IDE0 will have different results depending on if an absolute or relative path name is used. If an absolute pathname is specified (_SB_.PCI0.IDE0._CRS) an error will result since the object does not exist. Access using a single segment name (_CRS) will actually access the _SB_.PCI0._CRS object. Notice that the access will occur successfully with no errors.

5.3.1 Defined Root Namespaces

The following namespaces are defined under the namespace root.

Table 5-30 Namespaces Defined Under the Namespace Root

Name	Description
_GPE	General events in GPE register block.
_PR	ACPI 1.0 Processor Namespace. ACPI 1.0 requires all Processor objects to be defined under this namespace. ACPI 2.0 allows Processor object definitions under the _SB namespace. ACPI 2.0-compatible systems may maintain the _PR namespace for compatibility with ACPI 1.0 operating systems. An ACPI 2.0-compatible namespace may define Processor objects in either the _SB or _PR scope but not both. For more information about defining Processor objects, see section 8, “Processor Control.”
_SB	All Device/Bus Objects are defined under this namespace.
_SI	System indicator objects are defined under this namespace. For more information about defining system indicators, see section 10.1, _S1 System Indicators.”
_TZ	ACPI 1.0 Thermal Zone namespace. ACPI 1.0 requires all Thermal Zone objects to be defined under this namespace. ACPI 2.0 allows Thermal Zone object definitions under the _SB namespace. ACPI 2.0-compatible systems may maintain the _TZ namespace for compatibility with ACPI 1.0 operating systems. An ACPI 2.0-compatible namespace may define Thermal Zone objects in either the _SB or _TZ scope but not both. For more information about defining Thermal Zone objects, see section 12, “Thermal Management.”

5.3.2 Objects

All objects, except locals, have a global scope. Local data objects have a per-invocation scope and lifetime and are used to process the current invocation from beginning to end.

The contents of objects vary greatly. Nevertheless, most objects refer to data variables of any supported data type, a control method, or system software-provided functions.

5.4 Definition Block Encoding

This section specifies the encoding used in a Definition Block to define names (load time only), objects, and packages. The Definition Block is encoded as a stream from beginning to end. The lead byte in the stream comes from the AML encoding tables shown in section 16, “ACPI Source Language Reference,” and signifies how to interpret some number of following bytes, where each following byte can in turn signify how to interpret some number of following bytes. For a full specification of the AML encoding, see section 16, “ACPI Source Language Reference.”

Within the stream there are two levels of data being defined. One is the packaging and object declarations (load time), and the other is an object reference (package contents/run-time).

All encodings are such that the lead byte of an encoding signifies the type of declaration or reference being made. The type either has an implicit or explicit length in the stream. All explicit length declarations take the form shown below, where *PkgLength* is the length of the inclusive length of the data for the operation.

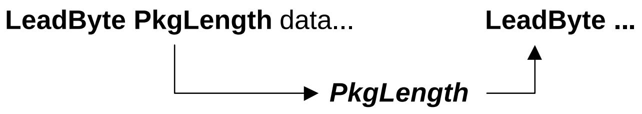

Figure 5-6 AML Encoding

Encodings of implicit length objects either have fixed length encodings or allow for nested encodings that, at some point, either result in an explicit or implicit fixed length.

The *PkgLength* is encoded as a series of 1 to 4 bytes in the stream with the most significant two bits of byte zero, indicating how many following bytes are in the *PkgLength* encoding. The next two bits are only used in one-byte encodings, which allows for one-byte encodings on a length up to 0x3F. Longer encodings, which do not use these two bits, have a maximum length of the following: two-byte encodings of 0x0FFF, three-byte encodings of 0x0FFFFF, and four-byte length encodings of 0x0FFFFFFFFF.

It is fatal for a package length to not fall on a logical boundary. For example, if a package is contained in another package, then by definition its length must be contained within the outer package, and similarly for a datum of implicit length.

At some point, the system software decides to “load” a Definition Block. Loading is accomplished when the system makes a pass over the data and populates the ACPI namespace and initializes objects accordingly. The namespace for which population occurs is either from the *current namespace location*, as defined by all nested packages or from the root if the name is preceded with ‘\’.

The first object present in a Definition Block must be a named control method. This is the Definition Block’s initialization control.

Packages are objects that contain an ordered reference to one or more objects. A package can also be considered a vertex of an array, and any object contained within a package can be another package. This permits multidimensional arrays of fixed or dynamic depths and vertices.

Unnamed objects are used to populate the contents of named objects. Unnamed objects cannot be created in the “root.” Unnamed objects can be used as arguments in control methods.

Control method execution may generate errors when creating objects. This can occur if a Method that creates named objects blocks and is reentered while blocked. This will happen because all named objects have an absolute path. This is true even if the object name specified is relative. For example, the following ASL code segments are functionally identical.

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

Notice that in the above example the execution of the DEAD method will always fail because the object _SB_.FOO.BAR is created at load time.

5.5 Using the ACPI Control Method Source Language

OEMs and BIOS vendors write definition blocks using the ACPI Control Method Source language (ASL) and use a translator to produce the byte stream encoding described in section 5.4. For example, the ASL statements that produce the example byte stream shown in that earlier section are shown in the following ASL example. For a full specification of the ASL statements, see section 16, “ACPI Source Language Reference.”

```
// ASL Example
DefinitionBlock (
    "forbook.aml",      // Output Filename
    "DSDT",             // Signature
    0x02,               // DSDT Compliance Revision
    "OEM",              // OEMID
    "forbook",          // TABLE ID
    0x1000              // OEM Revision
    )
{   // start of definition block
    OperationRegion(\GIO, SystemIO, 0x125, 0x1)
    Field(\GIO, ByteAcc, NoLock, Preserve)   {
            CT01,   1,
    }

    Scope(\_SB){   // start of scope
        Device(PCI0)   {   // start of device
            PowerResource(FET0, 0, 0) {       // start of pwr
                Method(_ON){
                    Store (Ones, CT01)        // assert power
                    Sleep (30)                // wait 30ms
                }
                Method(_OFF) {
                    Store (Zero, CT01)        // assert reset#
                }
                Method(_STA) {
                    Return (CT01)
                }
            } // end of pwr
        } // end of device
    } // end of scope
} // end of definition block
```

5.5.1 ASL Statements

ASL is principally a declarative language. ASL statements declare objects. Each object has three parts, two of which can be null:

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

FixedList refers to a list of known length that supplies data that all instances of a given *ObjectType* must have. It is written as (a, b, c,), where the number of arguments depends on the specific ObjectType, and some elements can be nested objects, that is (a, b, (q, r, s, t), d). Arguments to a *FixedList* can have default values, in which case they can be skipped. Some ObjectTypes can have a null *FixedList*.

VariableList refers to a list, not of predetermined length, of child objects that help define the parent. It is written as {x, y, z, aa, bb, cc}, where any argument can be a nested object. ObjectType determines what terms are legal elements of the *VariableList*. Some ObjectTypes can have a null variable list.

For a detailed specification of the ASL language, see section 16, “ACPI Source Language Reference.” For a detailed specification of the ACPI Control Method Machine Language (AML), upon which the output of the ASL translator is based, see section 17, “ACPI Machine Language Specification.”

5.5.2 ASL Macros

The ASL compiler supports some built in macros to assist in various ASL coding operations. The following table lists the supported directives and an explanation of their function.

Table 5-31 ASL Built-in Macros

ASL Statement	Description
Offset (*a*)	Used in a *FieldLis*t parameter to supply the byte offset of the next defined field within its parent region. This can be used instead of defining the bit lengths that need to be skipped. All offsets are defined from beginning to end of a region.
EISAID (*Id*)	Macro that converts the 7-character text argument into its corresponding 4-byte numeric EISA ID encoding. This can be used when declaring IDs for devices that are EISA IDs.
ResourceTemplate ()	Macro used to supply Plug and Play resource descriptor information in human readable form, which is then translated into the appropriate binary Plug and Play resource descriptor encodings. For more information about resource descriptor encodings, see section 6.4, "Resource Data Types for ACPI."
Unicode (string)	Macro that converts an ASCII string to a Unicode string contained in a buffer.

5.5.3 Control Method Execution

The operating software will initiate well-defined control methods as necessary to either interrogate or adjust system-level hardware state. This is called an invocation.

A control method can use other internal, or well defined, control methods to accomplish the task at hand, which can include defined control methods provided by the operating software. Interpretation of a Control Method is not preemptive, but it can block. When a control method does block, the operating software can initiate or continue the execution of a different control method. A control method can only assume that access to global objects is exclusive for any period the control method does not block.

Global objects are those NameSpace objects created at table load time.

5.5.3.1 Control Methods, Objects, and Operation Regions

Control Methods can reference any objects anywhere in the Namespace as well as address spaces defined in operation regions. Control methods must have exclusive access to the any address accessed via OpRegions. Control methods do not directly access any other hardware registers, including the ACPI-defined register blocks. Some of the ACPI registers, in the defined ACPI registers blocks, are maintained on behalf of control method execution. For example, the GPEx_BLK is not directly accessed by a control method but is used to provide an extensible interrupt handling model for control method invocation.

Note: Accessing an OpRegion may block, even if the OpRegion is not protected by a mutex. For example, because of the slow nature of embedded controller, embedded controller OpRegion field access may block.

5.5.4 Control Method Arguments, Local Variables, and Return Values

Control methods can be passed up to seven arguments. Each argument is an object, and could in turn be a “package” style object that refers to other objects. Access to the argument objects is via the ASL ArgTerm language elements. The number of arguments passed to any control method is fixed and is defined when the control method package is created.

Control methods can access up to eight local data objects. Access to the local data objects have shorthand encodings. On initial control method execution, the local data objects are NULL. Access to local objects is via the ASL LocalTerm language elements.

Upon control method execution completion, one object can be returned that can be used as the result of the execution of the method. The “caller” must either use the result or save it to a different object if it wants to preserve it. See the description of the Return ASL operator for additional details.
NameSpace objects created within the scope of a method are dynamic. They exist only for the duration of the method execution. They are created when specified by the code and are destroyed on exit. A method may create dynamic objects outside of the current scope in the NameSpace using the scope operator or using full path names. These objects will still be destroyed on method exit. Objects created at load time outside of the scope of the method are static. For example:

```
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
    } // end method
} // end scope
```

The object \XYZ.BAR is a static object created when the table that contains the above ASL is loaded. The object \XYZ.FOO.BAR is a dynamic object that is created when the `Name (BAR, 7)` statement in the FOO method is executed. The object \XYZ.FOOB is a dynamic object created by the \XYZ.FOO method when the `Name (\XYZ.FOOB, 3)` statement is executed. Notice that the \XYZ.FOOB object is destroyed after the \XYZ.FOO method exits.

5.6 ACPI Event Programming Model

The ACPI event programming model is based on the SCI interrupt and General-Purpose Event (GPE) register. ACPI provides an extensible method to raise and handle the SCI interrupt, as described in this section.

5.6.1 ACPI Event Programming Model Components

The components of the ACPI event programming model are the following:

- OSPM
- FADT
- PM1a_STS, PM1b_STS and PM1a_EN, PM1b_EN fixed register blocks
- GPE0_BLK and GPE1_BLK register blocks
- GPE register blocks defined in GPE block devices
- SCI interrupt
- ACPI AML code general-purpose event model
- ACPI device-specific model events
- ACPI Embedded Controller event model

The role of each component in the ACPI event programming model is described in the following table.

Table 5-32 ACPI Event Programming Model Components

Component	Description
OSPM	Receives all SCI interrupts raised (receives all SCI events). Either handles the event or masks the event off and later invokes an OEM-provided control method to handle the event. Events handled directly by OSPM are fixed ACPI events; interrupts handled by control methods are general-purpose events.
FADT	Specifies the base address for the following fixed register blocks on an ACPI-compatible platform: PM1x_STS and PM1x_EN fixed registers and the GPEx_STS and GPEx_EN fixed registers.
PM1*x*_STS and PM1*x*_EN fixed registers	PM1x_STS bits raise fixed ACPI events. While a PM1x_STS bit is set, if the matching PM1x_EN bit is set, the ACPI SCI event is raised.
GPE*x*_STS and GPE*x*_EN fixed registers	GPEx_STS bits that raise general-purpose events. For every event bit implemented in GPEx_STS, there must be a comparable bit in GPEx_EN. Up to 256 GPEx_STS bits and matching GPEx_EN bits can be implemented. While a GPEx_STS bit is set, if the matching GPEx_EN bit is set, then the general-purpose SCI event is raised.
SCI interrupt	A level-sensitive, shareable interrupt mapped to a declared interrupt vector. The SCI interrupt vector can be shared with other low-priority interrupts that have a low frequency of occurrence.
ACPI AML code general-purpose event model	A model that allows OEM AML code to use GPEx_STS events. This includes using GPEx_STS events as “wake” sources as well as other general service events defined by the OEM (“button pressed,” “thermal event,” “device present/not present changed,” and so on).
ACPI device-specific model events	Devices in the ACPI namespace that have ACPI-specific device IDs can provide additional event model functionality. In particular, the ACPI embedded controller device provides a generic event model.
ACPI Embedded Controller event model	A model that allows OEM AML code to use the response from the Embedded Controller Query command to provide general-service event defined by the OEM.

5.6.2 Types of ACPI Events

At the direct ACPI hardware level, two types of events can be signaled by an SCI interrupt:

- Fixed ACPI events
- General-purpose events

In turn, the general-purpose events can be used to provide further levels of events to the system. And, as in the case of the embedded controller, a well-defined second-level event dispatching is defined to make a third type of typical ACPI event. For the flexibility common in today's designs, two first-level general-purpose event blocks are defined, and the embedded controller construct allows a large number of embedded controller second-level event-dispatching tables to be supported. Then if needed, the OEM can also build additional levels of event dispatching by using AML code on a general-purpose event to sub-dispatch in an OEM defined manner.

5.6.2.1 Fixed ACPI Event Handling

When OSPM receives a fixed ACPI event, it directly reads and handles the event registers itself. The following table lists the fixed ACPI events. For a detailed specification of each event, see section 4, "ACPI Hardware Specification."

Table 5-33 Fixed ACPI Events

Event	Comment
Power management timer carry bit set.	For more information, see the description of the TMR_STS and TMR_EN bits of the PM1x fixed register block in section 4.7.3.1, "PM1 Event Grouping," as well as the TMR_VAL register in the PM_TMR_BLK in section 4.7.3.3, "Power Management Timer."
Power button signal	A power button can be supplied in two ways. One way is to simply use the fixed status bit, and the other uses the declaration of an ACPI power device and AML code to determine the event. For more information about the alternate-device based power button, see section 4.7.2.2.1.2, Control Method Power Button." Notice that during the S0 state, both the power and sleep buttons merely notify OSPM that they were pressed. If the system does not have a sleep button, it is recommended that OSPM use the power button to initiate sleep operations as requested by the user.
Sleep button signal	A sleep button can be supplied in one of two ways. One way is to simply use the fixed status button. The other way requires the declaration of an ACPI sleep button device and AML code to determine the event.
RTC alarm	ACPI-defines an RTC wake alarm function with a minimum of one-month granularity. The ACPI status bit for the device is optional. If the ACPI status bit is not present, the RTC status can be used to determine when an alarm has occurred. For more information, see the description of the RTC_STS and RTC_EN bits of the PM1x fixed register block in section 4.7.3.1, "PM1 Event Grouping."
Wake status	The wake status bit is used to determine when the sleeping state has been completed. For more information, see the description of the WAK_STS and WAK_EN bits of the PM1x fixed register block in section 4.7.3.1, "PM1 Event Grouping."

Table 5-33 Fixed ACPI Events *(continued)*

Event	Comment
System bus master request	The bus-master status bit provides feedback from the hardware as to when a bus master cycle has occurred. This is necessary for supporting the processor C3 power savings state. For more information, see the description of the BM_STS bit of the PM1x fixed register block in section 4.7.3.1, “PM1 Event Grouping.”
Global release status	This status is raised as a result of the Global Lock protocol, and is handled by OSPM as part of Global Lock synchronization. For more information, see the description of the GBL_STS bit of the PM1x fixed register block in section 4.7.3.1, “PM1 Event Grouping.” For more information on Global Lock, see section 5.2.9.1, “Global Lock.”

5.6.2.2 General-Purpose Event Handling

When OSPM receives a general-purpose event, it either passes control to an ACPI-aware driver, or uses an OEM-supplied control method to handle the event. An OEM can implement up to 128 general-purpose event inputs in hardware per GPE block, each as either a level or edge event. It is also possible to implement a single 256-pin block as long as it’s the only block defined in the system.

An example of a general-purpose event is specified in section 4, “ACPI Hardware Specification,” where EC_STS and EC_EN bits are defined to enable OSPM to communicate with an ACPI-aware embedded controller device driver. The EC_STS bit is set when either an interface in the embedded controller space has generated an interrupt or the embedded controller interface needs servicing. Notice that if a platform uses an embedded controller in the ACPI environment, then the embedded controller’s SCI output must be directly and exclusively tied to a single GPE input bit.

Hardware can cascade other general-purpose events from a bit in the GPEx_BLK through status and enable bits in Operational Regions (I/O space, memory space, PCI configuration space, or embedded controller space). For more information, see the specification of the General-Purpose Event Blocks (GPEx_BLK) in section 4.7.4.1, “General-Purpose Event Register Blocks.”

OSPM manages the bits in the GPEx blocks directly, although the source to those events is not directly known and is connected into the system by control methods. When OSPM receives a general-purpose event (the event is from a GPEx_BLK STS bit), OSPM does the following:

1. Disables the interrupt source (GPEx_BLK EN bit).
2. If an edge event, clears the status bit.
3. Performs one of the following:
    - Dispatches to an ACPI-aware device driver.
    - Queues the matching control method for execution.
    - Manages a wake event using device _PRW objects.
4. If a level event, clears the status bit.
5. Enables the interrupt source.

The OEM AML code can perform OEM-specific functions custom to each event the particular platform might generate by executing a control method that matches the event. For GPE events, OSPM will execute the control method of the name _GPE._*TXX* where *XX* is the hex value format of the event that needs to be handled and *T* indicates the event handling type (*T* must be either ‘E’ for an *edge* event or ‘L’ for a *level* event). The event values for status bits in GPE0_BLK start at zero (_*T*00) and end at the (GPE0_BLK_LEN / 2) **-** 1. The event values for status bits in GPE1_BLK start at GPE1_BASE and end at GPE1_BASE + (GPE1_BLK_LEN / 2) **-** 1. GPE0_BLK_LEN, GPE1_BASE, and GPE1_BLK_LEN are all defined in the FADT.

For OSPM to manage the bits in the GPEx_BLK blocks directly:

- Enable bits must be read/write.
- Status bits must be latching.
- Status bits must be read/clear, and cleared by writing a “1” to the status bit.

5.6.2.2.1 Wake Events

An important use of the general-purpose events is to implement device wake events. The components of the ACPI event programming model interact in the following way:

When a device asserts its wake signal, the general-purpose status event bit used to track that device is set.

While the corresponding general-purpose enable bit is enabled, the SCI interrupt is asserted.

If the system is sleeping, this will cause the hardware, if possible, to transition the system into the S0 state.

Once the system is running, OSPM will dispatch the corresponding GPE handler.

The handler needs to determine which device object has signaled wake and performs a wake Notify command on the corresponding device object(s) that have asserted wake.

In turn OSPM will notify OSPM native driver(s) for each device that will wake its device to service it.

Events that wake may not be intermixed with non-wake events on the same GPE input. Also, all wake events not exclusively tied to a GPE input (for example, one input is shared for multiple wake events) need to have individual enable and status bits in order to properly handle the semantics used by the system.

5.6.2.2.2 Dispatching to an ACPI-Aware Device Driver

Certain device support, such as an embedded controller, requires a dedicated GPE to service the device. Such GPEs are dispatched to native OS code to be handled and not to the corresponding GPE-specific control method.

In the case of the embedded controller, an OS-native, ACPI-aware driver is given the GPE event for its device. This driver services the embedded controller device and determines when events are to be reported by the embedded controller by using the Query command. When an embedded controller event occurs, the ACPI-aware driver dispatches the requests to other ACPI-aware drivers that have registered to handle the embedded controller queries or queues control methods to handle each event. If there is no device driver to handle specific queries, OEM AML code can perform OEM-specific functions that are customized to each event on the particular platform by including specific control methods in the namespace to handle these events. For an embedded controller event, OSPM will queue the control method of the name _Q*XX*, where *XX* is the hex format of the query code. Notice that each embedded controller device can have query event control methods.

Similarly, for an SMBus driver, if no driver registers for SMBus alarms, the SMBus driver will queue control methods to handle these. Methods must be placed under the SMBus device with the name _QXX where XX is the hex format of the SMBus address of the device sending the alarm.

5.6.2.2.3 Queuing the Matching Control Method for Execution

When a general-purpose event is raised, OSPM uses a naming convention to determine which control method to queue for execution and how the GPE EOI is to be handled. The GPEx_STS bits in the GPEx_BLK are indexed with a number from 0 through FF. The name of the control method to queue for an event raised from an enable status bit is always of the form _GPE._*Txx* where *xx* is the event value and *T* indicates the event EOI protocol to use (either edge or level). The event values for status bits in GPE0_BLK start at zero (_*T*00), end at the (GPE0_BLK_LEN / 2) - 1, and correspond to each status bit index within GPE0_BLK. The event values for status bits in GPE1_BLK are offset by GPE_BASE and therefore start at GPE1_BASE and end at GPE1_BASE + (GPE1_BLK_LEN / 2) - 1.

For example, suppose an OEM supplies a wake event for a communications port and uses bit 4 of the GPE0_STS bits to raise the wake event status. In an OEM-provided Definition Block, there must be a Method declaration that uses the name _GPE._L04 or \GPE._E04 to handle the event. An example of a control method declaration using such a name is the following:

```
Method (\_GPE._L04) {         // GPE 4 level wake handler
    Notify (\_SB.PCIO.COM0, 2)
}
```

The control method performs whatever action is appropriate for the event it handles. For example, if the event means that a device has appeared in a slot, the control method might acknowledge the event to some other hardware register and signal a change notify request of the appropriate device object. Or, the cause of the general-purpose event can result from more then one source, in which case the control method for that event determines the source and takes the appropriate action.

When a general-purpose event is raised from the GPE bit tied to an embedded controller, the embedded controller driver uses another naming convention defined by ACPI for the embedded controller driver to determine which control method to queue for execution. The queries that the embedded controller driver exchanges with the embedded controller are numbered from 0 through FF, yielding event codes 01 through FF. (A query response of 0 from the embedded controller is reserved for "no outstanding events.") The name of the control method to queue is always of the form _Q*xx* where *xx* is the number of the query acknowledged by the embedded controller. An example declaration for a control method that handles an embedded controller query is the following:

```
Method(_Q34) {        // embedded controller event for thermal
    Notify (\_SB.TZ0.THM1, 0x80)
}
```

When an SMBus alarm is handled by the SMBus driver, the SMBus driver uses a similar naming convention defined by ACPI for the driver to determine the control method to queue for execution. When an alarm is received by the SMBus host controller, it generally receives the SMBus address of the device issuing the alarm and one word of data. On implementations that use SMBALERT# for notifications, only the device address will be received. The name of the control method to queue is always of the form _Q*xx* where *xx* is the SMBus address of the device that issued the alarm. The SMBus address is 7 bits long corresponding to hex values 0 through 7F, although some addresses are reserved and will not be used. The control method will always be queued with one argument that contains the word of data received with the alarm. An exception is the case of an SMBus using SMBALERT# for notifications, in this case the argument will be 0. An example declaration for a control method that handles a SMBus alarm follows:

```
Method(_Q18, 1) {      // Thermal sensor device at address 0011 000

    // Arg0 contains notification value (if any)
    // Arg0 = 0 if devuice supports only SMBALERT#

    Notify (\_SB.TZ0.THM1, 0x80)
}
```

5.6.2.2.4 Managing a Wake Event Using Device _PRW Objects

A device's _PRW object provides the zero-based bit index into the general-purpose status register block to indicate which general-purpose status bit from either GPE0_BLK or GPE1_BLK is used as the specific device's wake mask. Although the hardware must maintain individual device wake enable bits, the system can have multiple devices using the same general-purpose event bit by using OEM-specific hardware to provide second-level status and enable bits. In this case, the OEM AML code is responsible for the second-level enable and status bits.

OSPM enables or disables the device wake function by enabling or disabling its corresponding GPE and by executing its _PSW control method (which is used to take care of the second-level enables). When the GPE is asserted, OSPM still executes the corresponding GPE control method that determines which device wakes are asserted and notifies the corresponding device objects. The native OS driver is then notified that its device has asserted wake, for which the driver powers on its device to service it.

If the system is in a sleeping state when the enabled GPE bit is asserted the hardware will transition the system into the S0 state, if possible.

5.6.3 Device Object Notifications

Some objects need to notify the OSPM of various object-related events. All such notifications are accomplished using the Notify operator, which supplies the ACPI object and a notification value that signifies the type of notification being performed. Notification values from 0 through 0x7F are common across any device object type. Notification values of 0x80 and above are device-specific and defined by each such device. For more information on the Notify operator, see section 16.2.3.4.1.9, “Notify (Notify).”

Table 5-34 Device Object Notification Types

Value	Description
0	**Bus Check.** This notification is performed on a device object to indicate to OSPM that it needs to perform the Plug and Play re-enumeration operation on the device tree starting from the point where it has been notified. OSPM will only perform this operation at boot, and when notified. It is the responsibility of the ACPI AML code to notify OSPM at any other times that this operation is required. The more accurately and closer to the actual device tree change the notification can be done, the more efficient the operating system’s response will be; however, it can also be an issue when a device change cannot be confirmed. For example, if the hardware cannot notice a device change for a particular location during a system sleeping state, it issues a Bus Check notification on wake to inform OSPM that it needs to check the configuration for a device change.
1	**Device Check.** Used to notify OSPM that the device either appeared or disappeared. If the device has appeared, OSPM will re-enumerate from the parent. If the device has disappeared, OSPM will invalidate the state of the device. OSPM may optimize out re-enumeration. If _DCK is present, then Notify(*object*,1) is assumed to indicate an undock request.
2	**Device Wake.** Used to notify OSPM that the device has signaled its wake event, and that OSPM needs to notify OSPM native device driver for the device. This is only used for devices that support _PRW.
3	**Eject Request.** Used to notify OSPM that the device should be ejected, and that OSPM needs to perform the Plug and Play ejection operation. OSPM will run the _EJx method.
4	**Device Check Light.** Used to notify OSPM that the device either appeared or disappeared. If the device has appeared, OSPM will re-enumerate from the device itself, not the parent. If the device has disappeared, OSPM will invalidate the state of the device.
5	**Frequency Mismatch.** Used to notify OSPM that a device inserted into a slot cannot be attached to the bus because the device cannot be operated at the current frequency of the bus. For example, this would be used if a user tried to hot-plug a 33 MHz PCI device into a slot that was on a bus running at greater than 33 MHz.
6	**Bus Mode Mismatch.** Used to notify OSPM that a device has been inserted into a slot or bay that cannot support the device in its current mode of operation. For example, this would be used if a user tried to hot-plug a PCI device into a slot that was on a bus running in PCI-X mode.
7	**Power Fault.** Used to notify OSPM that a device cannot be moved out of the D3 state because of a power fault.
8-7F	**Reserved.**

Below are the notification values defined for specific ACPI devices. For more information concerning the object-specific notification, see the section on the corresponding device/object.

Table 5-35 Control Method Battery Device Notification Values

Hex value	Description
80	**Battery Status Changed.** Used to notify OSPM that the Control Method Battery device status has changed.
81	**Battery Information Changed.** Used to notify OSPM that the Control Method Battery device information has changed. This only occurs when a battery is replaced.
>81	**Reserved.**

Table 5-36 Power Source Object Notification Values

Hex value	Description
80	**Power Source Status Changed**. Used to notify OSPM that the power source status has changed.
>80	**Reserved.**

Table 5-37 Thermal Zone Object Notification Values

Hex value	Description
80	**Thermal Zone Status Changed.** Used to notify OSPM that the thermal zone temperature has changed.
81	**Thermal Zone Trip points Changed.** Used to notify OSPM that the thermal zone trip points have changed.
82	**Device Lists Changed.** Used to notify OSPM that the thermal zone device lists (_ALx, _PSL, _TZD) have changed.
>82	**Reserved.**

Table 5-38 Control Method Power Button Notification Values

Hex value	Description
80	**S0 Power Button Pressed.** Used to notify OSPM that the power button has been pressed while the system is in the S0 state. Notice that when the button is pressed while the system is in the S1-S4 state, a Device Wake notification must be issued instead.
>80	**Reserved.**

Table 5-39 Control Method Sleep Button Notification Values

Hex value	Description
80	**S0 Sleep Button Pressed.** Used to notify OSPM that the sleep button has been pressed while the system is in the S0 state. Notice that when the button is pressed while the system is in the S1-S4 state, a Device Wake notification must be issued instead.
>80	**Reserved.**

Table 5-40 Control Method Lid Notification Values

Hex value	Description
80	**Lid Status Changed.** Used to notify OSPM that the control method lid device status has changed.
>80	**Reserved.**

Table 5-41 Processor Device Notification Values

Hex value	Description
80	**Performance Present Capabilities Changed**. Used to notify OSPM that the number of supported processor performance states has changed. This notification causes OSPM to re-evaluate the _PPC object. See section 8, “Processor Control,” for more information.
81	**C States Changed.** Used to notify OSPM that the number or type of supported processor C States has changed. This notification causes OSPM to re-evaluate the _CST object. See section 8, “Processor Control,” for more information.
>81	**Reserved.**

5.6.4 Device Class-Specific Objects

Most device objects are controlled through generic objects and control methods and they have generic device IDs. These generic objects, control methods, and device IDs are specified in sections 6, 7, 8, 10, 11, and 12. Section 5.6.5, “Defined Generic Objects and Control Methods,” lists all the generic objects and control methods defined in this specification.

However, certain integrated devices require support for some device-specific ACPI controls. This section lists these devices, along with the device-specific ACPI controls that can be provided.

Some of these controls are for ACPI-aware devices and as such have Plug and Play IDs that represent these devices. The following table lists the Plug and Play IDs defined by the ACPI specification.

Table 5-42 ACPI Device IDs

Plug and Play ID	Description
PNP0C08	**ACPI.** Not declared in ACPI as a device. This ID is used by OSPM for the hardware resources consumed by the ACPI fixed register spaces, and the operation regions used by AML code. It represents the core ACPI hardware itself.
PNP0A05	**Generic ISA Bus Device.** A bus only device whose bus settings are totally controlled by its ACPI resource information, and otherwise needs no bus-specific driver support.
PNP0A06	**Extended I/O Bus.** A special case of the PNP0A05 device, where the only difference is in the name of the device. There is no functional difference between the two IDs.
PNP0C09	**Embedded Controller Device.** A host embedded controller controlled through an ACPI-aware driver.
PNP0C0A	**Control Method Battery.** A device that solely implements the ACPI Control Method Battery functions. A device that has some other primary function would use its normal device ID. This ID is used when the devices primary function is that of a battery.
PNP0C0B	**Fan.** A device that causes cooling when “on” (D0 device state).

Table 5-42 ACPI Device IDs *(continued)*

Plug and Play ID	Description
PNP0C0C	**Power Button Device.** A device controlled through an ACPI-aware driver that provides power button functionality. This device is only needed if the power button is not supported using the fixed register space.
PNP0C0D	**Lid Device.** A device controlled through an ACPI-aware driver that provides lid status functionality. This device is only needed if the lid state is not supported using the fixed register space.
PNP0C0E	**Sleep Button Device.** A device controlled through an ACPI-aware driver that provides power button functionality. This device is optional.
PNP0C0F	**PCI Interrupt Link Device.** A device that allocates an interrupt connected to a PCI interrupt pin. See section 6., “Configuration,” for more details.
PNP0C80	**Memory Device.** This device is a memory subsystem.
ACPI0001	**SMBus 1.0 Host Controller.** An SMBus host controller (SMB-HC) compatible with the embedded controller-based SMB-HC interface (as specified in section 13.9, “SMBus Host Controller Interface via Embedded Controller”) and implementing the SMBus 1.0 Specification.
ACPI0002	**Smart Battery Subsystem.** The Smart battery Subsystem specified in section 11, “Power Source Devices.”
ACPI0003	**AC Device.** The AC adapter specified in section 11, “Power Source Devices.”
ACPI0004	**Module Device.** This device is a container object that acts as a bus node in a namespace.
ACPI0005	**SMBus 2.0 Host Controller.** An SMBus host controller (SMB-HC compatible with the embedded controller-based SMB-HC interface (as specified in section 13.9, “SMBus Host Controller Interface via Embedded Controller”) and implementing the SMBus 2.0 Specification.
ACPI0006	**GPE Block Device.** This device allows a system designer to describe GPE blocks beyond the two that are described in the FADT.

5.6.5 Defined Generic Objects and Control Methods

The following table lists all of the generic object and control methods defined in this specification and provides a reference to the defining section of the specification.

Table 5-43 Defined Generic Object and Control Methods

Object	Description	Reference
_ACx	Thermal Zone object that returns active cooling policy threshold values in tenths of degrees Kelvin.	12.3.1
_ADR	Device object that evaluates to a device’s address on its parent bus. For the display output device, this object returns a unique ID. (B.5.1, “_ADR - Return the Unique ID for this Device.”)	6.1.1
_AL*x*	Thermal zone object containing a list of cooling device objects.	12.3.2
_ALN	Resource data type reserved field name	16.2.4
_ASI	Resource data type reserved field name	16.2.4.16

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_BAS	Resource data type reserved field name	16.2.4
_BBN	PCI bus number setup by the BIOS	6.5.5
_BCL	Returns a buffer of bytes indicating list of brightness control levels supported.	B.5.2
_BCM	Sets the brightness level of the built-in display output device.	B.5.3
_BDN	Correlates a docking station between ACPI and legacy interfaces.	6.5.3
_BFS	Control method executed immediately following a wake event.	7.3.1
_BIF	Control Method Battery information object	11.2.2.1
_BM	Resource data type reserved field name	16.2.4
_BST	Control Method Battery status object	11.2.2.2
_BTP	Sets Control Method Battery trip point	11.2.2.3
_CID	Device identification object that evaluates to a device's Plug and Play Compatible ID list.	6.1.2
_CRS	Device configuration object that specifies a device's *current* resource settings, or a control method that generates such an object.	6.2.1
_CRT	Thermal zone object that returns critical trip point in tenths of degrees Kelvin.	12.3.3
_CST	Processor power state declaration object	8.3.2
_DCK	Indicates that the device is a docking station.	6.5.2
_DCS	Returns the status of the display output device.	B.5.5
_DDC	Returns the EDID for the display output device	B.5.4
_DDN	Object that associates a logical software name (for example, COM1) with a device.	6.1.3
_DEC	Resource data type reserved field name	16.2.4
_DGS	Control method used to query the state of the output device.	B.5.6
_DIS	Device configuration control method that disables a device.	6.2.2
_DMA	Object that specifies a device's *current* resources for DMA transactions.	6.2.3
_DOD	Control method used to enumerate devices attached to the display adapter.	B.4.2
_DOS	Control method used to enable/disable display output switching.	B.4.1
_DSS	Control method used to set display device state.	B.5.7
_Exx	Control method executed as a result of a general-purpose event.	5.6.2.2, 5.6.2.2.3
_EC	Control Method used to define the offset address and Query value of an SMB-HC defined within an embedded controller device.	13.12
_EDL	Device removal object that returns a packaged list of devices that are dependent on a device.	6.3.1
_EJx	Device insertion/removal control method that ejects a device.	6.3.3

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_EJD	Device removal object that evaluates to the name of a device object upon which a device is dependent. Whenever the named device is ejected, the dependent device must receive an ejection notification.	6.3.2
_FDE	Object that indicates the presence or absence of floppy disks.	10.9.1
_FDI	Object that returns floppy drive information.	10.9.2
_FDM	Control method that changes the mode of floppy drives.	10.9.3
_FIX	Object used to provide correlation between the fixed hardware register blocks defined in the FADT and the devices that implement these fixed hardware registers.	6.2.4
_GL	OS-defined Global Lock mutex object	5.7.1
_ GLK	Indicates the need to acquire the Global Lock, must be acquired when accessing the device.	6.5.7
_GPD	Control method that returns which VGA device will be posted at boot	B.4.4
_GPE	1. General-Purpose Events root name space 2. Object that returns the SCI interrupt within the GPx_STS register that is connected to the EC.	5.3.1 13.11
_GRA	Resource data type reserved field name.	16.2.4
_GTF	IDE device control method to get the Advanced Technology Attachement (ATA) task file needed to re-initialize the drive to bootup defaults.	10.8.1
_GTM	IDE device control method to get the IDE controller timing information.	10.8.2
_GTS	Control method executed just prior to setting the sleep enable (SLP_EN) bit.	7.3.3
_HE	Resource data type reserved field name	16.2.4
_HID	Device identification object that evaluates to a device's Plug and Play Hardware ID.	6.1.4
_HPP	An object that specifies the Cache-line size, Latency timer, SERR enable, and PERR enable values to be used when configuring a PCI device inserted into a hot-plug slot or initial configuration of a PCI device at system boot.	6.2.5
_INI	Device initialization method that performs device specific initialization.	6.5.1
_INT	Resource data type reserved field name	16.2.4
_IRC	Power management object that signifies the device has a significant inrush current draw.	7.2.11
_Lxx	Control method executed as a result of a general-purpose event.	5.6.2.2, 5.6.2.2.3
_LCK	Device insertion/removal control method that locks or unlocks a device.	6.3.4
_LEN	Resource data type reserved field name	16.2.4
_LID	Object that returns the status of the Lid on a mobile system.	10.3.1
_LL	Resource data type reserved field name	16.2.4

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_MAF	Resource data type reserved field name	16.2.4
_MAT	Object evaluates to a buffer of MADT APIC Structure entries.	6.2.6
_MAX	Resource data type reserved field name	16.2.4
_MEM	Resource data type reserved field name	16.2.4
_MIF	Resource data type reserved field name	16.2.4
_MIN	Resource data type reserved field name	16.2.4
_MSG	System indicator control that indicates messages are waiting.	10.1.2
_OFF	Power resource object that sets the resource off.	7.1.2
_ON	Power resource object that sets the resource on.	7.1.3
_OS	Object that evaluates to a string that identifies the operating system.	5.7.2
_PCL	Power source object that contains a list of devices powered by a power source.	11.3.2
_PCT	Processor performance control object	8.3.3.1
_PIC	Control method that conveys interrupt model in use to the system firmware.	5.8.1
_PPC	Control method used to determine number of performance states currently supported by the platform.	8.3.3.3
_PR	ACPI 1.0 Processor Namespace	5.3.1
_PR0	Power management object that evaluates to the device's power requirements in the D0 device state (device fully on).	7.2.6
_PR1	Power management object that evaluates to the device's power requirements in the D1 device state. Only devices that can achieve the defined D1 device state according to its given device class would supply this level.	7.2.7
_PR2	Power management object that evaluates to the device's power requirements in the D2 device state. Only devices that can achieve the defined D2 device state according to its given device class would supply this level.	7.2.8
_PRS	Device configuration object that specifies a device's *possible* resource settings, or a control method that generates such an object.	6.2.7
_PRT	An object that specifies the PCI interrupt Routing Table.	6.2.8
_PRW	Power management object that evaluates to the device's power requirements in order to wake the system from a system sleeping state.	7.2.9
_PS0	Power management control method that puts the device in the D0 device state. (device fully on).	7.2.1
_PS1	Power management control method that puts the device in the D1 device state.	7.2.2
_PS2	Power management control method that puts the device in the D2 device state.	7.2.3

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_PS3	Power management control method that puts the device in the D3 device state (device off).	7.2.4
_PSC	Power management object that evaluates to the device's current power state.	7.2.5
_PSL	Thermal zone object that returns list of passive cooling device objects.	12.3.4
_PSR	Power source object that returns present power source device.	11.3.1
_PSS	Object indicates the number of supported processor performance states.	8.3.3.2
_PSV	Thermal zone object that returns Passive trip point in tenths of degrees Kelvin.	12.3.5
_PSW	Power management control method that enables or disables the device's wake function.	7.2.10
_PTC	Object used to define a processor throttling control register.	8.3.1
_PTS	Control method used to prepare to sleep.	7.3.2
_PXM	Object used to describe proximity domains within a machine.	6.2.9
_Q*xx*	Embedded Controller Query and SMBus Alarm control method	5.6.2.2.3
_RBO	Resource data type reserved field name	16.2.4
_RBW	Resource data type reserved field name	16.2.4
_REG	Notifies AML code of a change in the availability of an operation region.	6.5.4
_REV	Revision of the ACPI specification that OSPM implements.	5.7.3
_RMV	Device insertion/removal object that indicates that the given device is removable.	6.3.5
_RNG	Resource data type reserved field name	16.2.4
_ROM	Control method used to get a copy of the display devices' ROM data.	B.4.3
_RW	Resource data type reserved field name	16.2.4
_S0	Power management package that defines system _S0 state mode.	7.3.4.1
_S1	Power management package that defines system _S1 state mode.	7.3.4.2
_S2	Power management package that defines system _S2 state mode.	7.3.4.3
_S3	Power management package that defines system _S3 state mode.	7.3.4.4
_S4	Power management package that defines system _S4 state mode.	7.3.4.5
_S5	Power management package that defines system _S5 state mode.	7.3.4.6
_S1D	Highest D-state supported by the device in the S1 state.	7.2.12
_S2D	Highest D-state supported by the device in the S2 state.	7.2.13
_S3D	Highest D-state supported by the device in the S3 state.	7.2.14
_S4D	Highest D-state supported by the device in the S4 state.	7.2.15
_SB	System bus scope	5.3.1

Table 5-43 Defined Generic Object and Control Methods *(continued)*

Object	Description	Reference
_SBS	Smart Battery object that returns Smart Battery configuration.	11.1.2
_SCP	Thermal zone object that sets user cooling policy (Active or Passive).	12.3.7
_SEG	Bus identification object that evaluates to a bus's segment number.	6.5.6
_SHR	Resource data type reserved field name	16.4.2
_SI	System indicators scope	5.3.1
_SIZ	Resource data type reserved field name	16.4.2
_SPD	Control method used to update which video device will be posted at boot.	B.4.5
_SRS	Device configuration control method that sets a device's settings.	6.2.10
_SST	System indicator control method that indicates the system status.	10.1.1
_STA	1. Device insertion/removal control method that returns a device's status. 2. Power resource object that evaluates to the current on or off state of the Power Resource.	6.3.6 7.1.4
_STM	IDE device control method used to set the IDE controller transfer timings.	10.8.3
_STR	Object evaluates to a Unicode string to describe a device.	6.1.5
_SUN	Object that evaluates to the slot unique ID number for a slot.	6.1.6
_T_x	Reserved for use by the ASL compiler.	16.2.1.1
_TC1	Thermal zone object that contains thermal constant for Passive cooling.	12.3.7
_TC2	Thermal zone object that contains thermal constant for Passive cooling.	12.3.8
_TMP	Thermal zone object that returns current temperature in tenths of degrees Kelvin.	12.3.9
_TRA	Resource data type reserved field name	16.4.2
_TRS	Resource data type reserved field name	16.4.2
_TSP	Thermal zone object that contains thermal sampling period for Passive cooling.	12.3.10
_TTP	Resource data type reserved field name	16.4.2
_TYP	Resource data type reserved field name	16.4.2
_TZ	ACPI 1.0 thermal zone scope	5.3.1
_TZD	Object evaluates to a package of device names associated with a Thermal Zone.	12.3.11
_TZP	Thermal zone polling frequency in tenths of seconds.	12.3.12
_UID	Device identification object that specifies a device's unique persistent ID, or a control method that generates it.	6.1.7
_VPO	Returns 32-bit integer indicating the video post options.	B.4.6
_WAK	Power management control method run once system is awakened.	7.3.5

5.7 Operating System-Defined Object Names

A list of OS supplied object names are shown in the following table.

Table 5-44 Predefined Global Events

Name	Description
_GL	Global Lock
_OS	Name of the operating system
_REV	Revision of the ACPI specification that OSPM implements.

5.7.1 _GL (Global Lock Mutex)

This object is a Mutex object that behaves like a Mutex as defined in section 16.2.3.3.1.13, “Mutex (Declare Synchronization/Mutex Object),” with the added behavior that acquiring this Mutex also acquires the shared environment Global Lock defined in section 5.2.11, “Global System Interrupts.” This allows Control Methods to explicitly synchronize with the Global Lock if necessary

5.7.2 _OS (OS Name Object)

This object evaluates to a string that identifies the operating system. In robust OSPM implementations, _OS evaluates differently for each OS release. This may allow AML code to accommodate differences in OSPM implementations. This value does not change with different revisions of the AML interpreter.

5.7.3 _REV (Revision Data Object)

This object evaluates to the revision of the ACPI Specification that the specified _OS implements as a DWORD. Larger values are newer revisions of the ACPI specification.

5.8 System Configuration Objects

5.8.1 _PIC Method

The _PIC optional method is to report to the BIOS the current interrupt model used by the OS. This control method returns nothing. The argument passed into the method signifies the interrupt model OSPM has chosen, PIC mode, APIC mode, or SAPIC mode. Notice that calling this method is optional for OSPM. If the method is never called, the BIOS must assume PIC mode. It is important that the BIOS save the value passed in by OSPM for later use during wake operations.

_PIC(x):

_PIC(0) => PIC Mode

_PIC(1) => APIC Mode

_PIC(2) => SAPIC Mode

_PIC(3-*n*) => Reserved

6 Configuration

This section specifies the objects OSPM expects to be used in control methods to configure devices. There are three types of configuration objects:

- Device identification objects associate platform devices with Plug and Play IDs.
- Device configuration objects declare and configure hardware resources and characteristics for devices enumerated via ACPI.
- Device insertion and removal objects provide mechanisms for handling dynamic insertion and removal of devices.

This section also defines the ACPI device–resource descriptor formats. Device–resource descriptors are used as parameters by some of the device configuration control method objects.

6.1 Device Identification Objects

Device identification objects associate each platform device with a Plug and Play device ID for each device. All the device identification objects are listed Table 6-1:

Table 6-1 Device Identification Objects

Object	Description
_ADR	Object that evaluates to a device's address on its parent bus.
_CID	Object that evaluates to a device's Plug and Play-compatible ID list.
_DDN	Object that associates a logical software name (for example, COM1) with a device.
_HID	Object that evaluates to a device's Plug and Play hardware ID.
_SUN	Object that evaluates to the slot-unique ID number for a slot.
_STR	Object that contains a Unicode identifier for a device.
_UID	Object that specifies a device's unique persistent ID, or a control method that generates it.

For any device that is not on an enumerable type of bus (for example, an ISA bus), OSPM enumerates the devices' Plug and Play ID(s) and the ACPI BIOS must supply an _HID object (plus an optional _CID object) for each device to enable OSPM to do that. For devices on an enumerable type of bus, such as a PCI bus, the ACPI system must identify which device on the enumerable bus is identified by a particular Plug and Play ID; the ACPI BIOS must supply an _ADR object for each device to enable this. A device object must contain either an _HID object or an _ADR object, but can contain both.

If any of these objects are implemented as control methods, these methods may depend on operation regions. Since the control methods may be evaluated before an operation region provider becomes available, the control method must be structured to execute in the absence of the operation region provider. (_REG methods notify the BIOS of the presence of operation region providers.) When a control method cannot determine the current state of the hardware due to a lack of operation region provider, it is recommended that the control method should return the condition that was true at the time that control passed from the BIOS to the OS. (The control method should return a default, boot value).

6.1.1 _ADR (Address)

This object is used to supply OSPM with the address of a device on its parent bus. An _ADR object must be used to specify the address of any device on a bus that has a standard enumeration algorithm.

An _ADR object can be used to provide capabilities to the specified address even if a device is not present. This allows the system to provide capabilities to a slot on the parent bus.

OSPM infers the parent bus from the location of the _ADR object's device package in the ACPI namespace. For more information about the positioning of device packages in the ACPI namespace, see section 16.2.3.3.1.9, "Device–Declare Bus/Device Package."

_ADR object information must be static and can be defined for the following bus types listed in Table 6-2.

Table 6-2 _ADR Object Bus Types

BUS	**Address encoding**
EISA	EISA slot number 0–F
Floppy Bus	Drive select values used for programming the floppy controller to access the specified INT13 unit number. The _ADR Objects should be sorted based on drive select encoding from 0-3.
IDE Controller	0–Primary Channel, 1–Secondary Channel
IDE Channel	0–Master drive, 1–Slave drive
PCI	High word–Device #, Low word–Function #. (for example, device 3, function 2 is 0x00030002). To refer to all the functions on a device #, use a function number of FFFF).
PCMCIA	Socket #; 0–First Socket
PC CARD	Socket #; 0–First Socket
SMBus	Lowest Slave Address
USB Root HUB	Only one child of the host controller. It must have an _ADR of 0. No other children or values of _ADR are allowed.
USB Ports	Port number

6.1.2 _CID (Compatible ID)

This optional object is used to supply OSPM with a device's Plug and Play-Compatible Device ID. Use _CID objects when a device has no other defined hardware standard method to report its compatible IDs.

A _CID object evaluates to either:

- A single Compatible Device ID
- A package of Compatible Device IDs for the device — in the order of preference, highest preference first.

Each Compatible Device ID must be either:

- A valid HID value (a 32-bit compressed EISA type ID or a string such as "ACPI0004").
- A string that uses a bus-specific nomenclature. For example, _CID can be used to specify the PCI ID. The format of a PCI ID string is one of the following:

  "PCI\CC_ccss"
  "PCI\CC_ccsspp"
  "PCI\VEN_vvvv&DEV_dddd&SUBSYS_ssssssss&REV_rr"
  "PCI\VEN_vvvv&DEV_dddd&SUBSYS_ssssssss"
  "PCI\VEN_vvvv&DEV_dddd&REV_rr"
  "PCI\VEN_vvvv&DEV_dddd"

  Where:

cc – hexadecimal representation of the Class Code byte
ss – hexadecimal representation of the Subclass Code byte
pp – hexadecimal representation of the Programming interface byte
vvvv – hexadecimal representation of the Vendor ID
dddd – hexadecimal representation of the Device ID
ssssssss – hexadecimal representation of the Subsystem ID
rr – hexadecimal representation of the Revision byte

A compatible ID retrieved from a _CID object is only meaningful if it is a non-NULL value.

Example ASL:

```
Device (XYZ) {
    Name (_HID, EISAID ("PNP0303"))          // PC Keyboard Controller
    Name (_CID, EISAID ("PNP030B"))    }
```

6.1.3 _DDN (DOS Device Name)

This object is used to associate a logical name (for example, COM1) with a device. This name can be used by applications to connect to the device.

6.1.4 _HID (Hardware ID)

This object is used to supply OSPM with the device's Plug and Play hardware ID.[8] When describing a platform, use of any _HID objects is optional. However, a _HID object must be used to describe any device that will be enumerated by OSPM. OSPM only enumerates a device when no bus enumerator can detect the device ID. For example, devices on an ISA bus are enumerated by OSPM. Use the _ADR object to describe devices enumerated by bus enumerators other than OSPM.

A _HID object evaluates to either a numeric 32-bit compressed EISA type ID or a string. If a string, the format must be an alphanumeric PNP or ACPI ID with no asterisk or other leading characters.

Example ASL:

```
Name (_HID, EISAID ("PNP0C0C"))       // Control-Method Power Button
Name (_HID, EISAID ("INT0800"))       // Firmware Hub
Name (_HID, "ACPI0003")               // AC adapter device
```

6.1.5 _STR (String)

The _STR object evaluates to a Unicode string that may be used by an OS to provide information to an end user describing the device. This information is particularly valuable when no other information is available.

Example ASL:

```
Device (XYZ) {
          Name (_ADR, 0x00020001)
          Name ( _STR, Unicode("ACME super DVD controller"))
}
```

Then, when all else fails, an OS can use the info included in the _STR object to describe the hardware to the user.

6.1.6 _SUN (Slot User Number)

_SUN is used by OSPM UI to identify slots for the user. For example, this can be used for battery slots, PCMCIA slots, or swappable bay slots to inform the user of what devices are in each slot. _SUN evaluates to a DWORD that is the number to be used in the user interface. This number should match any slot number printed on the physical slot.

6.1.7 _UID (Unique ID)

This object provides OSPM with a serial number-style ID of a device (or battery), which does not change across reboots. This object is optional, but is required when the device has no other way to report a persistent unique device ID. When a system has two devices that report the same _HID, each device must have a _UID object. When reported, the UID needs to be unique only among devices with the same device ID. OSPM typically uses the unique device ID to ensure that the device-specific information, such as network protocol binding information, is remembered for the device even if its relative location changes. For most integrated devices, this object contains a unique identifier. For other devices, like a docking station, this object can be a control method that returns the unique docking station ID.

A _UID object evaluates to either a numeric value or a string.

[8] A Plug and Play (EISA) ID can be obtained by sending e-mail to pnpid@microsoft.com.

6.2 Device Configuration Objects

Device configuration objects are used to configure hardware resources for devices enumerated via ACPI. Device configuration objects provide information about current and possible resource requirements, the relationship between shared resources, and methods for configuring hardware resources.

Note: these objects must only be provided for devices that cannot be configured by any other hardware standard such as PCI, PCMCIA, and so on.

When OSPM enumerates a device, it calls _PRS to determine the resource requirements of the device. It may also call _CRS to find the current resource settings for the device. Using this information, the Plug and Play system determines what resources the device should consume and sets those resources by calling the device's _SRS control method.

In ACPI, devices can consume resources (for example, legacy keyboards), provide resources (for example, a proprietary PCI bridge), or do both. Unless otherwise specified, resources for a device are assumed to be taken from the nearest matching resource above the device in the device hierarchy.

Some resources, however, may be shared amongst several devices. To describe this, devices that share a resource (resource consumers) must use the extended resource descriptors (0x7-0xA) described in section 6.4.3, "Large Resource Data Type." These descriptors point to a single device object (resource producer) that claims the shared resource in its _PRS. This allows OSPM to clearly understand the resource dependencies in the system and move all related devices together if it needs to change resources. Furthermore, it allows OSPM to allocate resources only to resource producers when devices that consume that resource appear.

The device configuration objects are listed in Table 6-3.

Table 6-3 Device Configuration Objects

Object	Description
_CRS	Object that specifies a device's *current* resource settings, or a control method that generates such an object.
_DIS	Control method that disables a device.
_DMA	Object that specifies a device's *current* resources for DMA transactions.
_FIX	Object used to provide correlation between the fixed-hardware register blocks defined in the FADT and the devices that implement these fixed-hardware registers.
_HPP	Object that specifies the cache-line size, latency timer, SERR enable, and PERR enable values to be used when configuring a PCI device inserted into a hot-plug slot or initial configuration of a PCI device at system boot.
_MAT	Object that evaluates to a buffer of MADT APIC Structure entries.
_PRS	An object that specifies a device's *possible* resource settings, or a control method that generates such an object.
_PRT	Object that specifies the PCI interrupt routing table.
_PXM	Object that specifies a proximity domain for a device.
_SRS	Control method that sets a device's settings.

6.2.1 _CRS (Current Resource Settings)

This required object evaluates to a byte stream that describes the system resources currently allocated to a device. Additionally, a bus device must supply the resources that it decodes and can assign to its children devices. If a device is disabled, then _CRS returns a valid resource template for the device, but the actual resource assignments in the return byte stream are ignored. If the device is disabled when _CRS is called, it must remain disabled.

The format of the data contained in a _CRS object follows the formats defined in section 6.4, “Resource Data Types for ACPI,” a compatible extension of the formats specified in the PNPBIOS specification.[9] The resource data is provided as a series of data structures, with each of the resource data structures having a unique tag or identifier. The resource descriptor data structures specify the standard PC system resources, such as memory address ranges, I/O ports, interrupts, and DMA channels.

Arguments:
None

Result Code:
Byte stream

6.2.2 _DIS (Disable)

This control method disables a device. When the device is disabled, it must not be decoding any hardware resources. Prior to running this control method, OSPM will have already put the device in the D3 state.

When a device is disabled via the _DIS, the _STA control method for this device must return with the Disabled bit set.

Arguments:
None

Result Code:
None

6.2.3 _DMA (Direct Memory Access)

This optional object returns a byte stream in the same format as a _CRS object. _DMA is only defined under devices that represent buses. It specifies the ranges the bus controller (bridge) decodes on the child-side of its interface. (This is analogous to the _CRS object, which describes the resources that the bus controller decodes on the parent-side of its interface.) Any ranges described in the resources of a _DMA object can be used by child devices for DMA or bus master transactions.

The _DMA object is only valid if a _CRS object is also defined. OSPM must re-evaluate the _DMA object after an _SRS object has been executed because the _DMA ranges resources may change depending on how the bridge has been configured.

If the _DMA object is not present for a bus device, the OS assumes that any address placed on a bus by a child device will be decoded either by a device on the bus or by the bus itself, (in other words, all address ranges can be used for DMA).

For example, if a platform implements a PCI bus that cannot access all of physical memory, it has a _DMA object under that PCI bus that describes the ranges of physical memory that can be accessed by devices on that bus.

[9] Plug and Play BIOS Specification Version 1.0A, May 5, 1994, Compaq Computer Corp., Intel Corp., Phoenix Technologies Ltd.

A _DMA object is not meant to describe any “map register” hardware that is set up for each DMA transaction. It is meant only to describe the DMA properties of a bus that cannot be changed without reevaluating the _SRS method.

Arguments:
None

Result Code:

Byte stream

6.2.4 _FIX (Fixed Register Resource Provider)

This optional object is used to provide a correlation between the fixed-hardware register blocks defined in the FADT and the devices in the ACPI namespace that implement these fixed-hardware registers. This object evaluates to a package of Plug and Play-compatible IDs (32-bit compressed EISA type IDs) that correlate to the fixed-hardware register blocks defined in the FADT. The device under which _FIX appears plays a role in the implementation of the fixed-hardware (for example, implements the hardware or decodes the hardware’s address). _FIX conveys to OSPM whether a given device can be disabled, powered off, or should be treated specially by conveying its role in the implementation of the ACPI fixed-hardware register interfaces. This object takes no arguments.

The _CRS object describes a device’s resources. That _CRS object may contain a superset of the resources in the FADT, as the device may actually decode resources beyond what the FADT requires. Furthermore, in a machine that performs translation of resources within I/O bridges, the processor-relative resources in the FADT may not be the same as the bus-relative resources in the _CRS.

Each of fields in the FADT has its own corresponding Plug and Play ID, as shown below:

- PNP0C20 - SMI_CMD
- PNP0C21 - PM1a_EVT_BLK / X_ PM1a_EVT_BLK
- PNP0C22 - PM1b_EVT_BLK / X_PM1b_EVT_BLK
- PNP0C23 - PM1a_CNT_BLK / X_PM1a_CNT_BLK
- PNP0C24 - PM1b_CNT_BLK / X_ PM1b_CNT_BLK
- PNP0C25 - PM2_CNT_BLK / X_ PM2_CNT_BLK
- PNP0C26 - PM_TMR_BLK / X_ PM_TMR_BLK
- PNP0C27 - GPE0_BLK / X_GPE0_BLK
- PNP0C28 - GPE1_BLK / X_ GPE1_BLK

Example ASL for _FIX usage:

```
Scope(\_SB) {
   Device(PCI0) {                     // Root PCI Bus
      Name(_HID, EISAID("PNP0A03"))   // Need _HID for root device
      Name(_ADR,0)                    // Device 0 on this bus
        Method (_CRS,0){       // Need current resources for root device
            // Return current resources for root bridge 0
        }
        Name(_PRT, Package(){  // Need PCI IRQ routing for PCI bridge
            // Package with PCI IRQ routing table information
        })
      Name(_FIX, Package(1) {
            EISAID("PNP0C25")}        //PM2 control ID
        )

        Device (PX40) {                   // ISA
            Name(_ADR,0x00070000)
            Name(_FIX, Package(1) {
                EISAID("PNP0C20")}    //SMI command port
            )
            Device (NS17) {               // NS17  (Nat. Semi 317, an ACPI part)
                Name(_HID, EISAID("PNP0C02"))
                Name(_FIX, Package(3) {
                    EISAID("PNP0C22"),    //PM1b event ID
                    EISAID("PNP0C24"),    //PM1b control ID
                    EISAID("PNP0C28")} //GPE1 ID
            }
        }          // end PX40

        Device (PX43) {               // PM Control
            Name(_ADR,0x00070003)
            Name(_FIX, Package(4) {
                EISAID("PNP0C21"),    //PM1a event ID
                EISAID("PNP0C23"),    //PM1a control ID
                EISAID("PNP0C26"),    //PM Timer ID
                EISAID("PNP0C27")} //GPE0 ID
            )
        }          // end PX43

    } // end PCI0

}   // end scope SB
```

6.2.5 _HPP (Hot Plug Parameters)

This optional object evaluates to the cache-line size, latency timer, SERR enable, and PERR enable values to be used when configuring a PCI device inserted into a hot-plug slot or for performing configuration of a PCI devices not configured by the BIOS at system boot. The object is placed under a PCI bus where this behavior is desired, such as a bus with hot-plug slots. _HPP provided settings apply to all child buses, until another _HPP object is encountered.

Arguments:

None

Result Code:

```
Method (_HPP, 0) {
    Return (Package(4){
        0x08,     // CacheLineSize in DWORDS
        0x40,     // LatencyTimer in PCI clocks
        0x01,     // Enable SERR (Boolean)
        0x00      // Enable PERR (Boolean)
        })
```

Table 6-4 _HPP

Field	Format	Definition
Cache-line size	INTEGER	Cache-line size reported in number of DWORDs.
Latency timer	INTEGER	Latency timer value reported in number of PCI clock cycles.
Enable SERR	INTEGER	When set to 1, indicates that action must be performed to enable SERR in the command register.
Enable PERR	INTEGER	When set to 1, indicates that action must be performed to enable PERR in the command register.

6.2.5.1 Example: Using _HPP

```
Scope(\_SB) {
    Device(PCI0) {                       // Root PCI Bus
        Name(_HID, EISAID("PNP0A03"))    // _HID for root device
        Name(_ADR,0)                     // Device 0 on this bus
        Method (_CRS,0){                 // Need current resources for root dev
                   // Return current resources for root bridge 0
        }
        Name(_PRT, Package(){  // Need PCI IRQ routing for PCI bridge
                   // Package with PCI IRQ routing table information
        })

        Device (P2P1) {                  // First PCI-to-PCI bridge (No Hot Plug slots)
            Name(_ADR,0x000C0000)        // Device#Ch, Func#0 on bus PCI0
            Name(_PRT, Package(){        // Need PCI IRQ routing for PCI bridge
                       // Package with PCI IRQ routing table information
            })
        } // end P2P1

        Device (P2P2) {       // Second PCI-to-PCI bridge (Bus contains Hot plug slots)
            Name(_ADR,0x000E0000)        // Device#Eh, Func#0 on bus PCI0
            Name(_PRT, Package(){        // Need PCI IRQ routing for PCI bridge
                       // Package with PCI IRQ routing table information
            })
            Name(_HPP, Package(){0x08,0x40, 0x01, 0x00})

            // Device definitions for Slot 1- HOT PLUG SLOT
            Device (S1F0) {              // Slot 1, Func#0 on bus P2P2
                Name(_ADR,0x00020000)
                Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S1F1) {              // Slot 1, Func#1 on bus P2P2
                Name(_ADR,0x00020001)
                Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S1F2) {              // Slot 1, Func#2 on bus P2P2
                Name(_ADR,0x000200 02)
                Method(_EJ0, 1) {  //Remove all power to device}           }
            Device (S1F3) {              // Slot 1, Func#3 on bus P2P2
                Name(_ADR,0x00020003)
                Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S1F4) {              // Slot 1, Func#4 on bus P2P2
                Name(_ADR,0x00020004)
                Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S1F5) {              // Slot 1, Func#5 on bus P2P2
                Name(_ADR,0x00020005)
                Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S1F6) {              // Slot 1, Func#6 on bus P2P2
                Name(_ADR,0x00020006)
                Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S1F7) {              // Slot 1, Func#7 on bus P2P2
                Name(_ADR,0x00020007)
                Method(_EJ0, 1) {  //Remove all power to device}
            }
            // Device definitions for Slot 2- HOT PLUG SLOT
            Device (S2F0) {              // Slot 2, Func#0 on bus P2P2
               Name(_ADR,0x00030000)
               Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S2F1) {              // Slot 2, Func#1 on bus P2P2
               Name(_ADR,0x00030001)
               Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S2F2) {              // Slot 2, Func#2 on bus P2P2
               Name(_ADR,0x00030002)
               Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S2F3) {              // Slot 2, Func#3 on bus P2P2
               Name(_ADR,0x00030003)
               Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S2F4) {              // Slot 2, Func#4 on bus P2P2
               Name(_ADR,0x00030004)
               Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S2F5) {              // Slot 2, Func#5 on bus P2P2
               Name(_ADR,0x00030005)
               Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S2F6) {              // Slot 2, Func#6 on bus P2P2
               Name(_ADR,0x00030006)
               Method(_EJ0, 1) {  //Remove all power to device}
            }
            Device (S2F7) {              // Slot 2, Func#7 on bus P2P2
               Name(_ADR,0x00030007)
               Method(_EJ0, 1) {  //Remove all power to device}
            }

        }   // end P2P2
    }   // end PCI0
}   // end Scope (\_SB)
```

OSPM will configure a PCI device on a card hot-plugged into slot 1 or slot 2, with a cache line size of 32 (Notice this field is in DWORDs), latency timer of 64, enable SERR, but leave PERR alone.

6.2.6 _MAT (Multiple APIC Table Entry)

This optional object evaluates to a buffer returning data in the format of a series of Multiple APIC Description Table (MADT) APIC Structure entries. This object can appear under an I/O APIC or processor object definition as processors may contain Local APICs. Specific types of entries from section 5.2.11 “Global System Interrupt Vectors” are meaningful to (in other words, is processed by) OSPM when returned via the evaluation of this object as described below. Other entry types returned by the evaluation of _MAT are ignored by OSPM.

When _MAT appears under a Processor object, OSPM processes Local APIC (section 5.2.10.5, “Processor Local APIC”), Local SAPIC (section 5.2.10.13, “Local SAPIC Structure”), and local APIC NMI (section 5.2.10.10, “Local APIC NMI”) entries returned from the object’s evaluation. Other entry types are ignored by OSPM. OSPM uses the ACPI processor ID in the entries returned from the object’s evaluation to identify the entries corresponding to the ACPI processor ID of the Processor object.

When _MAT appears under an I/O APIC, OSPM processes I/O APIC (section 5.2.10.6), I/O SAPIC (section 5.2.10.12, “I/O SAPIC Structure”), non-maskable interrupt sources (section 5.2.10.9, “Non-Maskable Interrupt Sources”), interrupt source overrides (section 5.2.10.8, “Interrupt Source Overrides”), and platform interrupt source structure (section 5.2.10.14, “Platform Interrupt Source Structure”) entries returned from the object’s evaluation. Other entry types are ignored by OSPM.

Arguments:

None

Result Code:

A buffer

Example ASL for _MAT usage:

```
Scope(\_SB) {
   Device(PCI0) {                     // Root PCI Bus
      Name(_HID, EISAID("PNP0A03"))   // Need _HID for root device
      Name(_ADR,0)                    // Device 0 on this bus
      Method (_CRS,0){                // Need current resources for root device
           // Return current resources for root bridge 0
      }
      Name(_PRT, Package(){   // Need PCI IRQ routing for PCI bridge
           // Package with PCI IRQ routing table information
      })

      Device (P64A) {     // P64A ACPI
         Name(_ADR,0)
           OperationRegion(TABD, SystemMemory, //Physical address of first
               // data byte of multiple ACPI table, Length of tables)
           Field (TABD, ByteAcc, NoLock, Preserve){
               MATD, Length of tables x 8
           }
           Method(_MAT, 0){
         return (MATD)}
        }          // end P64A

    } // end PCI0

}   // end scope SB
```

6.2.7 _PRS (Possible Resource Settings)

This optional object evaluates to a byte stream that describes the *possible* resource settings for the device. When describing a platform, specify a _PRS for all the configurable devices. Static (non-configurable) devices do not specify a _PRS object. The information in this package is used by OSPM to select a conflict-free resource allocation without user intervention. This method must not reference any operation regions that have not been declared available by a _REG method.

The format of the data in a _PRS object follows the same format as the _CRS object (for more information, see the _CRS object definition).

If the device is disabled when _PRS is called, it must remain disabled.

Arguments:

None

Result Code:

Byte stream

6.2.8 _PRT (PCI Routing Table)

PCI interrupts are inherently non-hierarchical. PCI interrupt pins are wired to interrupt inputs of the interrupt controllers. The _PRT object provides a mapping from PCI interrupt pins to the interrupt inputs of the interrupt controllers. The _PRT object is required under all PCI root bridges. _PRT evaluates to a package that contains a list of packages, each of which describes the mapping of a PCI interrupt pin.

Note: The PCI function number in the *Address* field of the_PRT packages must be 0xFFFF, indicating "any" function number or "all functions".

The _PRT mapping packages have the fields listed in Table 6-5.

Table 6-5 Mapping Fields

Field	Type	Description
Address	DWORD	The address of the device (uses the same format as _ADR).
Pin	BYTE	The PCI pin number of the device (0–INTA, 1–INTB, 2–INTC, 3–INTD).
Source	NamePath Or BYTE	Name of the device that allocates the interrupt to which the above pin is connected. The name can be a fully qualified path, a relative path, or a simple name segment that utilizes the namespace search rules. **Note**: This field is a NamePath and not a String literal, meaning that it should not be surrounded by quotes. If this field is the integer constant Zero (or a BYTE value of 0), then the interrupt is allocated from the global interrupt pool.
Source Index	DWORD	Index that indicates which resource descriptor in the resource template of the device pointed to in the Source field this interrupt is allocated from. If the **Source** field is the BYTE value zero, then this field is the global system interrupt number to which the pin is connected.

There are two ways that _PRT can be used. Typically, the interrupt input that a given PCI interrupt is on is configurable. For example, a given PCI interrupt might be configured for either IRQ 10 or 11 on an 8259 interrupt controller. In this model, each interrupt is represented in the ACPI namespace as a PCI Interrupt Link Device.

These objects have _PRS, _CRS, _SRS, and _DIS control methods to allocate the interrupt. Then, OSPM handles the interrupts not as interrupt inputs on the interrupt controller, but as PCI interrupt pins. The driver looks up the device's pins in the _PRT to determine which device objects allocate the interrupts. To move the PCI interrupt to a different interrupt input on the interrupt controller, OSPM uses _PRS, _CRS, _SRS, and _DIS control methods for the PCI Interrupt Link Device.

In the second model, the PCI interrupts are hardwired to specific interrupt inputs on the interrupt controller and are not configurable. In this case, the Source field in _PRT does not reference a device, but instead contains the value zero, and the Source Index field contains the global system interrupt to which the PCI interrupt is hardwired.

6.2.8.1 Example: Using _PRT to Describe PCI IRQ Routing

The following example describes two PCI slots and a PCI video chip. Notice that the interrupts on the two PCI slots are wired differently (barber-poled).

```
Scope(\_SB) {
    Device(LNKA){
        Name(_HID, EISAID("PNP0C0F"))                    // PCI interrupt link
        Name(_UID, 1)
        Name(_PRS, ResourceTemplate(){
            Interrupt(ResourceProducer,…) {10,11}        // IRQs 10,11
        })
        Method(_DIS) {…}
        Method(_CRS) {…}
        Method(_SRS, 1) {…}
    }
    Device(LNKB){
        Name(_HID, EISAID("PNP0C0F"))                    // PCI interrupt link
        Name(_UID, 2)
        Name(_PRS, ResourceTemplate(){
            Interrupt(ResourceProducer,…) {11,12}        // IRQs 11,12
        })
        Method(_DIS) {…}
        Method(_CRS) {…}
        Method(_SRS, 1) {…}
    }
    Device(LNKC){
        Name(_HID, EISAID("PNP0C0F"))                    // PCI interrupt link
        Name(_UID, 3)
        Name(_PRS, ResourceTemplate(){
            Interrupt(ResourceProducer,…) {12,14}        // IRQs 12,14
        })
        Method(_DIS) {…}
        Method(_CRS) {…}
        Method(_SRS, 1) {…}
    }
    Device(LNKD){
        Name(_HID, EISAID("PNP0C0F"))                    // PCI interrupt link
        Name(_UID, 4)
        Name(_PRS, ResourceTemplate(){
            Interrupt(ResourceProducer,…) {10,15}        // IRQs 10,15
        })
        Method(_DIS) {…}
        Method(_CRS) {…}
        Method(_SRS, 1) {…}
    }
    Device(PCI0){
        …
        Name(_PRT, Package{
            Package{0x0004ffff, 0, \_SB_.LNKA, 0},    // Slot 1, INTA    // A fully
            Package{0x0004ffff, 1, \_SB_.LNKB, 0},    // Slot 1, INTB    // qualified
            Package{0x0004ffff, 2, \_SB_.LNKC, 0},    // Slot 1, INTC    // pathname
            Package{0x0004ffff, 3, \_SB_.LNKD, 0},    // Slot 1, INTD    // can be used,
            Package{0x0005ffff, 0, LNKB, 0},          // Slot 2, INTA    // or a simple
            Package{0x0005ffff, 1, LNKC, 0},          // Slot 2, INTB    // name segment
            Package{0x0005ffff, 2, LNKD, 0},          // Slot 2, INTC    // utilizing the
            Package{0x0005ffff, 3, LNKA, 0},          // Slot 2, INTD    // search rules
            Package{0x0006ffff, 0, LNKC, 0}           // Video, INTA
        })
    }
}
```

6.2.9 _PXM (Proximity)

This optional object is used to describe proximity domains within a machine. _PXM evaluates to an integer that identifies the device as belonging to a specific proximity domain. The OS assumes that two devices in the same proximity domain are tightly coupled. An OS could choose to optimize its behavior based on this. For example, in a system with four processors and six memory devices, there might be two separate proximity domains (0 and 1), each with two processors and three memory devices. In this case, the OS may decide to run some software threads on the processors in proximity domain 0 and others on the processors in proximity domain 1. Furthermore, for performance reasons, it could choose to allocate memory for those threads from the memory devices inside the proximity domain common to the processor and the memory device rather than from a memory device outside of the processor's proximity domain. _PXM can be used to identify any device belonging to a proximity domain. Children of a device belong to the same proximity domain as their parent unless they contain an overriding _PXM. Proximity domains do not imply any ejection relationships.

An OS makes no assumptions about the proximity or nearness of different proximity domains. The difference between two integers representing separate proximity domains does not imply distance between the proximity domains (in other words, proximity domain 1 is not assumed to be closer to proximity domain 0 than proximity domain 6).

Arguments:

None

Result Code:

An integer

6.2.10 _SRS (Set Resource Settings)

This optional control method takes one byte stream argument that specifies a new resource allocation for a device. The resource descriptors in the byte stream argument must be specified in the same order as listed in the _CRS byte stream (for more information, see the _CRS object definition). A _CRS object can be used as a template to ensure that the descriptors are in the correct format.

The settings must take effect before the _SRS control method returns.

This method must not reference any operation regions that have not been declared available by a _REG method.

If the device is disabled, _SRS enables the device at the specified resources. _SRS is not used to disable a device; use the _DIS control method instead.

Arguments:

Byte stream

Result Code:

None

6.3 Device Insertion and Removal Objects

Device insertion and removal objects provide mechanisms for handling dynamic insertion and removal of devices. These same mechanisms are used for docking and undocking. These objects give information about whether or not devices are present, which devices are physically in the same device (independent of which bus the devices live on), and methods for controlling ejection or interlock mechanisms.

The system is more stable when removable devices have a software-controlled, VCR-style ejection mechanism instead of a "surprise-style" ejection mechanism. In this system, the eject button for a device does not immediately remove the device, but simply signals the operating system. OSPM then shuts down the device, closes open files, unloads the driver, and sends a command to the hardware to eject the device.

In ACPI, the sequence of events for dynamically inserting a device follows the process below. Notice that this process supports hot, warm, and cold insertion of devices.

1. If the device is physically inserted while the computer is in the working state (in other words, hot insertion), the hardware generates a general-purpose event.
2. The control method servicing the event uses the **Notify**(*device*,0) command to inform OSPM of the bus that the new device is on or the device object for the new device. If the Notify command points to the device object for the new device, the control method must have changed the device's status returned by _STA to indicate that the device is now present. The performance of this process can be optimized by having the object of the Notify as close as possible, in the namespace hierarchy, to where the new device resides. The Notify command can also be used from the _WAK control method (for more information about _WAK, see section 7.3.5 "_WAK (System Wake)") to indicate device changes that may have occurred while the computer was sleeping. For more information about the Notify command, see section 5.6.3 "Device Object Notification."."
3. OSPM uses the identification and configuration objects to identify, configure, and load a device driver for the new device and any devices found below the device in the hierarchy.
4. If the device has a _LCK control method, OSPM may later run this control method to lock the device.

The new device referred to in step 2 need not be a single device, but could be a whole tree of devices. For example, it could point to the PCI-PCI bridge docking connector. OSPM will then load and configure all devices it found below that bridge. The control method can also point to several different devices in the hierarchy if the new devices do not all live under the same bus. (in other words, more than one bus goes through the connector).

For removing devices, ACPI supports both hot removal (system is in the S0 state), and warm removal (system is in a sleep state: S1-S4). This is done using the _EJ*x* control methods. Devices that can be ejected include an _EJ*x* control method for each sleeping state the device supports (a maximum of 2 _EJ*x* objects can be listed). For example, hot removal devices would supply an _EJ0; warm removal devices would use one of _EJ1-EJ4. These control methods are used to signal the hardware when an eject is to occur.

The sequence of events for dynamically removing a device goes as follows:

1. The eject button is pressed and generates a general-purpose event. (If the system was in a sleeping state, it should wake the computer).
2. The control method for the event uses the **Notify**(*device*, 3) command to inform OSPM which specific device the user has requested to eject. Notify does not need to be called for every device that may be ejected, but for the top-level device. Any child devices in the hierarchy or any ejection-dependent devices on this device (as described by _EJD, below) are automatically removed.
3. The OS shuts down and unloads devices that will be removed.
4. If the device has a _LCK control method, OSPM runs this control method to unlock the device.
5. The OS looks to see what _EJ*x* control methods are present for the device. If the removal event will cause the system to switch to battery power (in other words, an undock) and the battery is low, dead, or not present, OSPM uses the lowest supported sleep state _EJ*x* listed; otherwise it uses the highest state _EJ*x*. Having made this decision, OSPM runs the appropriate _EJ*x* control method to prepare the hardware for eject.
6. Warm removal requires that the system be put in a sleep state. If the removal will be a warm removal, OSPM puts the system in the appropriate S*x* state. If the removal will be a hot removal, OSPM skips to step 8, below.
7. For warm removal, the system is put in a sleep state. Hardware then uses any motors, and so on, to eject the device. Immediately after ejection, the hardware transitions the computer to S0. If the system was sleeping when the eject notification came in, the OS returns the computer to a sleeping state consistent with the user's wake settings.
8. OSPM calls _STA to determine if the eject successfully occurred. (In this case, control methods do not need to use the **Notify**(*device*,3) command to tell OSPM of the change in _STA) If there were any mechanical failures, _STA returns 3: device present and not functioning, and OSPM informs the user of the problem.

Note: This mechanism is the same for removing a single device and for removing several devices, as in an undock.

ACPI does not disallow surprise-style removal of devices; however, this type of removal is not recommended because system and data integrity cannot be guaranteed when a surprise-style removal occurs. Because the OS is not informed, its device drivers cannot save data buffers and it cannot stop accesses to the device before the device is removed. To handle surprise-style removal, a general-purpose event must be raised. Its associated control method must use the Notify command to indicate which bus the device was removed from.

The device insertion and removal objects are listed in Table 6-6.

Table 6-6 Device Insertion and Removal Objects

Object	Description
_EDL	Object that evaluates to a package of namespace references of device objects that depend on the device containing _EDL. Whenever the named device is ejected, OSPM ejects all dependent devices.
_EJD	Object that evaluates to the name of a device object on which a device depends. Whenever the named device is ejected, the dependent device must receive an ejection notification.
_EJ*x*	Control method that ejects a device.
_LCK	Control method that locks or unlocks a device.
_RMV	Object that indicates that the given device is removable.
_STA	Control method that returns a device's status.

6.3.1 _EDL (Eject Device List)

This object evaluates to a package of namespace references containing the names of device objects that depend on the device under which the _EDL object is declared. This is primarily used to support docking stations. Before the device under which the _EDL object is declared may be ejected, OSPM prepares the devices listed in the _EDL object for physical removal.

Before OSPM ejects a device via the device's _EJx methods, all dependent devices listed in the package returned by _EDL are prepared for removal. Notice that _EJx methods under the dependent devices are not executed.

When describing a platform that includes a docking station, an _EDL object is declared under the docking station device. For example, if a mobile system can attach to two different types of docking stations, _EDL is declared under both docking station devices and evaluates to the packaged list of devices that must be ejected when the system is ejected from the docking station.

An ACPI 2.0-compliant OS evaluates the _EDL method just prior to ejecting the device.

6.3.2 _EJD (Ejection Dependent Device)

This object is used to specify the name of a device on which the device, under which this object is declared, is dependent. This object is primarily used to support docking stations. Before the device indicated by _EJD is ejected, OSPM will prepare the dependent device (in other words, the device under which this object is declared) for removal.

_EJD is evaluated once when the ACPI table loads. The EJx methods of the device indicated by _EJD will be used to eject all the dependent devices. A device's dependents will be ejected when the device itself is ejected.

Note: OSPM will not execute a dependent device's _EJx methods when the device indicated by _EJD is ejected.

When describing a platform that includes a docking station, usually more than one _EJD object will be needed. For example, if a dock attaches both a PCI device and an ACPI-configured device to a mobile system, then both the PCI device description package and the ACPI-configured device description package must include an _EJD object that evaluates to the name of the docking station (the name specified in an _ADR or _HID object in the docking station's description package). Thus, when the docking connector signals an eject request, OSPM first attempts to disable and unload the drivers for both the PCI and ACPI configured devices.

Note: An ACPI 1.0 OS evaluates the _EJD methods only once during the table load process. This greatly restricts a table designer's freedom to describe dynamic dependencies such as those created in scenarios with multiple docking stations. This restriction is illustrated in the example below; the _EJD information supplied via and ACPI 1.0-compatible namespace omits the IDE2 device from DOCK2's list of ejection dependencies. In ACPI 2.0, OSPM will be presented with a more in-depth view of the ejection dependencies in a system by use of the _EDL methods.

Example

An example use of _EJD and _EDL is as follows:

```
Scope(\_SB.PCI0) {

    Device(DOCK1) {    // Pass through dock – DOCK1
        Name(_ADR, …)
        Method(_EJ0, 0) {…}
        Method(_DCK, 1) {…}
        Name(_BDN, …)
        Method(_STA, 0) {0xF}
        Name(_EDL, Package( ) {   // DOCK1 has two dependent devices – IDE2 and CB2
                \_SB.PCI0.IDE2,
            \_SB.PCI0.CB2})
        }
    Device(DOCK2) {    // Pass through dock – DOCK2
        Name(_ADR, …)
        Method(_EJ0, 0) {…}
        Method(_DCK, 1) {…}
        Name(_BDN, …)
        Method(_STA, 0) {0x0}
        Name(_EDL, Package( ) {   // DOCK2 has one dependent device – IDE2
            \_SB.PCI0.IDE2})
        }

    Device(IDE1) {     // IDE Drive1 not dependent on the dock
        Name(_ADR, …)
        }

    Device(IDE2) {     // IDE Drive2
        Name(_ADR, …)
        Name(_EJD,"\\_SB.PCI0.DOCK1") // Dependent on DOCK1
        }

    Device(CB2) {      // CardBus Controller
        Name(_ADR, …)
        Name(_EJD,"\\_SB.PCI0.DOCK1") // Dependent on DOCK1
        }
} // end \_SB.PCIO
```

6.3.3 _EJx (Eject)

These control methods are optional and are supplied for devices that support a software-controlled VCR-style ejection mechanism or that require an action be performed such as isolation of power/data lines before the device can be removed from the system. To support warm (system is in a sleep state) and hot (system is in S0) removal, an _EJ*x* control method is listed for each sleep state from which the device supports removal, where *x* is the sleeping state supported. For example, _EJ0 indicates the device supports hot removal; _EJ1–EJ4 indicate the device supports warm removal.

For hot removal, the device must be immediately ejected when OSPM calls the _EJ0 control method. The _EJ0 control method does not return until ejection is complete. After calling _EJ0, OSPM verifies the device no longer exists to determine if the eject succeeded. For _HID devices, OSPM evaluates the _STA method. For _ADR devices, OSPM checks with the bus driver for that device.

For warm removal, the _EJ1–_EJ4 control methods do not cause the device to be immediately ejected. Instead, they set proprietary registers to prepare the hardware to eject when the system goes into the given sleep state. The hardware ejects the device only after OSPM has put the system in a sleep state by writing to the SLP_EN register. After the system resumes, OSPM calls _STA to determine if the eject succeeded.

The _EJ*x* control methods take one parameter to indicate whether eject should be enabled or disabled:

1–Hot eject or mark for ejection
0–Cancel mark for ejection (EJ0 will never be called with this value)

A device object may have multiple _EJ*x* control methods. First, it lists an EJx control method for the preferred sleeping state to eject the device. Optionally, the device may list an EJ4 control method to be used when the system has no power (for example, no battery) after the eject. For example, a hot-docking notebook might list _EJ0 and _EJ4.

6.3.4 _LCK (Lock)

This control method is optional and is required only for a device that supports a software-controlled locking mechanism. When the OS invokes this control method, the associated device is to be locked or unlocked based upon the value of the argument that is passed. On a lock request, the control method must not complete until the device is completely locked.

The _LCK control method takes one parameter that indicates whether or not the device should be locked:

1 –Lock the device.
0–Unlock the device.

When describing a platform, devices use either a _LCK control method or an _EJ*x* control method for a device.

6.3.5 _RMV (Remove)

The optional _RMV object indicates to OSPM whether the device can be removed while the system is in the working state and does not require any ACPI system firmware actions to be performed for the device to be safely removed from the system (in other words, any device that only supports surprise-style removal). Any such removable device that does not have _LCK or _EJx control methods must have an _RMV object. This allows OSPM to indicate to the user that the device can be removed and to provide a way for shutting down the device before removing it. OSPM will transition the device into D3 before telling the user it is safe to remove the device.

This method is reevaluated after a device-check notification.

Arguments:

None

Result Code:

0–The device cannot be removed.

1–The device can be removed.

Note: Operating Systems implementing ACPI 1.0 interpret the presence of this object to mean that the device is removable.

6.3.6 _STA (Status)

This object returns the status of a device, which can be one of the following: enabled, disabled, or removed.

Arguments:
None

Result Code (bitmap):
Bit 0–Set if the device is present.
Bit 1–Set if the device is enabled and decoding its resources.
Bit 2–Set if the device should be shown in the UI.
Bit 3–Set if the device is functioning properly (cleared if the device failed its diagnostics).
Bit 4–Set if the battery is present.
Bits 5–31–Reserved (must be cleared).

If bit 0 is cleared, then bit 1 must also be cleared (in other words, a device that is not present cannot be enabled).

A device can only decode its hardware resources if both bits 0 and 1 are set. If the device is not present (bit 0 cleared) or not enabled (bit 1 cleared), then the device must not decode its resources.

If a device is present in the machine, but should not be displayed in OSPM user interface, bit 2 is cleared. For example, a notebook could have joystick hardware (thus it is present and decoding its resources), but the connector for plugging in the joystick requires a port replicator. If the port replicator is not plugged in, the joystick should not appear in the UI, so bit 2 is cleared.

If a device object does not have an _STA object, then OSPM assumes that all of the above bits are set (in other words, the device is present, enabled, shown in the UI, and functioning).

This method must not reference any operation regions that have not been declared available by a _REG method.

6.4 Resource Data Types for ACPI

The _CRS, _PRS, and _SRS control methods use packages of resource descriptors to describe the resource requirements of devices.

6.4.1 ASL Macros for Resource Descriptors

ASL includes some macros for creating resource descriptors. The ASL syntax for these macros is defined in section 16.2.4, “ASL Macros for Resource Descriptors.”

6.4.2 Small Resource Data Type

A small resource data type may be 2 to 8 bytes in size and adheres to the following format:

Table 6-7 Small Resource Data Type Tag Bit Definitions

Offset	**Field**		
Byte 0	**Tag Bit[7]**	**Tag Bits[6:3]**	**Tag Bits [2:0]**
	Type–0	Small item name	Length–*n* bytes
Bytes 1 to *n*	Data bytes		

The following small information items are currently defined for Plug and Play devices:

Table 6-8 Small Resource Items

Small Item Name	Value
Reserved	0x1
Reserved	0x2
Reserved	0x3
IRQ format	0x4
DMA format	0x5
Start dependent Function	0x6
End dependent Function	0x7
I/O port descriptor	0x8
Fixed location I/O port descriptor	0x9
Reserved	0xA–0xD
Vendor defined	0xE
End tag	0xF

6.4.2.1 IRQ Format (Type 0, Small Item Name 0x4, Length=2 or 3)

The IRQ data structure indicates that the device uses an interrupt level and supplies a mask with bits set indicating the levels implemented in this device. For standard PC-AT implementation there are 15 possible interrupts so a two-byte field is used. This structure is repeated for each separate interrupt required.

Table 6-9 IRQ Descriptor Definition

Offset	Field Name
Byte 0	Value = 0010001nB (Type = 0, small item name = 0x4, length = (2 or 3))
Byte 1	IRQ mask bits[7:0], _INT Bit[0] represents IRQ0, bit[1] is IRQ1, and so on.
Byte 2	IRQ mask bits[15:8], _INT Bit[0] represents IRQ8, bit[1] is IRQ9, and so on.

Table 6-9 IRQ Descriptor Definition *(continued)*

Offset	Field Name
Byte 3	IRQ Information. Each bit, when set, indicates this device is capable of driving a certain type of interrupt. (Optional—if not included then assume edge sensitive, high true interrupts) **Note:** These bits can be used both for reporting and setting IRQ resources. **Note:** This descriptor is meant for describing interrupts that are connected to PIC-compatible interrupt controllers, which can only be programmed for Active-High-Edge-Triggered or Active-Low-Level-Triggered interrupts. Any other combination is illegal. The Extended Interrupt Descriptor can be used to describe other combinations. Bit[7:5] *Reserved (must be 0)* Bit[4] Interrupt is sharable, _SHR Bit[3] Interrupt Polarity, _LL 0: Active-High–This interrupt is sampled when the signal is high, or true. 1: Active-Low–This interrupt is sampled when the signal is low, or false. Bit[2:1] *Ignored* Bit[0] Interrupt Mode, _HE 0: Level-Triggered–This interrupt is triggered in response to the signal being in a low state. 1: Edge-Triggered–This interrupt is triggered in response to a change in signal state from low to high.

Note: Low true, level sensitive interrupts may be electrically shared, but the process of how this might work is beyond the scope of this specification.

Note: If byte 3 is not included, High true, edge sensitive, non-shareable is assumed.

See section 16.2.4.1, “ASL Macro for IRQ Descriptor,” for a description of the ASL macro that creates an IRQ descriptor.

6.4.2.2 DMA Format (Type 0, Small Item Name 0x5, Length=2)

The DMA data structure indicates that the device uses a DMA channel and supplies a mask with bits set indicating the channels actually implemented in this device. This structure is repeated for each separate channel required.

Table 6-10 DMA Descriptor Definition

Offset	Field Name
Byte 0	Value = 00101010B (Type = 0, small item name = 0x5, length = 2)
Byte 1	DMA channel mask bits[7:0], _DMA Bit[0] is channel 0
Byte 2	Bit[7] *Reserved (must be 0)* Bits[6:5]DMA channel speed supported, _TYP Status 00 Indicates compatibility mode 01 Indicates Type A DMA as described in the EISA Specification 10 Indicates Type B DMA 11 Indicates Type F Bits[4:3]*Ignored* Bit[2] Logical device bus master status, _BM Status 0 Logical device is not a bus master 1 Logical device is a bus master Bits[1:0]DMA transfer type preference, _SIZ Status 00 8-bit only 01 8- and 16-bit 10 16-bit only 11 *Reserved*

See section 16.2.4.2, “ASL Macro for DMA Descriptor,” for a description of the ASL macro that creates a DMA descriptor.

6.4.2.3 Start Dependent Functions (Type 0, Small Item Name 0x6, Length=0 or 1)

Each logical device requires a set of resources. This set of resources may have interdependencies that need to be expressed to allow arbitration software to make resource allocation decisions about the logical device. Dependent functions are used to express these interdependencies. The data structure definitions for dependent functions are shown here. For a detailed description of the use of dependent functions refer to the next section.

Table 6-11 Start Dependent Functions

Offset	Field Name
Byte 0	Value = 0_0110_00nB (Type = 0, small item name = 0x6, length =(0 or 1))

Start Dependent Function fields may be of length 0 or 1 bytes. The extra byte is optionally used to denote the compatibility or performance/robustness priority for the resource group following the Start DF tag. The compatibility priority is a ranking of configurations for compatibility with legacy operating systems. This is the same as the priority used in the PNPBIOS interface. For example, for compatibility reasons, the preferred configuration for COM1 is IRQ4, I/O 3F8-3FF. The performance/robustness performance is a ranking of configurations for performance and robustness reasons. For example, a device may have a high-performance, bus mastering configuration that may not be supported by legacy operating systems. The bus-mastering configuration would have the highest performance/robustness priority while its polled I/O mode might have the highest compatibility priority.

If the Priority byte is not included, this indicates the dependent function priority is ‘acceptable’. This byte is defined as:

Table 6-12 Start Dependent Function Priority Byte Definition

Bits	Definition
1:0	Compatibility priority. Acceptable values are: 0–Good configuration: Highest Priority and preferred configuration 1–Acceptable configuration: Lower Priority but acceptable configuration 2–Sub-optimal configuration: Functional configuration but not optimal 3–Reserved
3:2	Performance/robustness. Acceptable values are: 0–Good configuration: Highest Priority and preferred configuration 1–Acceptable configuration: Lower Priority but acceptable configuration 2–Sub-optimal configuration: Functional configuration but not optimal 3–Reserved
7:4	Reserved (must be 0)

Notice that if multiple Dependent Functions have the same priority, they are further prioritized by the order in which they appear in the resource data structure. The Dependent Function that appears earliest (nearest the beginning) in the structure has the highest priority, and so on.

See section 16.2.4.3, “ASL Macro for Start-Dependent Function Descriptor,” for a description of the ASL macro that creates a Start Dependent Function descriptor.

6.4.2.4 End Dependent Functions (Type 0, Small Item Name 0x7, Length=0)

Table 6-13 End Dependent Functions

Offset	Field Name
Byte 0	Value = 0_0111_000B (Type = 0, small item name = 0x7 length =0)

Notice that only one End Dependent Function item is allowed per logical device. This enforces the fact that Dependent Functions cannot be nested.

See section 16.2.4.4, “ASL Macro for End-Dependent Functions Descriptor,” for a description of the ASL macro that creates an End Dependent Functions descriptor.

6.4.2.5 I/O Port Descriptor (Type 0, Small Item Name 0x8, Length=7)

There are two types of descriptors for I/O ranges. The first descriptor is a full function descriptor for programmable devices. The second descriptor is a minimal descriptor for old ISA cards with fixed I/O requirements that use a 10-bit ISA address decode. The first type descriptor can also be used to describe fixed I/O requirements for ISA cards that require a 16-bit address decode. This is accomplished by setting the range minimum base address and range maximum base address to the same fixed I/O value.

Table 6-14 I/O Port Descriptor Definition

Offset	Field Name	Definition
Byte 0	I/O port descriptor	Value = 01000111B (Type = 0, Small item name = 0x8, Length = 7)
Byte 1	Information	Bits[7:1] are reserved and must be 0 Bit[0] (_DEC) If set, indicates the logical device decodes 16-bit addresses. If bit[0] is not set, this indicates the logical device only decodes address bits[9:0].
Byte 2	Range minimum base address, _MIN bits[7:0]	Address bits[7:0] of the minimum base I/O address that the card may be configured for.
Byte 3	Range minimum base address, _MIN bits[15:8]	Address bits[15:8] of the minimum base I/O address that the card may be configured for.
Byte 4	Range maximum base address, _MAX bits[7:0]	Address bits[7:0] of the maximum base I/O address that the card may be configured for.
Byte 5	Range maximum base address, _MAX bits[15:8]	Address bits[15:8] of the maximum base I/O address that the card may be configured for.
Byte 6	Base alignment, _ALN	Alignment for minimum base address, increment in 1-byte blocks.
Byte 7	Range length, _LEN	The number of contiguous I/O ports requested.

See section 16.2.4.5, “ASL Macro for I/O Port Descriptor,” for a description of the ASL macro that creates an I/O Port descriptor.

6.4.2.6 Fixed Location I/O Port Descriptor (Type 0, Small Item Name 0x9, Length=3)

This descriptor is used to describe 10-bit I/O locations.

Table 6-15 Fixed-Location I/O Port Descriptor Definition

Offset	Field Name	Definition
Byte 0	Fixed Location I/O port descriptor	Value = 01001011B (Type = 0, Small item name = 0x9, Length = 3)
Byte 1	Range base address, _BAS bits[7:0]	Address bits[7:0] of the base I/O address that the card may be configured for. This descriptor assumes a 10-bit ISA address decode.
Byte 2	Range base address, _BAS bits[9:8]	Address bits[9:8] of the base I/O address that the card may be configured for. This descriptor assumes a 10-bit ISA address decode.
Byte 3	Range length, _LEN	The number of contiguous I/O ports requested.

See section 16.2.4.6, “ASL Macro for Fixed I/O Port Descriptor,” for a description of the ASL macro that creates a Fixed I/O Port descriptor.

6.4.2.7 Vendor Defined (Type 0, Small Item Name 0xE, Length=1-7)

The vendor defined resource data type is for vendor use.

Table 6-16 Vendor-Defined Resource Descriptor Definition

Offset	Field Name
Byte 0	Value = 01110nnnB (Type = 0, small item name = 0xE, length = (1-7))
Byte 1 to 7	Vendor defined

See section 16.2.4.7, “ASL Macro for Short Vendor-Defined Descriptor,” for a description of the ASL macro that creates a short Vendor Defined descriptor.

6.4.2.8 End Tag (Type 0, Small Item Name 0xF, Length 1)

The End tag identifies an end of resource data.

Note: If the checksum field is zero, the resource data is treated as if the checksum operation succeeded. Configuration proceeds normally.

Table 6-17 End Tag Definition

Offset	Field Name
Byte 0	Value = 01111001B (Type = 0, small item name = 0xF, length = 1)
Byte 1	Check sum covering all resource data after the serial identifier. This check sum is generated such that adding it to the sum of all the data bytes will produce a zero sum.

The End Tag is automatically generated by the ASL compiler at the end of the **ResourceTemplate** statement.

6.4.3 Large Resource Data Type

To allow for larger amounts of data to be included in the configuration data structure the large format is shown below. This includes a 16-bit length field allowing up to 64 KB of data.

Table 6-18 Large Resource Data Type Tag Bit Definitions

Offset	Field Name
Byte 0	Value = 1xxxxxxxB (Type = 1, Large item name = xxxxxxx)
Byte 1	Length of data items bits[7:0]
Byte 2	Length of data items bits[15:8]
Bytes 3 to *n*	Actual data items

The following large information items are currently defined for Plug and Play ISA devices:

Table 6-19 Large Resource Items

Large Item Name	Value
24-bit memory range descriptor	0x1
Generic register descriptor	0x2
Reserved	0x3
Vendor defined	0x4
32-bit memory range descriptor	0x5
32-bit fixed location memory range descriptor	0x6
DWORD address space descriptor	0x7
WORD address space descriptor	0x8
Extended IRQ descriptor	0x9
QWORD address space descriptor	0xA
Reserved	0xB–0x7F

6.4.3.1 24-Bit Memory Range Descriptor (Type 1, Large Item Name 0x1)

The 24-bit memory range descriptor describes a device’s memory range resources within a 24-bit address space.

Table 6-20 Large Memory Range Descriptor Definition

Offset	Field Name, ASL Field Name	Definition
Byte 0	Memory range descriptor	Value = 10000001B (Type = 1, Large item name = 0x1)
Byte 1	Length, bits[7:0]	Value = 00001001B (9)
Byte 2	Length, bits[15:8]	Value = 00000000B (0)

Table 6-20 Large Memory Range Descriptor Definition *(continued)*

Offset	Field Name, ASL Field Name	Definition
Byte 3	Information	This field provides extra information about this memory. Bit[7:1] *Ignored* Bit[0] Write status, _RW 1 writeable (read/write) 0 non-writeable (read-only)
Byte 4	Range minimum base address, _MIN bits[7:0]	Address bits[15:8] of the minimum base memory address for which the card may be configured.
Byte 5	Range minimum base address, _MIN bits[15:8]	Address bits[23:16] of the minimum base memory address for which the card may be configured
Byte 6	Range maximum base address, _MAX, bits[7:0]	Address bits[15:8] of the maximum base memory address for which the card may be configured.
Byte 7	Range maximum base address, _MAX, bits[15:8]	Address bits[23:16] of the maximum base memory address for which the card may be configured
Byte 8	Base alignment, _ALN, bits[7:0]	This field contains the lower eight bits of the base alignment. The base alignment provides the increment for the minimum base address. (0x0000 = 64 KB)
Byte 9	Base alignment, _ALN, bits[15:8]	This field contains the upper eight bits of the base alignment. The base alignment provides the increment for the minimum base address. (0x0000 = 64 KB)
Byte 10	Range length, _LEN, bits[7:0]	This field contains the lower eight bits of the memory range length. The range length provides the length of the memory range in 256 byte blocks.
Byte 11	Range length, _LEN, bits[15:8]	This field contains the upper eight bits of the memory range length. The range length field provides the length of the memory range in 256 byte blocks.

Notes: Address bits [7:0] of memory base addresses are assumed to be 0.
A Memory range descriptor can be used to describe a fixed memory address by setting the range minimum base address and the range maximum base address to the same value.
24-bit Memory Range descriptors are used for legacy devices.
Mixing of 24-bit and 32-bit memory descriptors on the same device is not allowed.

See section 16.2.4.8, “ASL Macro for 24-Bit Memory Descriptor,” for a description of the ASL macro that creates a 24-bit Memory descriptor.

6.4.3.2 Vendor Defined (Type 1, Large Item Name 0x4)

The vendor defined resource data type is for vendor use.

Table 6-21 Large Vendor-Defined Resource Descriptor Definition

Offset	Field Name	Definition
Byte 0	Vendor defined	Value = 10000100B (Type = 1, Large item name = 0x4)
Byte 1	Length, bits[7:0]	Lower eight bits of vendor defined data length
Byte 2	Length, bits[15:8]	Upper eight bits of vendor defined data length
N * bytes	Vendor Defined	Vendor defined data bytes

See section 16.2.4.9, “ASL Macro for Long Vendor-Defined Descriptor,” for a description of the ASL macro that creates a long Vendor Defined descriptor.

6.4.3.3 32-Bit Memory Range Descriptor (Type 1, Large Item Name 0x5)

This memory range descriptor describes a device’s memory resources within a 32-bit address space.

Table 6-22 Large 32-Bit Memory Range Descriptor Definition

Offset	Field Name	Definition
Byte 0	Memory range descriptor	Value = 10000101B (Type = 1, Large item name = 0x5)
Byte 1	Length, bits[7:0]	Value = 00010001B (17)
Byte 2	Length, bits[15:8]	Value = 00000000B (0)
Byte 3	Information	This field provides extra information about this memory. Bit[7:1] *Ignored* Bit[0] Write status, _RW 1 writeable (read/write) 0 non-writeable (read-only)
Byte 4	Range minimum base address, _MIN bits[7:0]	Address bits[7:0] of the minimum base memory address for which the card may be configured.
Byte 5	Range minimum base address, _MIN bits[15:8]	Address bits[15:8] of the minimum base memory address for which the card may be configured.
Byte 6	Range minimum base address, _MIN bits[23:16]	Address bits[23:16] of the minimum base memory address for which the card may be configured.
Byte 7	Range minimum base address, _MIN bits[31:24]	Address bits[31:24] of the minimum base memory address for which the card may be configured.

Table 6-22 Large 32-Bit Memory Range Descriptor Definition *(continued)*

Offset	Field Name	Definition
Byte 8	Range maximum base address, _MAX bits[7:0]	Address bits[7:0] of the maximum base memory address for which the card may be configured.
Byte 9	Range maximum base address, _MAX bits[15:8]	Address bits[15:8] of the maximum base memory address for which the card may be configured.
Byte 10	Range maximum base address, _MAX bits[23:16]	Address bits[23:16] of the maximum base memory address for which the card may be configured.
Byte 11	Range maximum base address, _MAX bits[31:24]	Address bits[31:24] of the maximum base memory address for which the card may be configured.
Byte 12	Base alignment, _ALN bits[7:0]	This field contains Bits[7:0] of the base alignment. The base alignment provides the increment for the minimum base address.
Byte 13	Base alignment, _ALN bits[15:8]	This field contains Bits[15:8] of the base alignment. The base alignment provides the increment for the minimum base address.
Byte 14	Base alignment, _ALN bits[23:16]	This field contains Bits[23:16] of the base alignment. The base alignment provides the increment for the minimum base address.
Byte 15	Base alignment, _ALN bits[31:24]	This field contains Bits[31:24] of the base alignment. The base alignment provides the increment for the minimum base address.
Byte 16	Range length, _LEN bits[7:0]	This field contains Bits[7:0] of the memory range length. The range length provides the length of the memory range in 1-byte blocks.
Byte 17	Range length, _LEN bits[15:8]	This field contains Bits[15:8] of the memory range length. The range length provides the length of the memory range in 1-byte blocks.
Byte 18	Range length, _LEN bits[23:16]	This field contains Bits[23:16] of the memory range length. The range length provides the length of the memory range in 1-byte blocks.
Byte 19	Range length, _LEN bits[31:24]	This field contains Bits[31:24] of the memory range length. The range length provides the length of the memory range in 1-byte blocks.

Note: Mixing of 24-bit and 32-bit memory descriptors on the same device is not allowed.

See section 16.2.4.10, “ASL Macro for 32-Bit Memory Descriptor,” for a description of the ASL macro that creates a 32-bit Memory descriptor.

6.4.3.4 32-Bit Fixed Location Memory Range Descriptor (Type 1, Large Item Name 0x6)

This memory range descriptor describes a device's memory resources within a 32-bit address space.

Table 6-23 Large Fixed-Location Memory Range Descriptor Definition

Offset	Field Name	Definition
Byte 0	Memory range descriptor	Value = 10000110B (Type = 1, Large item name = 6)
Byte 1	Length, bits[7:0]	Value = 00001001B (9)
Byte 2	Length, bits[15:8]	Value = 00000000B (0)
Byte 3	Information	This field provides extra information about this memory. Bit[7:1] *Ignored* Bit[0] Write status, _RW 1 writeable (read/write) 0 non-writeable (read-only))
Byte 4	Range base address, _BAS bits[7:0]	Address bits[7:0] of the base memory address for which the card may be configured.
Byte 5	Range base address, _BAS bits[15:8]	Address bits[15:8] of the base memory address for which the card may be configured.
Byte 6	Range base address, _BAS bits[23:16]	Address bits[23:16] of the base memory address for which the card may be configured.
Byte 7	Range base address, _BAS bits[31:24]	Address bits[31:24] of the base memory address for which the card may be configured.
Byte 8	Range length, _LEN bits[7:0]	This field contains Bits[7:0] of the memory range length. The range length provides the length of the memory range in 1-byte blocks.
Byte 9	Range length, _LEN bits[15:8]	This field contains Bits[15:8] of the memory range length. The range length provides the length of the memory range in 1-byte blocks.
Byte 10	Range length, _LEN bits[23:16]	This field contains Bits[23:16] of the memory range length. The range length provides the length of the memory range in 1-byte blocks.
Byte 11	Range length, _LEN bits[31:24]	This field contains Bits[31:24] of the memory range length. The range length provides the length of the memory range in 1-byte blocks.

Note: Mixing of 24-bit and 32-bit memory descriptors on the same device is not allowed.

See section 16.2.4.11, "ASL Macro for 32-Bit Fixed Memory Descriptor," for a description of the ASL macro that creates a 32-bit Fixed Memory descriptor.

6.4.3.5 Address Space Descriptors

The QWORD, DWORD, and WORD Address Space Descriptors are general-purpose structures for describing a variety of types of resources. These resources also include support for advanced server architectures (such as multiple root buses), and resource types found on some RISC processors. These descriptors can describe various kinds of resources. The following table defines the valid combination of each field and how they should be interpreted.

Table 6-24 Valid combination of Address Space Descriptors fields

_LEN	_MIF	_MAF	Definition
0	0	0	Variable size, variable location resource descriptor for _PRS. If _MIF is set, _MIN must be a multiple of (_GRA+1). If _MAF is set, _MAX must be (a multiple of (_GRA+1))-1. OS can pick the resource range that satisfies following conditions: • If _MIF is not set, start address is a multiple of (_GRA+1) and greater or equal to _MIN. Otherwise, start address is _MIN. • If _MAF is not set, end address is (a multiple of (_GRA+1))-1 and less or equal to _MAX. Otherwise, end address is _MAX.
0	0	1	
0	1	0	
0	1	1	(Illegal combination)
Non-0	0	0	Fixed size, variable location resource descriptor for _PRS. _LEN must be a multiple of (_GRA+1). OS can pick the resource range that satisfies following conditions: • Start address is a multiple of (_GRA+1) and greater or equal to _MIN. • End address is (start address+_LEN-1) and less or equal to _MAX.
Non-0	0	1	(Illegal combination)
Non-0	1	0	(Illegal combination)
Non-0	1	1	Fixed size, fixed location resource descriptor. _GRA must be 0 and _LEN must be (_MAX - _MIN +1).

6.4.3.5.1 QWORD Address Space Descriptor (Type 1, Large Item Name 0xA)

The QWORD address space descriptor is used to report resource usage in a 64-bit address space (like memory and I/O).

Table 6-25 QWORD Address Space Descriptor Definition

Offset	Field Name	Definition
Byte 0	QWORD Address Space Descriptor	Value=10001010B (Type = 1, Large item name = 0xA)
Byte 1	Length, bits[7:0]	Variable: Value = 43 (minimum)
Byte 2	Length, bits[15:8]	Variable: Value = 0 (minimum)
Byte 3	Resource Type	Indicates which type of resource this descriptor describes. Defined values are: 0 Memory range 1 I/O range 2 Bus number range 3–255 Reserved
Byte 4	General Flags	Flags that are common to all resource types: Bits[7:4] Reserved (must be 0) Bit[3] _MAF: 1–The specified max address is fixed. 0–The specified max address is not fixed andcan be changed. Bit[2] _MIF: 1–The specified min address is fixed. 0–The specified min address is not fixed andcan be changed. Bit[1] _DEC: 1–This bridge subtractively decodes this address (top level bridges only). 0–This bridge positively decodes this address. Bit[0]. 1–This device consumes this resource. 0–This device produces and consumes this resource.
Byte 5	Type Specific Flags	Flags that are specific to each resource type. The meaning of the flags in this field depends on the value of the Resource Type field (see above).

Table 6-25 QWORD Address Space Descriptor Definition *(continued)*

Offset	Field Name	Definition
Byte 6	Address space granularity, _GRA bits[7:0]	A set bit in this mask means that this bit is decoded. All bits less significant than the most significant set bit must be set. That is, the value of the full Address Space Granularity field (all 32 bits) must be a number (2^n-1).
Byte 7	Address space granularity, _GRA bits[15:8]	
Byte 8	Address space granularity, _GRA bits[23:16]	
Byte 9	Address space granularity, _GRA bits[31:24]	
Byte 10	Address space granularity, _GRA bits[39:32]	
Byte 11	Address space granularity, _GRA bits[47:40]	
Byte 12	Address space granularity, _GRA bits[55:48]	
Byte 13	Address space granularity, _GRA bits[63:56]	
Byte 14	Address range minimum, _MIN bits[7:0]	For bridges that translate addresses, this is the address space on the secondary side of the bridge.
Byte 15	Address range minimum, _MIN bits[15:8]	
Byte 16	Address range minimum, _MIN bits[23:16]	
Byte 17	Address range minimum, _MIN bits[31:24]	
Byte 18	Address range minimum, _MIN bits[39:32]	
Byte 19	Address range minimum, _MIN bits[47:40]	

Table 6-25 QWORD Address Space Descriptor Definition *(continued)*

Offset	Field Name	Definition
Byte 20	Address range minimum, _MIN bits[55:48]	
Byte 21	Address range minimum, _MIN bits[63:56]	
Byte 22	Address range maximum, _MAX bits[7:0]	For bridges that translate addresses, this is the address space on the secondary side of the bridge.
Byte 23	Address range maximum, _MAX bits[15:8]	
Byte 24	Address range maximum, _MAX bits[23:16]	
Byte 25	Address range maximum, _MAX bits[31:24]	
Byte 26	Address range maximum, _MAX bits[39:32]	For bridges that translate addresses, this is the address space on the secondary side of the bridge.
Byte 27	Address range maximum, _MAX bits[47:40]	
Byte 28	Address range maximum, _MAX bits[55:48]	
Byte 29	Address range maximum, _MAX bits[63:56]	
Byte 30	Address Translation offset, _TRA bits[7:0]	For bridges that translate addresses across the bridge, this is the offset that must be added to the address on the secondary side to obtain the address on the primary side. Non-bridge devices must list 0 for all Address Translation offset bits.
Byte 31	Address Translation offset, _TRA bits[15:8]	
Byte 32	Address Translation offset, _TRA bits[23:16]	
Byte 33	Address Translation offset, _TRA bits[31:24]	

Table 6-25 QWORD Address Space Descriptor Definition *(continued)*

Offset	Field Name	Definition
Byte 34	Address Translation offset, _TRA bits[39:32]	
Byte 35	Address Translation offset, _TRA bits[47:40]	
Byte 36	Address Translation offset, _TRA bits[55:48]	
Byte 37	Address Translation offset, _TRA bits[63:56]	
Byte 38	Address length, _LEN bits[7:0]	
Byte 39	Address length, _LEN, bits[15:8]	
Byte 40	Address length, _LEN bits[23:16]	
Byte 41	Address length, _LEN bits[31:24]	
Byte 42	Address length, _LEN bits[39:32]	
Byte 43	Address length, _LEN bits[47:40]	
Byte 44	Address length, _LEN bits[55:48]	
Byte 45	Address length, _LEN bits[63:56]	
Byte 46	Resource Source Index	(Optional) Only present if Resource Source (below) is present. This field gives an index to the specific resource descriptor that this device consumes from in the current resource template for the device object pointed to in Resource Source.
String	Resource Source	(Optional) If present, the device that uses this descriptor consumes its resources from the resources produced by the named device object. If not present, the device consumes its resources out of a global pool. If not present, the device consumes this resource from its hierarchical parent.

See section 16.2.4.12, “ASL Macros for QWORD Address Space Descriptor,” for a description of the ASL macro that creates a QWORD Address Space descriptor.

6.4.3.5.2 DWORD Address Space Descriptor (Type 1, Large Item Name 0x7)

The DWORD address space descriptor is used to report resource usage in a 32-bit address space (like memory and I/O).

Table 6-26 DWORD Address Space Descriptor Definition

Offset	Field Name	Definition
Byte 0	DWORD Address Space Descriptor	Value=10000111B (Type = 1, Large item name = 0x7)
Byte 1	Length, bits[7:0]	Variable: Value = 23 (minimum)
Byte 2	Length, bits[15:8]	Variable: Value = 0 (minimum)
Byte 3	Resource Type	Indicates which type of resource this descriptor describes. Defined values are: 0 Memory range 1 I/O range 2 Bus number range 3-255 Reserved
Byte 4	General Flags	Flags that are common to all resource types: Bits[7:4] Reserved (must be 0) Bit[3] _MAF: 1–The specified max address is fixed. 0–The specified max address is not fixed and can be changed. Bit[2] _MIF: 1–The specified min address is fixed. 0–The specified min address is not fixed and can be changed. Bit[1] _DEC: 1–This bridge subtractively decodes this address (top level bridges only). 0–This bridge positively decodes this address. Bit[0] 1–This device consumes this resource. 0–This device produces and consumes this resource.
Byte 5	Type Specific Flags	Flags that are specific to each resource type. The meaning of the flags in this field depends on the value of the Resource Type field (see above).

Table 6-26 DWORD Address Space Descriptor Definition *(continued)*

Offset	Field Name	Definition
Byte 6	Address space granularity, _GRA bits[7:0]	A set bit in this mask means that this bit is decoded. All bits less significant than the most significant set bit must be set. (in other words, the value of the full Address Space Granularity field (all 32 bits) must be a number (2^n-1).
Byte 7	Address space granularity, _GRA bits[15:8]	
Byte 8	Address space granularity, _GRA bits [23:16]	
Byte 9	Address space granularity, _GRA bits [31:24]	
Byte 10	Address range minimum, _MIN bits [7:0]	For bridges that translate addresses, this is the address space on the secondary side of the bridge.
Byte 11	Address range minimum, _MIN bits [15:8]	
Byte 12	Address range minimum, _MIN bits [23:16]	
Byte 13	Address range minimum, _MIN bits [31:24]	
Byte 14	Address range maximum, _MAX bits [7:0]	For bridges that translate addresses, this is the address space on the secondary side of the bridge.
Byte 15	Address range maximum, _MAX bits [15:8]	
Byte 16	Address range maximum, _MAX bits [23:16]	
Byte 17	Address range maximum, _MAX bits [31:24]	
Byte 18	Address Translation offset, _TRA bits [7:0]	For bridges that translate addresses across the bridge, this is the offset that must be added to the address on the secondary side to obtain the address on the primary side. Non-bridge devices must list 0 for all Address Translation offset bits.

Table 6-26 DWORD Address Space Descriptor Definition *(continued)*

Byte 19	Address Translation offset, _TRA bits [15:8]	
Byte 20	Address Translation offset, _TRA bits [23:16]	
Byte 21	Address Translation offset, _TRA bits [31:24]	
Byte 22	Address Length, _LEN, bits [7:0]	
Byte 23	Address Length, _LEN, bits [15:8]	
Byte 24	Address Length, _LEN, bits [23:16]	
Byte 25	Address Length, _LEN, bits [31:24]	
Byte 26	Resource Source Index	(Optional) Only present if Resource Source (below) is present. This field gives an index to the specific resource descriptor that this device consumes from in the current resource template for the device object pointed to in Resource Source.
String	Resource Source	(Optional) If present, the device that uses this descriptor consumes its resources from the resources produced by the named device object. If not present, the device consumes its resources out of a global pool. If not present, the device consumes this resource from its hierarchical parent.

See section 16.2.4.13, “ASL Macro for DWORD Address Space Descriptor,” for a description of the ASL macro that creates a DWORD Address Space descriptor.

6.4.3.5.3 WORD Address Space Descriptor (Type 1, Large Item Name 0x8)

The WORD address space descriptor is used to report resource usage in a 16-bit address space (like memory and I/O).

Note: This descriptor is exactly the same as the DWORD descriptor specified in Table 6-23; the only difference is that the address fields are 16 bits wide rather than 32 bits wide.

Table 6-27 WORD Address Space Descriptor Definition

Offset	Field Name	Definition
Byte 0	WORD Address Space Descriptor	Value=10001000B (Type = 1, Large item name = 0x8)
Byte 1	Length, bits[7:0]	Variable: Value = 13 (minimum)
Byte 2	Length, bits[15:8]	Variable: Value = 0 (minimum)
Byte 3	Resource Type	Indicates which type of resource this descriptor describes. Defined values are: 0 Memory range 1 I/O range 2 Bus number range 3-255 Reserved
Byte 4	General Flags	Flags that are common to all resource types: Bits[7:4] Reserved (must be 0) Bit[3] _MAF: 1–The specified max address is fixed. 0–The specified max address is not fixed and can be changed. Bit[2] _MIF: 1–The specified min address is fixed. 0–The specified min address is not fixed and can be changed. Bit[1] _DEC: 1–This bridge subtractively decodes this address (top level bridges only). 0–This bridge positively decodes this address. Bit[0] 1–This device consumes this resource. 0–This device produces and consumes this resource.
Byte 5	Type Specific Flags	Flags that are specific to each resource type. The meaning of the flags in this field depends on the value of the Resource Type field (see above).

Table 6-27 WORD Address Space Descriptor Definition *(continued)*

Offset	Field Name	Definition
Byte 6	Address space granularity, _GRA bits[7:0]	A set bit in this mask means that this bit is decoded. All bits less significant than the most significant set bit must be set. (in other words, the value of the full Address Space Granularity field (all 16 bits) must be a number (2^n-1).
Byte 7	Address space granularity, _GRA bits[15:8]	
Byte 8	Address range minimum, _MIN bits [7:0]	For bridges that translate addresses, this is the address space on the secondary side of the bridge.
Byte 9	Address range minimum, _MIN bits [15:8]	
Byte 10	Address range maximum, _MAX bits [7:0]	For bridges that translate addresses, this is the address space on the secondary side of the bridge.
Byte 11	Address range maximum, _MAX bits [15:8]	
Byte 12	Address Translation offset, _TRA bits [7:0]	For bridges that translate addresses across the bridge, this is the offset that must be added to the address on the secondary side to obtain the address on the primary side. Non-bridge devices must list 0 for all Address Translation offset bits.
Byte 13	Address Translation offset, _TRA bits [15:8]	
Byte 14	Address Length, _LEN, bits [7:0]	
Byte 15	Address Length, _LEN, bits [15:8]	

Table 6-27 WORD Address Space Descriptor Definition *(continued)*

Offset	Field Name	Definition
Byte 16	Resource Source Index	(Optional) Only present if Resource Source (below) is present. This field gives an index to the specific resource descriptor that this device consumes from in the current resource template for the device object pointed to in Resource Source.
String	Resource Source	(Optional) If present, the device that uses this descriptor consumes its resources from the resources produced by the named device object. If not present, the device consumes its resources out of a global pool. If not present, the device consumes this resource from its hierarchical parent.

See section 16.2.4.14, “ASL Macro for WORD Address Descriptor,” for a description of the ASL macro that creates a WORD address descriptor.

6.4.3.5.4 Resource Type Specific Flags

The meaning of the flags in the Type Specific Flags field of the Address Space Descriptors depends on the value of the Resource Type field in the descriptor. The flags for each resource type are defined in the following tables:

Table 6-28 Memory Resource Flag (Resource Type = 0) Definitions

Bits	Meaning
Bits[7:6]	Reserved (must be 0)
Bit[5]	Memory to I/O Translation, _TTP 1–TypeTranslation: This resource, which is memory on the secondary side of the bridge, is I/O on the primary side of the bridge. 0–TypeStatic: This resource, which is memory on the secondary side of the bridge, is also memory on the primary side of the bridge.
Bits[4:3]	Memory attributes, _MTP. These bits are only defined if this memory resource describes system RAM. For a definition of the labels described here, see section 15, “System Address Map Interfaces.” Value and Meaning 0 AddressRangeMemory 1 AddressRangeReserved 2 AddressRangeACPI 3 AddressRangeNVS

Table 6-28 Memory Resource Flag (Resource Type = 0) Definitions *(continued)*

Bits	Meaning
Bits[2:1]	Memory attributes, _MEM Value andMeaning 0 The memory is non-cacheable. 1 The memory is cacheable. 2 The memory is cacheable and supports write combining. 3 The memory is cacheable and prefetchable.
Bit[0]	Write status, _RW 1–This memory range is read-write. 0–This memory range is read-only.

Table 6-29 I/O Resource Flag (Resource Type = 1) Definitions

Bits	Meaning
Bits[7:6]	Reserved (must be 0)
Bit[5]	Sparse Translation, _TRS. This bit is only meaningful if Bit[4] is set. 1–SparseTranslation: The primary-side memory address of any specific I/O port within the secondary-side range can be found using the following function. address = (((port & 0xfffc) << 10) || (port & 0xfff)) + _TRA In the address used to access the I/O port, bits[11:2] must be identical to bits[21:12], this gives four bytes of I/O ports on each 4 KB page. 0–DenseTranslation: The primary-side memory address of any specific I/O port within the secondary-side range can be found using the following function. address = port + _TRA
Bit[4]	I/O to Memory Translation, _TTP 1 -– TypeTranslation: This resource, which is I/O on the secondary side of the bridge, is memory on the primary side of the bridge. 0–TypeStatic: This resource, which is I/O on the secondary side of the bridge, is also I/O on the primary side of the bridge.
Bit[3:2]	Reserved (must be 0)

Table 6-29 I/O Resource Flag (Resource Type = 1) Definitions *(continued)*

Bits	Meaning
Bit[1]	_RNG This flag is for bridges on systems with multiple bridges. Setting this bit means the memory window specified in this descriptor is limited to the ISA I/O addresses that fall within the specified window. The ISA I/O ranges are: n000-n0FF, n400-n4FF, n800-n8FF, nC00-nCFF. This bit can only be set for bridges entirely configured through ACPI namespace.
Bit[0]	_RNG This flag is for bridges on systems with multiple bridges. Setting this bit means the memory window specified in this descriptor is limited to the non-ISA I/O addresses that fall within the specified window. The non-ISA I/O ranges are: n100-n3FF, n500-n7FF, n900-nBFF, nD00-nFFF. This bit can only be set for bridges entirely configured through ACPI namespace.

Table 6-30 Bus Number Range Resource Flag (Resource Type = 2) Definitions

Bits	Meaning
Bit[7:0]	Reserved (must be 0)

6.4.3.6 Extended Interrupt Descriptor (Type 1, Large Item Name 0x9)

The Extended Interrupt Descriptor is necessary to describe interrupt settings and possibilities for systems that support interrupts above 15.

To specify multiple interrupt numbers, this descriptor allows vendors to list an array of possible interrupt numbers, any one of which can be used.

Table 6-31 Extended Interrupt Descriptor Definition

Offset	Field Name	Definition
Byte 0	Extended Interrupt Descriptor	Value=10001001B (Type = 1, Large item name = 0x9)
Byte 1	Length, bits[7:0]	Variable: Value = 6 (minimum)
Byte 2	Length, bits[15:8]	Variable: Value = 0 (minimum)

Table 6-31 Extended Interrupt Descriptor Definition *(continued)*

Offset	**Field Name**	**Definition**
Byte 3	Interrupt Vector Flags	Interrupt Vector Information. Bit[7:4] Reserved (must be 0) Bit[3] Interrupt is shareable, _SHR Bit[2] Interrupt Polarity, _LL 0–Active-High: This interrupt is sampled when the signal is high, or true. 1–Active-Low: This interrupt is sampled when the signal is low, or false. Bit[1] Interrupt Mode, _HE 0–Level-Triggered: This interrupt is triggered in response to the signal being in either a high or low state. 1–Edge-Triggered: This interrupt is triggered in response to a change in signal state, either high to low or low to high. Bit[0] 1–This device consumes this resource. 0–This device produces and consumes this resource.
Byte 4	Interrupt table length	Indicates the number of interrupt numbers that follow. When this descriptor is returned from _CRS, or when OSPM passes this descriptor to _SRS, this field must be set to 1.
Byte 4*n*+5	Interrupt Number, _INT bits [7:0]	Interrupt number
Byte 4*n*+6	Interrupt Number, _INT bits [15:8]	
Byte 4*n*+7	Interrupt Number, _INT bits [23:16]	
Byte 4*n*+8	Interrupt Number, _INT bits [31:24]	
…	…	Additional interrupt numbers

Table 6-31 Extended Interrupt Descriptor Definition *(continued)*

Offset	**Field Name**	**Definition**
Byte *x*	Resource Source Index	(Optional) Only present if Resource Source (below) is present. This field gives an index to the specific resource descriptor that this device consumes from in the current resource template for the device object pointed to in Resource Source.
String	Resource Source	(Optional) If present, the device that uses this descriptor consumes its resources from the resources produces by the named device object. If not present, the device consumes its resources out of a global pool. If not present, the device consumes this resource from its hierarchical parent.

Note: Low true, level sensitive interrupts may be electrically shared, the process of how this might work is beyond the scope of this specification.

If the OS is running using the 8259 interrupt model, only interrupt number values of 0-15 will be used, and interrupt numbers greater than 15 will be ignored.

See section 16.2.4.15, “ASL Macro for Extended Interrupt Descriptor,” for a description of the ASL macro that creates an Extended Interrupt descriptor.

6.4.3.7 Generic Register Descriptor (Type 1, Large Item Name 0x2)

The generic register descriptor describes the location of a fixed width register within any of the ACPI-defined address spaces.

Table 6-32 Generic Register Descriptor Definition

Offset	**Field Name, ASL Field Name**	**Definition**
Byte 0	Generic register descriptor	Value = 10000010B (Type = 1, Large item name = 0x2)
Byte 1	Length, bits[7:0]	Value = 00001100B (12)
Byte 2	Length, bits[15:8]	Value = 00000000B (0)
Byte 3	Address Space ID, _ASI	The address space where the data structure or register exists. Defined values are: 0–System Memory 1–System I/O 2–PCI Configuration Space 3–Embedded Controller 4–SMBus 0x7F–Functional Fixed Hardware
Byte 4	Register Bit Width, _RBW	Indicates the register width in bits.
Byte 5	Register Bit Offset, _RBO	Indicates the offset to the start of the register in bits from the Register Address.

Table 6-32 Generic Register Descriptor Definition *(continued)*

Offset	Field Name, ASL Field Name	Definition
Byte 6	Reserved	Must be 0.
Byte 7	Register Address, _ADR bits[7:0]	Register Address
Byte 8	Register Address, _ADR bits[15:8]	
Byte 9	Register Address, _ADR bits[23:16]	
Byte 10	Register Address, _ADR bits[31:24]	
Byte 11	Register Address, _ADR bits[39:32]	
Byte 12	Register Address, _ADR bits[47:40]	
Byte 13	Register Address, _ADR bits[55:48]	
Byte 14	Register Address, _ADR bits[63:56]	

See section 16.2.4.16, “ASL Macro for Generic Register Descriptor,” for a description of the ASL macro that creates a Generic Register descriptor.

6.5 Other Objects and Control Methods

Table 6-33 Other Objects and Methods

Object	Description
_INI	Device initialization method that is run shortly after ACPI has been enabled.
_DCK	Indicates that the device is a docking station.
_BDN	Correlates a docking station between ACPI and legacy interfaces.
_REG	Notifies AML code of a change in the availability of an operation region.
_BBN	PCI bus number set up by the BIOS.
_SEG	Indicates a bus segment location.
_GLK	Indicates the Global Lock must be acquired when accessing a device.

6.5.1 _INI (Init)

_INI is a device initialization object that performs device specific initialization. This control method is located under a device object and is run only when OSPM loads a description table. There are restrictions related to when this method is called and governing writing code for this method. The _INI method must only access Operation Regions that have been indicated to available as defined by the _REG method. The _REG method is described in section 6.5.4, “_REG (Region).” This control method is run before _ADR, _CID, _HID, _SUN, and _UID are run.

If the _STA method indicates that the device is present, OSPM will evaluate the __INI for the device (if the _INI method exists) and will examine each of the children of the device for _INI methods. If the _STA method indicates that the device is not present, OSPM will not run the _INI and will not examine the children of the device for _INI methods. If the device becomes present after the table has already been loaded, OSPM will not evaluate the _INI method, nor examine the children for _INI methods.

The _INI control method is generally used to switch devices out of a legacy operating mode. For example, BIOSes often configure CardBus controllers in a legacy mode to support legacy operating systems. Before enumerating the device with an ACPI operating system, the CardBus controllers must be initialized to CardBus mode. For such systems, the vendor can include an _INI control method under the CardBus controller to switch the device into CardBus mode.

In addition to device initialization, OSPM unconditionally evaluates an _INI object under the _SB namespace, if present, at the beginning of namespace initialization.

6.5.2 _DCK (Dock)

This control method is located in the device object that represents the docking station (that is, the device object with all the _EJx control methods for the docking station). The presence of _DCK indicates to the OS that the device is really a docking station.

_DCK also controls the isolation logic on the docking connector. This allows an OS to prepare for docking before the bus is activated and devices appear on the bus.

Arguments:

Arg0

1–Dock (that is, remove isolation from connector)

0–Undock (isolate from connector)

Return Code:

1 if successful, 0 if failed.

Note: When _DCK is called with 0, OSPM will ignore the return value. The _STA object that follows the _EJx control method will notify whether or not the portable has been ejected.

6.5.3 _BDN (BIOS Dock Name)

_BDN is used to correlate a docking station reported via ACPI and the same docking station reported via legacy interfaces. It is primarily used for upgrading over non-ACPI environments.

_BDN must appear under a device object that represents the dock, that is, the device object with _Ejx methods. This object must return a DWORD that is the EISA-packed DockID returned by the Plug and Play BIOS Function 5 (Get Docking Station Identifier) for a dock.

Note: If the machine does not support PNPBIOS, this object is not required.

6.5.4 _REG (Region)

The OS runs _REG control methods to inform AML code of a change in the availability of an operation region. When an operation region handler is unavailable, AML cannot access data fields in that region. (Operation region writes will be ignored and reads will return indeterminate data.).

Except for the cases shown below, control methods must assume all operation regions inaccessible until the _REG(RegionSpace, 1) method is executed. Once _REG has been executed for a particular operation region, indicating that the operation region handler is ready, a control method can access fields in the operation region. Conversely, control methods must not access fields in operation regions when _REG method execution has not indicated that the operation region handler is ready.

For example, until the Embedded Controller driver is ready, the control methods cannot access the Embedded Controller. Once OSPM has run _REG(EmbeddedControl, 1), the control methods can then access operation regions in Embedded Controller address space. Furthermore, if OSPM executes _REG(EmbeddedControl, 0), control methods must stop accessing operation regions in the Embedded Controller address space.

The exceptions for this rule are:

1. OSPM must guarantee that the following operation regions must always be accessible:
   - PCI_Config operation regions on a PCI root bus containing a _BBN object.
   - I/O operation regions.
   - Memory operation regions when accessing memory returned by the System Address Map reporting interfaces.
2. OSPM must make Embedded Controller operation regions, accessed via the Embedded Controllers described in ECDT, available before executing any control method. These operation regions may become inaccessible after OSPM runs _REG(EmbeddedControl, 0).

Place _REG in the same scope as operation region declarations. The OS will run the _REG in a given scope when the operation regions declared in that scope are available for use.

For example:

```
Scope(\_SB.PCI0) {
    OperationRegion(OPR1, PCI_Config, ...)
    Method(_REG, 2) {...}  // OSPM executes this when PCIO operation region handler
                           //  status changes
    Device(PCI1) {
        Method(_REG, 2) {...}
        Device(ETH0) {
         OperationRegion(OPR2, PCI_Config, ...)
            Method(_REG,2) {...}
        }
    }
    Device(ISA0) {
        OperationRegion(OPR3, I/O, ...)
      Method(_REG, 2) {...}    // OSPM executes this when ISAO operation region handler
                              //  status changes

        Device(EC0) {
            Name(_HID, EISAID("PNP0C09"))
            OperationRegion(OPR4, EC, ...)
         Method(_REG, 2) {...} // OSPM executes this when EC operation region
                                 //  handler status changes

        }
    }
}
```

When the PCI0 operation region handler is ready, OSPM will run the _REG method declared in PCI0 scope to indicate that PCI Config space operation region access is available within the PCI0 scope (in other words, OPR1 access is allowed). When the ISA0 operation handler is ready, OSPM will run the _REG method in the ISA0 scope to indicate that the I/O space operation region access is available within that scope (in other words, OPR3 access is allowed). Finally, when the Embedded Controller operation region handler is ready, OSPM will run the _REG method in the EC0 scope to indicate that EC space operation region access is available within the EC0 scope (in other words, OPR4 access is allowed). It should be noted that PCI Config Space Operation Regions are ready as soon the host controller or bridge controller has been programmed with a bus number. PCI1's _REG method would not be run until the PCI-PCI bridge has been properly configured. At the same time, the OS will also run ETH0's _REG method since its PCI Config Space would be also available. The OS will again run ETH0's _REG method when the ETH0 device is started. Also, when the host controller or bridge controller is turned off or disabled, PCI Config Space Operation Regions for child devices are no longer available. As such, ETH0's _REG method will be run when it is turned off and will again be run when PCI1 is turned off.

Note: The OS only runs _REG methods that appear in the same scope as operation region declarations that use the operation region type that has just been made available. For example, _REG in the EC device would not be run when the PCI bus driver is loaded since the operation regions declared under EC do not use any of the operation region types made available by the PCI driver (namely, config space, I/O, and memory).

Arguments:

Arg0: Integer: Operation region space:

0–Memory

1–I/O

2–PCI_Config

3–Embedded Controller

4–SMBus

5–CMOS

6–PCIBARTarget

0x80-0xff–OEM region space handler

Arg1: Integer: 1 for connecting the handler, 0 for disconnecting the handler

6.5.5 _BBN (Base Bus Number)

For multi-root PCI machines, _BBN is the PCI bus number that the BIOS assigns. This is needed to access a PCI_Config operation region for the specific bus. The _BBN object must be unique for every host bridge within a segment since it is the PCI bus number.

6.5.6 _SEG (Segment)

The _SEG object indicates a bus segment location. _SEG is a level higher than _BBN. Each segment is composed of up to 256 PCI Buses.

```
Device(ND0) { // this is a node 0
    Name(_HID, “ACPI0004”)

    // Returns the "Current Resources"
    Name(_CRS,
        ResourceTemplate() {
            …
        }
    )

    Device(PCI0) {
        Name(_HID, EISAID(“PNP0A03”))
        Name(_ADR, 0x00000000)
        Name(_SEG, 0) // The buses below the host bridge belong to PCI segment 0
            …
        Name(_BBN, 0)
        …
    }
    Device(PCI1) {
        …
        Name(_SEG, 0) // The buses below the host bridge belong to PCI segment 0
        …
        Name(_BBN, 16)
        …
    }
    …
 }

 Device(ND1) { // this is a node 1
    Name(_HID, “ACPI0004”)

    // Returns the "Current Resources"
    Name(_CRS,
        ResourceTemplate() {
            …
        }
    )


    Device(PCI0) {
        Name(_HID, EISAID(“PNP0A03”))
        Name(_ADR, 0x00000000)
        Name(_SEG, 1) // The buses below the host bridge belong to PCI segment 1
            …
        Name(_BBN, 0)
        …
    }
    Device(PCI1) {
        …
        Name(_SEG, 1) // The buses below the host bridge belong to PCI segment 1
        …
        Name(_BBN, 16)
        …
    }
    …
 }
```

6.5.7 _GLK (Global Lock)

This optional named object is located in a device object. This object returns a value that indicates to any entity that accesses this device (in other words, OSPM or any device driver) whether the Global Lock must be acquired when accessing the device. OS-based device accesses must be performed while in acquisition of the Global Lock when potentially contentious accesses to device resources are performed by non-OS code, such as System Management Mode (SMM)-based code in Intel architecture-based systems.

An example of this device resource contention is a device driver for an SMBus-based device contending with SMM-based code for access to the Embedded Controller, SMB-HC, and SMBus target device. In this case, the device driver must acquire and release the Global Lock when accessing the device to avoid resource contention with SMM-based code that accesses any of the listed resources.

Return Codes:

1 Global Lock required, 0 Global Lock not required

7 Power and Performance Management

This section specifies the device power management objects and system power management objects. OSPM uses these objects to manage the platform by achieving a desirable balance between performance and energy conservation goals.

Device performance states (Px states) are power consumption and capability states within the active (D0) device power state. Performance states allow OSPM to make tradeoffs between performance and energy conservation. Device performance states have the greatest impact when the implementation is such that the states invoke different device efficiency levels as opposed to a linear scaling of performance and energy consumption. Since performance state transitions occur in the active device states, care must be taken to ensure that performance state transitions do not adversely impact the system.

Device performance state objects, when necessary, are defined on a per device class basis as described in the device class specifications (See Appendix A).

The system state indicator objects are also specified in this section.

7.1 Declaring a Power Resource Object

An ASL **PowerResource** statement is used to declare a **PowerResource** object. A Power Resource object refers to a software-controllable power plane, clock plane, or other resource upon which an integrated ACPI power-managed device might rely. Power resource objects can appear wherever is convenient in namespace.

The syntax of a **PowerResource** statement is:

PowerResource(*resourcename, systemlevel, resourceorder***) {NamedList}**

where the *systemlevel* parameter is a number and the *resourceorder* parameter is a numeric constant (a WORD). For a formal definition of the **PowerResource** statement syntax, see section 16, “ACPI Source Language Reference.”

Systemlevel is the lowest power system sleep level OSPM must maintain to keep this power resource on (0 equates to S0, 1 equates to S1, and so on).

Each power-managed ACPI device lists the resources it requires for its supported power levels. OSPM multiplexes this information from all devices and then enables and disables the required Power Resources accordingly. The *resourceorderl* field in the Power Resource object is a unique value per Power Resource, and it provides the system with the order in which Power Resources must be enabled or disabled. Power Resources are enabled from low values to high values and are disabled from high values to low values. The operating software enables or disables all affected Power Resources in any one *resourceorder* level at a time before moving on to the next ordered level. Putting Power Resources in different order levels provides power sequencing and serialization where required.

A Power Resource can have named objects under its Namespace location. For a description of the ACPI-defined named objects for a Power Resource, see section 7.2, "Device Power Management Objects."

The following block of ASL sample code shows a use of **PowerResource**.

```
PowerResource(PIDE, 0, 0) {
    Method(_STA) {
        Return (Xor (GIO.IDEI, One, Zero))   // inverse of isolation
    }
    Method(_ON) {
        Store (One, GIO.IDEP)      // assert power
        Sleep (10)                 // wait 10ms
        Store (One, GIO.IDER)      // de-assert reset#
        Stall (10)                 // wait 10us
        Store (Zero, GIO.IDEI)     // de-assert isolation
    }
    Method(_OFF) {
        Store (One, GIO.IDEI)      // assert isolation
        Store (Zero, GIO.IDER)     // assert reset#
        Store (Zero, GIO.IDEP)     // de-assert power
    }
}
```

7.1.1 Defined Child Objects for a Power Resource

Each power resource object is required to have the following control methods to allow basic control of each power resource. As OSPM changes the state of device objects in the system, the power resources that are needed will also change causing OSPM to turn power resources on and off. To determine the initial power resource settings the _STA method can be used.

Table 7-1 Power Resource Child Objects

Object	Description
_OFF	Set the resource off.
_ON	Set the resource on.
_STA	Object that evaluates to the current on or off state of the Power Resource. 0–OFF, 1–ON

7.1.2 _OFF

This power resource control method puts the power resource into the OFF state. The control method does not complete until the power resource is off. OSPM only turns on or off one resource at a time, so the AML code can obtain the proper timing sequencing by using Stall or Sleep within the ON (or OFF) method to cause the proper sequencing delays between operations on power resources.

Arguments:
None

Result Code:
None

7.1.3 _ON

This power resource control method puts the power resource into the ON state. The control method does not complete until the power resource is on. OSPM only turns on or off one resource at a time, so the AML code can obtain the proper timing sequencing by using Stall or Sleep within the ON (or OFF) method to cause the proper sequencing delays between operations on power resources.

Arguments:
None

Result Code:
None

7.1.4 _STA (Status)

Returns the current ON or OFF status for the power resource.

Arguments:
None

Result Code:
0 indicates the power resource is currently off.
1 indicates the power resource is currently on.

7.2 Device Power Management Objects

For a device that is power-managed using ACPI, a Definition Block contains one or more of the objects found in the table below. Power management of a device is done using two different paradigms:

- Power Resource control
- Device-specific control

Power Resources are resources that could be shared amongst multiple devices. The operating software will automatically handle control of these devices by determining which particular Power Resources need to be in the ON state at any given time. This determination is made by considering the state of all devices connected to a Power Resource.

By definition, a device that is OFF does not have any power resource or system power state requirements. Therefore, device objects do not list power resources for the OFF power state.

For OSPM to put the device in the D3 state, the following must occur:

- All Power Resources no longer referenced by any device in the system must be in the OFF state.
- If present, the _PS3 control method is executed to set the device into the D3 device state.

The only transition allowed from the D3 device state is to the D0 device state.

For many devices the Power Resource control is all that is required; however, device objects may include their own device-specific control method.

These two types of power management controls (through Power Resources and through specific devices) can be applied in combination or individually as required.

For systems that do not control device power states through power plane management, but whose devices support multiple D-states, more information is required by the OS to determine the S-state to D-state mapping for the device. The ACPI BIOS can give this information to OSPM by way of the _S*x*D methods. These methods tell OSPM for S-state "*x*", the highest D-state supported by the device is "*y*." OSPM is allowed to pick a lower D-state for a given S-state, but OSPM is not allowed to exceed the given D-state.

Further rules that apply to device power management objects are:

- For a given S-state, a device cannot be in a higher D-state than its parent device.
- If there exists an ACPI Object to turn on a device (either through _PS*x* or _PR*x* objects), then a corresponding object to turn the device off must also be declared and vice versa.
- If there exists an ACPI Object that controls power (_PS*x* or _PR*x*, where *x* =0, 1, 2, or 3), then methods to set the device into D0 and D3 device states must be present.
- If a mixture of _PS*x* and _PR*x* methods is declared for the device, then the device states supported through _PS*x* methods must be identical to the device states supported through _PR*x* methods. ACPI system firmware may enable device power state control exclusively through _PS*x* (or _PR*x*) method declarations.

Table 7-2 Device Power Management Child Objects

Object	**Description**
_PS0	Control method that puts the device in the D0 device state (device fully on).
_PS1	Control method that puts the device in the D1 device state.
_PS2	Control method that puts the device in the D2 device state.
_PS3	Control method that puts the device in the D3 device state (device off).
_PSC	Object that evaluates to the device's current power state.
_PR0	Object that evaluates to the device's power requirements in the D0 device state (device fully on).
_PR1	Object that evaluates to the device's power requirements in the D1 device state. The only devices that supply this level are those that can achieve the defined D1 device state according to the related device class.
_PR2	Object that evaluates to the device's power requirements in the D2 device state. The only devices that supply this level are those that can achieve the defined D2 device state according to the related device class.
_PRW	Object that evaluates to the device's power requirements in order to wake the system from a system sleeping state.
_PSW	Control method that enables or disables the device's wake function.
_IRC	Object that signifies the device has a significant inrush current draw.
_S1D	Highest D-state supported by the device in the S1 state
_S2D	Highest D-state supported by the device in the S2 state
_S3D	Highest D-state supported by the device in the S3 state
_S4D	Highest D-state supported by the device in the S4 state

7.2.1 _PS0 (Power State 0)

This Control Method is used to put the specific device into its D0 state. This Control Method can only access Operation Regions that are either always available while in a system working state or that are available when the Power Resources references by the _PR0 object are all ON.

Arguments:
None

Result Code:
None

7.2.2 _PS1 (Power State 1)

This control method is used to put the specific device into its D1 state. This control method can only access Operation Regions that are either always available while in a system working state or that are available when the Power Resources references by the _PR1 object are all ON.

Arguments:
None

Result Code:
None

7.2.3 _PS2 (Power State 2)

This control method is used to put the specific device into its D2 state. This control method can only access Operation Regions that are either always available while in a system working state or that are available when the Power Resources references by the _PR2 object are all ON.

Arguments:
None

Result Code:
None

7.2.4 _PS3 (Power State 3)

This control method is used to put the specific device into its D3 state. This control method can only access Operation Regions that are always available while in a system working state.

A device in the D3 state must no longer be using its resources (for example, its memory space and I/O ports are available to other devices).

Arguments:
None

Result Code:
None

7.2.5 _PSC (Power State Current)

This control method evaluates to the current device state. This control method is not required if the device state can be inferred by the Power Resource settings. This would be the case when the device does not require a _PS0, _PS1, _PS2, or _PS3 control method.

Arguments:

None

Result Code:

The result codes are shown in Table 7-3.

Table 7-3 _PSC Control Method Result Codes

Result	**Device State**
0	D0
1	D1
2	D2
3	D3

7.2.6 _PR0 (Power Resources for D0)

This object evaluates to a package of the following definition:

Table 7-4 Power Resource Requirements Package

	Object	Description
1	object reference	Reference to required Power Resource #0
N	object reference	Reference to required Power Resource #N

For OSPM to put the device in the D0 device state, the following must occur:

1. All Power Resources referenced by elements 1 through N must be in the ON state.
2. All Power Resources no longer referenced by any device in the system must be in the OFF state.
3. If present, the _PS0 control method is executed to set the device into the D0 device state.

_PR0 must return the same data each time it is evaluated. All power resources referenced must exist in the namespace.

7.2.7 _PR1 (Power Resources for D1)

This object evaluates to a package as defined in Table 7-3. For OSPM to put the device in the D1 device state, the following must occur:

1. All Power Resources referenced by elements 1 through N must be in the ON state.
2. All Power Resources no longer referenced by any device in the system must be in the OFF state.
3. If present, the _PS1 control method is executed to set the device into the D1 device state.

_PR1 must return the same data each time it is evaluated. All power resources referenced must exist in the namespace.

7.2.8 _PR2 (Power Resources for D2)

This object evaluates to a package as defined in Table 7-3. For OSPM to put the device in the D2 device state, the following must occur:

1. All Power Resources referenced by elements 1 through N must be in the ON state.
2. All Power Resources no longer referenced by any device in the system must be in the OFF state.
3. If present, the _PS2 control method is executed to set the device into the D2 device state.

_PR2 must return the same data each time it is evaluated. All power resources referenced must exist in the namespace.

7.2.9 _PRW (Power Resources for Wake)

This object is only required for devices that have the ability to wake the system from a system sleeping state. This object evaluates to a package of the following definition:

Table 7-5 Wake Power Requirements Package

	Object Type	Description
0	Numeric or package	If the data type of this package element is numeric, then this _PRW package element is the bit index in the GPEx_EN, in the GPE blocks described in the FADT, of the enable bit that is enabled for the wake event. If the data type of this package element is a package, then this _PRW package element is itself a package containing two elements. The first is an object reference to the GPE Block device that contains the GPE that will be triggered by the wake event. The second element is numeric and it contains the bit index in the GPEx_EN, in the GPE Block referenced by the first element in the package, of the enable bit that is enabled for the wake event. For example, if this field is a package then it is of the form: Package() {_SB.PCI0.ISA.GPE, 2}
1	numeric	The lowest power system sleeping state that can be entered while still providing wake functionality.
2	object reference	Reference to required Power Resource #0
N	object reference	Reference to required Power Resource #N

For OSPM to have the defined wake capability properly enabled for the device, the following must occur:

1. All Power Resources referenced by elements 2 through N are put into the ON state.
2. If present, the _PSW control method is executed to set the device-specific registers to enable the wake functionality of the device.

Then, if the system wants to enter a sleeping state:

1. Interrupts are disabled.
2. The sleeping state being entered must be greater or equal to the power state declared in element 1 of the _PRW object.
3. The proper general-purpose register bits are enabled.

The system sleeping state specified must be a state that the system supports (in other words, a corresponding _S*x* object must exist in the namespace).

_PRW must return the same data each time it is evaluated. All power resources referenced must exist in the namespace.

7.2.10 _PSW (Power State Wake)

In addition to _PSR, this control method can be used to enable or disable the device's ability to wake a sleeping system. This control method can only access Operation Regions that are either always available while in a system working state or that are available when the Power Resources references by the _PRW object are all ON. For example, do not put a power plane control for a bus controller within configuration space located behind the bus.

Arguments:

0– Enable / Disable:	0 to disable the device's wake capabilities.
	1 to enable the device's wake capabilities.

Result Code:

None

7.2.11 _IRC (In Rush Current)

The presence of this object signifies that transitioning the device to its D0 state causes a system-significant in-rush current load. In general, such operations need to be serialized such that multiple operations are not attempted concurrently. Within ACPI, this type of serialization can be accomplished with the *resourceorder* parameter of the device's Power Resources; however, this does not serialize ACPI-controlled devices with non-ACPI controlled devices. IRC is used to signify this fact outside of OSPM to OSPM such that OSPM can serialize all devices in the system that have in-rush current serialization requirements. OSPM can only transition one device flagged with _IRC to the D0 state at a time.

7.2.12 _S1D (S1 Device State)

This object evaluates to an integer that conveys to OSPM the highest power (lowest number) D-state supported by this device in the S1 system sleeping state. _S1D must return the same integer each time it is evaluated. This value overrides an S-state to D-state mapping OSPM may ascertain from the device's power resource declarations. See Table 7-2 for the result code.

7.2.13 _S2D (S2 Device State)

This object evaluates to an integer that conveys to OSPM the highest power (lowest number) D-state supported by this device in the S2 system sleeping state. _S2D must return the same integer each time it is evaluated. This value overrides an S-state to D-state mapping OSPM may ascertain from the device's power resource declarations. See Table 7-2 for the result code.

7.2.14 _S3D (S3 Device State)

This object evaluates to an integer that conveys to OSPM the highest power (lowest number) D-state supported by this device in the S3 system sleeping state. _S3D must return the same integer each time it is evaluated. This value overrides an S-state to D-state mapping OSPM may ascertain from the device's power resource declarations. See Table 7-2 for the result code.

7.2.15 _S4D (S4 Device State)

This object evaluates to an integer that conveys to OSPM the highest power (lowest number) D-state supported by this device in the S4 system sleeping state. _S4D must return the same integer each time it is evaluated. This value overrides an S-state to D-state mapping OSPM may ascertain from the device's power resource declarations. See Table 7-2 for the result code.

7.3 OEM-Supplied System-Level Control Methods

An OEM-supplied Definition Block provides some number of controls appropriate for system-level management. These are used by OSPM to integrate to the OEM-provided features. The following table lists the defined OEM system controls that can be provided.

Table 7-6 BIOS-Supplied Control Methods for System-Level Functions

Object	Description
_BFS	Control method executed immediately following a wake event.
_PTS	Control method used to prepare to sleep.
_GTS	Control method executed just prior to setting the sleep enable (SLP_EN) bit.
_S0	Package that defines system _S0 state mode.
_S1	Package that defines system _S1 state mode.
_S2	Package that defines system _S2 state mode.
_S3	Package that defines system _S3 state mode.
_S4	Package that defines system _S4 state mode.
_S5	Package that defines system _S5 state mode.
_WAK	Control method run once awakened.

7.3.1 _BFS (Back From Sleep)

_BFS is an optional control method. If it exists, OSPM must execute the _BFS method immediately following wake from any sleeping state S1, S2, S3, or S4. _BFS allows ACPI system firmware to perform any required system specific functions when returning a system sleep state. OSPM will execute the _GTS control method before performing any other physical I/O or enabling any interrupt servicing upon returning from a sleeping state. A value that indicates the sleeping state from which the system was awoken (in other words, 1=S1, 2=S2, 3=S3, 4=S4) is passed as an argument to the _BFS control method.

The _BFS method must be self-contained (not call other methods). Additionally, _BFS may only access OpRegions that are currently available (see the _REG method for details).

Arguments:

0: The value of the previous sleeping state (1 for S1, 2 for S2, and so on).

7.3.2 _PTS (Prepare To Sleep)

The _PTS control method is executed by the OS at the beginning of the sleep process for S1, S2, S3, S4, and for orderly S5 shutdown. The sleeping state value (1, 2, 3, 4, or 5) is passed to the _PTS control method. Before OSPM notifies native device drivers and prepares the system software for a system sleeping state, it executes this ACPI control method. Thus, this control method can be executed a relatively long time before actually entering the desired sleeping state. In addition, OSPM can abort the sleeping operation without notification to OSPM, in which case another _PTS would occur some time before the next attempt by OSPM to enter a sleeping state.

The _PTS control method cannot modify the current configuration or power state of any device in the system. For example, _PTS would simply store the sleep type in the embedded controller in sequencing the system into a sleep state when the SLP_EN bit is set.

Arguments:

0: The value of the sleeping state (1 for S1, 2 for S2, and so on).

7.3.3 _GTS (Going To Sleep)

_GTS is an optional control method. If it exists, OSPM must execute the _GTS control method just prior to setting the sleep enable (SLP_EN) bit in the PM1 control register when entering the S1, S2, S3, and S4 sleeping states and when entering S5 for orderly shutdown. _GTS allows ACPI system firmware to perform any required system specific functions prior to entering a system sleep state. OSPM will set the sleep enable (SLP_EN) bit in the PM1 control register immediately following the execution of the _GTS control method without performing any other physical I/O or allowing any interrupt servicing. The sleeping state value (1, 2, 3, 4, or 5) is passed as an argument to the _GTS control method. The _GTS method must not attempt to directly place the system into a sleeping state. OSPM performs this function by setting the sleep enable bit upon return from _GTS. In the case of entry into the S5 soft off state however, _GTS may indeed perform operations that place the system into the S5 state as OSPM will not regain control.

The _GTS method must be self-contained (not call other methods). Additionally, _GTS may only access OpRegions that are currently available (see the _REG method for details).

Arguments:

0: The value of the sleeping state (1 for S1, 2 for S2, and so on).

7.3.4 System _S*x* states

All system states supported by the system must provide a package containing the DWORD value of the following format in the static Definition Block. The system states, known as S0–S5, are referenced in the namespace as _S0–_S5 and for clarity the short S*x* names are used unless specifically referring to the named _S*x* object. For each S*x* state, there is a defined system behavior.

Table 7-7 System State Package

Byte Length	Byte Offset	Description
1	0	Value for PM1a_CNT.SLP_TYP register to enter this system state.
1	1	Value for PM1b_CNT.SLP_TYP register to enter this system state. To enter any given state, OSPM must write the PM1a_CNT.SLP_TYP register before the PM1b_CNT.SLP_TYP register.
2	2	Reserved

States S1–S4 represent some system sleeping state. The S0 state is the system working state. Transition into the S0 state from some other system state (such as sleeping) is automatic, and, by virtue that instructions are being executed, OSPM assumes the system to be in the S0 state. Transition into any system sleeping state is only accomplished by the operating software directing the hardware to enter the appropriate state, and the operating software can only do this within the requirements defined in the Power Resource and Bus/Device Package objects.

All run-time system state transitions (for example, to and from the S0 state), except S4 and S5, are done similarly such that the code sequence to do this is the following:

```
/*
    Intel Architecture SetSleepingState example
*/

        ULONG
        SetSystemSleeping (
            IN  ULONG   NewState
        )
        {
        PROCESSOR_CONTEXT  Context;
        ULONG              PowerSeqeunce;
        BOOLEAN            FlushCaches;
        USHORT             SlpTyp;

// Required environment: Executing on the system boot
// processor. All other processors stopped.  Interrupts
// disabled.  All  Power Resources (and devices) are in
// corresponding device state to support NewState.

        // Get h/w attributes for this system state
        FlushCaches= SleepType[NewState].FlushCache;
        SlpTyp     = SleepType[NewState].SlpTyp & SLP_TYP_MASK;

        _asm {
        lea    eax, OsResumeContext
        push   eax                    ; Build real mode handler the resume
        push   offset  sp50           ; context, with eip = sp50
        call   SaveProcessorState

        mov    eax, ResumeVector      ; set firmware’s resume vector
        mov    [eax], offset OsRealModeResumeCode

        mov    edx, PM1a_STS          ;Make sure wake status is clear
        mov    ax, WAK_STS            ; (cleared by asserting the bit
        out    dx, ax                 ; in the status register)

        mov    edx, PM1b_STS          ;
        out    dx, ax                 ;

        and    eax, not SLP_TYP_MASK
        or     eax, SlpTyp            ; set SLP_TYP
        or     ax, SLP_EN             ; set SLP_EN

        cmp    FlushCaches, 0
        jz     short sp10             ; If needed, ensure no dirty data in

        call   FlushProcessorCaches   ; the caches while sleeping

sp10:   mov    edx, PM1a_SLP_TYP      ; get address for PM1a_SLP_TYP
        out    dx, ax                 ; start h/w sequencing
        mov    edx, PM1b_SLP_TYP      ; get address for PM1b_SLP_TYP
        out    dx, ax                 ; start h/w sequencing

        mov    edx, PM1a_STS          ; get address for PM1x_STS
        mov    ecx, PM1b_STS

sp20:   in     ax, dx                 ; wait for WAK status
        xchg   edx, ecx
        test   ax, WAK_STS
        jz     short sp20

sp50:
}
        // Done..
        *ResumeVector = NULL;
        return 0;
        }
```

7.3.4.1 System _S0 State (Working)

While the system is in the S0 state, it is in the system working state. The behavior of this state is defined as:

- The processors are in the C0, C1, C2, or C3 states. The processor-complex context is maintained and instructions are executed as defined by any of these processor states.
- Dynamic RAM context is maintained and is read/write by the processors.
- Devices states are individually managed by the operating software and can be in any device state (D0, D1, D2, or D3).
- Power Resources are in a state compatible with the current device states.

Transition into the S0 state from some system sleeping state is automatic, and by virtue that instructions are being executed OSPM, assumes the system to be in the S0 state.

7.3.4.2 System _S1 State (Sleeping with Processor Context Maintained)

While the system is in the S1 sleeping state, its behavior is the following:

- The processors are not executing instructions. The processor-complex context is maintained.
- Dynamic RAM context is maintained.
- Power Resources are in a state compatible with the system S1 state. All Power Resources that supply a System-Level reference of S0 are in the OFF state.
- Devices states are compatible with the current Power Resource states. Only devices that solely reference Power Resources that are in the ON state for a given device state can be in that device state. In all other cases, the device is in the D3 (off) state[10].
- Devices that are enabled to wake the system and that can do so from their current device state can initiate a hardware event that transitions the system state to S0. This transition causes the processor to continue execution where it left off.

To transition into the S1 state, the OSPM must flush all processor caches.

7.3.4.3 System _S2 State

The S2 sleeping state is logically lower than the S1 state and is assumed to conserve more power. The behavior of this state is defined as:

- The processors are not executing instructions. The processor-complex context is not maintained.
- Dynamic RAM context is maintained.
- Power Resources are in a state compatible with the system S2 state. All Power Resources that supply a System-Level reference of S0 or S1 are in the OFF state.
- Devices states are compatible with the current Power Resource states. Only devices that solely reference Power Resources that are in the ON state for a given device state can be in that device state. In all other cases, the device is in the D3 (off) state.
- Devices that are enabled to wake the system and that can do so from their current device state can initiate a hardware event that transitions the system state to S0. This transition causes the processor to begin execution at its boot location. The BIOS performs initialization of core functions as needed to exit an S2 state and passes control to the firmware resume vector. See section 9.3.2, “BIOS Initialization of Memory,” for more details on BIOS initialization.

Because the processor context can be lost while in the S2 state, the transition to the S2 state requires that the operating software flush all dirty cache to dynamic RAM (DRAM).

[10] Or it is at least assumed to be in the D3 state by its device driver. For example, if the device doesn’t explicitly describe how it can stay in some state non-off state while the system is in a sleeping state, the operating software must assume that the device can lose its power and state.

7.3.4.4 System _S3 State

The S3 state is logically lower than the S2 state and is assumed to conserve more power. The behavior of this state is defined as follows:

- The processors are not executing instructions. The processor-complex context is not maintained.
- Dynamic RAM context is maintained.
- Power Resources are in a state compatible with the system S3 state. All Power Resources that supply a System-Level reference of S0, S1, or S2 are in the OFF state.
- Devices states are compatible with the current Power Resource states. Only devices that solely reference Power Resources that are in the ON state for a given device state can be in that device state. In all other cases, the device is in the D3 (off) state.
- Devices that are enabled to wake the system and that can do so from their current device state can initiate a hardware event that transitions the system state to S0. This transition causes the processor to begin execution at its boot location. The BIOS performs initialization of core functions as necessary to exit an S3 state and passes control to the firmware resume vector. See section 9.3.2, "BIOS Initialization of Memory," for more details on BIOS initialization.

From the software viewpoint, this state is functionally the same as the S2 state. The operational difference can be that some Power Resources that could be left ON to be in the S2 state might not be available to the S3 state. As such, additional devices may need to be in a logically lower D0, D1, D2, or D3 state for S3 than S2. Similarly, some device wake events can function in S2 but not S3.

Because the processor context can be lost while in the S3 state, the transition to the S3 state requires that the operating software flush all dirty cache to DRAM.

7.3.4.5 System _S4 State

While the system is in this state, it is in the system S4 sleeping state. The state is logically lower than the S3 state and is assumed to conserve more power. The behavior of this state is defined as follows:

- The processors are not executing instructions. The processor-complex context is not maintained.
- DRAM context is not maintained.
- Power Resources are in a state compatible with the system S4 state. All Power Resources that supply a System-Level reference of S0, S1, S2, or S3 are in the OFF state.
- Devices states are compatible with the current Power Resource states. In other words, all devices are in the D3 state when the system state is S4.
- Devices that are enabled to wake the system and that can do so from their S4 device state can initiate a hardware event that transitions the system state to S0. This transition causes the processor to begin execution at its boot location.

After OSPM has executed the _PTS control method and has put the entire system state into main memory, there are two ways that OSPM may handle the next phase of the S4 state transition; saving and restoring main memory. The first way is to use the operating system's drivers to access the disks and file system structures to save a copy of memory to disk and then initiate the hardware S4 sequence by setting the SLP_EN register bit. When the system wakes, the firmware performs a normal boot process and transfers control to the OS via the firmware_waking_vector loader. The OS then restores the system's memory and resumes execution.

The alternate method for entering the S4 state is to utilize the BIOS via the S4BIOS transition. The BIOS uses firmware to save a copy of memory to disk and then initiates the hardware S4 sequence. When the system wakes, the firmware restores memory from disk and wakes OSPM by transferring control to the FACS waking vector.

The S4BIOS transition is optional, but any system that supports this mechanism must support entering the S4 state via the direct OS mechanism. Thus the preferred mechanism for S4 support is the direct OS mechanism as it provides broader platform support. The alternate S4BIOS transition provides a way to achieve S4 support on operating systems that do not have support for the direct method.

7.3.4.6 System _S5 State (Soft Off)

The S5 state is similar to the S4 state except that OSPM does not save any context. The system is in the soft off state and requires a complete boot when awakened (BIOS and OS). Software uses a different state value to distinguish between this state and the S4 state to allow for initial boot operations within the BIOS to distinguish whether or not the boot is going to wake from a saved memory image. OSPM will not disable wake events before setting the SLP_EN bit when entering the S5 sleeping state. This provides support for remote management initiatives by enabling Remote Power On (RPO) capability. This is a change from ACPI 1.0 behavior.

An ACPI 2.0-compliant OS must provide an end user accessible mechanism for disabling all wake devices, with the exception of the system power button, from a single point in the user interface.

7.3.5 _WAK (System Wake)

After the system wakes from a sleeping state, it will invoke the _WAK method and pass the sleeping state value that has ended. This operation occurs asynchronously with other driver notifications in the system and is not the first action to be taken when the system wakes. The AML code for this control method issues device, thermal, and other notifications to ensure that OSPM checks the state of devices, thermal zones, and so on, that could not be maintained during the system sleeping state. For example, if the system cannot determine whether a device was inserted or removed from a bus while in the S2 state, the _WAK method would issue a *devicecheck* type of notification for that bus when issued with the sleeping state value of 2 (for more information about types of notifications, see section 5.6.3, “Device Object Notifications”). Notice that a device check notification from the _SB node will cause OSPM to re-enumerate the entire tree[11].

Hardware is not obligated to track the state needed to supply the resulting status; however, this method must return status concerning the last sleep operation initiated by OSPM. The result codes can be used to provide additional information to OSPM or user.

Arguments:

0	The value of the sleeping state (1 for S1, 2 for S2, and so on).

Result Code (2 DWORD package):

Status	Bit field of defined conditions that occurred during sleep.	
	0x00000000	Wake was signaled and was successful
	0x00000001	Wake was signaled but failed due to lack of power.
	0x00000002	Wake was signaled but failed due to thermal condition.
	Other	Reserved
PSS	If non-zero, the effective S-state the power supply really entered.	

This value is used to detect when the targeted S-state was not entered because of too much current being drawn from the power supply. For example, this might occur when some active device’s current consumption pushes the system’s power requirements over the low power supply mark, thus preventing the lower power mode from being entered as desired.

[11] Only buses that support hardware-defined enumeration methods are done automatically at run-time. This would include ACPI-enumerated devices.

8 Processor Control

This section describes OSPM run-time aspects of managing the processor's performance, power consumption, and other controls while the system is in the working state[12]. The major controls over the processors are:

- Processor power states: C0, C1, C2, C3…Cn
- Processor clock throttling
- Processor performance states: P0, P1, … Pn

These controls are used in combination by OSPM to achieve the desired balance of the following sometimes conflicting goals:

- Performance
- Power consumption and battery life
- Thermal requirements
- Noise-level requirements

Because the goals interact with each other, the operating software needs to implement a policy as to when and where tradeoffs between the goals are to be made[13]. For example, the operating software would determine when the audible noise of the fan is undesirable and would trade off that requirement for lower thermal requirements, which can lead to lower processing performance. Each processor control is discussed in the following sections along with how the control interacts with the various goals.

8.1 Processor Power States

ACPI supports placing system processors into one of four power states while in the G0 working state[14]. Processor power states include C0, C1, C2, and C3. The C0 power state is an active power state where the CPU executes instructions. The C1, C2, and C3 power states are processor sleeping states where the processor consumes less power and dissipates less heat than leaving the processor in the C0 state. While in a sleeping state, the processor does not execute any instructions. Each processor sleeping state has a latency associated with entering and exiting that corresponds to the power savings. In general, the longer the entry/exit latency, the greater the power savings when in the state. To conserve power, OSPM places the processor into one of its supported sleeping states when idle. While in the C0 state, ACPI allows the performance of the processor to be altered through a defined "throttling" process and through transitions into multiple performance states (P-states). A diagram of processor power states is provided below.

[12] In any system sleeping state, the processors are not executing instructions (that is, they are not run-time), and the power consumption is fixed as a property of that system state.

[13] A thermal warning leaves room for operating system tradeoffs to occur (to start the fan or to reduce performance), but a critical thermal alert does not occur.

[14] Notice that these CPU states map into the G0 working state. The state of the CPU is undefined in the G3 sleeping state, the C*x* states only apply to the G0 state.

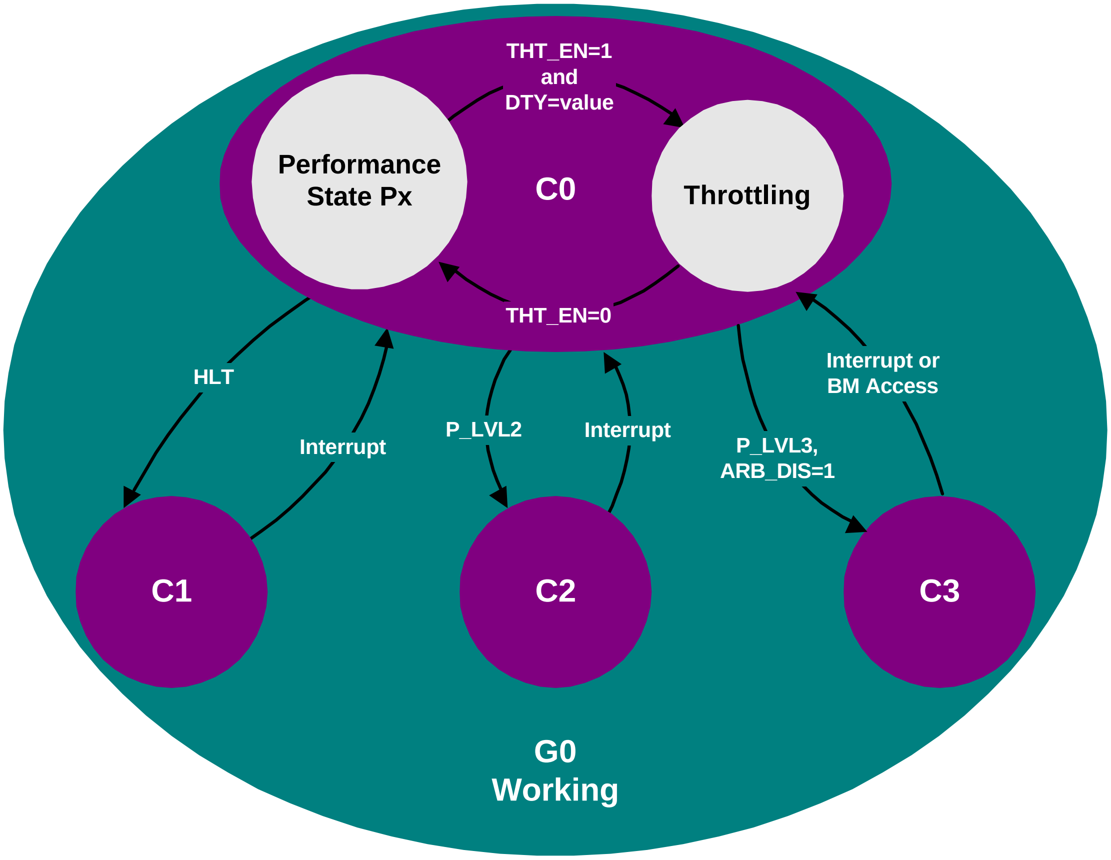

Figure 8-1 Processor Power States

ACPI defines logic on a per-CPU basis that OSPM uses to transition between the different processor power states. This logic is optional, and is described through the FADT table and processor objects (contained in the hierarchical namespace). The fields and flags within the FADT table describe the symmetrical features of the hardware, and the processor object contains the location for the particular CPU's clock logic (described by the P_BLK register block and _CST objects).

The P_LVL2 and P_LVL3 registers provide optional support for placing the system processors into the C2 or C3 states. The P_LVL2 register is used to sequence the selected processor into the C2 state, and the P_LVL3 register is used to sequence the selected processor into the C3 state. Additional support for the C3 state is provided through the bus master status and arbiter disable bits (BM_STS in the PM1_STS register and ARB_DIS in the PM2_CNT register). System software reads the P_LVL2 or P_LVL3 registers to enter the C2 or C3 power state. The Hardware must put the processor into the proper clock state precisely on the read operation to the appropriate P_LVLx register.

Processor power state support is symmetric; OSPM assumes all processors in a system support the same power states. If processors have non-symmetric power state support, then the BIOS will choose and use the lowest common power states supported by all the processors in the system through the FADT table. For example, if the CPU0 processor supports all power states up to and including the C3 state, but the CPU1 processor only supports the C1 power state, then OSPM will only place idle processors into the C1 power state (CPU0 will never be put into the C2 or C3 power states). Notice that the C1 power state must be supported. The C2 and C3 power states are optional (see the PROC_C1 flag in the FADT table description in section 5.2.5, “System Description Table Header”).

The following sections describe processor power states in detail.

8.1.1 Processor Power State C0

While the processor is in the C0 power state, it executes instructions. While in the C0 power state, OSPM can generate a policy to run the processor at less than maximum performance. The clock throttling mechanism provides OSPM with the functionality to perform this task in addition to thermal control. The mechanism allows OSPM to program a value into a register that reduces the processor's performance to a percentage of maximum performance.

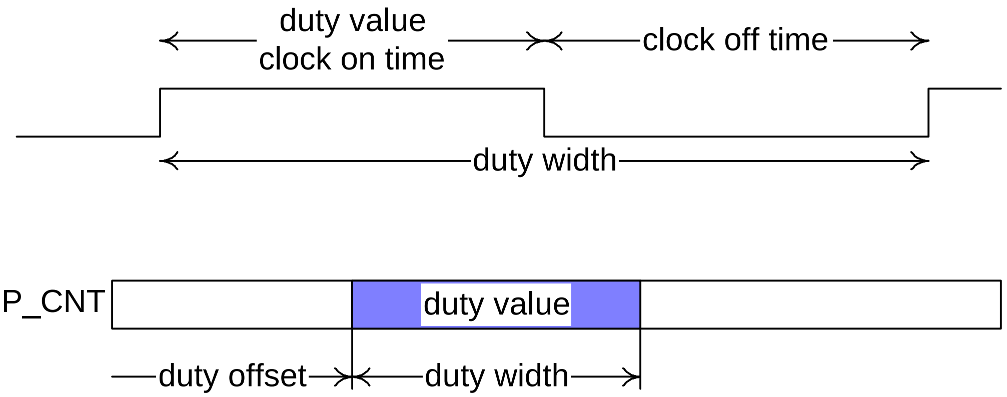

Figure 8-2 Throttling Example

The FADT contains the duty offset and duty width values. The duty offset value determines the offset within the P_CNT register of the duty value. The duty width value determines the number of bits used by the duty value (which determines the granularity of the throttling logic). The performance of the processor by the clock logic can be expressed with the following equation:

$$\% \, Performance = \frac{dutysetting}{2^{dutywidth}} * 100\%$$

Equation 1 Duty Cycle Equation

Nominal performance is defined as "close as possible, but not below the indicated performance level." OSPM will use the duty offset and duty width to determine how to access the duty setting field. OSPM will then program the duty setting based on the thermal condition and desired power of the processor object. OSPM calculates the nominal performance of the processor using the equation expressed in Equation 1. Notice that a *dutysetting* of zero is reserved.

For example, the clock logic could use the stop grant cycle to emulate a divided processor clock frequency on an IA processor (through the use of the STPCLK# signal). This signal internally stops the processor's clock when asserted LOW. To implement logic that provides eight levels of clock control, the STPCLK# pin could be asserted as follows (to emulate the different frequency settings):

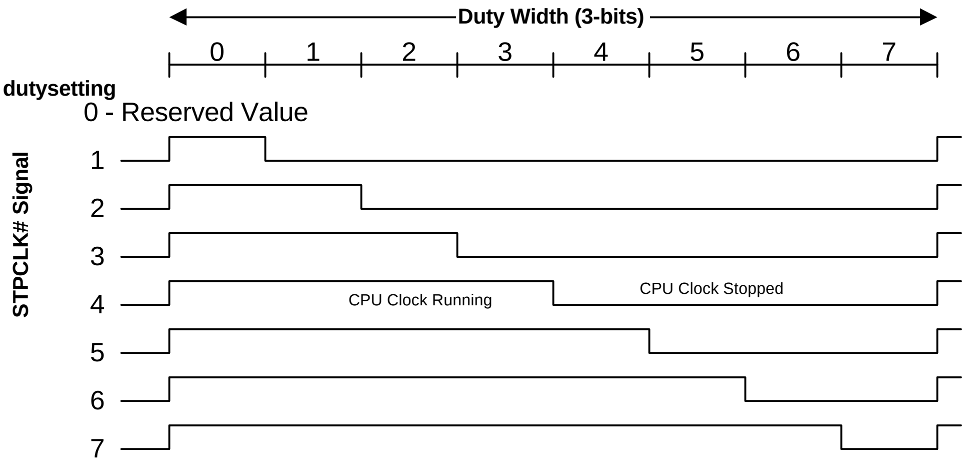

Figure 8-3 Example Control for the STPCLK#

To start the throttling logic OSPM sets the desired duty setting and then sets the THT_EN bit HIGH. To change the duty setting, OSPM will first reset the THT_EN bit LOW, then write another value to the duty setting field while preserving the other unused fields of this register, and then set the THT_EN bit HIGH again.

The example logic model is shown below:

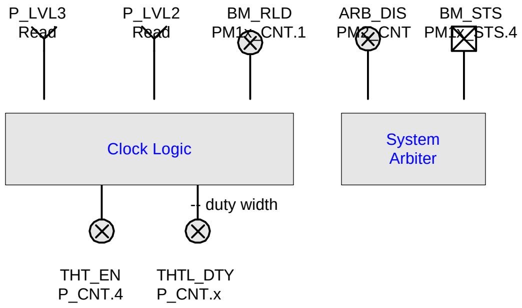

Figure 8-4 ACPI Clock Logic (One per Processor)

Implementation of the ACPI processor power state controls minimally requires the support a single CPU sleeping state (C1). All of the CPU power states occur in the G0/S0 system state; they have no meaning when the system transitions into the sleeping state(S1-S4). ACPI defines the attributes (semantics) of the different CPU states (defines four of them). It is up to the platform implementation to map an appropriate low-power CPU state to the defined ACPI CPU state.

ACPI clock control is supported through the optional processor register block (P_BLK). ACPI requires that there be a unique processor register block for each CPU in the system. Additionally, ACPI requires that the clock logic for multiprocessor systems be symmetrical; if the P0 processor supports the C1, C2, and C3 states, but P1 only supports the C1 state, then OSPM will limit all processors to enter the C1 state when idle.

The following sections define the different ACPI CPU sleeping states.

8.1.2 Processor Power State C1

All processors must support this power state. This state is supported through a native instruction of the processor (HLT forIA 32-bit processors), and assumes no hardware support is needed from the chipset. The hardware latency of this state must be low enough that OSPM does not consider the latency aspect of the state when deciding whether to use it. Aside from putting the processor in a power state, this state has no other software-visible effects. In the C1 power state, the processor is able to maintain the context of the system caches.

The hardware can exit this state for any reason, but must always exit this state when an interrupt is to be presented to the processor.

8.1.3 Processor Power State C2

This processor power state is optionally supported by the system. If present, the state offers improved power savings over the C1 state and is entered by using the P_LVL2 command register for the local processor. The worst-case hardware latency for this state is declared in the FADT and OSPM can use this information to determine when the C1 state should be used instead of the C2 state. Aside from putting the processor in a power state, this state has no other software-visible effects. OSPM assumes the C2 power state has lower power and higher exit latency than the C1 power state.

The C2 power state is an optional ACPI clock state that needs chipset hardware support. This clock logic consists of a P_LVL2 register that, when read, will cause the processor complex to precisely transition into a C2 power state. In a C2 power state, the processor is assumed capable of keeping its caches coherent; for example, bus master and multiprocessor activity can take place without corrupting cache context.

The C2 state puts the processor into a low-power state optimized around multiprocessor and bus master systems. OSPM will cause an idle processor complex to enter a C2 state if there are bus masters or Multiple processor activity (which will prevent OSPM from placing the processor complex into the C3 state). The processor complex is able to snoop bus master or multiprocessor CPU accesses to memory while in the C2 state.

The hardware can exit this state for any reason, but must always exit this state whenever an interrupt is to be presented to the processor.

8.1.4 Processor Power State C3

This processor power state is optionally supported by the system. If present, the state offers improved power savings over the C1 and C2 state and is entered by using the P_LVL3 command register for the local processor. The worst-case hardware latency for this state is declared in the FADT, and OSPM can use this information to determine when the C1 or C2 state should be used instead of the C3 state. While in the C3 state, the processor's caches maintain state but the processor is not required to snoop bus master or multiprocessor CPU accesses to memory.

The hardware can exit this state for any reason, but must always exit this state when an interrupt is to be presented to the processor or when BM_RLD is set and a bus master is attempting to gain access to memory.

OSPM is responsible for ensuring that the caches maintain coherency. In a uniprocessor environment, this can be done by using the PM2_CNT.ARB_DIS bus master arbitration disable register to ensure bus master cycles do not occur while in the C3 state. In a multiprocessor environment, the processors' caches can be flushed and invalidated such that no dynamic information remains in the caches before entering the C3 state.

There are two mechanisms for supporting the C3 power state:

- Having OSPM flush and invalidate the caches prior to entering the C3 state.
- Providing hardware mechanisms to prevent masters from writing to memory (uniprocessor-only support).

In the first case, OSPM will flush the system caches prior to entering the C3 state. As there is normally much latency associated with flushing processor caches, OSPM is likely to only support this in multiprocessor platforms for idle processors. Flushing of the cache is accomplished through one of the defined ACPI mechanisms (described below in section 8.2.4.1, “Flushing Caches”).

In uniprocessor-only platforms that provide the needed hardware functionality (defined in this section), OSPM will attempt to place the platform into a mode that will prevent system bus masters from writing into memory while the processor is in the C3 state. This is accomplished by disabling bus masters prior to entering a C3 power state. Upon a bus master requesting an access, the CPU will awaken from the C3 state and re-enable bus master accesses.

OSPM uses the BM_STS bit to determine the power state to enter when considering a transition to or from the C2/C3 power state. The BM_STS is an optional bit that indicates when bus masters are active. OSPM uses this bit to determine the policy between the C2 and C3 power states: alot of bus master activity demotes the CPU power state to the C2 (or C1 if C2 is not supported), no bus master activity promotes the CPU power state to the C3 power state. OSPM keeps a running history of the BM_STS bit to determine CPU power state policy.

The last hardware feature used in the C3 power state is the BM_RLD bit. This bit determines if the C*x* power state was exited as a result of bus master requests. If set, then the C*x* power state was exited upon a request from a bus master. If reset, the power state was not exited upon bus master requests. In the C3 state, bus master requests need to transition the CPU back to the C0 state (as the system is capable of maintaining cache coherency), but such a transition is not needed for the C2 state. OSPM can optionally set this bit when using a C3 power state, and clear it when using a C1 or C2 power state.

8.1.5 Additional Processor Power States

ACPI 2.0 introduces optional processor power states beyond C3. These power states, C4… Cn, are conveyed to OSPM through the _CST object defined in section 8.3.2, “_CST (C-States).” These additional power states are characterized by equivalent semantics to the C1 through C3 power states, as defined in the previous sections, but with different entry/exit latencies and power savings. See section 8.3.2, “_CST (C-States),” for more information.

8.2 Flushing Caches

To support the C3 power state without using the ARB_DIS feature, the hardware must provide functionality to flush and invalidate the processors’ caches (for an IA processor, this would be the WBINVD instruction). To support the S1, S2 or S3 sleeping states, the hardware must provide functionality to flush the platform caches. Flushing of caches is supported by one of the following mechanisms:

- Processor instruction to write back and invalidate system caches (WBINVD instruction for IA processors).
- Processor instruction to write back but not invalidate system caches (WBINVD instruction for IA processors and some chipsets with partial support; that is, they don’t invalidate the caches).

The ACPI specification expects all platforms to support the local CPU instruction for flushing system caches (with support in both the CPU and chipset), and provides some limited “best effort” support for systems that don’t currently meet this capability. The method used by the platform is indicated through the appropriate FADT fields and flags indicated in this section.

ACPI specifies parameters in the FADT that describe the system’s cache capabilities. If the platform properly supports the processor’s write back and invalidate instruction (WBINVD for IA processors), then this support is indicated to OSPM by setting the WBINVD flag in the FADT.

If the platform supports neither of the first two flushing options, then OSPM can attempt to manually flush the cache if it meets the following criteria:

- A cache-enabled sequential read of contiguous physical memory of not more than 2 MB will flush the platform caches.

There are two additional FADT fields needed to support manual flushing of the caches:

- FLUSH_SIZE, typically twice the size of the largest cache in the system.
- FLUSH_STRIDE, typically the smallest cache line size in the system.

8.3 Declaring a Processor Object

A processor object is declared for each processor in the system using an ASL **Processor** statement. A processor object provides processor configuration information and points to the processor register block (P_BLK).

ACPI 2.0 processor objects are declared under the _SB namespace. This allows OSPM to treat processors in a device-like manner. For example, in a multiprocessor system, processors may be ejected or dynamically inserted. ACPI 2.0-compatible systems may maintain the ACPI 1.0-defined _PR namespace for compatibility with ACPI 1.0 operating systems. An ACPI 2.0-compatible namespace may define Processor objects in either the _SB or _PR scope but not both.

ACPI 2.0 expands the processor object definition by defining processor-specific objects that may be included in the processor object's optional object list. These objects serve multiple purposes including providing alternative definitions for the registers described by the processor register block (P_BLK) and processor performance state control. Additionally, under ACPI 2.0, other ACPI-defined device-related objects may be included in the processor object's object list (for example, the unique identifier object _UID).

With device-like characteristics attributed to processors in ACPI 2.0, it is implied that a processor device driver will be loaded by OSPM to, at a minimum, process device notifications. OSPM will enumerate processors in the system using the ACPI Namespace, processor-specific native identification instructions, and optionally the _HID method.

OSPM will ignore definitions of ACPI-defined objects in an object list of a processor object declared under the _PR namespace. Processor-specific objects are described in the following sections.

For more information on the declaration of the processor object, see section 16.2.3.3.1.16, "PowerResource (Declare Power Resource)."

8.3.1 _PTC (Processor Throttling Control)

_PTC is an optional object used to define a processor throttling control register alternative to the I/O address spaced-based P_BLK throttling control register (P_CNT) described in section 4, "ACPI Hardware Specification. The processor throttling control register mechanism remains as defined in section 8.1.1, " Processor Power State C0."

The _PTC object contains data in the following format:

Name (_PTC, *Processor_Control_Register* //ResourceTemplateTerm-Generic Register Descriptor)

Notice that if the _PTC object exists, the specified register is used instead of the P_CNT register specified in the Processor term. Also notice that if the _PTC object exists and the _CST object does **not** exist, OSPM will use the processor control register from the _PTC object and the P_LVLx registers from the P_BLK.

EXAMPLE
This is an example usage of the _PTC object in a Processor object list:

```
Processor (
    \_SB.CPU0,     // Processor Name
    1,             // ACPI Processor number
    0x120,         // PBlk system IO address
    6 )            // PBlkLen
{   //Object List

    Name(_PTC, ResourceTemplate()

        {
        Register(FFixedHW, 0, 0, 0)
        }

    ) //End of _PTC Object

}   // End of Object List
```

EXAMPLE
This is an example usage of the _PTC object using the values defined in ACPI 1.0. This is an illustrative example to demonstrate the mechanism with well-known values.

```
Processor (
    \_SB.CPU0, // Processor Name
    1,         // ACPI Processor number
    0x120,     // PBLK system IO address
    6 )        // PBLK Len

{   //Object List

    Name(_PTC, // 32 bit wide IO space-based register at the <P_BLK> address
        ResourceTemplate()
        {
            Register(SystemIO, 32, 0, 0x120)
        }

    ) //End of _PTC Object

}   // End of Object List
```

8.3.2 _CST (C States)

_CST is an optional object that provides an alternative method to declare the supported processor power states (C States). Values provided by the _CST object override P_LVLx values in P_BLK and P_LVLx_LAT values in the FADT. The _CST object allows the number of processor power states to be expanded beyond C1, C2, and C3 to an arbitrary number of power states. The entry semantics for these expanded states, (in other words), the considerations for entering these states, are conveyed to OSPM by the C-state_Type field and correspond to the entry semantics for C1, C2, and C3 as described in sections 8.1.2 through 8.1.4. _CST defines ascending C-states characterized by lower power and higher entry/exit latency.

The _CST object evaluates to a package that declares the available C-states as follows:

```
Name (_CST, Package()
{// Field Name              Field Type
C States_Defined,           //ByteConst

Package ()       // C State Definition - 0
{
C State_Register,           //ResourceTemplateTerm-Generic Register Descriptor
C State_Type,               // ByteConst
Latency,                    // WordConst
Power_Consumption           // DWordConst
},
.
.
.
Package ()       // C State Definition - n
{
C State_Register,           //ResourceTemplateTerm-Generic Register Descriptor
C State_Type,               // ByteConst
Latency,                    // WordConst
Power_Consumption           // DWordConst
}

} )        // End of _CST object
```

The C States_Defined field indicates the number of C state entries that follow. Each C State definition entry is a package that describes the C State. A read of the C State_Register places the CPU in the corresponding C State. The Generic Register Descriptor format is described in section 6.4.3.7, “Generic Register Descriptor (Type 1, Large Item Name 0x2).” The description of the remaining package fields is as follows:

- **C State_Type**. The C State type (for example, 0=C0, 1=C1, and so on).This field conveys the semantics used by OSPM when entering the C state.
- **Latency.** The worst-case latency in microseconds to enter and exit the C –State.
- **Power Consumption.** Average power consumption in milliwatts when in the C –State.

Notice that if the _CST object exists, the power states specified in the _CST object are used in lieu of P_LVL2 and P_LVL3 registers defined in P_BLK and the P_LVLx_LAT values defined in the FADT. Also notice that if the _CST object exists and the _PTC object does **not** exist, OSPM will use the processor control register defined in P_BLK and the P_LVLx registers in the _CST object.

The number or type of available C States may change dynamically. As such, ACPI 2.0 supports Notify events on the processor object. Notify events of type 0x81 will cause OSPM to re-evaluate any _CST objects residing under the particular processor object notified. This allows AML code to notify OSPM when the number of supported C States may have changed as a result of an asynchronous event (AC insertion/removal, and so on).

```
The fields in the processor structure remain for backward compatibility.
```

EXAMPLE
This is an example usage of the _CST structure in a Processor structure.

```
Processor (
    \_SB.CPU0,     // Processor Name
    1,             // ACPI Processor number
    0x120,         // PBlk system IO address
    6 )            // PBlkLen
{
    Name(_CST, Package()
    {
    4,         // There are four C-states defined here with three semantics
               // The third and fourth C-states defined have the same C3 entry semantics
    Package(){ResourceTemplate(){Register(FFixedHW, 0, 0, 0)}, 1,  20, 1000},
    Package(){ResourceTemplate(){Register(SystemIO, 8, 0, 0x161)}, 2,  40,  750},
    Package(){ResourceTemplate(){Register(SystemIO, 8, 0, 0x162)}, 3,  60,  500},
    Package(){ResourceTemplate(){Register(SystemIO, 8, 0, 0x163)}, 3, 100,  250}
    })
}
```

EXAMPLE
This is an example usage of the _CST structure using the values defined in ACPI 1.0.

```
Processor (
    \_SB.CPU0,     // Processor Name
    1,             // ACPI Processor number
    0x120,         // PBLK system IO address
    6 )            // PBLK Len
{
    Name(_CST, Package()
    {
    2,         //  There are two C-states defined here – C2 and C3
    Package(){ResourceTemplate(){Register(SystemIO, 8, 0, 0x124)}, 2, 2,  750},
    Package(){ResourceTemplate(){Register(SystemIO, 8, 0, 0x125)}, 3, 65, 500}
    })
}
```

The platform will issue a **Notify**(_SB.CPU0, 0x81*)* to inform OSPM to re-evaluate this object when the number of available processor power states changes.

8.3.3 Processor Performance Control

Processor performance control is implemented through three optional objects whose presence indicates to OSPM that the platform and CPU are capable of supporting multiple performance states. The platform must supply all three objects if processor performance control is implemented. The processor performance control objects define the supported processor performance states, allow the processor to be placed in a specific performance state, and report the number of performance states currently available on the system.

In a multiprocessing environment, all CPUs must support the same number of performance states and each processor performance state must have identical performance and power-consumption parameters. Performance objects must be present under each processor object in the system for OSPM to utilize this feature.

Processor performance control objects include the '_PCT' package, '_PSS' package, and the '_PPC' method as detailed below.

8.3.3.1 _PCT (Performance Control)

This optional object declares an interface that allows OSPM to transition the processor into a performance state. OSPM performs processor performance transitions by writing the performance state–specific control value to a Performance Control Register (PERF_CTRL).

OSPM may select a processor performance state as indicated by the performance state value returned by the _PPC method, or any lower power (higher numbered) state. The control value to write is contained in the corresponding _PSS entry's "Control" field.

Success or failure of the processor performance transition is determined by reading a Performance Status Register (PERF_STATUS) to determine the processor's current performance state. If the transition was successful, the value read from PERF_STATUS will match the "Status" field in the _PSS entry that corresponds to the desired processor performance state.

This object evaluates to a package that declares the above-mentioned transition control and status addresses as follows:

```
Name (_PCT, Package()
{
Perf_Ctrl_Register,      //ResourceTemplateTerm-Generic Register Descriptor
Perf_Status_Register     //ResourceTemplateTerm-Generic Register Descriptor
})       // End of _PCT
```

8.3.3.2 _PSS (Performance Supported States)

This optional object indicates to OSPM the number of supported processor performance states that any given system can support. This object evaluates to a packaged list of information about available performance states including internal CPU core frequency, typical power dissipation, control register values needed to transition between performance states, and status register values that allow OSPM to verify performance transition status after any OS-initiated transition change request. The list is sorted in descending order by typical power dissipation. As a result, the zeroth entry describes the highest performance state and the ‘*n*th’ entry describes the lowest performance state.

```
Name (_PSS, Package()
{// Field Name              Field Type


Package ()        // Performance State 0 Definition – P0
{
CoreFreq,                              // DWordConst
Power,                                 // DWordConst
TransitionLatency,                     // DWordConst
BusMasterLatency,                      // DWordConst
Control,                               // DWordConst
Status                                 // DWordConst
},
.
.
.
Package ()        // Performance State n Definition – Pn
{
CoreFreq,                              // DWordConst
Power,                                 // DWordConst
TransitionLatency,                     // DWordConst
BusMasterLatency,                      // DWordConst
Control,                               // DWordConst
Status                                 // DWordConst
}


} )      // End of _PSS object
```

Each performance state entry contains six data fields as follows:

- ***CoreFreq.*** Indicates the core CPU operating frequency (in MHz).
- ***Power.*** Indicates the typical power dissipation (in milliWatts).
- ***TransitionLatency.*** Indicates the worst-case latency in microseconds that the CPU is unavailable during a transition from any performance state to this performance state.
- ***BusMasterLatency.*** Indicates the worst-case latency in microseconds that Bus Masters are prevented from accessing memory during a transition from any performance state to this performance state.
- ***Control.*** Indicates the value to be written to the Performance Control Register (PERF_CTRL) in order to initiate a transition to the performance state.
- ***Status.*** Indicates the value that OSPM will compare to a value read from the Performance Status Register (PERF_STATUS) to ensure that the transition to the performance state was successful. OSPM may always place the CPU in the lowest power state, but additional states are only available when indicated by the _PPC method.

8.3.3.3 _PPC (Performance Present Capabilities)

This optional object is a method that dynamically indicates to OSPM the number of performance states currently supported by the platform. This method returns a number that indicates the _PSS entry number of the highest performance state that OSPM can use at a given time. OSPM may choose the corresponding state entry in the _PSS as indicated by the value returned by the _PPC method or any lower power (higher numbered) state entry in the _PSS.

Arguments:

None

Returned Value:

Number of states supported (integer)

0 – states 0 .. nth state available (all states available)

1 – state 1 .. nth state available

2 – state 2 .. nth state available

…

n – state *n* available only

In order to support dynamic changes of _PPC object, ACPI 2.0 supports Notify events on the processor object. Notify events of type 0x80 will cause OSPM to reevaluate any _PPC objects residing under the particular processor object notified. This allows AML code to notify OSPM when the number of supported states may have changed as a result of an asynchronous event (AC insertion/removal, docked, undocked, and so on).

8.3.3.4 Processor Performance Control Example

EXAMPLE:

This is an example of processor performance control objects in a processor object list.

In this example, a uniprocessor platform that has processor performance capabilities with support for three performance states as follows:

1. 500 MHz (8.2W) supported at any time
2. 600 MHz (14.9W) supported only when AC powered
3. 650 MHz (21.5W) supported only when docked

It takes no more than 500 microseconds to transition from one performance state to any other performance state.

During a performance transition, bus masters are unable to access memory for a maximum of 300 microseconds.

The PERF_CTRL and PERF_STATUS registers are implemented as Functional Fixed Hardware.

The following ASL objects are implemented within the system:

_SB.DOCK: Evaluates to 1 if system is docked, zero otherwise.

_SB.AC: Evaluates to 1 if AC is connected, zero otherwise.

```
Processor (
\_SB.CPU0,     // Processor Name
1,             // ACPI Processor number
0x120,         // PBlk system IO address
6 )            // PBlkLen
{
Name(_PCT, Package ()  // Performance Control object
{
     ResourceTemplate(){Register(FFixedHW, 0, 0, 0)},       // PERF_CTRL
     ResourceTemplate(){Register(FFixedHW, 0, 0, 0)}        // PERF_STATUS
})  // End of _PCT object

Name (_PSS, Package()
{
 Package(){650, 21500, 500, 300, 0x00, 0x08},   // Performance State zero (P0)
 Package(){600, 14900, 500, 300, 0x01, 0x05},   // Performance State one (P1)
 Package(){500, 8200,  500, 300, 0x02, 0x06}    // Performance State two (P2)
})             // End of _PSS object

Method (_PPC, 0)      // Performance Present Capabilities method
{
         If (\_SB.DOCK)
         {
              Return(0)       // All _PSS states available (650, 600, 500).
         }

         If (\_SB.AC)
         {
              Return(1)       // States 1 and 2 available (600, 500).
         }

         Else
         {
              Return(2)       // State 2 available (500)
         }
    }   // End of _PPC method

}   // End of processor object list
```

The platform will issue a **Notify**(_SB.CPU0, 0x80) to inform OSPM to re-evaluate this object when the number of available processor performance states changes.

9 Waking and Sleeping

ACPI defines a mechanism to transition the system between the working state (G0) and a sleeping state (G1) or the soft-off (G2) state. During transitions between the working and sleeping states, the context of the user's operating environment is maintained. ACPI defines the quality of the G1 sleeping state by defining the system attributes of four types of ACPI sleeping states (S1, S2, S3, and S4). Each sleeping state is defined to allow implementations that can tradeoff cost, power, and wake latencies. Additionally, ACPI defines the sleeping states such that an ACPI platform can support multiple sleeping states, allowing the platform to transition into a particular sleeping state for a predefined period of time and then transition to a lower power/higher wake latency sleeping state (transitioning through the G0 state) [15].

ACPI defines a programming model that provides a mechanism for OSPM to initiate the entry into a sleeping or soft-off state (S1-S5); this consists of a 3-bit field SLP_TYPx[16] that indicates the type of sleep state to enter, and a single control bit SLP_EN to start the sleeping process.

Note: Systems containing processors without a hardware mechanism to place the processor in a low-power state may additionally require the execution of appropriate native instructions to place the processor in a low-power state after OSPM sets the SLP_EN bit. The hardware may implement a number of low-power sleeping states and then associate these states with the defined ACPI sleeping states (through the SLP_TYPx fields). The ACPI system firmware creates a sleeping object associated with each supported sleeping state (unsupported sleeping states are identified by the lack of the sleeping object). Each sleeping object contains two constant 3-bit values that OSPM will program into the SLP_TYPa and SLP_TYPb fields (in fixed register space).

ACPI also defines an alternate mechanism for entering and exiting the S4 state that passes control to the BIOS to save and restore platform context. Context ownership is similar in definition to the S3 state, but hardware saves and restores the context of memory to non-volatile storage (such as a disk drive), and OSPM treats this as an S4 state with implied latency and power constraints. This alternate mechanism of entering the S4 state is referred to as the S4BIOS transition.

Prior to entering a sleeping state (S1-S4), OSPM will execute OEM-specific AML/ASL code contained in the _PTS (Prepare To Sleep) control method. One use of the _PTS control method is that it can indicate to the embedded controller what sleeping state the system will enter when the SLP_EN bit is set. The embedded controller can then respond by executing the proper power-plane sequencing upon this bit being set.

Immediately prior to entering a system sleeping state (as well as the S5 soft-off state), OSPM will execute the _GTS (Going To Sleep) control method. _GTS allows ACPI system firmware to perform any necessary system specific functions prior to entering a system sleeping state.

Upon waking, OSPM will execute the _BFS (Back From Sleep) control method. This allows ACPI system firmware to perform any necessary system specific functions prior to returning control to OSPM. The _WAK (Wake) control method is then executed. This control method again contains OEM-specific AML/ASL code. One use of the _WAK control method requests OSPM to check the platform for any devices that might have been added or removed from the system while the system was asleep. For example, a PC Card controller might have had a PC Card added or removed, and because the power to this device was off in the sleeping state, the status change event was not generated.

[15] OSPM uses the RTC wakeup feature to program in the time transition delay. Prior to sleeping, OSPM will program the RTC alarm to the closest (in time) wakeup event: either a transition to a lower power sleeping state, or a calendar event (to run some application).

[16] Notice that there can be two fixed PM1x_CNT registers, each pointing to a different system I/O space region. Normally a register grouping only allows a bit or bit field to reside in a single register group instance (a or b); however, each platform can have two instances of the SLP_TYP (one for each grouping register: a and b). The _Sx control method gives a package with two values: the first is the SLP_TYPa value and the second is the SLP_TYPb value.

This section discusses the system initialization sequence of an ACPI-enabled platform. This includes the boot sequence, different wake scenarios, and an example to illustrate how to use the system address map reporting interfaces. This sequence is part of the ACPI event programming model.

For detailed information on the power management control methods described above, see section 7, “Power and Performance Management.”

9.1 Sleeping States

The illustration below shows the transitions between the working state, the sleeping states, and the Soft Off state.

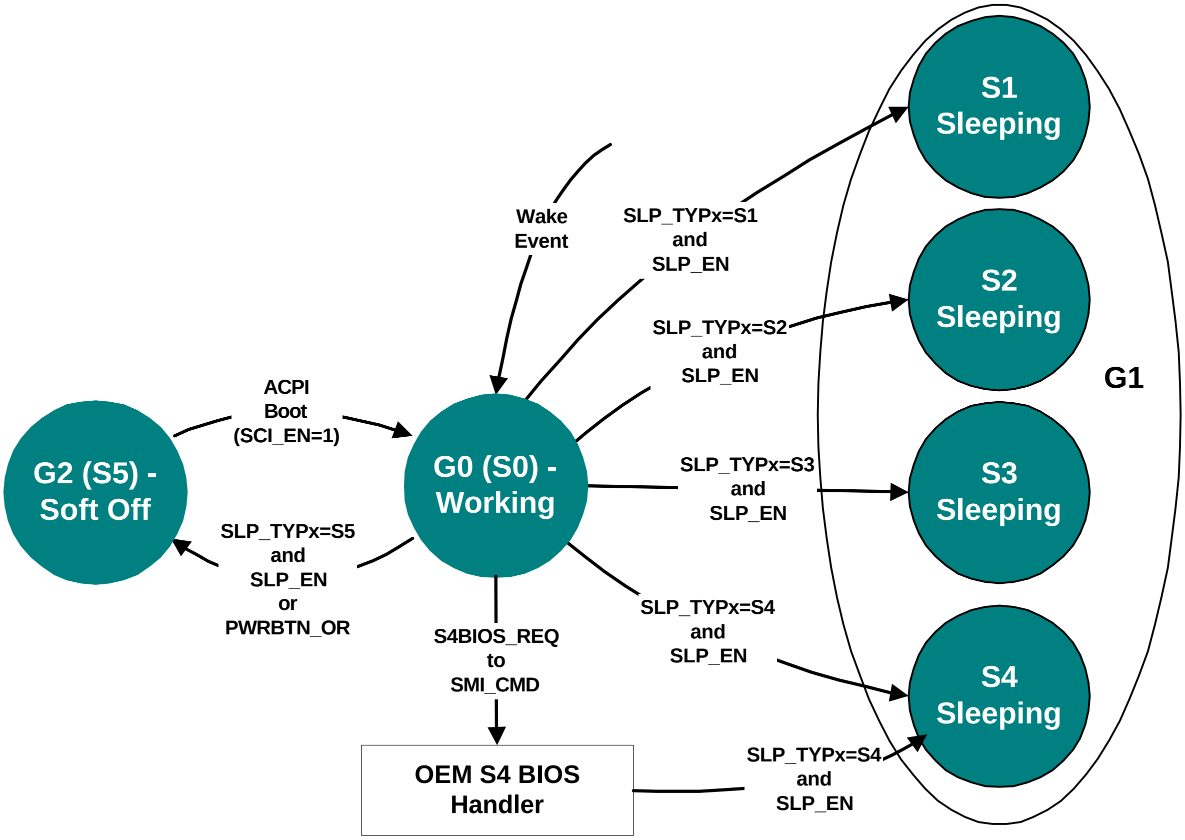

Figure 9-1 Example Sleeping States

ACPI defines distinct differences between the G0 and G1 system states.

- In the G0 state, work is being performed by the OS/application software and the hardware. The CPU or any particular hardware device could be in any one of the defined power states (C0-C3 or D0-D3); however, some work will be taking place in the system.
- In the G1 state, the system is assumed to be doing no work. Prior to entering the G1 state, OSPM will place devices in a device power state compatible with the system sleeping state to be entered; if a device is enabled to wake the system, then OSPM will place these devices into the lowest D*x* state from which the device supports wake. This is defined in the power resource description of that device object. This definition of the G1 state implies:
  - The CPUs execute no instructions in the G1 state.

- Hardware devices are not operating (except possibly to generate a wake event).
- ACPI registers are affected as follows:
  - Wake event bits are enabled in the corresponding fixed or general-purpose registers according to enabled wake options.
  - PM1 control register is programmed for the desired sleeping state.
  - WAK_STS is set by hardware in the sleeping state.

All sleeping states have these specifications. ACPI defines additional attributes that allow an ACPI platform to have up to four different sleeping states, each of which has different attributes. The attributes were chosen to allow differentiation of sleeping states that vary in power, wake latency, and implementation cost tradeoffs.

Running processors at reduced levels of performance is not an ACPI sleeping state (G1); this is a working (G0) state–defined event.

The CPU cannot execute any instructions when in the sleeping state; OSPM relies on this fact. A platform designer might be tempted to support a sleeping system by reducing the clock frequency of the system, which allows the platform to maintain a low-power state while at the same time maintaining communication sessions that require constant interaction (as with some network environments). This is definitely a G0 activity where an OS policy decision has been made to turn off the user interface (screen) and run the processor in a reduced performance mode. This type of reduced performance state as a sleeping state is not defined by the ACPI specification; ACPI assumes no code execution during sleeping states.

ACPI defines attributes for four sleeping states: S1, S2, S3 and S4. (Notice that S4 and S5 are very similar from a hardware standpoint.) At least one sleeping state, S1-S4, must be implemented by an ACPI-compatible system. Platforms can support multiple sleeping states. ACPI specifies that a 3-bit binary number be associated with each sleeping state (these numbers are given objects within ACPI's root namespace: _S0, _S1, _S2, _S3, _S4 and _S5). When entering a system sleeping state, OSPM will do the following:

1. Pick the deepest sleeping state supported by the platform and enabled waking devices.
2. Execute the _PTS control method (which passes the type of intended sleep state to OEM AML code) if it is an S1–S4 sleeping state.
3. If OS policy decides to enter the S4 state and chooses to use the S4BIOS mechanism and S4BIOS is supported by the platform, OSPM will pass control to the BIOS software by writing the S4BIOS_REQ value to the SMI_CMD port.
4. If not using the S4BIOS mechanism, OSPM gets the SLP_TYPx value from the associated sleeping object (_S1, _S2, _S3, _S4 or _S5).
5. Program the SLP_TYP*x* fields with the values contained in the selected sleeping object.
6. Execute the _GTS control method, passing an argument that indicates the sleeping state to be entered (1, 2, 3, or 4 representing S1, S2, S3, and S4).
7. If entering S1, S2, or S3, flush the processor caches.
8. If not entering S4BIOS, set the SLP_EN bit to start the sleeping sequence. (This actually occurs on the same write operation that programs the SLP_TYPx field in the PM1_CNT register.) If entering S4BIOS, write the S4BIOS_REQ value into the SMI_CMD port.
9. On systems containing processors without a hardware mechanism to place the processor in a low-power state, execute appropriate native instructions to place the processor in a low-power state.

The _PTS control method provides the BIOS a mechanism for performing some housekeeping, such as writing the sleep type value to the embedded controller, before entering the system sleeping state. Control method execution occurs "just prior" to entering the sleeping state and is not an event synchronized with the write to the PM1_CNT register. Execution can take place several seconds prior to the system actually entering the sleeping state. As such, no hardware power-plane sequencing takes place by execution of the _PTS control method.

Upon waking, the _BFS control method is executed. OSPM then executes the _WAK control method. This control method executes OEM-specific ASL/AML code that can search for any devices that have been added or removed during the sleeping state.

The following sections describe the sleeping state attributes.

9.1.1 S1 Sleeping State

The S1 state is defined as a low wake-latency sleeping state. In this state, all system context is preserved with the exception of CPU caches. Before setting the SLP_EN bit, OSPM will flush the system caches. If the platform supports the WBINVD instruction (as indicated by the WBINVD and WBINVD_FLUSH flags in the FADT), OSPM will execute the WBINVD instruction. The hardware is responsible for maintaining all other system context, which includes the context of the CPU, memory, and chipset.

Examples of S1 sleeping state implementation alternatives follow.

9.1.1.1 Example 1: S1 Sleeping State Implementation

This example references an IA processor that supports the stop grant state through the assertion of the STPCLK# signal. When SLP_TYPx is programmed to the S1 value (the OEM chooses a value, which is then placed in the _S1 object) and the SLP_ENx bit is subsequently set, the hardware can implement an S1 state by asserting the STPCLK# signal to the processor, causing it to enter the stop grant state.

In this case, the system clocks (PCI and CPU) are still running. Any enabled wake event causes the hardware to de-assert the STPCLK# signal to the processor whereby OSPM must first invalidate the CPU caches and then transition back into the working state.

9.1.1.2 Example 2: S1 Sleeping State Implementation

When SLP_TYPx is programmed to the S1 value and the SLP_ENx bit is subsequently set, the hardware will implement an S1 sleeping state transition by doing the following:

1. Placing the processor into the stop grant state.
2. Stopping the processor’s input clock, placing the processor into the stop clock state.
3. Placing system memory into a self-refresh or suspend-refresh state. Refresh is maintained by the memory itself or through some other reference clock that is not stopped during the sleeping state.
4. Stopping all system clocks (asserts the standby signal to the system PLL chip). Normally the RTC will continue running.

In this case, all clocks in the system have been stopped (except for the RTC). Hardware must reverse the process (restarting system clocks) upon any enabled wake event whereby OSPM must first invalidate the CPU caches and then transition back into the working state.

9.1.2 S2 Sleeping State

The S2 state is defined as a low wake latency sleep state. This state is similar to the S1 sleeping state where any context except for system memory may be lost. Additionally, control starts from the processor’s reset vector after the wake event. Before setting the SLP_EN bit, OSPM will flush the system caches. If the platform supports the WBINVD instruction (as indicated by the WBINVD and WBINVD_FLUSH flags in the FADT), OSPM will execute the WBINVD instruction. The hardware is responsible for maintaining chip set and memory context. An example of an S2 sleeping state implementation follows.

9.1.2.1 Example: S2 Sleeping State Implementation

When the SLP_TYPx register(s) are programmed to the S2 value (found in the _S2 object) and the SLP_EN bit is set, the hardware will implement an S2 sleeping state transition by doing the following:

1. Stopping system clocks (the only running clock is the RTC).
2. Placing system memory into a self-refresh or suspend-refresh state.
3. Powering off the CPU and cache subsystem.

In this case, the CPU is reset upon detection of the wake event; however, core logic and memory maintain their context. Execution control starts from the CPU’s boot vector. The BIOS is required to:

- Program the initial boot configuration of the CPU (such as the CPU’s MSR and MTRR registers).
- Initialize the cache controller to its initial boot size and configuration.
- Enable the memory controller to accept memory accesses.
- Jump to the waking vector.

9.1.3 S3 Sleeping State

The S3 state is defined as a low wake-latency sleep state. From the software viewpoint, this state is functionally the same as the S2 state. The operational difference is that some Power Resources that may have been left ON in the S2 state may not be available to the S3 state. As such, some devices may be in a lower power state when the system is in S3 state than when the system is in the S2 state. Similarly, some device wake events can function in S2 but not S3. An example of an S3 sleeping state implementation follows.

9.1.3.1 Example: S3 Sleeping State Implementation

When the SLP_TYPx register(s) are programmed to the S3 value (found in the _S3 object) and the SLP_EN bit is set, the hardware will implement an S3 sleeping state transition by doing the following:

1. Placing the memory into a low-power auto-refresh or self-refresh state.
2. Devices that are maintaining memory isolating themselves from other devices in the system.
3. Removing power from the system. At this point, only devices supporting memory are powered (possibly partially powered). The only clock running in the system is the RTC clock.

In this case, the wake event repowers the system and resets most devices (depending on the implementation).

Execution control starts from the CPU’s boot vector. The BIOS is required to:

1. Program the initial boot configuration of the CPU (such as the MSR and MTRR registers).
2. Initialize the cache controller to its initial boot size and configuration.
3. Enable the memory controller to accept memory accesses.
4. Jump to the waking vector.

Notice that if the configuration of cache memory controller is lost while the system is sleeping, the BIOS is required to reconfigure it to either the pre-sleeping state or the initial boot state configuration. The BIOS can store the configuration of the cache memory controller into the reserved memory space, where it can then retrieve the values after waking. OSPM will call the _PTS method once per session (prior to sleeping).

The BIOS is also responsible for restoring the memory controller’s configuration. If this configuration data is destroyed during the S3 sleeping state, then the BIOS needs to store the pre-sleeping state or initial boot state configuration in a non-volatile memory area (as with RTC CMOS RAM) to enable it to restore the values during the waking process.

When OSPM re-enumerates buses coming out of the S3 sleeping state, it will discover any devices that have been inserted or removed, and configure devices as they are turned on.

9.1.4 S4 Sleeping State

The S4 sleeping state is the lowest-power, longest wake-latency sleeping state supported by ACPI. In order to reduce power to a minimum, it is assumed that the hardware platform has powered off all devices. Because this is a sleeping state, the platform context is maintained. Depending on how the transition into the S4 sleeping state occurs, the responsibility for maintaining system context changes. S4 supports two entry mechanisms: OS initiated and BIOS-initiated. The OSPM-initiated mechanism is similar to the entry into the S1-S3 sleeping states; OSPM driver writes the SLP_TYPx fields and sets the SLP_EN bit. The BIOS-initiated mechanism occurs by OSPM transferring control to the BIOS by writing the S4BIOS_REQ value to the SMI_CMD port.

In OSPM-initiated S4 sleeping state, OSPM is responsible for saving all system context. Before entering the S4 state, OSPM will save context of all memory with the exception of memory reported as typeAddressRangeReserved (see section 15, “System Address Map Interfaces,” for more information). Upon waking, OSPM will then restore the system context. When OSPM re-enumerates buses coming out of the S4 sleeping state, it will discover any devices that have come and gone, and configure devices as they are turned on.

In the BIOS-initiated S4 sleeping state, OSPM is responsible for the same system context as described in the S3 sleeping state (BIOS restores the memory and some chip set context). The S4BIOS transition transfers control to the BIOS, allowing it to save context to non-volatile memory (such as a disk partition).

9.1.4.1 Operating System-Initiated S4 Transition

If OSPM supports OSPM-initiated S4 transition, it will not generate a BIOS-initiated S4 transition. Platforms that support the BIOS-initiated S4 transition also support OSPM-initiated S4 transition.

OSPM-initiated S4 transition is initiated by OSPM by saving system context, writing the appropriate values to the SLP_TYPx register(s), and setting the SLP_EN bit. Upon exiting the S4 sleeping state, the BIOS restores the chipset to its POST condition, updates the hardware signature (described later in this section), and passes control to OSPM through a normal boot process.

When the BIOS builds the ACPI tables, it generates a hardware signature for the system. If the hardware configuration has changed during an OS-initiated S4 transition, the BIOS updates the hardware signature in the FACS table. A change in hardware configuration is defined to be any change in the platform hardware that would cause the platform to fail when trying to restore the S4 context; this hardware is normally limited to boot devices. For example, changing the graphics adapter or hard disk controller while in the S4 state should cause the hardware signature to change. On the other hand, removing or adding a PC Card device from a PC Card slot should not cause the hardware signature to change.

9.1.4.2 The S4BIOS Transition

The BIOS-initiated S4 transition begins with OSPM writing the S4BIOS_REQ value into the SMI_CMD port (as specified in the FADT). Once gaining control, the BIOS then saves the appropriate memory and chip set context, and then places the platform into the S4 state (power off to all devices).

In the FACS memory table, there is the S4BIOS_F bit that indicates hardware support for the BIOS-initiated S4 transition. If the hardware platform supports the S4BIOS state, it sets the S4BIOS_F flag within the FACS memory structure prior to booting the OS. If the S4BIOS_F flag in the FACS table is set, this indicates that OSPM can request the BIOS to transition the platform into the S4BIOS sleeping state by writing the S4BIOS_REQ value (found in the FADT) to the SMI_CMD port (identified by the SMI_CMD value in the FADT).

Upon waking the BIOS, software restores memory context and jumps to the waking vector (similar to wake from an S3 state). Coming out of the S4BIOS state, the BIOS must only configure boot devices (so it can read the disk partition where it saved system context). When OSPM re-enumerates buses coming out of the S4BIOS state, it will discover any devices that have come and gone, and configure devices as they are turned on.

9.1.5 S5 Soft Off State

The S5 soft off state is used by OSPM to turn the machine off. Notice that *the S5 state is not a sleeping state* (it is a G2 state) and no context is saved by OSPM or hardware. Also notice that from a hardware perspective, the S4 and S5 states are nearly identical. When initiated, the hardware will sequence the system to a state similar to the off state. The hardware has no responsibility for maintaining any system context (memory or I/O); however, it does allow power on due to a power button press or wake event (Remote Power On event). Upon power on, the BIOS does normal power-on reset, loads the boot sector, and executes (but not the waking vector, as all ACPI table context is lost when entering the S5 state).

9.1.6 Transitioning from the Working to the Sleeping State

On a transition of the system from the working to the sleeping state, the following occurs:

1. OSPM decides (through a policy scheme) to place the system into the sleeping state.
2. OSPM examines all devices enabled to wake the system and determines the deepest possible sleeping state the system can enter to support the enabled wake functions. The _PRW named object under each device is examined, as well as the power resource object it points to.
3. OSPM places all device drivers into their respective D*x* state. If the device is enabled for wake, it enters the D*x* state associated with the wake capability. If the device is not enabled to wake the system, it enters the D3 state.
4. OSPM executes the _PTS control method, passing an argument that indicates the desired sleeping state (1, 2, 3, or 4 representing S1, S2, S3, and S4).
5. OSPM saves any other processor's context (other than the local processor) to memory.
6. OSPM writes the waking vector into the FACS table in memory.
7. OSPM executes the _GTS control method, passing an argument that indicates the sleeping state to be entered (1, 2, 3, or 4 representing S1, S2, S3, and S4).
8. OSPM clears the WAK_STS in the PM1a_STS and PM1b_STS registers.
9. OSPM saves the local processor's context to memory.
10. OSPM flushes caches (only if entering S1, S2 or S3).
11. OSPM sets GPE enable registers to ensure that all appropriate wake signals are armed.
12. If entering an S4 state using the S4BIOS mechanism, OSPM writes the S4BIOS_REQ value (from the FADT) to the SMI_CMD port. This passes control to the BIOS, which then transitions the platform into the S4BIOS state.
13. If not entering an S4BIOS state, then OSPM writes SLP_TYPa (from the associated sleeping object) with the SLP_ENa bit set to the PM1a_CNT register.
14. OSPM writes SLP_TYPb with the SLP_EN bit set to the PM1b_CNT register.
15. On systems containing processors without a hardware mechanism to place the processor in a low-power state, OSPM executes appropriate native instructions to place the processor in a low-power state.
16. OSPM loops on the WAK_STS bit (in both the PM1a_CNT and PM1b_CNT registers).
17. The system enters the specified sleeping state.

Note: this is accomplished after step 14 or 15 above.

9.1.7 Transitioning from the Working to the Soft Off State

On a transition of the system from the working to the soft off state, the following occurs:

1. OSPM executes the _PTS control method, passing the argument 5.
2. OSPM prepares its components to shut down (flushing disk caches).
3. OSPM executes the _GTS control method, passing the argument 5.
4. OSPM writes SLP_TYPa (from the _S5 object) with the SLP_ENa bit set to the PM1a_CNT register.
5. OSPM writes SLP_TYPb (from the _S5 object) with the SLP_ENb bit set to the PM1b_CNT register.
6. The system enters the Soft Off state.

9.2 Flushing Caches

Before entering the S1, S2 or S3 sleeping states, OSPM is responsible for flushing the system caches. ACPI provides a number of mechanisms to flush system caches. These include:

- Using a native instruction (for example, the IA32 WBINVD instruction) to flush and invalidate platform caches.
  WBINVD_FLUSH flag set (1) in the FADT indicates the system provides this support level.
- Using the IA32 instruction WBINVD to flush but **not** invalidate the platform caches.
  WBINVD flag set (1) in the FADT indicates the system provides this support level.

The manual flush mechanism has two caveats:

- Largest cache is 1 MB in size (FLUSH_SIZE is a maximum value of 2 MB).
- No victim caches (for which the manual flush algorithm is unreliable).

Processors with built-in victim caches will not support the manual flush mechanism and are therefore required to support the WBINVD mechanism to use the S2 or S3 state.

The manual cache-flushing mechanism relies on the two FADT fields:

- **FLUSH_SIZE.** Indicates twice the size of the largest cache in bytes.
- **FLUSH_STRIDE.** Indicates the smallest line size of the caches in bytes.

The cache flush size value is typically twice the size of the largest cache size, and the cache flush stride value is typically the size of the smallest cache line size in the platform. OSPM will flush the system caches by reading a contiguous block of memory indicated by the cache flush size.

9.3 Initialization

This section covers the initialization sequences for an ACPI platform. After a reset or wake from an S2, S3, or S4 sleeping state (as defined by the ACPI sleeping state definitions), the CPU will start execution from its boot vector. At this point, the initialization software has many options, depending on what the hardware platform supports. This section describes at a high level what should be done for these different options. Figure 9-2 illustrates the flow of the boot-up software.

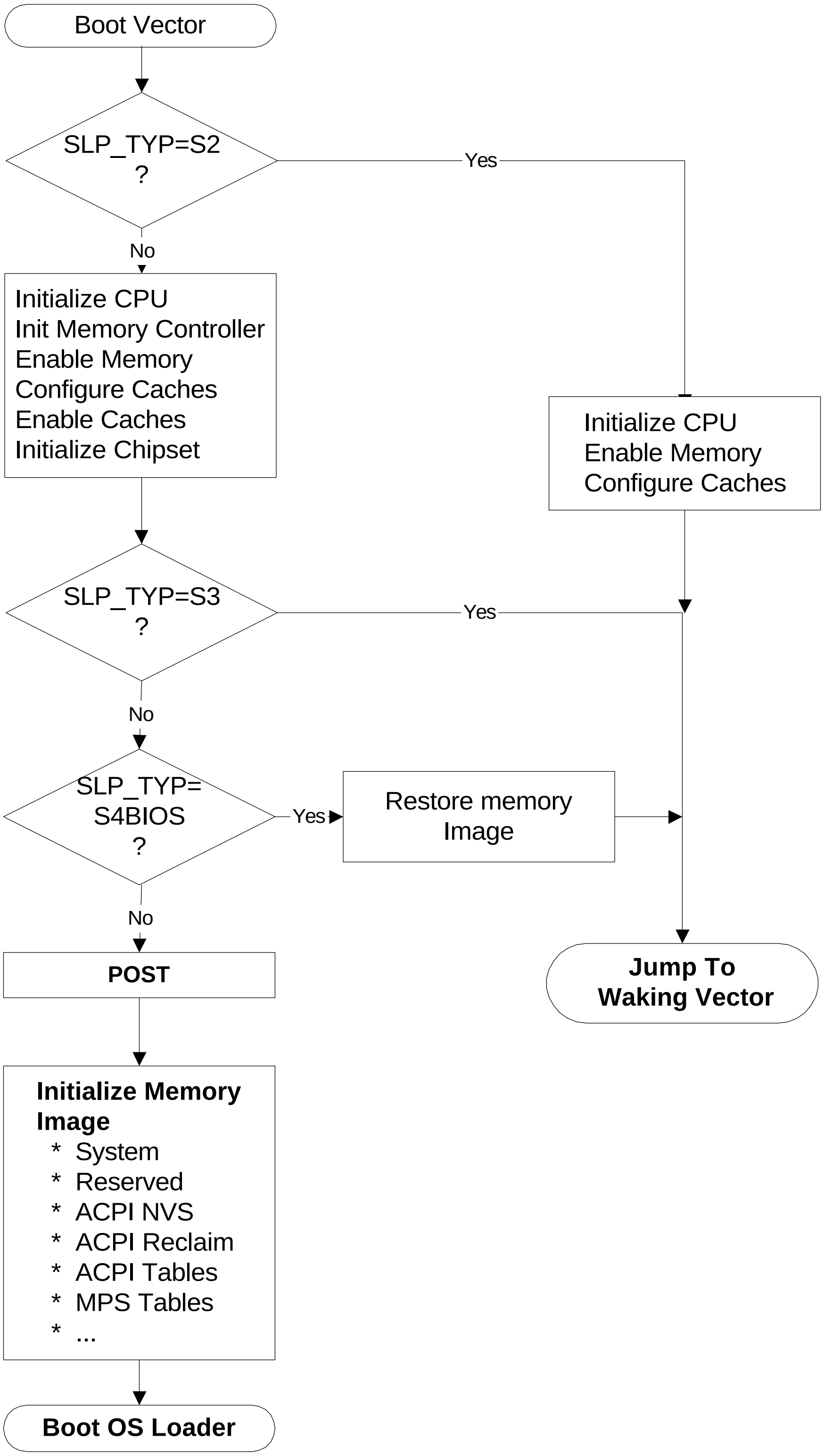

Figure 9-2 BIOS Initialization

The processor will start executing at its power-on reset vector when waking from an S2, S3, or S4 sleeping state, during a power-on sequence, or as a result of a hard or soft reset.

When executing from the power-on reset vector as a result of a power-on sequence, a hard or soft reset, **or waking from an S4 sleep state**, the platform firmware performs complete hardware initialization; placing the system in a boot configuration. The firmware then passes control to the operating system boot loader.

When executing from the power-on reset vector as a result of waking from an S2 or S3 sleep state, the platform firmware performs only the hardware initialization required to restore the system to either the state the platform was in prioir to the initial operating system boot, or to the pre-sleep configuration state. In multiprocessor systems, non-boot processors should be placed in the same state as prior to the initial operating system boot. The platform firmware then passes control back to OSPM system by jumping to either the Firmware_Waking_Vector or the X_Firmware_Waking_Vector in the FACS (see table 5-11 for more information). The contents of operating system memory contents may not be changed during the S2 or S3 sleep state.

First, the BIOS determines whether this is a wake from S2 or S3 by examining the SLP_TYP register value, which is preserved between sleeping sessions. If this is an S2 or S3 wake, then the BIOS restores minimum context of the system before jumping to the waking vector. This includes:

- **CPU configuration.** BIOS restores the pre-sleep configuration or initial boot configuration of each CPU (MSR, MTRR, BIOS update, SMBase, and so on). Interrupts must be disabled (for IA-32 processors, disabled by CLI instruction).
- **Memory controller configuration.** If the configuration is lost during the sleeping state, the BIOS initializes the memory controller to its pre-sleep configuration or initial boot configuration.
- **Cache memory configuration.** If the configuration is lost during the sleeping state, the BIOS initializes the cache controller to its pre-sleep configuration or initial boot configuration.
- **Functional device configuration.** The BIOS doesn't need to configure/restore context of functional devices such as a network interface (even if it is physically included in chipset) or interrupt controller. OSPM is responsible for restoring all context of these devices. The only requirement for the hardware and BIOS is to ensure that interrupts are not asserted by devices when the control is passed to OS.
- **ACPI registers.** SCI_EN bit must be set. All event status/enable bits (PM1x_STS, PM1x_EN, GPEx_STS and GPEx_EN) must not be changed by BIOS.

Note: The BIOS may reconfigure the CPU, memory controller and cache memory controller to either the pre-sleeping configuration or the initial boot configuration. OSPM must accommodate both configurations.

When waking from an S4BIOS sleeping state, the BIOS initializes a minimum number of devices such as CPU, memory, cache, chipset and boot devices. After initializing these devices, the BIOS restores memory context from non-volatile memory such as hard disk, and jumps to waking vector.

As mentioned previously, waking from an S4 state is treated the same as a cold boot: the BIOS runs POST and then initializes memory to contain the ACPI system description tables. After it has finished this, it can call OSPM loader, and control is passed to OSPM.

When waking from S4 (either S4OS or S4BIOS), the BIOS may optionally set SCI_EN bit before passing control to OSPM. In this case, interrupts must be disabled (for IA-32 processors, disabled CLI instruction) until the control is passed to OSPM and the chipset must be configured in ACPI mode.

9.3.1 Placing the System in ACPI Mode

When a platform initializes from a cold boot (mechanical off or from an S4 or S5 state), the hardware platform may be configured in a legacy configuration. From these states, the BIOS software initializes the computer as it would for a legacy operating system. When control is passed to the operating system, OSPM will check the SCI_EN bit and if it is not set will then enable ACPI mode by first finding the ACPI tables, and then by generating a write of the ACPI_ENABLE value to the SMI_CMD port (as described in the FADT). The hardware platform will set the SCI_EN bit to indicate to OSPM that the hardware platform is now configured for ACPI.

Note: Before SCI is enabled, no SCI interrupt can occur. Nor can any SCI interrupt occur immediately after ACPI is on. The SCI interrupt can only be signaled after OSPM has enabled one of the GPE/PM1 enable bits.

When the platform is waking from an S1, S2 or S3 state, OSPM assumes the hardware is already in the ACPI mode and will not issue an ACPI_ENABLE command to the SMI_CMD port.

9.3.2 BIOS Initialization of Memory

During a power-on reset, an exit from an S4 sleeping state, or an exit from an S5 soft-off state, the BIOS needs to initialize memory. This section explains how the BIOS should configure memory for use by a number of features including:

- ACPI tables.
- BIOS memory that wants to be saved across S4 sleeping sessions and should be cached.
- BIOS memory that does not require saving and should be cached.

For example, the configuration of the platform's cache controller requires an area of memory to store the configuration data. During the wake sequence, the BIOS will re-enable the memory controller and can then use its configuration data to reconfigure the cache controllers. To support these three items, IA-PC-based systems contain system address map reporting interfaces that return the following memory range types:

- **ACPI Reclaim Memory.** Memory identified by the BIOS that contains the ACPI tables. This memory can be any place above 8 MB and contains the ACPI tables. When OSPM is finished using the ACPI tables, it is free to reclaim this memory for system software use (application space).
- **ACPI Non-Volatile-Sleeping Memory (NVS).** Memory identified by the BIOS as being reserved by the BIOS for its use. OSPM is required to tag this memory as cacheable, and to save and restore its image before entering an S4 state. Except as directed by control methods, OSPM is not allowed to use this physical memory. OSPM will call the _PTS control method some time before entering a sleeping state, to allow the platform's AML code to update this memory image before entering the sleeping state. After the system awakes from an S4 state, OSPM will restore this memory area and call the _WAK control method to enable the BIOS to reclaim its memory image.

Note: The memory information returned from the system address map reporting interfaces should be the same before and after an S4 sleep.

When the system is first booting, OSPM will invoke E820 interfaces on IA-PC-based legacy systems or the GetMemoryMap() interface on EFI-enabled systems to obtain a system memory map (see section 15, "System Address Map Interfaces," for more information). As an example, the following memory map represents a typical IA-PC-based legacy platform's physical memory map.

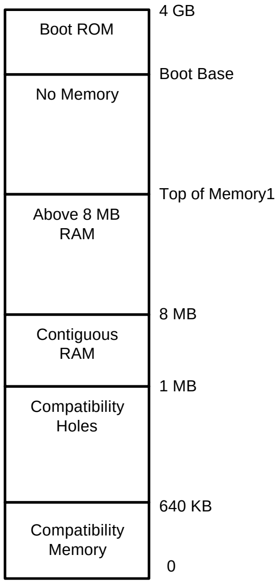

Figure 9-3 Example Physical Memory Map

The names and attributes of the different memory regions are listed below:

- **0–640 KB.** Compatibility Memory. Application executable memory for an 8086 system.
- **640 KB–1 MB.** Compatibility Holes. Holes within memory space that allow accesses to be directed to the PC-compatible frame buffer (A0000h-BFFFFh), to adapter ROM space (C0000h-DFFFFh), and to system BIOS space (E0000h-FFFFFh).
- **1 MB–8 MB.** Contiguous RAM. An area of contiguous physical memory addresses. Operating systems may require this memory to be contiguous in order for its loader to load the OS properly on boot up. (No memory-mapped I/O devices should be mapped into this area.)
- **8 MB–Top of Memory1.** This area contains memory to the "top of memory1" boundary. In this area, memory-mapped I/O blocks are possible.
- **Boot Base–4 GB.** This area contains the bootstrap ROM.

The BIOS should decide where the different memory structures belong, and then configure the E820 handler to return the appropriate values.

For this example, the BIOS will report the system memory map by E820 as shown in Figure 9-4. Notice that the memory range from 1 MB to top of memory is marked as system memory, and then a small range is additionally marked as ACPI reclaim memory. A legacy OS that does not support the E820 extensions will ignore the extended memory range calls and correctly mark that memory as system memory.

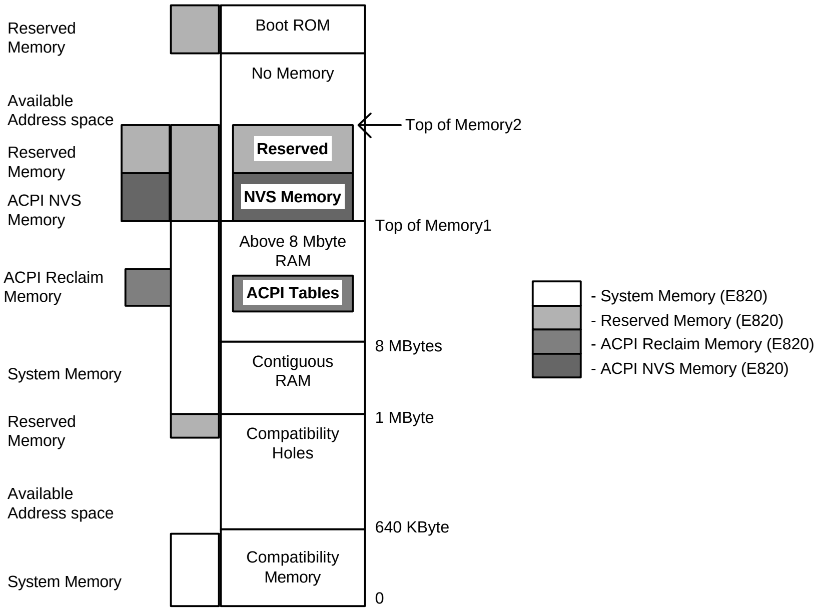

Figure 9-4 Memory as Configured after Boot

Also, from the Top of Memory1 to the Top of Memory2, the BIOS has set aside some memory for its own use and has marked as reserved both ACPI NVS Memory and Reserved Memory. A legacy OS will throw out the ACPI NVS Memory and correctly mark this as reserved memory (thus preventing this memory range from being allocated to any add-in device).

OSPM will call the _PTS control method prior to initiating a sleep (by programming the sleep type, followed by setting the SLP_EN bit). During a catastrophic failure (where the integrity of the AML code interpreter or driver structure is questionable), if OSPM decides to shut the system off, it will not issue a _PTS, but will immediately issue a SLP_TYP of “soft off” and then set the SLP_EN bit. Hence, the hardware should not rely solely on the _PTS control method to sequence the system to the “soft off” state. After waking from an S4 state, OSPM will restore the ACPI NVS memory image and then issue the _WAK control method that informs BIOS that its memory image is back.

9.3.3 OS Loading

At this point, the BIOS has passed control to OSPM, either by using OSPM boot loader (a result of waking from an S4/S5 or boot condition) or OSPM waking vector (a result of waking from an S2 or S3 state). For the Boot OS Loader path, OSPM will get the system address map via one of the mechanisms describe in section 15, “System Address Map Interfaces.” If OSPM is booting from an S4 state, it will then check the NVS image file’s hardware signature with the hardware signature within the FACS table (built by BIOS) to determine whether it has changed since entering the sleeping state (indicating that the platforms fundamental hardware configuration has changed during the current sleeping state). If the signature has changed, OSPM will not restore the system context and can boot from scratch (from the S4 state). Next, for an S4 wake, OSPM will check the NVS file to see whether it is valid. If valid, then OSPM will load the NVS image into system memory. Next, OSPM will check the SCI_EN bit and if it is not set, will write the ACPI_ENABLE value to the SMI_CMD register to switch into the system into ACPI mode and will then reload the memory image from the NVS file.

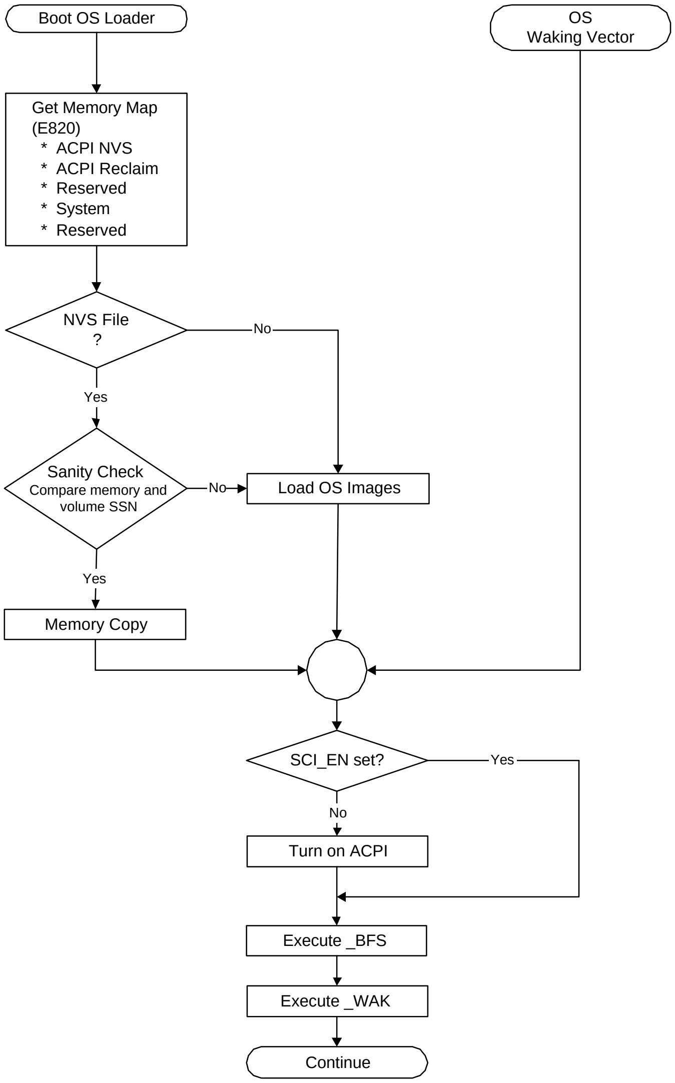

Figure 9-5 OS Initialization

If an NVS image file did not exist, then OSPM loader will load OSPM from scratch. At this point, OSPM will generate a _WAK call that indicates to the BIOS that its ACPI NVS memory image has been successfully and completely updated.

9.3.4 Exiting ACPI Mode

For machines that do not boot in ACPI mode, ACPI provides a mechanism that enables the OS to disable ACPI. The following occurs:

1. OSPM unloads all ACPI drivers (including the ACPI driver).
2. OSPM disables all ACPI events.
3. OSPM finishes using all ACPI registers.
4. OSPM issues an I/O access to the port at the address contained in the SMI_CMD field (in the FADT) with the value contained in the ACPI_DISABLE field (in the FADT).
5. BIOS then remaps all SCI events to legacy events and resets the SCI_EN bit.
6. Upon seeing the SCI_EN bit cleared, the ACPI OS enters the legacy OS mode.

When and if the legacy OS returns control to the ACPI OS, if the legacy OS has not maintained the ACPI tables (in reserved memory and ACPI NVS memory), the ACPI OS will reboot the system to allow the BIOS to re-initialize the tables.

10 ACPI-Specific Device Objects

This section specifies the ACPI device-specific objects. The system status indicator objects, declared under the _SI scope in the ACPI Namespace, are also specified in this section.

The device-specific objects specified in this section are objects for the following types of devices:

- Control method battery devices (for more information about Control Method Battery devices, see section 11.2, “Control Method Batteries”).
- Control method lid devices (for more information about control method lid devices, see section 10.3, “Control Method Lid Device”).
- Control method power and sleep button devices (for more information about control method power and sleep button devices, see section 4.7.2.2, “Buttons”).
- Embedded controller devices (for more information about embedded controller devices, see section 13, “ACPI Embedded Controller Interface Specification”).
- System Management Bus (SMBus) host controller (for more information, see section 13.9, “SMBus Host Controller Interface via Embedded Controller”).
- Fan devices (for more information about fan devices, see section 12, “Thermal Management”).
- Generic bus bridge devices.
- IDE control methods.

For a list of the ACPI Plug and Play ID values for all these devices, see section 5.6.4, “Device Class-Specific Objects.”

10.1 _SI System Indicators

ACPI provides an interface for a variety of simple and icon-style indicators on a system. All indicator controls are in the _SI portion of the namespace. The following table lists all defined system indicators. (Notice that there are also per-device indicators specified for battery devices).

Table 10-1 System Indicator Control Methods

Object	Description
_SST	System status indicator
_MSG	Messages waiting indicator

10.1.1 _SST (System Status)

Operating software invokes this control method to set the system status indicator as desired.

Arguments:
0 – No system state indication. Indicator off.
1–Working
2–Waking
3–Sleeping. Used to indicate system state S1, S2 or S3.
4–Sleeping with context saved to non-volatile storage.

10.1.2 _MSG (Message)

This control method sets the system's message-waiting status indicator.

Arguments:

0	Number of messages waiting

10.2 Battery Device

A battery device is required to either have an ACPI Smart Battery Table or a Control Method Battery interface. In the case of an ACPI Smart Battery Table, the Definition Block needs to include a Bus/Device Package for the SMBus host controller. This will install an OS specific driver for the SMBus, which in turn will locate the Smart Battery System Manager or Smart Battery Selector and Smart Battery Charger SMBus devices.

The Control Method Battery interface is defined in section 11.2, "Control Method Batteries."

10.3 Control Method Lid Device

For systems with a lid, the lid status can either be implemented using the fixed register space as defined in section 4, "ACPI Hardware Specification," or implemented in AML code as a control method lid device.

To implement a control method lid device, implement AML code that issues notifications for the device whenever the lid status has changed. The _LID control method for the lid device must be implemented to report the current state of the lid as either opened or closed.

The lid device can support _PRW and _PSW methods to select the wake functions for the lid when the lid transitions from closed to opened.

The Plug and Play ID of an ACPI control method lid device is PNP0C0D.

Table 10-2 Control Method Lid Device

Object	Description
_LID	Returns the current status of the lid.

10.3.1 _LID

Evaluates to the current status of the lid.

Result Code:

Zero:	The lid is closed
Non-zero:	The lid is open

10.4 Control Method Power and Sleep Button Devices

The system's power or sleep button can either be implemented using the fixed register space as defined in section 4.7.2.2, "Buttons," or implemented in AML code as a control method power button device. In either case, the power button override function or similar unconditional system power or reset functionality is still implemented in external hardware.

To implement a control method power-button or sleep-button device, implement AML code that delivers two types of notifications concerning the device. The first is Notify(*Object*, 0x80) to signal that the button was pressed while the system was in the S0 state to indicate that the user wants the machine to transition from S0 to some sleeping state. The other notification is Notify(*Object*, 0x2) to signal that the button was pressed while the system was in an S1 to S4 state and to cause the system to wake. When the button is used to wake the system, the wake notification (Notify(*Object*, 0x2)) must occur after OSPM actually wakes, and a button-pressed notification (Notify(*Object*, 0x80)) must not occur.

The Wake Notification indicates that the system is awake because the user pressed the button and therefore a complete system resume should occur (for example, turn on the display immediately, and so on).

10.5 Embedded Controller Device

Operation of the embedded controller host controller register interface requires that the embedded controller driver has ACPI-specific knowledge. Specifically, the driver needs to provide an "operational region" of its embedded controller address space, and needs to use a general-purpose event (GPE) to service the host controller interface. For more information about an ACPI-compatible embedded controller device, see section 13, "ACPI Embedded Controller Interface Specification."

The embedded controller device object provides the _HID of an ACPI-integrated embedded controller device of PNP0C09 and the host controller register locations using the device standard methods. In addition, the embedded controller must be declared as a named device object that includes a set of control methods. For more information, see section 13.11, "Defining an Embedded Controller Device in ACPI Namespace").

10.6 Fan Device

A fan device is assumed to be in operation when it is in the D0 state. Thermal zones reference fan device(s) as being responsible primarily for cooling within that zone. Notice that multiple fan devices can be present for any one thermal zone. They might be actual different fans, or they might be used to implement one fan of multiple speeds (for example, by turning both "fans" on the one fan will run full speed).

The Plug and Play ID of a fan device is PNP0C0B. For more information about fan devices, see section 12, "Thermal Management."

10.7 Generic ISA Bus Device

A generic ISA bus device is a bridge that does not require a special OS driver because the bridge does not provide or require any features not described within the normal ACPI device functions. The resources the bridge requires are specified via normal ACPI resource mechanisms. Device enumeration for child devices is supported via ACPI namespace device enumeration and OS drivers require no other features of the bus. Such a bridge device is identified with the Plug and Play ID of PNP0A05 or PNP0A06.

A generic bus bridge device is typically used for integrated bridges that have no other means of controlling them and that have a set of well-known devices behind them. For example, a portable computer can have a "generic bus bridge" known as an EIO bus that bridges to some number of Super-I/O devices. The bridged resources are likely to be positively decoded as either a function of the bridge or the integrated devices. In this example, a generic bus bridge device would be used to declare the bridge then child devices would be declared below the bridge; representing the integrated Super-I/O devices.

10.8 IDE Controller Device

Most device drivers can save and restore the registers of their device. For IDE controllers and drives, this is not true because there are several drive settings for which ATA does not provide mechanisms to read. Further, there is no industry standard for setting timing information for IDE controllers. Because of this, ACPI interface mechanisms are necessary to provide the operating system information about the current settings for the drive and channel, and for setting the timing for the channel.

OSPM and the IDE driver will follow these steps when powering off the IDE subsystem:

1. The IDE driver will call the _GTM control method to get the current transfer timing settings for the IDE channel. This includes information about DMA and PIO modes.
2. The IDE driver will call the standard OS services to power down the drives and channel.
3. As a result, OSPM will execute the appropriate _PS3 methods and turn off unneeded power resources.

To power on the IDE subsystem, OSPM and the IDE driver will follow these steps:

1. The IDE driver will call the standard OS services to turn on the drives and channel.
2. As a result, OSPM will execute the appropriate _PS0 methods and turn on required power resources.
3. The IDE driver will call the _STM control method passing in transfer timing settings for the channel, as well as the ATA drive ID block for each drive on the channel. The _STM control method will configure the IDE channel based on this information.
4. For each drive on the IDE channel, the IDE driver will run the _GTF to determine the ATA commands required to reinitialize each drive to boot up defaults.
5. The IDE driver will finish initializing the drives by sending these ATA commands to the drives, possibly modifying or adding commands to suit the features supported by the operating system.

The following shows the namespace for these objects:

```
\_SB                                - System bus
            PCI0                            - PCI bus
                    IDE1                    - IDE channel
_ADR    - Indicates address of the channel on the PCI bus
_GTM    - Control method to get current IDE channel settings
_STM    - Control method to set current IDE channel settings
_PR0    - Power resources needed for D0 power state
                                DRV1    - Drive 0
                                        _ADR    - Indicates address of master IDE device
                                        _GTF    - Control method to get task file
                                DRV2    - Drive 1
                                        _ADR    - Indicates address of slave IDE device
                                        _GTF    - Control method to get task file
                    IDE2                    - Second IDE channel
_ADR    - Indicates address of the channel on the PCI bus
_GTM    - Control method to get current IDE channel settings
_STM    - Control method to set current IDE channel settings
```

_PR0 - Power resources needed for D0 power state

DRV1 - Drive 0

_ADR - Indicates address of master IDE device

_GTF - Control method to get task file

DRV2 - Drive 1

_ADR - Indicates address of slave IDE device

_GTF - Control method to get task file

The sequential order of operations is as follows:

Powering down:

Call _GTM.

Power down drive (calls _PS3 method and turns off power planes).

Powering up:

Power up drive (calls _PS0 method if present and turns on power planes).

Call _STM passing info from _GTM (possibly modified), with ID data from each drive.

Initialize the channel.

May modify the results of _GTF.

For each drive:

Call _GTF.

Execute task file (possibly modified).

Table 10-3 IDE Specific Controls

Object	Description
_GTF	Optional control method to get the ATA task file needed to re-initialize the drive to boot up defaults.
_GTM	Optional control method to get the IDE controller timing information.
_STM	Optional control method to set the IDE controller's transfer timing settings.

10.8.1 _GTF (Get Task File)

This Control Method returns a buffer containing the ATA commands used to restore the drive to boot up defaults (that is, the state of the drive after POST). The returned buffer is an array with each element in the array consisting of 7 8-bit register values (56 bits) corresponding to ATA task registers 1F1 thru 1F7. Each entry in the array defines a command to the drive.

Note: ACPI 1.0b defines _GTF as evaluating to a buffer containing a header byte (1-based) that indicates the number of commands following in the array. The de facto standard OSPM implementations supporting the _GTF method do not support the ACPI 1.0b definition including this header byte. As such, the _GTF definition has been updated, removing the header byte for ACPI 2.0.

ATA task file array definition:
    Seven register values for command 1
        Reg values: (1F1, 1F2, 1F3, 1F4, 1F5, 1F6, 1F7)
    Seven register values for command 2
        Reg values: (1F1, 1F2, 1F3, 1F4, 1F5, 1F6, 1F7)
    Seven register values for command 3
        Reg values: (1F1, 1F2, 1F3, 1F4, 1F5, 1F6, 1F7)

Etc……

After powering up the drive, the OS will send these commands to the drive, in the order specified. The IDE driver may modify some of the feature commands or append its own to better tune the drive for OSPM features before sending the commands to the drive.

This Control Method is listed under each drive device object. _GTF must be called after calling _STM.

Arguments:
    None

Result Code:
    A buffer that is a byte stream of ATA commands to send to the drive.

Example of the return from _GTF:

```
Method(_GTF, 0x0, NotSerialized)
{
    Return(GTF0)
}
Name(GTF0, Buffer(0x1c)
{
   0x03, 0x00, 0x00, 0x00, 0x00, 0xa0, 0xef, 0x03, 0x00, 0x00, 0x00, 0x00,
   0xa0, 0xef, 0x00, 0x10, 0x00, 0x00, 0x00, 0xa0, 0xc6, 0x00, 0x00, 0x00,
   0x00, 0x00, 0xa0, 0x91
}
```

10.8.2 _GTM (Get Timing Mode)

This Control Method returns the current settings for the IDE channel.

This control method is listed under each channel device object.

Arguments:
None

Result Code:
A buffer with the current settings for the IDE channel:

```
Buffer (){
       PIO Speed 0                   //DWORD
       DMA Speed 0                   //DWORD
       PIO Speed 1                   //DWORD
       DMA Speed 1                   //DWORD
       Flags                         //DWORD
}
```

Table 10-4 _GTM Method Result Codes

Field	Format	Description
PIO Speed 0	DWORD	The PIO bus-cycle timing for drive 0 in nanoseconds. 0xFFFFFFFF indicates that this mode is not supported by the channel. If the chipset cannot set timing parameters independently for each drive, this field represents the timing for both drives.
DMA Speed 0	DWORD	The DMA bus-cycle for drive 0 timing in nanoseconds. If Bit 0 of the Flags register is set, this DMA timing is for UltraDMA mode, otherwise the timing is for multi-word DMA mode. 0xFFFFFFFF indicates that this mode is not supported by the channel. If the chipset cannot set timing parameters independently for each drive, this field represents the timing for both drives.
PIO Speed 1	DWORD	The PIO bus-cycle timing for drive 1 in nanoseconds. 0xFFFFFFFF indicates that this mode is not supported by the channel. If the chipset cannot set timing parameters independently for each drive, this field must be 0xffffffff.
DMA Speed 1	DWORD	The DMA bus-cycle timing for drive 1 in nanoseconds. If Bit 0 of the Flags register is set, this DMA timing is for UltraDMA mode, otherwise the timing is for multi-word DMA mode. 0xFFFFFFFF indicates that this mode is not supported by the channel. If the chipset cannot set timing parameters independently for each drive, this field must be 0xFFFFFFFF.
Flags	DWORD	Mode flags Bit[0]: 1 indicates using UltraDMA on drive 0 Bit[1]: 1 indicates IOChannelReady is used on drive 0 Bit[2]: 1 indicates using UltraDMA on drive 1 Bit[3]: 1 indicates IOChannelReady is used on drive 1 Bit[4]: 1 indicates chipset can set timing independently for each drive Bits[5-31]: reserved (must be 0)

10.8.3 _STM (Set Timing Mode)

This Control Method sets the IDE channel’s transfer timings to the setting requested. The AML code is required to convert and set the nanoseconds timing to the appropriate transfer mode settings for the IDE controller. _STM may also make adjustments so that _GTF control methods return the correct commands for the current channel settings.

This control method takes three arguments: Channel timing information (as described in Table 10-4), and the ATA drive ID block for each drive on the channel. The channel timing information is not guaranteed to be the same values as returned by _GTM; the OS may tune these values as needed.

The ATA drive ID block is the raw data returned by the Identify Drive, ATA command, which has the command code “0ech.” The _STM control method is responsible for correcting for drives that misreport their timing information.

Arguments:

Arg0	Buffer	Channel timing information (formatted as described in table 10-4)
Arg1	Buffer	ATA drive IDE block for drive 0
Arg2	Buffer	ATA drive IDE block for drive 1

Result Code:

None

10.9 Floppy Controller Device Objects

10.9.1 _FDE (Floppy Disk Enumerate)

Enumerating devices attached to a floppy disk controller is a time-consuming function. In order to speed up the process of floppy enumeration, ACPI defines an optional enumeration object that is defined directly under the device object for the floppy disk controller. It returns a buffer of five 32-bit values. The first four values are Boolean values indicating the presence or absence of the four floppy drives that are potentially attached to the controller. A non-zero value indicates that the floppy device is present. The fifth value returned indicates the presence or absence of a tape controller. Definitions of the tape presence value can be found in Table 10-5.

Arguments:

None

Result Code:

A buffer containing values that indicate the presence or absence of floppy devices.

```
Buffer (){
       Floppy 0                      // Boolean DWORD
       Floppy 1                      // Boolean DWORD
       Floppy 2                      // Boolean DWORD
       Floppy 3                      // Boolean DWORD
       Tape                          // See table below
}
```

Table 10-5 Tape Presence

Value	Description
0	Unknown if device is present
1	Device is present
2	Device is never present
>2	Reserved

10.9.2 _FDI (Floppy Disk Information)

This object returns information about a floppy disk drive. This information is the same as that returned by the INT 13 Function 08H on IA-PCs.

Result Code:

```
Package {
    Drive Number                //BYTE
    Device Type                 //BYTE
    Maximum Cylinder Number     //WORD
    Maximum Sector Number       //WORD
    Maximum Head Number         //WORD
    disk_specify_1              //BYTE
    disk_specify_2              //BYTE
    disk_motor_wait             //BYTE
    disk_sector_siz             //BYTE
    disk_eot                    //BYTE
    disk_rw_gap                 //BYTE
    disk_dtl                    //BYTE
    disk_formt_gap              //BYTE
    disk_fill                   //BYTE
    disk_head_sttl              //BYTE
    disk_motor_strt             //BYTE
    }
```

Table 10-6 ACPI Floppy Drive Information

Field	**Format**	**Definition**
Drive Number	BYTE	As reported by _INT 13 Function 08H
Device Type	BYTE	As reported by _INT 13 Function 08H
Maximum Cylinder Number	WORD	As reported by _INT 13 Function 08H
Maximum Sector Number	WORD	As reported by _INT 13 Function 08H
Maximum Head Number	WORD	As reported by _INT 13 Function 08H
Disk_specify_1	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_specify_2	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_motor_wait	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_sector_siz	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_eot	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_rw_gap	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_dtl	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_formt_gap	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_fill	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_head_sttl	BYTE	As reported in ES:D1 from INT 13 Function 08H
Disk_motor_strt	BYTE	As reported in ES:D1 from INT 13 Function 08H

10.9.3 _FDM (Floppy Disk Drive Mode)

This control method switches the mode (300RPM/360RPM) of all floppy disk drives attached to this controller. If this control method is implemented, the platform must reset the mode of all drives to 300RPM mode after a D*x* to D0 transition of the controller.

Arguments:

0 – Set the mode of all drives to 300RPM mode.

1 – Set the mode of all drives to 360RPM mode.

Result Code:

None

10.10 GPE Block Device

The GPE Block device is an optional device that allows a system designer to describe GPE blocks beyond the two that are described in the FADT. Control methods associated with the GPE pins of GPE block devices exist as children of the GPE Block device, not within the _GPE namespace.

A GPE Block device consumes I/O or memory address space, as specified by its _PRS or _CRS child objects. The interrupt vector used by the GPE block does not need to be the same as the SCI_INT field. The interrupt used by the GPE block device is specified in the _CRS and _PRS methods associated with the GPE block.

A GPE Block device must have a _HID or a _CID of “ACPI0006.”

Note: A system designer must describe the GPE block necessary to bootstrap the system in the FADT as a GPE0/GPE1 block. GPE Block devices cannot be used to implement these GPE inputs.

To represent the GPE block associated with the FADT, the system designer needs only to include the ACPI0006 device in the tree, and **not** have any _CRS, _PRS, _SRS, or other GPE-specific methods associated with that block. Any block that does not represent the GPE block of the FADT must contain the _Lxx, _Exx, _CRS, _PRS, or _SRS methods required to use/program that block. OSPM assumes the first ACPI0006 device without a _CRS is the GPE device that is associated with the FADT.

```
// ASL example of root GPE block
Device(\_SB.PCI0.GPE0) {
    Name(_HID,”ACPI0006”)
    Name(_UID,1)
}

// ASL example of a non-root GPE block
Device(\_SB.PCI0.GPE1) {
    Name(_HID,”ACPI0006”)
    Name(_UID,2)
    Name(_CRS,Buffer() {
        IO(Decode16, FC00, FC03, 4, 4,)
        IRQ( Level, ActiveHigh, Shared,) { 5 }
        }
    }
    Method(_L02) { … }
    Method(_E07) { … }
```

Notice that it is legal to replace the I/O descriptors with Memory descriptors if the register is memory mapped.

If the system must run any GPEs to bootstrap the system (for example, when Embedded Controller events are required), the associated block of GPEs must be described in the FADT. This register block is not relocatable and will always be available for the life of the operating system boot.

The GPE block associated with the ACPI0006 device can be stopped, ejected, reprogrammed, and so on. The system can also have multiple such GPE blocks.

10.10.1 Matching Control Methods for General-Purpose Events in a GPE Block Device

When a GPE Device raises an interrupt, OSPM executes a corresponding control method (as described in section 5.6.2.2.3, “Queuing the Matching Control Method for Execution”). These control methods (of the form _Lxx and _Exx) for GPE Devices are not within the _GPE namespace. They are children of the GPE Block device.

For example:

```
Device(GPE5) {

    Name(_HID, “ACPI0006”)

    Method(_L02) { … }
    Method(_E07) { … }
}
```

10.11 Module Device

This optional device is a container object that acts as a bus node in a namespace. It may contain child objects that are devices or buses. The module device is declared using the ACPI0004 hardware indentifier (HID).

If the module device contains a _CRS object, the “bus” described by this object is assumed to have these resources available for consumption by its child devices. Any resources not described in the module device’s _CRS object may not be allocated to child devices.

For example, consider a Module Device containing three child memory devices. If the _CRS object for the Module Device contains memory from 2 GB through 6 GB, then the child memory devices may only be assigned addresses within this range.

Example:

```
Device(\_SB.NOD0) {
    Name(_HID, "ACPI0004")        // Module device
    Name(_UID, 0)
    Name(_PRS, ResourceTemplate() {
        WordIo(ResourceProducer,
            MinFixed,      // _MIF
            MaxFixed,,,    // _MAF
            0x0000,        // _GRA
            0x0000,        // _MIN
            0x7FFF,        // _MAX
            0x0,           // _TRA
            0x8000)        // _LEN
        DWordMemory(ResourceProducer,,    // For Main Memory + PCI
            MinNotFixed,       // _MIF
            MaxNotFixed,       // _MAF
            Cacheable,     // _MEM
            ReadWrite,     // _RW
            0x0FFFFFFF,    // _GRA
            0x40000000,    // _MIN
            0x7FFFFFFF,    // _MAX
            0x0,           // _TRA
            0x00000000)    // _LEN
         })
    Method(_SRS, 1) { ... }
    Method(_CRS, 0) { ... }

    Device(MEM0) {            // Main Memory (256MB module)
        Name(_HID, EISAID("PNP0C80"))
        Name(_UID, 0)
        Method(_STA, 0) {
            // If memory not present --> Return(0x00)
            // Else if memory is disabled --> Return(0x0D)
            // Else --> Return(0x0F)
        }
        Name(_PRS, ResourceTemplate() {
            DWordMemory(,,,,
                Cacheable, // _MEM
                ReadWrite, // _RW
                0x0FFFFFFF,// _GRA
                0x40000000,// _MIN
                0x7FFFFFFF,// _MAX
                0x0,       // _TRA
                0x10000000)// _LEN
        })
        Method(_CRS, 0) { ... }
        Method(_SRS, 1) { ... }
        Method(_DIS, 0) { ... }
    }
    Device(MEM1) {            // Main Memory (512MB module)
        Name(_HID, EISAID("PNP0C80"))
        Name(_UID, 1)
        Method(_STA, 0) {
            // If memory not present --> Return(0x00)
            // Else if memory is disabled --> Return(0x0D)
            // Else --> Return(0x0F)
        }
        Name(_PRS, ResourceTemplate() {
            DWordMemory(,,,,
                Cacheable, // _MEM
                ReadWrite, // _RW
                0x1FFFFFFF,// _GRA
                0x40000000,// _MIN
                0x7FFFFFFF,// _MAX
                0x0,       // _TRA
                0x20000000)// _LEN
        })
        Method(_CRS, 0) { ... }
        Method(_SRS, 1) { ... }
        Method(_DIS, 0) { ... }
    }
    Device(PCI0) {            // PCI Root Bridge
        Name(_HID, EISAID("PNP0A03"))
        Name(_UID, 0)
        Name(_BBN, 0x00)
        Name(_PRS, ResourceTemplate() {
            WordBusNumber(ResourceProducer,
                MinFixed,  // _MIF
                MaxFixed,, // _MAF
                0x00,      // _GRA
                0x00,      // _MIN
                0x7F,      // _MAX
                0x0,       // _TRA
                0x80)      // _LEN
            WordIo(ResourceProducer,
                MinFixed,  // _MIF
                MaxFixed,,,// _MAF
                0x0000,    // _GRA
                0x0000,    // _MIN
                0x0CF7,    // _MAX
                0x0,       // _TRA
                0x0CF8)    // _LEN
            WordIo(ResourceProducer,
                MinFixed,  // _MIF
                MaxFixed,,,// _MAF
                0x0000,    // _GRA
                0x0D00,    // _MIN
                0x7FFF,    // _MAX
                0x0,       // _TRA
                0x7300)    // _LEN
            DWordMemory(ResourceProducer,,
                MinNotFixed,   // _MIF
                MaxNotFixed,   // _MAF
                NonCacheable,  // _MEM
                ReadWrite, // _RW
                0x0FFFFFFF,// _GRA
                0x40000000,// _MIN
                0x7FFFFFFF,// _MAX
                0x0,       // _TRA
                0x00000000)// _LEN
        })
        Method(_CRS, 0) { ... }
        Method(_SRS, 1) { ... }
    }
}
```

10.12 Memory Devices

Memory devices allow a platform designer to optionally describe the dynamic properties of memory. If a platform cannot have memory added or removed while the system is active, then memory devices are not necessary. Memory devices may describe exactly the same physical memory that the System Address Map interfaces describe (see section 15, “System Address Map Interfaces”). They do not describe how that memory is, or has been, used. If a region of physical memory is marked in the System Address Map interface as AddressRangeReserved or AddressRangeNVS and it is also described in a memory device, then it is the responsibility of the OS to guarantee that the memory device is never disabled.

It is not necessary to describe all memory in the system with memory devices if there is some memory in the system that is static in nature. If, for instance, the memory that is used for the first 16 MB of system RAM cannot be ejected, inserted, or disabled, that memory may only be represented by the System Address Map interfaces. But if memory can be ejected, inserted, or disabled, it must be represented by a memory device.

10.12.1 Address Decoding

Memory devices must provide a _CRS object that describes the physical address space that the memory decodes. If the memory can decode alternative ranges in physical address space, the devices may also provide _PRS, _SRS and _DIS objects. Other device objects may also apply if the device can be ejected.

10.12.2 Example: Memory Device

```
Scope(\_SB){
    Device(MEM0) {
        Name(_HID, EISAID(“PNP0C80”))
        Name(_CRS, ResourceTemplate() {
                QwordMemory(ResourceConsumer,
                            ,
                            MinFixed,
                            MaxFixed,
                            Cacheable,
                            ReadWrite,
                            0xfffffff,
                            0x10000000,
                            0x30000000,
                            0,
                            ,,)
        }
    }
}
```

11 Power Source Devices

This section specifies the battery and AC adapter device objects OSPM uses to manage power resources.

A battery device is required to either have a Smart Battery subsystem or a Control Method Battery interface as described in this section. OSPM is required to be able to connect and manage a battery on either of these interfaces. This section describes these interfaces.

In the case of a compatible ACPI Smart Battery Table, the Definition Block needs to include a Bus/Device package for the SMB-HC. This will install an OS-specific driver for the SMBus, which in turn will locate the components of the Smart Battery subsystem. In addition to the battery or batteries, the Smart Battery subsystem includes a charger and a manager device to handle subsystems with multiple batteries.

The Smart Battery System Manager is one implementation of a manager device that is capable of arbitrating among the available power sources (AC power and batteries) for a system. It provides a superset of the Smart Battery Selector functionality, such as safely responding to power events (AC versus battery power), inserting and removing batteries and notifying the OS of all such changes. Additionally, the Smart Battery System Manager is capable of handling configurations including simultaneous charging and discharging of multiple batteries. Unlike the Smart Battery Selector that shares responsibility for configuring the battery system with OSPM, the Smart Battery System Manager alone controls the safe configuration of the battery system and simply issues status changes to OSPM when the configuration changes. Smart Battery System Manager is the recommended solution for handling multiple-battery systems.

11.1 Smart Battery Subsystems

The Smart Battery subsystem is defined by the:

- System Management Bus Specification (SMBS)
- Smart Battery Data Specification (SBDS)
- Smart Battery Charger Specification (SBCS)
- Smart Battery System Manager Specification (SBSM)
- Smart Battery Selector Specification (SBSS)

An ACPI-compatible Smart Battery subsystem consists of:

- An SMB-HC(CPU to SMB-HC) interface
- At least one Smart Battery
- A Smart Battery Charger
- Either a Smart Battery System Manager or a Smart Battery Selector if more than one Smart Battery is supported

In such a subsystem, a standard way of communicating with a Smart Battery and Smart Battery Charger is through the SMBus physical protocols. The Smart Battery System Manager or Smart Battery Selector provides event notification (battery insertion/removal, and so on) and charger SMBus routing capability for any Smart Battery subsystem. A typical Smart Battery subsystem is illustrated below:

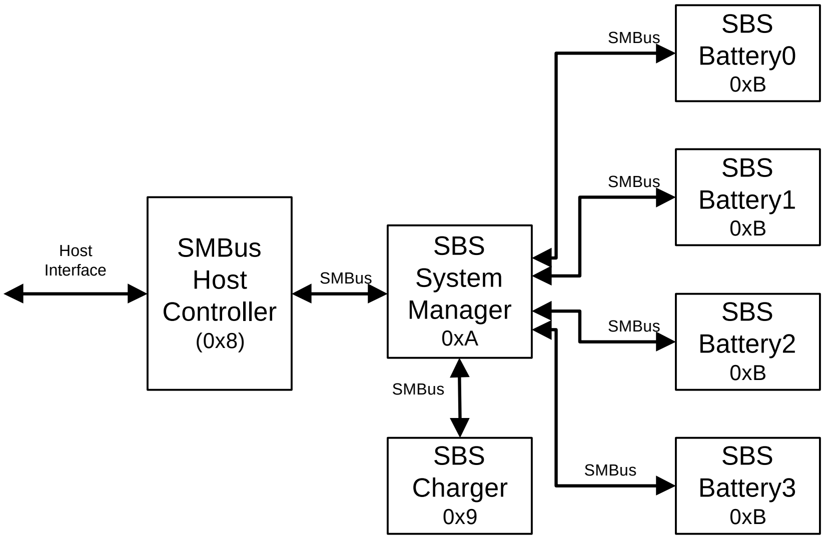

Figure 11-1 Typical Smart Battery Subsystem (SBS)

SMBus defines a fixed 7-bit slave address per device. This means that all batteries in the system have the same address (defined to be 0xB). The slave addresses associated with Smart Battery subsystem components are shown in the following table.

Table 11-1 Example SMBus Device Slave Addresses

SMBus Device Description	SMBus Slave Address (A0-A6)
SMBus Host Slave Interface	0x8
Smart Battery Charger/Charger Selector or Charger System Manager	0x9
Smart Battery System Manager or Smart Battery Selector	0xA
Smart Battery	0xB

Each SMBus device has up to 256 registers that are addressed through the SMBus protocol's *Command* value. SMBus devices are addressed by providing the slave address with the desired register's *Command* value. Each SMBus register can have non-linear registers; that is, command register 1 can have a 32-byte string, while command register 2 can have a byte, and command register 3 can have a word.

The SMBus host slave interface provides a standard mechanism for the host CPU to generate SMBus protocol commands that are required to communicate with SMBus devices (in other words, the Smart Battery components). ACPI defines such an SMB-HCthat resides in embedded controller address space; however, an OS can support any SMB-HCthat has a native SMB-HCdevice driver.

The Smart Battery System Manager provides a standard programming model to control multiple Smart Batteries in a Smart Battery subsystem. A Smart Battery System Manager provides the following types of battery management functions:

- Event notification for battery insertion and removal
- Event notification for AC power connected or disconnected
- Status of which Smart Battery is communicating with the SMB-HC
- Status of which Smart Battery(s) are powering the system
- Status of which Smart Battery(s) are connected to the charger
- Status of which Smart Batteries are present in the system
- Event notification when the Smart Battery System Manager switches from one power source to another
- Hardware-switching to an alternate Smart Battery when the Smart Battery supplying power runs low
- Hardware switching between battery-powered and AC-powered powered operation

The Smart Battery System Manager function can reside in a standalone SMBus slave device (Smart Battery System Manager that responds to the 0xA slave address), may be present within a smart charger device (Smart Battery Charger that responds to the 0x9 slave address), or may be combined within the embedded controller (that responds to the 0xA slave address). If both a Smart Battery Charger and a standalone Smart Battery System Manager are present in the same Smart Battery subsystem, then the driver assumes that the standalone Smart Battery System Manager is wired to the batteries.

The Smart Battery charger is an SMBus device that provides a standard programming model to control the charging of Smart Batteries present in a Smart Battery subsystem. For single battery systems, the Smart Battery Charger is also responsible for notifying the system of the battery and AC status.

The Smart Battery provides intelligent chemistry-independent power to the system. The Smart Battery is capable of informing the Smart Battery charger of its charging requirements (which provides chemistry independence) and providing battery status and alarm features needed for platform battery management.

11.1.1 ACPI Smart Battery Status Change Notification Requirements

The Smart Battery System Manager, the Smart Battery Selector, and the Smart Battery Charger each have an optional mechanism for notifying the system that the battery configuration or AC status has changed. ACPI requires that this interrupt mechanism be through the SMBus Alarm Notify mechanism.

For systems using an embedded controller as the SMBus host, a battery system device issues a status change notification by either mastering the SMBus to send the notification directly to the SMBus host, or by emulating it in the embedded controller. In either case, the process is the same. After the notification is received or emulated, the embedded controller asserts an SCI. The source of the SCI is identified by a GPE that indicates the SCI was caused by the embedded controller. The embedded controller’s status register alarm bit is set, indicating that the SMBus host received an alarm message. The Alarm Address Register contains the address of the SMBus device that originated the alarm and the Alarm Data Registers contain the contents of that device’s status register.

11.1.1.1 Smart Battery Charger

This requires a Smart Battery Charger, on a battery or AC status change, to generate an SMBus Alarm Notify. The contents of the Smart Battery Charger's ChargerStatus() command register (0x13) is placed in the embedded controller's Alarm Data Registers, the Smart Battery Charger's slave address[17] (0x09) is placed in the embedded controller's Alarm Address Register and the EC's Status Register's Alarm bit is set. The embedded controller then asserts an SCI.

11.1.1.2 Smart Battery Charger with optional System Manager or Selector

A Smart Battery Charger that contains the optional System Manager or Selector function (as indicated by the ChargerSpecInfo() command register, 0x11, bit 4) is required to generate an SMBus Alarm Notify on a battery or AC status change. The content of the Smart Battery Charger with an optional System Manager, the BatterySystemState() command register (0x21) (or in the case of an optional Selector, the SelectorState() (0x01)), is placed in the EC's Alarm Data Registers, the Smart Battery Charger's slave address (0x09) is placed in the embedded controller's Alarm Address Register, and the embedded controller's Status Register's Alarm bit is set. The embedded controller then asserts an SCI.

11.1.1.3 Smart Battery System Manager

The Smart Battery System Manager is required to generate an SMBus Alarm Notify on a battery or AC status change. The content of the Smart Battery System Manager's BatterySystemState() command register (0x01) is placed in the EC's Alarm Data Registers, the Smart Battery System Manager's slave address (0x0A) is placed in the EC's Alarm Address Register, and the embedded controller's Status Register's Alarm bit is set. The embedded controller then asserts an SCI.

11.1.1.4 Smart Battery Selector

The requirements for the Smart Battery Selector are the same as the requirements for the Smart Battery System Manager, with the exception that the contents of the SelectorState() command register (0x01) are used instead of BatterySystemState(). The Smart Battery Selector is a subset of the Smart Battery System Manager and does not have the added support for simultaneous charge/discharge of multiple batteries. The System Manager is the preferred implementation.

[17] Notice that the 1.0 SMBus protocol specification is ambiguous about the definition of the "slave address" written into the command field of the host controller. In this case, the slave address is actually the combination of the 7-bit slave address and the Write protocol bit. Therefore, bit 0 of the initiating device's slave address is aligned to bit 1 of the host controller's slave command register, bit 1 of the slave address is aligned to bit 2 of the controller's slave command register, and so on.

11.1.2 Smart Battery Objects

The Smart Battery subsystem requires a number of objects to define its interface. These are summarized below:

Table 11-2 Smart Battery Objects

Object	Description
_HID	This is the hardware ID named object that contains a string. For Smart Battery subsystems, this object returns the value of "ACPI0002." This identifies the Smart Battery subsystem to the Smart Battery driver.
_SBS	This is the Smart Battery named object that contains a DWORD. This named object returns the configuration of the Smart Battery subsystem and is encoded as follows: 0 – Maximum of one Smart Battery and no Smart Battery System Manager or Smart Battery Selector. 1 – Maximum of one Smart Battery and a Smart Battery System Manager or Smart Battery Selector. 2 – Maximum of two Smart Batteries and a Smart Battery System Manager or Smart Battery Selector. 3 – Maximum of three Smart Batteries and a Smart Battery System Manager or Smart Battery Selector. 4 – Maximum of four Smart Batteries and a Smart Battery System Manager or Smart Battery Selector. The maximum number of batteries is for the system. Therefore, if the platform is capable of supporting four batteries, but only two are normally present in the system, then this field should return 4. Notice that a value of 0 indicates a maximum support of one battery and there is no Smart Battery System Manager or Smart Battery Selector present in the system.

11.1.3 Smart Battery Subsystem Control Methods

As the SMBus is not an enumerable bus, all devices on the bus must be declared in the ACPI name space. As the Smart Battery driver understands Smart Battery, Smart Battery Charger, and Smart Battery System Manager or Smart Battery Selector; only a single device needs to be declared per Smart Battery subsystem. The driver gets information about the subsystem through the hardware ID (which defines a Smart Battery subsystem) and the number of Smart Batteries supported on this subsystem (_SBS named object). The ACPI Smart Battery table indicates the energy levels of the platform at which the system should warn the user and then enter a sleeping state. The Smart Battery driver then reflects these as threshold alarms for the Smart Batteries.

The _SBS control method returns the configuration of the Smart Battery subsystem. This named object returns a DWORD value with a number from 0 to 4. If the number of batteries is greater than 0, then the Smart Battery driver assumes that a Smart Battery System Manager or Smart Battery Selector is present. If 0, then the Smart Battery driver assumes a single Smart Battery and neither a Smart Battery System Manager nor Smart Battery Selector is present.

A Smart Battery device declaration in the ACPI name space requires the _GLK object if potentially contentious accesses to device resources are performed by non-OS code. See section 6.5.7, "_GLK (Global Lock)," for details about the _GLK object.

11.1.3.1 Example: Single Smart Battery Subsystem

This section illustrates how to define a Smart Battery subsystem containing a single Smart Battery and charger. The platform implementation is illustrated below:

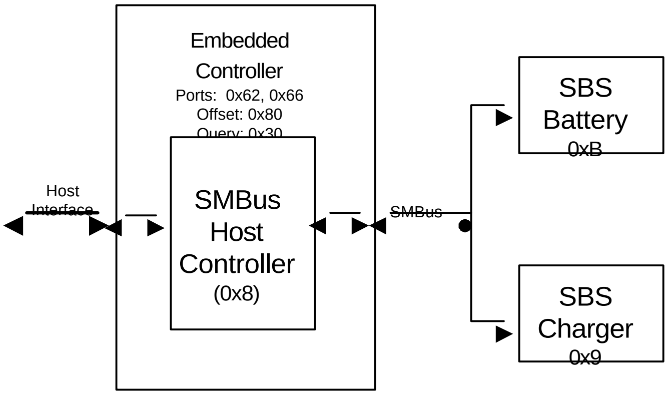

Figure 11-2 Single Smart Battery Subsystem

In this example, the platform is using an SMB-HC that resides within the embedded controller and meets the ACPI standard for an embedded controller interface and SMB-HCinterface. The embedded controller interface sits at system I/O port addresses 0x62 and 0x66. The SMB-HCis at base address 0x80 within embedded controller address space (as defined by the ACPI embedded controller specification) and responds to events on query value 0x30.

In this example the Smart Battery subsystem only supports a single Smart Battery. The ASL code for describing this interface is shown below:

```
Device(EC0) {
    Name(_HID, EISAID("PNP0C09"))
    Name(_CRS,
        ResourceTemplate(){       // port 0x62 and 0x66
            IO(Decode16, 0x62, 0x62, 0, 1),
            IO(Decode16, 0x66, 0x66, 0, 1)
            }
    )
    Name(_GPE, 0)
    Device (SMB0) {
        Name(_HID, "ACPI0001")        // Smart Battery Host Controller
        Name(_EC, 0x8030)             // EC offset (0x80), Query (0x30)
            Device(SBS0){             // Smart Battery Subsystem
                Name(_HID, "ACPI0002") // Smart Battery Subsystem ID
                Name(_SBS, 0x1)       // Indicates support for one battery
            }   // end of SBS0
        }       // end of SMB0
    }           // end of EC
```

11.1.3.2 Multiple Smart Battery Subsystem: Example

This section illustrates how to define a Smart Battery subsystem that contains three Smart Batteries, a Smart Battery System Manager, and a Smart Battery Charger. The platform implementation is illustrated below:

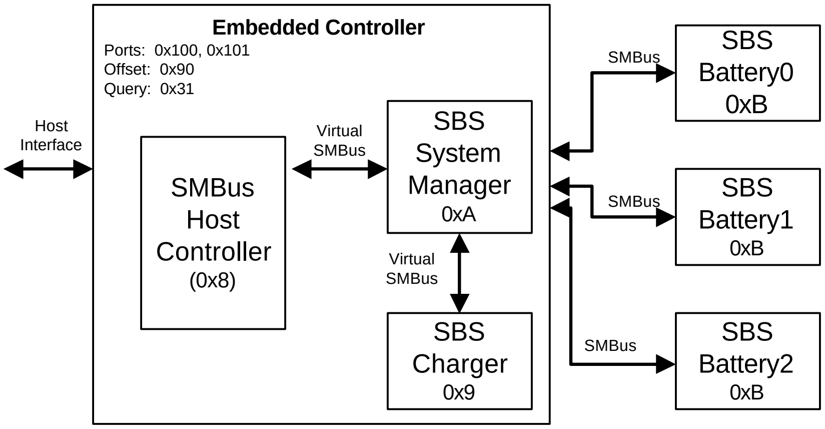

Figure 11-3 Smart Battery Subsystem

In this example, the platform is using an SMB-HCthat resides within the embedded controller and meets the ACPI standard for an embedded controller interface and SMB-HCinterface. The embedded controller interface sits at system I/O port addresses 0x100 and 0x101. The SMB-HCresides at base address 0x90 within embedded controller address space (as defined by the ACPI embedded controller specification) and responds to events on query value 0x31.

In this example the Smart Battery subsystem supports three Smart Batteries. The Smart Battery Charger and Smart Battery System Manager reside within the embedded controller, meet the Smart Battery System Manager and Smart Battery Charger interface specification, and respond to their 7-bit addresses (0xA and 0x9 respectively). The ASL code for describing this interface is shown below:

```
Device(EC1) {
    Name(_HID, EISAID("PNP0C09"))
    Name(_CRS,
        ResourceTemplate(){                   // port 0x100 and 0x101
            IO(Decode16, 0x100, 0x100, 0, 2)
            }
        )
    Name(_GPE, 1)
    Device (SMB1) {
        Name(_HID, "ACPI0001")     // Smart Battery Host Controller
        Name(_EC, 0x9031)          // EC offset (0x90), Query (0x31)
        Device(SBS1){              // Smart Battery Subsystem
            Name(_HID, "ACPI0002") // Smart Battery Subsystem ID
            Name(_SBS, 0x3)        // Indicates support for three batteries
            }   // end of SBS1
        }       // end of SMB1
    }           // end of EC
```

11.2 Control Method Batteries

The following section illustrates the operation and definition of the Control Method Battery.

11.2.1 Battery Events

The AML code handling an SCI for a battery event notifies the system of which battery's status may have changed. The OS uses the _BST control method to determine the current status of the batteries and what action, if any, should be taken (for more information about the _BST control method, see section 11.2.2, "Battery Control Methods"). The typical action is to notify applications monitoring the battery status to provide the user with an up-to-date display of the system battery state. But in some cases, the action may involve generating an alert or even forcing a system into a sleeping state. In any case, any changes in battery status should generate an SCI in a timely manner to keep the system power state UI consistent with the actual state of the system battery (or batteries).

Unlike most other devices, when a battery is inserted or removed from the system, the device itself (the battery bay) is still considered to be present in the system. For most systems, the _STA for this device will always return a value with bits 0-3 set and will toggle bit 4 to indicate the actual presence of a battery (see section 6.3.6, "_STA [Status]"). When this insertion or removal occurs, the AML code handler for this event should issue a **Notify**(*battery_device*, 0x81) to indicate that the static battery information has changed. For systems that have battery slots in a docking station or batteries that cannot be surprise-removed, it may be beneficial or necessary to indicate that the entire device has been removed. In this case, the standard methods and notifications described in section 6.3, "Device Insertion and Removal Objects," should be used.

When the present state of the battery has changed or when the trip point set by the _BTP control method is reached or crossed, the hardware will assert a general purpose event. The AML code handler for this event issues a **Notify**(*battery_device*, 0x80) on the battery device.

In the case where the remaining battery capacity becomes critically low, the AML code handler issues a **Notify**(*battery_device*, 0x80) and reports the battery critical flag in the _BST object. The OS performs an emergency shutdown. For a full description of the critical battery state, see section 3.9.4, "Low Battery Levels."

Sometimes the value to be returned from _BST or _BIF will be temporarily unknown. In this case, the method may return the value 0xFFFFFFFF as a placeholder. When the value becomes known, the appropriate notification (0x80 for _BST or 0x81 for BIF) should be issued, in like manner to any other change in the data returned by these methods. This will cause OSPM to re-evaluate the method—obtaining the correct data value.

11.2.2 Battery Control Methods

The Control Method Battery is a battery with an AML code interface between the battery and the host PC. The battery interface is completely accessed by AML code control methods, allowing the OEM to use any type of battery and any kind of communication interface supported by ACPI. OSPM requires accurate battery data to perform optimal power management policy and to provide the end user with a meaningful estimation of remaining battery life. As such, control methods that return battery information should calculate this information rather than return hard coded data.

A Control Method Battery is described as a device object. Each device object supporting the Control Method Battery interface contains the following additional control methods. When there are two or more batteries in the system, each battery will have an independent device object in the name space.

Table 11-3 Battery Control Methods

Object	Description
_BIF	Returns static information about a battery (in other words, model number, serial number, design voltage, and so on).
_BST	Returns the current battery status (in other words, dynamic information about the battery, such as whether the battery is currently charging or discharging, an estimate of the remaining battery capacity, and so on).
_BTP	Sets the Battery Trip point, which generates an SCI when batterycapacity reaches the specified point.
_PCL	List of pointers to the device objects representing devices powered by the battery.
_STA	Returns general status of the battery (for a description of the _STA control method, see section 6.3.6, “_STA [Status]”).

A Control Method Battery device declaration in the ACPI name space requires the _GLK object if potentially contentious accesses to device resources are performed by non-OS code. See section 6.5.7, “_GLK (Global Lock),” for details about the _GLK object.

11.2.2.1 BIF (Battery Information)

This object returns the static portion of the Control Method Battery information. This information remains constant until the battery is changed.

Arguments:
None

Result Code:

```
Package {
// ASCIIZ is ASCII character string terminated with
// a 0x00.
       Power Unit                        //DWORD
       Design Capacity                   //DWORD
       Last Full Charge Capacity         //DWORD
       Battery Technology                //DWORD
       Design Voltage                    //DWORD
       Design Capacity of Warning        //DWORD
       Design Capacity of Low            //DWORD
       Battery Capacity Granularity 1    //DWORD
       Battery Capacity Granularity 2    //DWORD
       Model Number                      //ASCIIZ
       Serial Number                     //ASCIIZ
       Battery Type                      //ASCIIZ
       OEM Information                   //ASCIIZ
}
```

Table 11-4 _BIF Method Result Codes

Field	**Format**	**Description**
Power Unit	DWORD	Indicates the units used by the battery to report its capacity and charge/discharge rate information to the OS. 0x00000000 – Capacity information is reported in [mWh] and charge/discharge rate information in [mW]. 0x00000001 – Capacity information is reported in [mAh] and charge/discharge rate information in [mA].
Design Capacity	DWORD	Battery’s design capacity. Design Capacity is the nominal capacity of a new battery. The *Design Capacity* value is expressed as power [mWh] or current [mAh] depending on the *Power Unit* value. 0x000000000 – 0x7FFFFFFF (in [mWh] or [mAh]) 0xFFFFFFFF – Unknown design capacity
Last Full Charge Capacity	DWORD	Predicted battery capacity when fully charged. The *Last Full Charge Capacity* value is expressed as power (mWh) or current (mAh) depending on the *Power Unit* value. 0x000000000h – 0x7FFFFFFF (in [mWh] or [mAh]) 0xFFFFFFFF – Unknown last full charge capacity
Battery Technology	DWORD	0x00000000 – Primary (for example, non-rechargeable) 0x00000001 – Secondary (for example, rechargeable)
Design Voltage	DWORD	Nominal voltage of a new battery. 0x000000000 – 0x7FFFFFFF in [mV] 0xFFFFFFFF – Unknown design voltage
Design capacity of Warning	DWORD	OEM-designed battery warning capacity. See section 3.9.4, “Low Battery Levels.” 0x000000000 – 0x7FFFFFFF in [mWh] or [mAh]
Design Capacity of Low	DWORD	OEM-designed low battery capacity. See section 3.9.4, “Low Battery Levels.” 0x000000000 – 0x7FFFFFFF in [mWh] or [mAh]
Battery Capacity Granularity 1	DWORD	Battery capacity granularity between low and warning in [mAh] or [mWh].
Battery Capacity Granularity 2	DWORD	Battery capacity granularity between warning and Full in [mAh] or [mWh].
Model Number	ASCIIZ	OEM-specific Control Method Battery model number
Serial Number	ASCIIZ	OEM-specific Control Method Battery serial number
Battery Type	ASCIIZ	The OEM-specific Control Method Battery type
OEM Information	ASCIIZ	OEM-specific information for the battery that the UI uses to display the OEM information about the Battery. If the OEM does not support this information, this should be reserved as 0x00.

Notes: A secondary-type battery should report the corresponding capacity (except for Unknown).

On a multiple-battery system, all batteries in the system should return the same granularity.

Operating systems prefer these control methods to report data in terms of power (watts).

11.2.2.2 BST (Battery Status)

This object returns the present battery status. Whenever the *Battery State* value changes, the system will generate an SCI to notify the OS.

Arguments:
None

Result Code:

```
Package{
       Battery State                //DWORD
       Battery Present Rate         //DWORD
       Battery Remaining Capacity   //DWORD
       Battery Present Voltage      //DWORD
}
```

Table 11-5 _BST Method Result Codes

Field	**Format**	**Description**
Battery State	DWORD	Bit values. Notice that the *Charging* bit and the *Discharging* bit are mutually exclusive and must not both be set at the same time. Even in critical state, hardware should report the corresponding charging/discharging state. Bit0 – 1 indicates the battery is discharging. Bit1 – 1 indicates the battery is charging. Bit2 – 1 indicates the battery is in the critical energy state (see section 3.9.3, “Low Battery Levels”). This does not mean battery failure.
Battery Present Rate	DWORD	Returns the power or current being supplied or accepted through the battery’s terminals (direction depends on the *Battery State* value). The *Battery Present Rate* value is expressed as power [mWh] or current [mAh] depending on the *Power Unit* value. Batteries that are rechargeable and are in the discharging state are required to return a valid *Battery Present Rate* value. 0x00000000 – 0x7FFFFFFF in [mW] or [mA] 0xFFFFFFFF – Unknown rate

Table 11-5 _BST Method Result Codes *(continued)*

Field	Format	Description
Battery Remaining Capacity	DWORD	Returns the estimated remaining battery capacity. The *Battery Remaining Capacity* value is expressed as power [mWh] or current [mAh] depending on the *Power Unit* value. Batteries that are rechargeable are required to return a valid *Battery Remaining Capacity* value. 0x00000000 – 0x7FFFFFFF in [mWh] or [mAh] 0xFFFFFFFF – Unknown capacity
Battery Present Voltage	DWORD	Returns the voltage across the battery's terminals. Batteries that are rechargeable must report *Battery Present Voltage.* 0x000000000 – 0x7FFFFFFF in [mV] 0xFFFFFFFF – Unknown voltage **Note:** Only a primary battery can report unknown voltage.

Notice that when the battery is a primary battery (a non-rechargeable battery such as an Alkaline-Manganese battery) and cannot provide accurate information about the battery to use in the calculation of the remaining battery life, the Control Method Battery can report the percentage directly to OS. It does so by reporting the Last Full Charged Capacity =100 and BatteryPresentRate=0xFFFFFFFF. This means that Battery Remaining Capacity directly reports the battery's remaining capacity [%] as a value in the range 0 through 100 as follows:

$$\textbf{Remaining Battery Percentage[\%]} = \frac{\textbf{Battery Remaining Capacity [=0} \sim \textbf{100]}}{\textbf{Last Full Charged Capacity [=100]}} \; \textbf{* 100}$$

$$\textbf{Remaining Battery Life [h]} = \frac{\textbf{Battery Remaining Capacity [mAh/mWh]}}{\textbf{Battery Present Rate [=0xFFFFFFFF]}} \quad \textbf{= unknown}$$

11.2.2.3 BTP (Battery Trip Point)

This object is used to set a trip point to generate an SCI whenever the *Battery Remaining Capacity* reaches or crosses the value specified in the _BTP object. Specifically, if *Battery Remaining Capacity* is less than the last argument passed to _BTP, a notification must be issued when the value of *Battery Remaining Capacity* rises to be greater than or equal to this trip-point value. Similarly, if *Battery Remaining Capacity* is greater than the last argument passed to _BTP, a notification must be issued when the value of *Battery Remaining Capacity* falls to be less than or equal to this trip-point value. The last argument passed to _BTP will be kept by the system.

If the battery does not support this function, the _BTP control method is not located in the name space. In this case, the OS must poll the *Battery Remaining Capacity* value.

Arguments:
Level at which to set the trip point:
0x00000001 – 0x7FFFFFFF (in units of mWh or mAh, depending on the *Power Units* value)
0x00000000 – Clear the trip point

Result Code:
None

11.3 AC Adapters and Power Source Objects

The Power Source objects describe the power source used to run the system.

Table 11-6 Power Source Control Methods

Object	Description
_PSR	Returns present power source device.
_PCL	List of pointers to powered devices.

11.3.1 PSR (Power Source)

Returns the current power source devices. Used for the AC adapter and is located under the AC adapter object in name space. Used to determine if system is running off the AC adapter.

Arguments:
None

Result Code:
0x00000000 – Off-line
0x00000001 – On-line

11.3.2 PCL (Power Consumer List)

This object evaluates to a list of pointers, each pointing to a device or a bus powered by the power source device. Pointing to a bus indicates that all devices under the bus are powered by the power source device.

11.4 Example: Power Source Name Space

The ACPI name space for a computer with an AC adapter and two batteries associated with a docking station that has an AC adapter and a battery is shown in Figure 11.4.

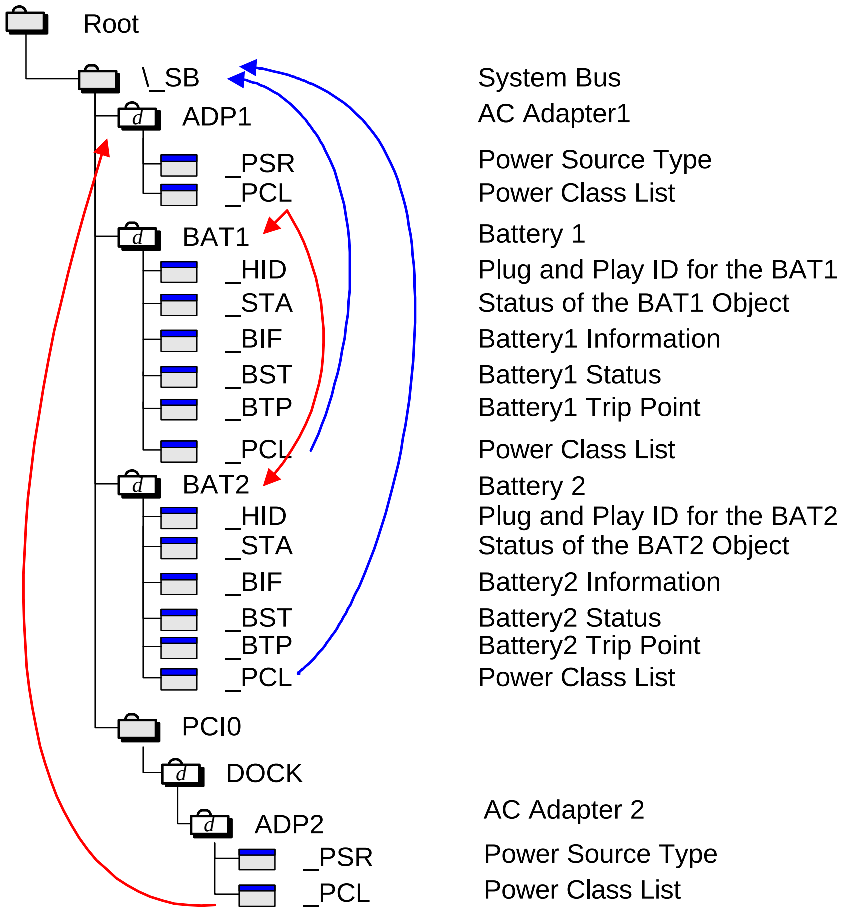

Figure 11-4 Power Source Name Space Example that Includes a Docking Station

12 Thermal Management

This section specifies the objects OSPM uses for thermal management of a platform.

12.1 Thermal Control

ACPI allows OSPM to be proactive in its system cooling policies. With OSPM in control of the operating environment, cooling decisions can be made based on the application load on the CPU and the thermal heuristics of the system. Graceful shutdown of the OS at critical heat levels becomes possible as well. The following sections describe the thermal objects available to OSPM to control platform temperature. ACPI expects all temperatures to be given in tenths of degrees Kelvin.

The ACPI thermal design is based around regions called *thermal zones.* Generally, the entire PC is one large thermal zone, but an OEM can partition the system into several thermal zones if necessary.

12.1.1 Active, Passive, and Critical Policies

There are three cooling policies that OSPM uses to control the thermal state of the hardware. The policies are *active, passive* and *critical.*

- **Active Cooling**. OSPM takes a direct action such as turning on a fan. Active cooling devices typically consume power and produce some amount of noise when enabled (_ON), but are able to cool a thermal zone without limiting system performance. The _ACx objects declare the temperature thresholds OSPM uses to decide when to start or stop different active cooling devices.
- **Passive Cooling**. OSPM reduces the power consumption of devices to reduce the temperature of a thermal zone, such as slowing (throttling) the processor clock. Passive cooling devices typically produce no user-noticeable noise. The _PSV control method specifies the temperature threshold where OSPM will start or stop passive cooling.
- **Critical Trip Points**. These are threshold temperatures at which OSPM performs an orderly, but critical, shutdown of the system. The _HOT object declares the critical temperature at which OSPM may choose to transition the system into the S4 sleeping state, if supported, The _CRT object declares the critical temperature at which OSPM must perform a critical shutdown.

When a thermal zone appears, OSPM runs control methods in the thermal zone to retrieve the temperature thresholds (trip points) at which it executes a cooling policy. When OSPM receives a temperature change notification it will run the _TMP control method, which returns the current temperature of the thermal zone. OSPM checks the current temperature against the temperature thresholds. If _TMP is greater than or equal to _AC*x* then OSPM will turn on the associated active cooling device(s). If _TMP is greater than or equal to _PSV then OSPM will perform passive cooling. If _TMP is greater than or equal to _HOT then OSPM may choose to transition the system into the S4 sleeping state, if supported. Finally, if _TMP is greater than or equal to _CRT then OSPM will shut the system down. OSPM must also evaluate _TMP when any thermal zone appears in the namespace (for example, during system initialization) and must initiate a cooling policy as warranted independent of receipt of a temperature change notification. This allows OSPM to cool systems containing a thermal zone whose temperature has already exceeded temperature thresholds at initialization time.

An optimally designed system that uses several thresholds can notify OSPM of thermal increase or decrease by raising an SCI every several degrees. This enables OSPM to anticipate thermal trends and incorporate heuristics to better manage the system's temperature.

The OS can also request that the platform change the priority of active cooling (performance) versus passive cooling (energy conservation/silence) by invoking the _SCP (Set Cooling Policy) method.

12.1.2 Dynamically Changing Cooling Temperatures

An OEM can reset _AC*x* and _PSV and notify OSPM to reevaluate the control methods to retrieve the new policy threshold settings. The following are the primary uses for this type of thermal notification:

- When OSPM changes the platform's cooling policy from one cooling mode to the other.
- When a swappable bay device is inserted or removed. A swappable bay is a slot that can accommodate several different devices that have identical form factors, such as a CD-ROM drive, disk drive, and so on. Many mobile PCs have this concept already in place.
- When the temperature reaches an _AC*x* or _PSV policy threshold to implement hysteresis.

In each situation, the OEM-provided AML code must execute a **Notify**(*thermal_zone*, 0x81) statement to request OSPM to re-evaluate the policy thresholds by obtaining the current values for the _AC*x* and _PSV objects.

12.1.2.1 OSPM Change of Cooling Policy

When OSPM changes the platform's cooling policy from one cooling mode to the other, the following occurs:

1. OSPM notifies the platform of the new cooling mode by running the Set Cooling Policy (_SCP) control method in all thermal zones.
2. Thresholds are updated in the hardware and OSPM is notified of the change.
3. OSPM re-evaluates _AC*x* and _PSV to obtain the new temperature thresholds.

12.1.2.2 Resetting Cooling Temperatures to Adjust to Bay Device Insertion or Removal

The platform can adjust the thermal zone temperature to accommodate the maximum operating temperature of a bay device as necessary. For example:

1. Hardware detects that a device was inserted into or removed from the bay, updates the temperature thresholds, and then notifies OSPM of the thermal policy change and device insertion events.
2. OSPM re-enumerates the devices and reevaluates _AC*x* and _PSV.

12.1.2.3 Resetting Cooling Temperatures to Implement Hysteresis

An OEM can build hysteresis into platform thermal design by dynamically resetting cooling temperature thresholds. For example:

1. When the temperature increases to the designated threshold, OSPM will turn on the associated active cooling device or perform passive cooling.
2. The platform resets the threshold value to a lower temperature (to implement hysteresis) and notifies OSPM of the change. Because of this new threshold value, the fan will be turned off at a lower temperature than when it was turned on (therefore implementing a negative hysteresis).
3. When the temperature hits the lower threshold value, OSPM will turn off the associated active cooling device or cease passive cooling. The hardware will reset _AC*x* to its original value and notify OSPM that the trip points have once again been altered.

12.1.3 Detecting Temperature Changes

The ability of hardware to asynchronously notify an ACPI-compatible OS of meaningful changes in the thermal zone's temperature is a highly desirable capability that relieves OSPM from implementing a poll-based policy and generally results in a much more responsive and accurate environment. Each notification instructs OSPM to evaluate whether a trip point has been crossed and allows OSPM to anticipate temperature trends for the thermal zone.

It is recognized that much of the hardware used to implement thermal zone functionality today is not capable of generating ACPI-visible notifications (SCIs) or only can do so with wide granularity (for example, only when the temperature crosses the critical threshold). In these environments, OSPM must poll the thermal zone's temperature periodically to implement an effective policy.

While ACPI specifies a mechanism that enables OSPM to poll thermal zone temperature, platform reliance on thermal zone polling is strongly discouraged by this specification. OEMs should design systems that asynchronously notify OSPM whenever a meaningful change in the zone's temperature occurs – relieving the OS of the overhead associated with polling. In some cases, embedded controller firmware can overcome limitations of existing thermal sensor capabilities to provide the desired asynchronous notification.

Notice that the _TZP (thermal zone polling) object is used to indicate whether a thermal zone must be polled by OSPM, and if so, a recommended polling frequency. See section 12.3.13, "_TZP," for more information.

12.1.3.1 Hardware Notifications

Hardware that supports asynchronous temperature change notifications does so using an SCI. The OEM-provided AML that responds to this SCI must execute a **Notify**(*thermal_zone*, 0x80) statement to inform OSPM that a meaningful change in temperature has occurred. When OSPM receives this thermal event, it will run the _TMP control method to evaluate the current temperature. OSPM will then compare the value to the cooling policy temperatures. If the temperature has crossed over any of the policy thresholds, then OSPM will actively or passively cool (or stop cooling) the system, or shut the system down entirely.

Both the number and granularity of thermal zone trip points are OEM-specific. However, it is important to notice that since OSPM can use heuristic knowledge to help cool the system, the more events OSPM receives the better understanding it will have of the system's thermal characteristic.

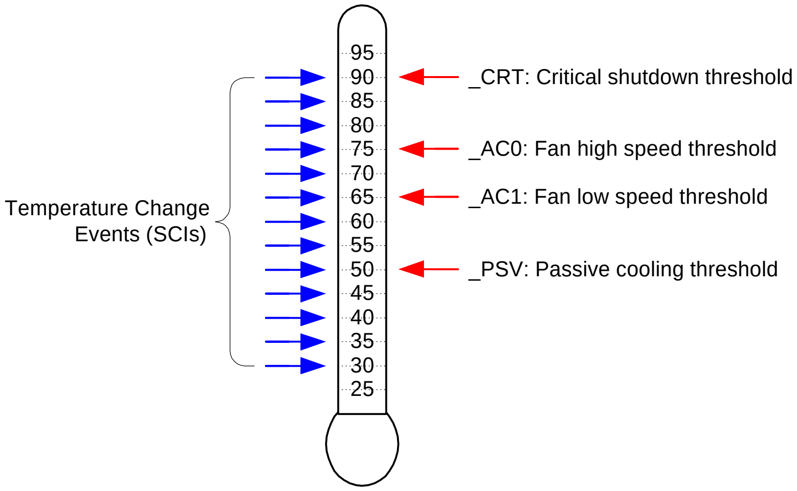
Temperature Change
Events (SCIs)
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
_CRT: Critical shutdown threshold
_AC0: Fan high speed threshold
_AC1: Fan low speed threshold
_PSV: Passive cooling threshold

Figure 12-1 SCI Events

For example, the thermal zone illustrated above includes hardware that will generate a temperature change notification using a 5° Celsius granularity. All thresholds (_PSV, _AC1, _AC0, and _CRT) exist within the monitored range and fall on 5° boundaries. This granularity is appropriate for this system as it provides sufficient opportunity for OSPM to detect when a threshold is crossed as well as to understand the thermal zone's basic characteristics (temperature trends).

Note: The ACPI specification defines Kelvin as the standard unit for temperature. All thermal zone objects must report temperatures in Kelvin. All figures and examples in this section of the specification use Celsius for reasons of clarity. ACPI allows Kelvin to be declared in precision of 1/10th of a degree (for example, 310.5). Kelvin is expressed as θ/K = T/°C + 273.2.

12.1.3.2 Polling

Platforms that are not capable of generating SCIs for thermal change events or that can only do so for a few thresholds should inform OSPM to implement a poll-based policy. OSPM does this to ensure that temperature changes across threshold boundaries are always detectable.

Polling can be done in conjunction with hardware notifications. For example, thermal zone hardware that only supports a single threshold might be configured to use this threshold as the critical temperature trip point. Assuming that hardware monitors the temperature at a finer granularity than OSPM would, this environment has the benefit of being more responsive when the system is overheating.

A thermal zone advertises the need to be polled by OSPM via the _TZP object. The absence of this control method informs OSPM to implement polling using an OS-provided default frequency. See section 12.3.13, "_TZP," for more information.

12.1.4 Active Cooling

Active cooling devices typically consume power and produce some amount of noise when enabled (_ON). These devices are able to cool a thermal zone without limiting the performance of a device.

The active cooling methods (_AC*x*) in conjunction with the active cooling lists (_AL*x*) allow an OEM to use a device that offers varying degrees of cooling capability or multiple cooling devices. The _AC*x* method designates the temperature where Active cooling is engaged or disengaged (depending upon the direction in which the temperature is changing). The _AL*x* object evaluates to a list of devices that actively cool the zone. For example:

- If a standard single-speed fan is the Active cooling device, then _AC0 evaluates to the temperature where active cooling is engaged and the fan is listed in _AL0.
- If the zone uses two independently controlled single-speed fans to regulate the temperature, then _AC0 will evaluate to the maximum cooling temperature using two fans, and _AC1 will evaluate to the standard cooling temperature using one fan.
- If a zone has a single fan with a low speed and a high speed, the _AC0 will evaluate to the temperature associated with running the fan at high-speed, and _AC1 will evaluate to the temperature associated with running the fan at low speed. _AL0 and _AL1 will both point to different device objects associated with the same physical fan, but control the fan at different speeds.

For ASL coding examples that illustrate these points, see sections 12.4, "Thermal Zone Object Requirements," and 12.5, "Thermal Zone Examples."

12.1.5 Passive Cooling

Passive cooling devices are able to cool a thermal zone without creating noise and without consuming additional power (actually saving power), but do so by lowering the performance of the system.

12.1.5.1 Processor Clock Throttling

The processor passive cooling threshold (_PSV) in conjunction with the processor list (_PSL) allows an OEM to indicate the temperature at which clock throttling will be applied to the processor(s) residing in a given thermal zone. Unlike other cooling policies, during passive cooling of processors OSPM takes the initiative to actively monitor the temperature in order to cool the platform.

On an ACPI-compatible platform that properly implements CPU throttling, the temperature transitions will be similar to the following figure.

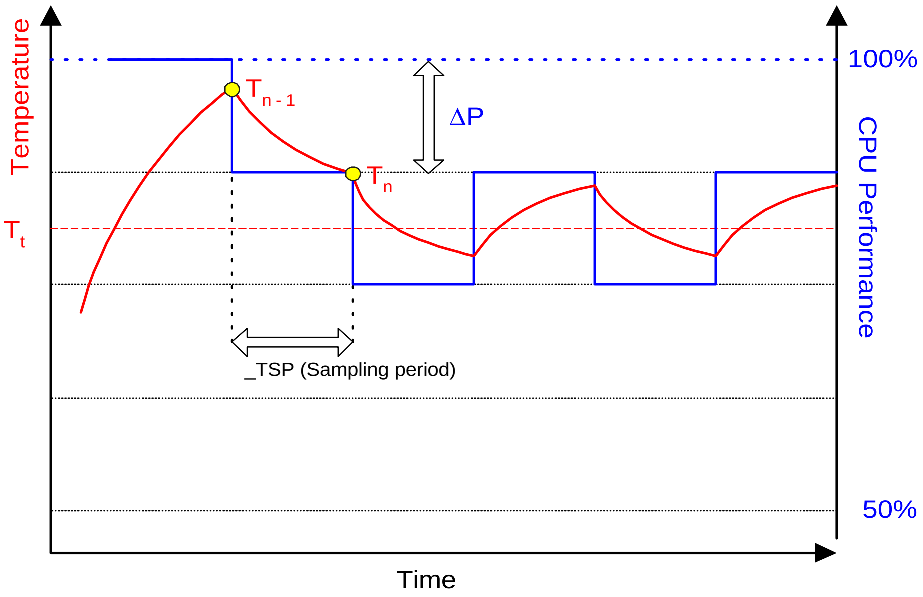

Figure 12-2 Temperature and CPU Performance Versus Time

The following equation should be used by OSPM to assess the optimum CPU performance change necessary to lower the thermal zone’s temperature:

Equation #1: ΔP [%] = _TC1 * (Tn - Tn-1) + _TC2 * (Tn - Tt)
Where:

Tn = current temperature

Tt = target temperature (_PSV)

The two coefficients _TC1 and _TC2 and the sampling period _TSP are hardware-dependent constants the OEM must supply to OSPM (for more information, see section 12.3, “Thermal Objects”). The object _TSP contains a time interval that OSPM uses to poll the hardware to sample the temperature. Whenever _TSP time has elapsed, OSPM will run _TMP to sample the current temperature (shown as Tn in the above equation). Then OSPM will use the sampled temperature and _PSV (which is the target temperature Tt) to evaluate the equation for ΔP. The granularity of ΔP is determined by the CPU duty width of the system.

Note: Equation #1 has an implied formula.

Equation #2: Pn = Pn-1 + HW[- ΔP] where 0% <= Pn <= 100%

For Equation #2, whenever Pn-1 + ΔP lies outside the range 0-100%, then Pn will be truncated to 0-100%. For hardware that cannot assume all possible values of Pn between 0 and 100%, a hardware-specific mapping function HW is used.

In addition, the hardware mapping function in Equation #2 should be interpreted as follows:

1. If the right hand side of Equation #1 is negative, HW[ΔP] is rounded to the next available higher setting of frequency.
2. If the right hand side of Equation #1 is positive, HW[ΔP] is rounded to the next available lower setting of frequency.

The calculated Pn becomes Pn-1 during the next sampling period.

For more information about CPU throttling, see section 8.1.1, Processor Power State C0.” A detailed explanation of this thermal feedback equation is beyond the scope of this specification.

12.1.6 Critical Shutdown

When the thermal zone temperature reaches the threshold indicated by _CRT, OSPM must immediately shut the system down. The system must disable the power either after the temperature reaches some hardware-determined level above _CRT or after a predetermined time has passed. Before disabling power, platform designers should incorporate some time that allows OSPM to run its critical shutdown operation. There is no requirement for a minimum shutdown operation window that commences immediately after the temperature reaches _CRT. This is because:

- Temperature might rise rapidly in some systems and slowly on others, depending on casing design and environmental factors.
- Shutdown can take several minutes on a server and only a few short seconds on a hand-held device.

Because of this indistinct discrepancy and the fact that a critical heat situation is a remarkably rare occurrence, ACPI does not specify a target window for a safe shutdown. It is entirely up to the OEM to build in a safe buffer that it sees fit for the target platform.

12.2 Cooling Preferences

A robust OSPM implementation provides the means for the end user to convey a preference (or a level of preference) for either performance or energy conservation to OSPM. Allowing the end user to choose this preference is most critical to mobile system users where maximizing system run-time on a battery charge often has higher priority over realizing maximum system performance. For example, if a user is taking notes on her PC in a quiet environment, such as a library or a corporate meeting, she may want the system to emphasize passive cooling so that the system operates quietly, even at the cost of system performance.

A user preference towards performance corresponds to the Active cooling mode while a user's preference towards energy conservation or quiet corresponds to the Passive cooling mode. ACPI defines an interface to convey the cooling mode to the platform. Active cooling can be performed with minimal OSPM thermal policy intervention. For example, the platform indicates through thermal zone parameters that crossing a thermal trip point requires a fan to be turned on. Passive cooling requires OSPM thermal policy to manipulate device interfaces that reduce performance to reduce thermal zone temperature.

Either cooling mode will be activated only when the thermal condition requires it. When the thermal zone is at an optimal temperature level where it does not warrant any cooling, both modes result in a system operating at its maximum potential with all fans turned off.

Thermal zones supporting the _SCP control method allow the user to switch the system's cooling mode emphasis. See section 12.3.7, "_SCP," for more information.

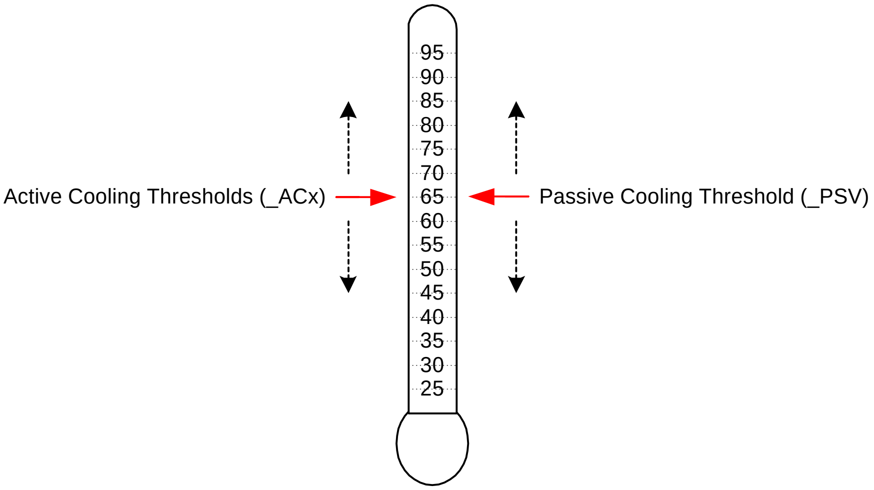
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
Active Cooling Thresholds (_ACx)
Passive Cooling Threshold (_PSV)

Figure 12-3 Active and Passive Threshold Values

As illustrated in Figure 12-3, OEMs must choose the value for each threshold to instruct OSPM to initiate the cooling policies at the desired target temperatures. OEMs can emphasize active or passive cooling modes by assigning different threshold values. Generally, if _ACx is set lower than _PSV, then the system emphasizes active cooling. Conversely, if _PSV is set lower than _ACx, then the emphasis is placed on passive cooling.

For example, a thermal zone that includes a processor and one single-speed fan may use _PSV to indicate the temperature value at which OSPM would enable passive cooling and _AC0 to indicate the temperature at which the fan would be turned on. If the value of _PSV is less than _AC0 then the system will favor passive cooling (for example, CPU clock throttling). On the other hand, if _AC0 is less than _PSV the system will favor active cooling (in other words, using the fan). See Figure 12-4 below.

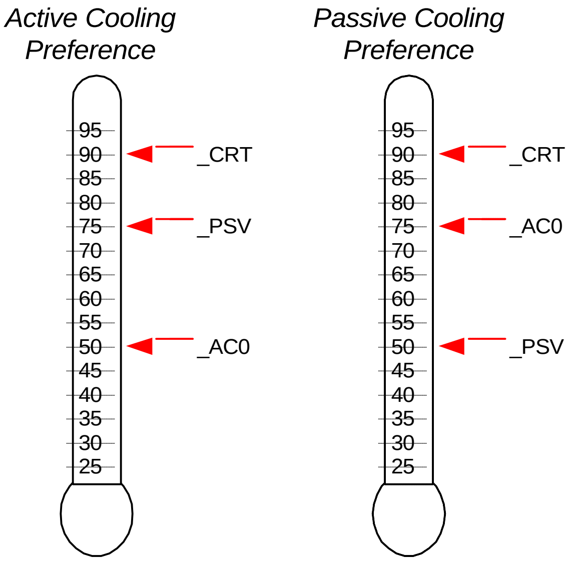

Figure 12-4 Cooling Preferences

The example on the left enables active cooling (for example, turn on a fan) when OSPM detects the temperature has risen above 50°. If for some reason the fan does not reduce the system temperature, then at 75° OSPM will initiate passive cooling (for example, CPU throttling) while still running the fan. If the temperature continues to climb, OSPM will quickly shut the system down when the temperature reaches 90°C. The example on the right is similar but the _AC0 and _PSV threshold values have been swapped to emphasize passive cooling.

The ACPI thermal model allows flexibility in the thermal zone design. An OEM that needs a less elaborate thermal implementation may consider using only a single threshold (for example, _CRT). Complex thermal implementations can be modeled using multiple active cooling thresholds and devices, or through the use of additional thermal zones.

12.2.1 Evaluating Thermal Device Lists

The **Notify**(*thermal_zone*, 0x82) statement can be used to inform OSPM that a change has been made to the thermal zone device lists. This thermal event instructs OSPM to re-evaluate the _ALx, _PSL, and _TZD objects.

For example, a system that supports the dynamic insertions of processors might issue this notification to inform OSPM of changes to _PSL following the insertion or removal of a processor. OSPM would re-evaluate all thermal device lists and adjust its policy accordingly.

Notice that this notification can be used with the **Notify**(*thermal_zone*, 0x81) statement to inform OSPM to both re-evaluate all device lists and all thresholds.

12.3 Thermal Objects

Control methods and objects related to thermal management are listed in Table 12-1.

Table 12-1 Thermal Control Methods

Object	**Description**
_AC*x*	Returns active cooling policy threshold values in tenths of degrees Kelvin.
_AL*x*	List of active cooling device objects.
_CRT	Returns critical trip point in tenths of degrees Kelvin where OSPM must perform a critical shutdown.
_HOT	Returns critical trip point in tenths of degrees Kelvin where OSPM may choose to transition the system into S4.
_PSL	List of processor device objects for clock throttling.
_PSV	Returns the passive cooling policy threshold value in tenths of degrees Kelvin.
_SCP	Sets platform cooling policy (active or passive).
_TC1	Thermal constant for passive cooling.
_TC2	Thermal constant for passive cooling.
_TMP	Returns the thermal zone’s current temperature in tenths of degrees Kelvin.
_TSP	Thermal sampling period for Passive cooling in tenths of seconds.
_TZD	List of devices whose temperature is measured by this thermal zone.
_TZP	Thermal zone polling frequency in tenths of seconds.

12.3.1 AC*x* (Active Cooling)

This object returns the temperature at which OSPM must start or stop Active cooling, where *x* is a value between 0 and 9 that designates multiple active cooling levels of the thermal zone. If the Active cooling device has one cooling level (that is, “on”) then that cooling level is named _AC0. If the cooling device has two levels of capability, such as a high fan speed and a low fan speed, then they are named _AC0 and _AC1 respectively. The smaller the value of *x*, the greater the cooling strength _AC*x* represents. In the above example, _AC0 represents the greater level of cooling (the faster fan speed) and _AC1 represents the lesser level of cooling (the slower fan speed). For every AC*x* method, there must be a matching AL*x* object.

Arguments:
None

Result Code:
Active cooling temperature threshold in tenths of degrees Kelvin.

The result code is an integer value that represents tenths of degrees Kelvin. For example, 300.0K is represented by the integer 3000.

12.3.2 AL*x* (Active List)

This object evaluates to a list of Active cooling devices to be turned on when the associated _AC*x* temperature threshold is exceeded. For example, these devices could be fans.

Arguments:

None

Result Code:

A package consisting of references to all active cooling devices that should be engaged when the associated active cooling threshold (_ACx) is exceeded.

12.3.3 CRT (Critical Temperature)

This object returns the critical temperature at which OSPM must shutdown the system.

Arguments:

None

Result Code:

Critical temperature threshold in tenths of degrees Kelvin.

The result is an integer value that represents tenths of degrees Kelvin. For example, 300.0K is represented by the integer 3000.

12.3.4 HOT (Hot Temperature)

This object returns the critical temperature at which OSPM may choose to transition the system into the S4 sleeping state. The platform vendor should define _HOT to be far enough below _CRT so as to allow OSPM enough time to transition the system into the S4 sleeping state. While dependent on the amount of installed memory, on typical platforms OSPM implementations can transition the system into the S4 sleeping state in tens of seconds.

Arguments:

None

Result Code:

Critical temperature threshold in tenths of degrees Kelvin.

The result is an integer value that represents tenths of degrees Kelvin. For example, 300.0K is represented by the integer 3000.

12.3.5 PSL (Passive List)

This object evaluates to a list of processor objects to be used for passive cooling.

Arguments:

None

Result Code:

A package consisting of references to all processor objects that will be used for passive cooling when the passive cooling threshold (_PSV) is exceeded.

12.3.6 PSV (Passive)

This object returns the temperature at which OSPM must activate passive cooling policy.

Arguments:

None

Result Code:

Passive cooling temperature threshold in tenths of degrees Kelvin.

The result code is an integer value that represents tenths of degrees Kelvin. For example, 300.0 Kelvin is represented by 3000.

12.3.7 SCP (Set Cooling Policy)

This control method sets the platform's cooling mode policy setting. The hardware can use this as a trigger to reassign _AC*x* and _PSV temperatures. The OS will automatically evaluate _AC*x* and _PSV objects after executing _SCP.

Arguments:

0 – Active

1 – Passive

Result Code:

None

12.3.8 TC1 (Thermal Constant 1)

This object evaluates to the constant _TC1 for use in the Passive cooling formula:

ΔPerformance [%]= _TC1 * (Tn - Tn-1) + _TC2 * (Tn. - Tt)

Arguments:

None

Result Code:

Integer value of thermal constant #1.

12.3.9 TC2 (Thermal Constant 2)

This object evaluates to the constant _TC2 for use in the Passive cooling formula:

ΔPerformance [%]= _TC1 * (Tn - Tn-1) + _TC2 *(Tn - Tt)

Arguments:

None

Result Code:

Integer value of thermal constant #2.

12.3.10 TMP (Temperature)

This control method returns the thermal zone’s current operating temperature in Kelvin.

Arguments:

None

Result Code:

The current temperature of the thermal zone in tenths of degrees Kelvin. For example, 300.0K is represented by the integer 3000.

12.3.11 TSP (Thermal Sampling Period)

This object evaluates to a thermal sampling period (in tenths of seconds) used by OSPM to implement the Passive cooling equation. This value, along with _TC1 and _TC2, will enable OSPM to provide the proper hysteresis required by the system to accomplish an effective passive cooling policy. The granularity of the sampling period is 0.1 seconds. For example, if the sampling period is 30.0 seconds, then _TSP needs to report 300; if the sampling period is 0.5 seconds, then it will report 5. OSPM can normalize the sampling over a longer period if necessary.

Arguments:

None

Result Code:

Thermal sampling period for passive cooling, in tenths of seconds.

12.3.12 TZD (Thermal Zone Devices)

This optional object evaluates to a package of device names. Each name corresponds to a device in the ACPI namespace that is associated with the thermal zone. The temperature reported by the thermal zone is roughly correspondent to that of each of the devices.

The list of devices returned by the control method need not be a complete and absolute list of devices affected by the thermal zone. However, the package should at least contain the devices that would uniquely identify where this thermal zone is located in the machine. For example, a thermal zone in a docking station should include a device in the docking station, a thermal zone for the CD-ROM bay, should include the CD-ROM.

Arguments:

None

Result Code:

A package consisting of references to devices associated with the thermal zone.

12.3.13 TZP (Thermal Zone Polling)

This optional object evaluates to a *recommended* polling frequency (in tenths of seconds) for this thermal zone. A value of zero indicates that OSPM does not need to poll the temperature of this thermal zone in order to detect temperature changes (the hardware is capable of generating asynchronous notifications). Notice that the absence of _TZP informs OSPM to implement polling using an OS-provided default frequency.

The use of polling is allowed but strongly discouraged by this specification. OEMs should design systems that asynchronously notify OSPM whenever a meaningful change in the zone’s temperature occurs—relieving the OS of the overhead associated with polling. See section 12.1.3, “Detecting Temperature Changes,” for more information.

This value is specified as tenths of seconds with a 1 second granularity. A minimum value of 30 seconds (_TZP evaluates to 300) and a maximum value of 300 seconds (in other words, 5 minutes) (_TZP evaluates to 3000) may be specified. As this is a *recommended* value, OSPM will consider other factors when determining the actual polling frequency to use.

Arguments:

None

Result Code:

The recommended polling frequency, in tenths of seconds. A value of zero indicates that polling is not necessary.

12.4 Thermal Zone Object Requirements

While not all thermal zone objects are required to be present in each thermal zone defined in the namespace, OSPM levies conditional requirements for the presence of specific thermal zone objects based on the definition of other related thermal zone objects. These requirements are outlined below:

- All thermal zones must contain the _TMP object.
- A thermal zone must define at least one trip point: _CRT, _HOT, _ACx, or _PSV.
- If _ACx is defined then an associated _ALx must be defined (e.g. defining _AC0 requires _AL0 also be defined).
- If _PSV is defined then either _PSL or _TZD must be defined. _PSL and _TZD may both be defined.
- If _PSL is defined then:

If a performance control register is defined (via either P_BLK or _PTC) for a processor defined in _PSL then _TC1, _TC2, and _TSP must be defined.

If a performance control register is not defined (via either P_BLK or _PTC) for a processor defined in _PSL then the processor must support processor performance states (in other words, the processor's processor object must include _PCT, _PSS, and _PPC).

- If _PSV is defined and _PSL is not defined (in other words, only _TZD is defined) then at least one device in the _TZD device list must support device performance states.
- _SCP is optional.
- _TZD is optional outside of the _PSV requirement outlined above.
- If _HOT is defined then the system must support the S4 sleeping state.

12.5 Thermal Zone Examples

12.5.1 Example: The Basic Thermal Zone

The following ASL describes a basic configuration where the entire system is treated as a single thermal zone. Cooling devices for this thermal zone consist of a processor and one single-speed fan. This is an example only.

Notice that this thermal zone object (TZ0) is defined in the _SB scope. Thermal zone objects should appear in the namespace under the portion of the system that comprises the thermal zone. For example, a thermal zone that is isolated to a docking station should be defined within the scope of the docking station device. Besides providing for a well-organized namespace, this configuration allows OSPM to dynamically adjust its thermal policy as devices are added or removed from the system.

```
Scope(\_SB) {
    Processor(
        CPU0,
        1,          // unique number for this processor
        0x110,      // system IO address of Pblk Registers
        0x06        // length in bytes of PBlk
        ) {}
Scope(\_SB.PCI0.ISA0) {
    Device(EC0) {
        Name(_HID, EISAID("PNP0C09"))     // ID for this EC
        // current resource description for this EC
        Name(_CRS,
                    ResourceTemplate() {
                        IO(Decode16,0x62,0x62,0,1)
                        IO(Decode16,0x66,0x66,0,1)
            })
        Name(_GPE, 0)      // GPE index for this EC

        // create EC's region and field for thermal support
        OperationRegion(EC0, EmbeddedControl, 0, 0xFF)
        Field(EC0, ByteAcc, Lock, Preserve) {
            MODE,   1,      // thermal policy (quiet/perform)
            FAN,    1,      // fan power (on/off)
            ,       6,      // reserved
            TMP,    8,      // current temp
            AC0,    8,      // active cooling temp (fan high)
            ,       8,      // reserved
            PSV,    8,      // passive cooling temp
            HOT     8,      // critical S4 temp
            CRT,    8       // critical temp
            }

        // following is a method that OSPM will schedule after
        // it receives an SCI and queries the EC to receive value 7
        Method(_Q07) {
            Notify (\_SB.PCI0.ISA0.EC0.TZ0, 0x80)
        }   // end of Notify method

        // fan cooling on/off - engaged at AC0 temp
        PowerResource(PFAN, 0, 0) {
            Method(_STA) { Return (\_SB.PCI0.ISA0.EC0.FAN) }     // check power state
            Method(_ON) { Store (One, \_SB.PCI0.ISA0.EC0.FAN) }         // turn on fan
            Method(_OFF) { Store ( Zero, \_SB.PCI0.ISA0.EC0.FAN) }      // turn off fan
        }

        // Create FAN device object
        Device (FAN) {
            // Device ID for the FAN
            Name(_HID, EISAID("PNP0C0B"))
            // list power resource for the fan
            Name(_PR0, Package(){PFAN})
        }

        // create a thermal zone
        ThermalZone (TZ0) {
            Method(_TMP) { Return (\_SB.PCI0.ISA0.EC0.TMP )} // get current temp
            Method(_AC0) { Return (\_SB.PCI0.ISA0.EC0.AC0) } // fan high temp
            Name(_AL0, Package(){\_SB.PCI0.ISA0.EC0.FAN})    // fan is act cool dev
            Method(_PSV) { Return (\_SB.PCI0.ISA0.EC0.PSV) } // passive cooling temp
            Name(_PSL, Package (){\_SB.CPU0})                // passive cooling devices
            Method(_HOT) { Return (\_SB.PCI0.ISA0.EC0.HOT) } // get critical S4 temp
            Method(_CRT) { Return (\_SB.PCI0.ISA0.EC0.CRT) } // get critical temp
            Method(_SCP, 1) { Store (Arg1, \_SB.PCI0.ISA0.EC0.MODE) } // set cooling mode
            Name(_TC1, 4)                                    // bogus example constant
            Name(_TC2, 3)                                    // bogus example constant
            Name(_TSP, 150)                                  // passive sampling = 15 sec
            Name(_TZP, 0)                                    // polling not required
        } // end of TZ0

    }   // end of ECO
}   // end of \_SB.PCI0.ISA0 scope-

} // end of \_SB scope
```

12.5.2 Example: Multiple-Speed Fans

The following ASL describes a thermal zone consisting of a processor and one dual-speed fan. As with the previous example, this thermal zone object (TZ0) is defined in the _SB scope and represents the entire system. This is an example only.

```
Scope(\_SB) {
    Processor(
        CPU0,
        1,          // unique number for this processor
        0x110,      // system IO address of Pblk Registers
        0x06        // length in bytes of PBlk
        ) {}


Scope(\_SB.PCI0.ISA0) {
    Device(EC0) {
        Name(_HID, EISAID("PNP0C09"))     // ID for this EC
        // current resource description for this EC
        Name(_CRS,
                    ResourceTemplate() {
                        IO(Decode16,0x62,0x62,0,1)
                        IO(Decode16,0x66,0x66,0,1)
            })
        Name(_GPE, 0)      // GPE index for this EC

        // create EC's region and field for thermal support
        OperationRegion(EC0, EmbeddedControl, 0, 0xFF)
        Field(EC0, ByteAcc, Lock, Preserve) {
            MODE,   1,      // thermal policy (quiet/perform)
            FAN0,   1,      // fan strength high/off
            FAN1,   1,      // fan strength low/off
            ,       5,      // reserved
            TMP,    8,      // current temp
            AC0,    8,      // active cooling temp (high)
            AC1,    8,      // active cooling temp (low)
            PSV,    8,      // passive cooling temp
            HOT     8,      // critical S4 temp
            CRT,    8       // critical temp
        }

        // following is a method that OSPM will schedule after it
        // receives an SCI and queries the EC to receive value 7
        Method(_Q07) {
            Notify (\_SB.PCI0.ISA0.EC0.TZ0, 0x80)
        } end of Notify method

        // fan cooling mode high/off - engaged at AC0 temp
        PowerResource(FN10, 0, 0) {
            Method(_STA) { Return (\_SB.PCI0.ISA0.EC0.FAN0) }    // check power state
            Method(_ON) { Store (One, \_SB.PCI0.ISA0.EC0.FAN0) } // turn on fan at high
            Method(_OFF) { Store (Zero, \_SB.PCI0.ISA0.EC0.FAN0) }// turn off fan
        }

        // fan cooling mode low/off   - engaged at AC1 temp
        PowerResource(FN11, 0, 0) {
            Method(_STA) { Return (\_SB.PCI0.ISA0.EC0.FAN1) }    // check power state
            Method(_ON) { Store (One, \_SB.PCI0.ISA0.EC0.FAN1) } // turn on fan at low
            Method(_OFF) { Store (Zero, \_SB.PCI0.ISA0.EC0.FAN1) }// turn off fan
        }
        // Following is a single fan with two speeds.  This is represented
        // by creating two logical fan devices.  When FN2 is turned on then
        // the fan is at a low speed.  When FN1 and FN2 are both on then
        // the fan is at high speed.
        //
        // Create FAN device object FN1
        Device (FN1) {
            // Device ID for the FAN
            Name(_HID, EISAID("PNP0C0B"))
            Name(_UID, 0)
            Name(_PR0, Package(){FN10, FN11})
        }

        // Create FAN device object FN2
        Device (FN2) {
            // Device ID for the FAN
            Name(_HID, EISAID("PNP0C0B"))
            Name(_UID, 1)
            Name(_PR0, Package(){FN10})
        }

        // create a thermal zone
        ThermalZone (TZ0) {
            Method(_TMP) { Return (\_SB.PCI0.ISA0.EC0.TMP )}  // get current temp
            Method(_AC0) { Return (\_SB.PCI0.ISA0.EC0.AC0) }  // fan high temp
            Method(_AC1) { Return (\_SB.PCI0.ISA0.EC0.AC1) }  // fan low temp
            Name(_AL0, Package() {\_SB.PCI0.ISA0.EC0.FN1})    // active cooling (high)
            Name(_AL1, Package() {\_SB.PCI0.ISA0.EC0.FN2})    // active cooling (low)
            Method(_PSV) { Return (\_SB.PCI0.ISA0.EC0.PSV) }  // passive cooling temp
            Name(_PSL, Package() {\_SB.CPU0})                // passive cooling devices
            Method(_HOT) { Return (\_SB.PCI0.ISA0.EC0.HOT) } // get critical S4 temp
            Method(_CRT) { Return (\_SB.PCI0.ISA0.EC0.CRT) } // get crit. temp
            Method(_SCP, 1) { Store (Arg1, \_SB.PCI0.ISA0.EC0.MODE) } // set cooling mode
            Name(_TC1, 4)                                    // bogus example constant
            Name(_TC2, 3)                                    // bogus example constant
            Name(_TSP, 150)                                  // passive sampling = 15 sec
            Name(_TZP, 0)                                    // polling not required
        } // end of TZ0

    } // end of ECO
}   // end of \_SB.PCI0.ISA0 scope


} // end of \_SB scope
```

13 ACPI Embedded Controller Interface Specification

ACPI defines a standard hardware and software communications interface between an OS driver and an embedded controller. This allows any OS to provide a standard driver that can directly communicate with an embedded controller in the system, thus allowing other drivers within the system to communicate with and use the resources of system embedded controllers. This in turn enables the OEM to provide platform features that the OS OSPM and applications can take advantage of.

ACPI also defines a standard hardware and software communications interface between an OS driver and an Embedded Controller-based SMB-HC (EC-SMB-HC).

The ACPI standard supports multiple embedded controllers in a system, each with its own resources. Each embedded controller has a flat byte-addressable I/O space, currently defined as 256 bytes. Features implemented in the embedded controller have an event “query” mechanism that allows feature hardware implemented by the embedded controller to gain the attention of an OS driver or ASL/AML code handler. The interface has been specified to work on the most popular embedded controllers on the market today, only requiring changes in the way the embedded controller is “wired” to the host interface.

Two interfaces are specified:

- A private interface, exclusively owned by the embedded controller driver.
- A shared interface, used by the embedded controller driver and some other driver.

This interface is separate from the traditional PC keyboard controller. Some OEMs might choose to implement the ACPI Embedded Controller Interface (ECI) within the same embedded controller as the keyboard controller function, but the ECI requires its own unique host resources (interrupt event and access registers).

This interface does support sharing the ECI with an inter-environment interface (such as SMI) and relies on the ACPI-defined “Global Lock” protocol. For information about the Global Lock interface, see section 5.2.9.1, “Global Lock.” Both the shared and private EC interfaces are described in the following sections.

The ECI has been designed such that a platform can use it in either the legacy or ACPI modes with minimal changes between the two operating environments. This is to encourage standardization for this interface to enable faster development of platforms as well as opening up features within these controllers to higher levels of software.

13.1 Embedded Controller Interface Description

Embedded controllers are the general class of microcontrollers used to support OEM-specific implementations. The ACPI specification supports embedded controllers in any platform design, as long as the microcontroller conforms to one of the models described in this section. The embedded controller is a unique feature in that it can perform complex low-level functions through a simple interface to the host microprocessor(s).

Although there is a large variety of microcontrollers in the market today, the most commonly used embedded controllers include a host interface that connects the embedded controller to the host data bus, allowing bi-directional communications. A bi-directional interrupt scheme reduces the host processor latency in communicating with the embedded controller.

Currently, the most common host interface architecture incorporated into microcontrollers is modeled after the standard IA-PC architecture keyboard controller. This keyboard controller is accessed at 0x60 and 0x64 in system I/O space. Port 0x60 is termed the data register, and allows bi-directional data transfers to and from the host and embedded controller. Port 0x64 is termed the command/status register; it returns port status information upon a read, and generates a command sequence to the embedded controller upon a write. This same class of controllers also includes a second decode range that shares the same properties as the keyboard interface by having a command/status register and a data register. The following diagram graphically depicts this interface.

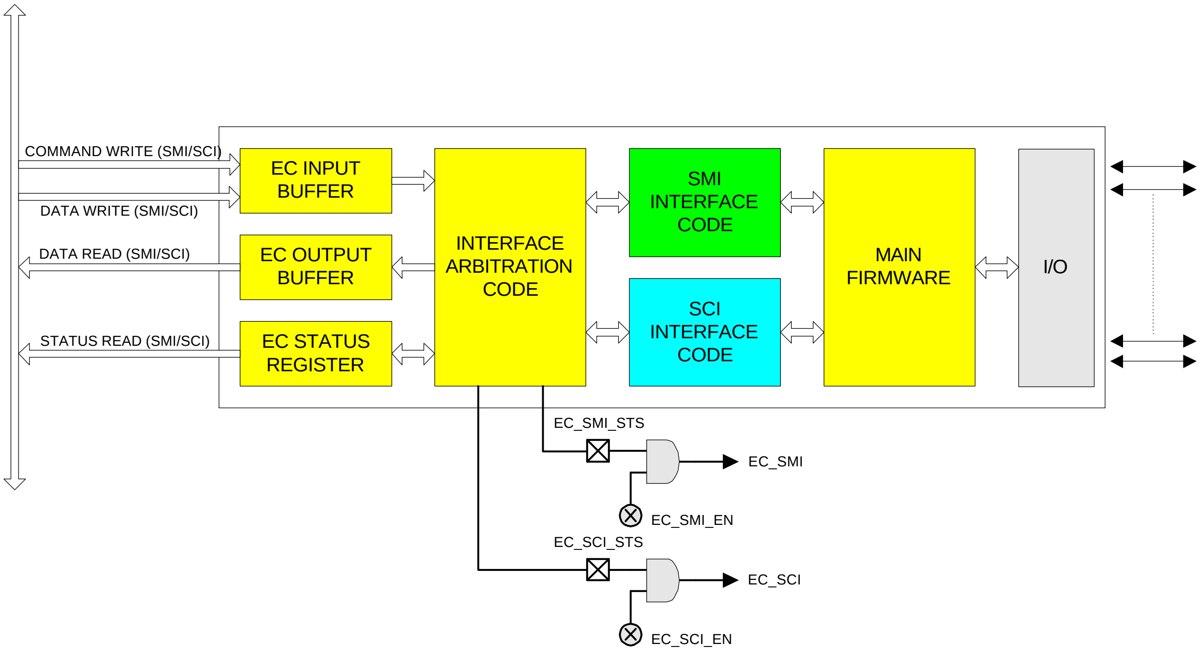

Figure 13-1 Shared Interface

The diagram above depicts the general register model supported by the ACPI Embedded Controller Interface.

The first method uses an embedded controller interface shared between OSPM and the system management code, which requires the Global Lock semaphore overhead to arbitrate ownership. The second method is a dedicated embedded controller decode range for sole use by OSPM driver. The following diagram illustrates the embedded controller architecture that includes a dedicated ACPI interface.

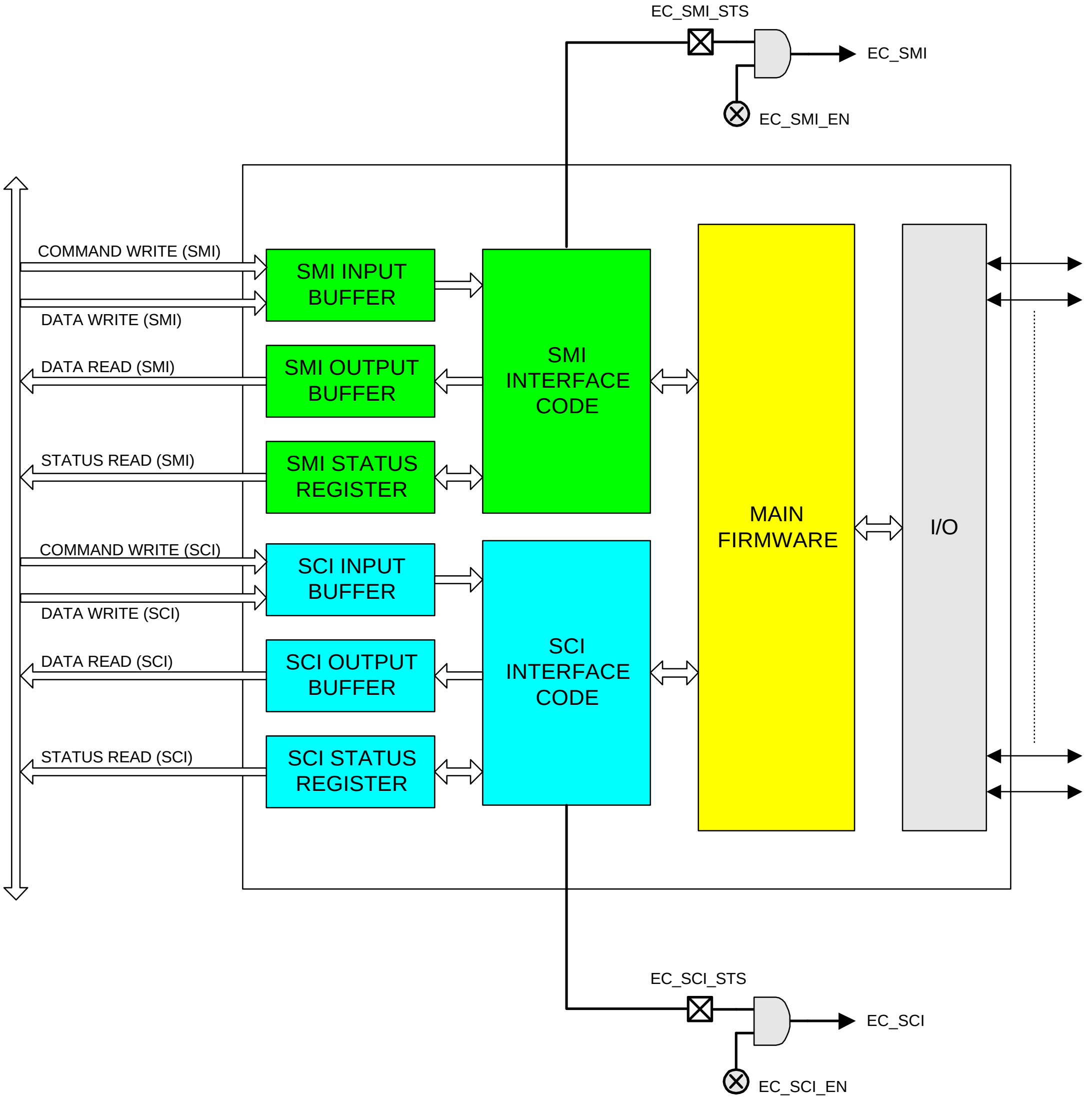

Figure 13-2 Private Interface

The private interface allows OSPM to communicate with the embedded controller without the additional software overhead associated with using the Global Lock. Several common system configurations can provide the additional embedded controller interfaces:

- Non-shared embedded controller. This will be the most common case where there is no need for the system management handler to communicate with the embedded controller when the system transitions to ACPI mode. OSPM processes all normal types of system management events, and the system management handler does not need to take any actions.
- Integrated keyboard controller and embedded controller. This provides three host interfaces as described earlier by including the standard keyboard controller in an existing component (chip set, I/O controller) and adding a discrete, standard embedded controller with two interfaces for system management activities.
- Standard keyboard controller and embedded controller.This provides three host interfaces by providing a keyboard controller as a distinct component, and two host interfaces are provided in the embedded controller for system management activities.
- **T**wo embedded controllers. This provides up to four host interfaces by using two embedded controllers; one controller for system management activities providing up to two host interfaces, and one controller for keyboard controller functions providing up to two host interfaces.
- Embedded controller and no keyboard controller. Future platforms might provide keyboard functionality through an entirely different mechanism, which would allow for two host interfaces in an embedded controller for system management activities.

To handle the general embedded controller interface (as opposed to a dedicated interface) model, a method is available to make the embedded controller a shareable resource between multiple tasks running under the operating system’s control and the system management interrupt handler. This method, as described in this section, requires several changes:

- Additional external hardware
- Embedded controller firmware changes
- System management interrupt handler firmware changes
- Operating software changes

Access to the shared embedded controller interface requires additional software to arbitrate between the operating system’s use of the interface and the system management handler’s use of the interface. This is done using the Global Lock as described in section 5.2.9.1, “Global Lock.”

This interface sharing protocol also requires embedded controller firmware changes, in order to ensure that collisions do not occur at the interface. A collision could occur if a byte is placed in the system output buffer and an interrupt is then generated. There is a small window of time when the incorrect recipient could receive the data. This problem is resolved by ensuring that the firmware in the embedded controller does not place any data in the output buffer until it is requested by OSPM or the system management handler.

More detailed algorithms and descriptions are provided in the following sections.

13.2 Embedded Controller Register Descriptions

The embedded controller contains three registers at two address locations: EC_SC and EC_DATA. The EC_SC, or Embedded Controller Status/Command register, acts as two registers: a status register for reads to this port and a command register for writes to this port. The EC_DATA (Embedded Controller Data register) acts as a port for transferring data between the host CPU and the embedded controller.

13.2.1 Embedded Controller Status, EC_SC (R)

This is a read-only register that indicates the current status of the embedded controller interface.

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
IGN	**SMI_EVT**	**SCI_EVT**	**BURST**	**CMD**	**IGN**	**IBF**	**OBF**

Where:

IGN:	Ignored
SMI_EVT:	1 – Indicates SMI event is pending (requesting SMI query). 0 – No SMI events are pending.
SCI_EVT:	1 – Indicates SCI event is pending (requesting SCI query). 0 – No SCI events are pending.
BURST:	1 – Controller is in burst mode for polled command processing. 0 – Controller is in normal mode for interrupt-driven command processing.
CMD:	1 – Byte in data register is a command byte (only used by controller). 0 – Byte in data register is a data byte (only used by controller).
IBF:	1 – Input buffer is full (data ready for embedded controller). 0 – Input buffer is empty.
OBF:	1 – Output buffer is full (data ready for host). 0 – Output buffer is empty.

The Output Buffer Full (OBF) flag is set when the embedded controller has written a byte of data into the command or data port but the host has not yet read it. After the host reads the status byte and sees the OBF flag set, the host reads the data port to get the byte of data that the embedded controller has written. After the host reads the data byte, the OBF flag is cleared automatically by hardware. This signals the embedded controller that the data has been read by the host and the embedded controller is free to write more data to the host.

The Input Buffer Full (IBF) flag is set when the host has written a byte of data to the command or data port, but the embedded controller has not yet read it. After the embedded controller reads the status byte and sees the IBF flag set, the embedded controller reads the data port to get the byte of data that the host has written. After the embedded controller reads the data byte, the IBF flag is automatically cleared by hardware. This is the signal to the host that the data has been read by the embedded controller and that the host is free to write more data to the embedded controller.

The SCI event (SCI_EVT) flag is set when the embedded controller has detected an internal event that requires the operating system’s attention. The embedded controller sets this bit in the status register, and generates an SCI to OSPM. OSPM needs this bit to differentiate command-complete SCIs from notification SCIs. OSPM uses the query command to request the cause of the SCI_EVT and take action. For more information, see section 13.3, “Embedded Controller Command Set.”)

The SMI event (SMI_EVT) flag is set when the embedded controller has detected an internal event that requires the system management interrupt handler’s attention. The embedded controller sets this bit in the status register before generating an SMI.

The Burst (BURST) flag indicates that the embedded controller has received the burst enable command from the host, has halted normal processing, and is waiting for a series of commands to be sent from the host. This allows OSPM or system management handler to quickly read and write several bytes of data at a time without the overhead of SCIs between the commands.

13.2.2 Embedded Controller Command, EC_SC (W)

This is a write-only register that allows commands to be issued to the embedded controller. Writes to this port are latched in the input data register and the input buffer full flag is set in the status register. Writes to this location also cause the command bit to be set in the status register. This allows the embedded controller to differentiate the start of a command sequence from a data byte write operation.

13.2.3 Embedded Controller Data, EC_DATA (R/W)

This is a read/write register that allows additional command bytes to be issued to the embedded controller, and allows OSPM to read data returned by the embedded controller. Writes to this port by the host are latched in the input data register, and the input buffer full flag is set in the status register. Reads from this register return data from the output data register and clear the output buffer full flag in the status register.

13.3 Embedded Controller Command Set

The embedded controller command set allows OSPM to communicate with the embedded controllers. ACPI defines the commands and their byte encodings for use with the embedded controller that are shown in the following table.

Table 13-1 Embedded Controller Commands

Embedded Controller Command	Command Byte Encoding
Read Embedded Controller (RD_EC)	0x80
Write Embedded Controller (WR_EC)	0x81
Burst Enable Embedded Controller (BE_EC)	0x82
Burst Disable Embedded Controller (BD_EC)	0x83
Query Embedded Controller (QR_EC)	0x84

13.3.1 Read Embedded Controller, RD_EC (0x80)

This command byte allows OSPM to read a byte in the address space of the embedded controller. This command byte is reserved for exclusive use by OSPM, and it indicates to the embedded controller to generate SCIs in response to related transactions (that is, IBF=0 or OBF=1 in the EC Status Register), rather than SMIs. This command consists of a command byte written to the Embedded Controller Command register (EC_SC), followed by an address byte written to the Embedded Controller Data register (EC_DATA). The embedded controller then returns the byte at the addressed location. The data is read at the data port after the OBF flag is set.

13.3.2 Write Embedded Controller, WR_EC (0x81)

This command byte allows OSPM to write a byte in the address space of the embedded controller. This command byte is reserved for exclusive use by OSPM, and it indicates to the embedded controller to generate SCIs in response to related transactions (that is, IBF=0 or OBF=1 in the EC Status Register), rather than SMIs. This command allows OSPM to write a byte in the address space of the embedded controller. It consists of a command byte written to the Embedded Controller Command register (EC_SC), followed by an address byte written to the Embedded Controller Data register (EC_DATA), followed by a data byte written to the Embedded Controller Data Register (EC_DATA); this is the data byte written at the addressed location.

13.3.3 Burst Enable Embedded Controller, BE_EC (0x82)

This command byte allows OSPM to request dedicated attention from the embedded controller and (except for critical events) prevents the embedded controller from doing tasks other than receiving command and data from the host processor (either the system management interrupt handler or OSPM). This command is an optimization that allows the host processor to issue several commands back to back, in order to reduce latency at the embedded controller interface. When the controller is in the burst mode, it should transition to the burst disable state if the host does not issue a command within the following guidelines:

- First Access – 400 microseconds
- Subsequent Accesses – 50 microseconds each
- Total Burst Time – 1 millisecond

In addition, the embedded controller can disengage the burst mode at any time to process a critical event. If the embedded controller disables burst mode for any reason other than the burst disable command, it should generate an SCI to OSPM to indicate the change.

While in burst mode, the embedded controller follows these guidelines for OSPM driver:

SCIs are generated as normal, including IBF=0 and OBF=1.

Accesses should be responded to within 50 microseconds.

Burst mode is entered in the following manner:

OSPM driver writes the Burst Enable Embedded Controller, BE_EC (0x82) command byte and then the Embedded Controller will prepare to enter the Burst mode. This includes processing any routine activities such that it should be able to remain dedicated to OSPM interface for ~ 1 microsecond.

The Embedded Controller sets the Burst bit of the Embedded Controller Status Register, puts the Burst Acknowledge byte (0x90) into the SCI output buffer, sets the OBF bit, and generates an SCI to signal OSPM that it is in Burst mode.

Burst mode is exited the following manner:

OSPM driver writes the Burst Disable Embedded Controller, BD_EC (0x83) command byte and then the Embedded Controller will exit Burst mode by clearing the Burst bit in the Embedded Controller Status register and generating an SCI signal (due to IBF=0).

The Embedded Controller clears the Burst bit of the Embedded Controller Status Register.

13.3.4 Burst Disable Embedded Controller, BD_EC (0x83)

This command byte releases the embedded controller from a previous burst enable command and allows it to resume normal processing. This command is sent by OSPM or system management interrupt handler after it has completed its entire queued command sequence to the embedded controller.

13.3.5 Query Embedded Controller, QR_EC (0x84)

OSPM driver sends this command when the SCI_EVT flag in the EC_SC register is set. When the embedded controller has detected a system event that must be communicated to OSPM, it first sets the SCI_EVT flag in the EC_SC register, generates an SCI, and then waits for OSPM to send the query (QR_EC) command. OSPM detects the embedded controller SCI, sees the SCI_EVT flag set, and sends the query command to the embedded controller. Upon receipt of the QR_EC command byte, the embedded controller places a notification byte with a value between 0-255, indicating the cause of the notification. The notification byte indicates which interrupt handler operation should be executed by OSPM to process the embedded controller SCI. The query value of zero is reserved for a spurious query result and indicates "no outstanding event."

13.4 SMBus Host Controller Notification Header (Optional), OS_SMB_EVT

This query command notification header is the special return code that indicates events with an SMBus controller implemented within an embedded controller. These events include:

- Command completion
- Command error
- Alarm reception

The actual notification value is declared in the EC-SMB-HC device object in the ACPI Namespace.

13.5 Embedded Controller Firmware

The embedded controller firmware must obey the following rules in order to be ACPI-compatible:

- **SMI Processing**. Although it is not explicitly stated in the command specification section, a shared embedded controller interface has a separate command set for communicating with each environment it plans to support. In other words, the embedded controller knows which environment is generating the command request, as well as which environment is to be notified upon event detection, and can then generate the correct interrupts and notification values. This implies that a system management handler uses commands that parallel the functionality of all the commands for ACPI including query, read, write, and any other implemented specific commands.
- **SCI/SMI Task Queuing**. If the system design is sharing the interface between both a system management interrupt handler and OSPM, the embedded controller should always be prepared to queue a notification if it receives a command. The embedded controller only sets the appropriate event flag in the status (EC_SC) register if the controller has detected an event that should be communicated to the OS or system management handler. The embedded controller must be able to field commands from either environment without loss of the notification event. At some later time, the OS or system management handler issues a query command to the embedded controller to request the cause of the notification event.
- **Notification Management**. The use of the embedded controller means using the query (QR_EC) command to notify OSPM of system events requiring action. If the embedded controller is shared with the operating system, the SMI handler uses the SMI_EVT flag and an SMI query command (not defined in this document) to receive the event notifications. The embedded controller doesn't place event notifications into the output buffer of a shared interface unless it receives a query command from OSPM or the system management interrupt handler.

13.6 Interrupt Model

The EC Interrupt Model uses pulsed interrupts to speed the clearing process. The Interrupt is firmware generated using an EC general-purpose output and has the waveform shown in Figure 13-3. The embedded controller SCI is always wired directly to a GPE input, and OSPM driver treats this as an edge event (the EC SCI GPE cannot be shared).

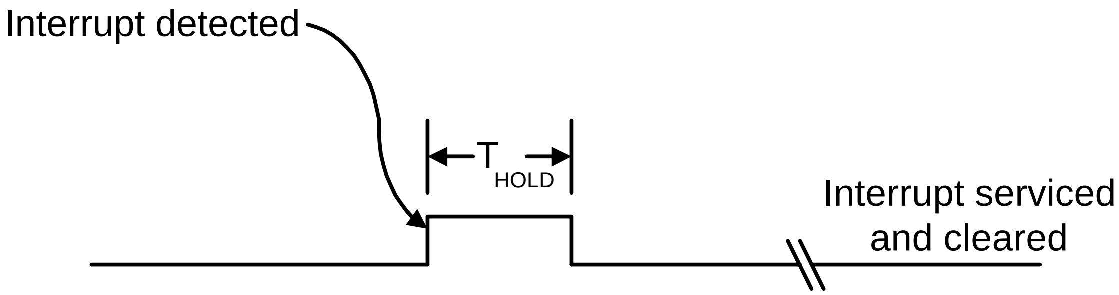

Figure 13-3 EC Interrupt Waveform

13.6.1 Event Interrupt Model

The embedded controller must generate SCIs for the events listed in the following table.

Table 13-2 Events for Which Embedded Controller Must Generate SCIs

Event	Description
IBF=0	Signals that the embedded controller has read the last command or data from the input buffer and the host is free to send more data.
OBF=1	Signals that the embedded controller has written a byte of data into the output buffer and the host is free to read the returned data.
SCI_EVT=1	Signals that the embedded controller has detected an event that requires OS attention. OSPM should issue a query (QR_EC) command to find the cause of the event.

13.6.2 Command Interrupt Model

The embedded controller must generate SCIs for commands as follows:

- **Read** Command (3 Bytes)

Byte #1	(Command byte Header)	Interrupt on IBF=0
Byte #2	(Address byte to read)	No Interrupt
Byte #3	(Data read to host)	Interrupt on OBF=1

- **Write** Command (3 Bytes)

Byte #1	(Command byte Header)	Interrupt on IBF=0
Byte #2	(Address byte to write)	Interrupt on IBF=0
Byte #3	(Data to read)	Interrupt on IBF=0

- **Query** Command (2 Bytes)

Byte #1	(Command byte Header)	No Interrupt
Byte #2	(Query value to host)	Interrupt on OBF=1

- **Burst Enable** Command (2 Bytes)

Byte #1	(Command byte Header)	No Interrupt
Byte #2	(Burst acknowledge byte)	Interrupt on OBF=1

- **Burst Disable** Command (1 Byte)

Byte #1	(Command byte Header)	Interrupt on IBF=0

13.7 Embedded Controller Interfacing Algorithms

To initiate communications with the embedded controller, OSPM or system management handler acquires ownership of the interface. This ownership is acquired through the use of the Global Lock (described in section 5.2.9.1, “Global Lock”), or is owned by default by OSPM as a non-shared resource (and the Global Lock is not required for accessibility).

After ownership is acquired, the protocol always consists of the passing of a command byte. The command byte will indicate the type of action to be taken. Following the command byte, zero or more data bytes can be exchanged in either direction. The data bytes are defined according to the command byte that is transferred.

The embedded controller also has two status bits that indicate whether the registers have been read. This is used to ensure that the host or embedded controller has received data from the embedded controller or host. When the host writes data to the command or data register of the embedded controller, the input buffer flag (IBF) in the status register is set within 1 microsecond. When the embedded controller reads this data from the input buffer, the input buffer flag is reset. When the embedded controller writes data into the output buffer, the output buffer flag (OBF) in the status register is set. When the host processor reads this data from the output buffer, the output buffer flag is reset.

13.8 Embedded Controller Description Information

Certain aspects of the embedded controller’s operation have OEM-definable values associated with them. The following is a list of values that are defined in the software layers of the ACPI specification:

- Status flag indicating whether the interface requires the use of the Global Lock.
- Bit position of embedded controller interrupt in general-purpose status register.
- Decode address for command/status register.
- Decode address for data register.
- Base address and query value of any EC-SMBus controller.

For implementation details of the above listed information, see sections 13.11, “Defining an Embedded Controller Device in ACPI Namespace,” and 13.12, “Defining an Embedded Controller SMBus Host Controller in ACPI Namespace.”

An embedded controller will require the inclusion of the GLK method in its ACPI namespace if potentially contentious accesses to device resources are performed by non-OS code. See section 6.5.7, “_GLK (Global Lock)” for details about the _GLK method.

13.9 SMBus Host Controller Interface via Embedded Controller

This section specifies a standard interface that an ACPI-compatible OS can use to communicate with embedded controller-based SMBus host controllers (EC-SMB-HC). This interface allows the host processor (under control of OSPM) to manage devices on the SMBus. Typical devices residing on the SMBus include Smart Batteries, Smart Battery Chargers, contrast/backlight control, and temperature sensors.

The EC-SMB-HC interface consists of a block of registers that reside in embedded controller space. These registers are used by software to initiate SMBus transactions and receive SMBus notifications. By using a well-defined register set, OS software can be written to operate with any vendor’s embedded controller hardware.

Certain SMBus segments have special requirements that the host controller filters certain SMBus commands (for example, to prevent an errant application or virus from potentially damaging the battery subsystem). This is most easily accomplished by implementing the host interface controller through an embedded controller—as embedded controller can easily filter out potentially problematic commands.

Notice that an EC-SMB-HC interface will require the inclusion of the GLK method in its ACPI namespace if potentially contentious accesses to device resources are performed by non-OS code. See section6.5.7, “_GLK (Global Lock” for details on using the _GLK method.

13.9.1 Register Description

The EC-SMBus host interface is a flat array of registers that are arranged sequentially in the embedded controller address space.

13.9.1.1 Status Register, SMB_STS

This register indicates general status on the SMBus. This includes SMB-HC command completion status, alarm received status, and error detection status (the error codes are defined later in this section). This register is cleared to zeroes (except for the ALRM bit) whenever a new command is issued using a write to the protocol (SMB_PRTCL) register. This register is always written with the error code before clearing the protocol register. The SMB-HCquery event (that is, an SMB-HCinterrupt) is raised after the clearing of the protocol register.

Note: OSPM must ensure the ALRM bit is cleared after it has been serviced by writing ‘00’ to the SMB_STS register.

Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
DONE	**ALRM**	**RES**	**STATUS**				

Where:

DONE:	Indicates the last command has completed and no error.
ALRM:	Indicates an SMBus alarm message has been received.
RES:	Reserved
STATUS:	Indicates SMBus communication status for one of the reasons listed in the following table.

Table 13-3 SMBus Status Codes

Status Code	**Name**	**Description**
00h	SMBus OK	Indicates the transaction has been successfully completed.
07h	SMBus Unknown Failure	Indicates failure because of an unknown SMBus error.
10h	SMBus Device Address Not Acknowledged	Indicates the transaction failed because the slave device address was not acknowledged.
11h	SMBus Device Error Detected	Indicates the transaction failed because the slave device signaled an error condition.
12h	SMBus Device Command Access Denied	Indicates the transaction failed because the SMBus host does not allow the specific command for the device being addressed. For example, the SMBus host might not allow a caller to adjust the Smart Battery Charger's output.
13h	SMBus Unknown Error	Indicates the transaction failed because the SMBus host encountered an unknown error.
17h	SMBus Device Access Denied	Indicates the transaction failed because the SMBus host does not allow access to the device addressed. For example, the SMBus host might not allow a caller to directly communicate with an SMBus device that controls the system's power planes.
18h	SMBus Timeout	Indicates the transaction failed because the SMBus host detected a timeout on the bus.
19h	SMBus Host Unsupported Protocol	Indicates the transaction failed because the SMBus host does not support the requested protocol.
1Ah	SMBus Busy	Indicates that the transaction failed because the SMBus host reports that the SMBus is presently busy with some other transaction. For example, the Smart Battery might be sending charging information to the Smart Battery Charger.
1Fh	SMBus PEC (CRC-8) Error	Indicates that a Packet Error Checking (PEC) error occurred during the last transaction.

All other error codes are reserved.

13.9.1.2 Protocol Register, SMB_PRTCL

This register determines the type of SMBus transaction generated on the SMBus. In addition to indicating the protocol type to the SMB-HC, a write to this register initiates the transaction on the SMBus. Notice that bit 7 of the protocol value is used to indicate whether packet error checking should be employed. A value of 1 (one) in this bit indicates that PEC format should be used for the specified protocol, and a value of 0 (zero) indicates the standard (non-PEC) format should be used.

Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
PEC	**PROTOCOL**						

Where:

PROTOCOL:	0x00 – Controller Not In Use
	0x01 – Reserved
	0x02 – Write Quick Command
	0x03 – Read Quick Command
	0x04 – Send Byte
	0x05 – Receive Byte
	0x06 – Write Byte
	0x07 – Read Byte
	0x08 – Write Word
	0x09 – Read Word
	0x0A – Write Block
	0x0B – Read Block
	0x0C – Process Call
	0x0D – Block Write-Block Read Process Call

For example, the protocol value of 0x09 would be used to communicate to a device that supported the standard *read word* protocol. If this device also supported packet error checking for this protocol, a value of 0x89 (*read word with PEC*) could optionally be used. See the SMBus specification for more information on packet error checking.

When OSPM initiates a new command such as write to the SMB_PRTCL register, the SMBus controller first updates the SMB_STS register and then clears the SMB_PRTCL register. After the SMB_PRTCL register is cleared, the host controller query value is raised.

All other protocol values are reserved.

13.9.1.3 Address Register, SMB_ADDR

This register contains the 7-bit address to be generated on the SMBus. This is the first byte to be sent on the SMBus for all of the different protocols.

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
ADDRESS (A6:A0)							**RES**

Where:

RES: Reserved

ADDRESS: 7-bit SMBus address. This address is not zero aligned (in other words, it is only a 7-bit address (A6:A0) that is aligned from bit 1-7).

13.9.1.4 Command Register, SMB_CMD

This register contains the command byte that will be sent to the target device on the SMBus and is used for the following protocols: send byte, write byte, write word, read byte, read word, process call, block read and block write. It is not used for the quick commands or the receive byte protocol, and as such, its value is a "don't care" for those commands.

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
COMMAND							

Where:

COMMAND:	Command byte to be sent to SMBus device.

13.9.1.5 Data Register Array, SMB_DATA[i], i=0-31

This bank of registers contains the remaining bytes to be sent or received in any of the different protocols that can be run on the SMBus. The SMB_DATA[i] registers are defined on a per-protocol basis and, as such, provide efficient use of register space.

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
DATA							

Where:

DATA:	One byte of data to be sent or received (depending upon protocol).

13.9.1.6 Block Count Register, SMB_BCNT

This register contains the number of bytes of data present in the SMB_DATA[i] registers preceding any write block and following any read block transaction. The data size is defined on a per protocol basis.

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
RES			**BCNT**				

13.9.1.7 Alarm Address Register, SMB_ALRM_ADDR

This register contains the address of an alarm message received by the host controller, at slave address 0x8, from the SMBus master that initiated the alarm. The address indicates the slave address of the device on the SMBus that initiated the alarm message. The status of the alarm message is contained in the SMB_ALRM_DATAx registers. Once an alarm message has been received, the SMB-HC will not receive additional alarm messages until the ALRM status bit is cleared.

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
ADDRESS (A6:A0)							**RES**

Where:

RES:	Reserved
ADDRESS:	Slave address (A6:A0) of the SMBus device that initiated the SMBus alarm message.

13.9.1.8 Alarm Data Registers, SMB_ALRM_DATA[0], SMB_ALRM_DATA[1]

These registers contain the two data bytes of an alarm message received by the host controller, at slave address 0x8, from the SMBus master that initiated the alarm. These data bytes indicate the specific reason for the alarm message, such that OSPM can take actions. Once an alarm message has been received, the SMB-HCwill not receive additional alarm messages until the ALRM status bit is cleared.

Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
DATA (D7:D0)							

Where:

DATA: Data byte received in alarm message.

The alarm address and alarm data registers are not read by OSPM until the alarm status bit is set. OSPM driver then reads the 3 bytes, and clears the alarm status bit to indicate that the alarm registers are now available for the next event.

13.9.2 Protocol Description

This section describes how to initiate the different protocols on the SMBus through the interface described in section 13.9.1, "Register Descriptions." The registers should all be written with the appropriate values before writing the protocol value that starts the SMBus transaction. All transactions can be completed in one pass.

13.9.2.1 Write Quick

Data Sent:

SMB_ADDR: Address of SMBus device.

SMB_PRTCL: Write 0x02 to initiate the write quick protocol.

Data Returned:

SMB_STS: Status code for transaction.

SMB_PRTCL: 0x00 to indicate command completion.

13.9.2.2 Read Quick

Data Sent:

SMB_ADDR: Address of SMBus device.

SMB_PRTCL: Write 0x03 to initiate the read quick protocol.

Data Returned:

SMB_STS: Status code for transaction.

SMB_PRTCL: 0x00 to indicate command completion.

13.9.2.3 Send Byte

Data Sent:

SMB_ADDR:	Address of SMBus device.
SMB_CMD:	Command byte to be sent.
SMB_PRTCL:	Write 0x04 to initiate the send byte protocol, or 0x84 to initiate the send byte protocol with PEC.

Data Returned:

SMB_STS:	Status code for transaction.
SMB_PRTCL:	0x00 to indicate command completion.

13.9.2.4 Receive Byte

Data Sent:

SMB_ADDR:	Address of SMBus device.
SMB_PRTCL:	Write 0x05 to initiate the receive byte protocol, or 0x85 to initiate the receive byte protocol with PEC.

Data Returned:

SMB_DATA[0]:	Data byte received.
SMB_STS:	Status code for transaction.
SMB_PRTCL:	0x00 to indicate command completion.

13.9.2.5 Write Byte

Data Sent:

SMB_ADDR:	Address of SMBus device.
SMB_CMD:	Command byte to be sent.
SMB_DATA[0]:	Data byte to be sent.
SMB_PRTCL:	Write 0x06 to initiate the write byte protocol, or 0x86 to initiate the write byte protocol with PEC.

Data Returned:

SMB_STS:	Status code for transaction.
SMB_PRTCL:	0x00 to indicate command completion.

13.9.2.6 Read Byte

Data Sent:

SMB_ADDR:	Address of SMBus device.
SMB_CMD:	Command byte to be sent.
SMB_PRTCL:	Write 0x07 to initiate the read byte protocol, or 0x87 to initiate the read byte protocol with PEC.

Data Returned:

SMB_DATA[0]:	Data byte received.
SMB_STS:	Status code for transaction.
SMB_PRTCL:	0x00 to indicate command completion.

13.9.2.7 Write Word

Data Sent:

SMB_ADDR:	Address of SMBus device.
SMB_CMD:	Command byte to be sent.
SMB_DATA[0]:	Low data byte to be sent.
SMB_DATA[1]:	High data byte to be sent.
SMB_PRTCL:	Write 0x08 to initiate the write word protocol, or 0x88 to initiate the write word protocol with PEC.

Data Returned:

SMB_STS:	Status code for transaction.
SMB_PRTCL:	0x00 to indicate command completion.

13.9.2.8 Read Word

Data Sent:

SMB_ADDR:	Address of SMBus device.
SMB_CMD:	Command byte to be sent.
SMB_PRTCL:	Write 0x09 to initiate the read word protocol, or 0x89 to initiate the read word protocol with PEC.

Data Returned:

SMB_DATA[0]:	Low data byte received.
SMB_DATA[1]:	High data byte received.
SMB_STS:	Status code for transaction.
SMB_PRTCL:	0x00 to indicate command completion.

13.9.2.9 Write Block

Data Sent:

SMB_ADDR: Address of SMBus device.

SMB_CMD: Command byte to be sent.

SMB_DATA[0-31]: Data bytes to write (1-32).

SMB_BCNT: Number of data bytes (1-32) to be sent.

SMB_PRTCL: Write 0x0A to initiate the write block protocol, or 0x8A to initiate the write block protocol with PEC.

Data Returned:

SMB_PRTCL: 0x00 to indicate command completion.

SMB_STS: Status code for transaction.

13.9.2.10 Read Block

Data Sent:

SMB_ADDR: Address of SMBus device.

SMB_CMD: Command byte to be sent.

SMB_PRTCL: Write 0x0B to initiate the read block protocol, or 0x8B to initiate the read block protocol with PEC.

Data Returned:

SMB_BCNT: Number of data bytes (1-32) received.

SMB_DATA[0-31]: Data bytes received (1-32).

SMB_STS: Status code for transaction.

SMB_PRTCL: 0x00 to indicate command completion.

13.9.2.11 Process Call

Data Sent:

SMB_ADDR: Address of SMBus device.

SMB_CMD: Command byte to be sent.

SMB_DATA[0]: Low data byte to be sent.

SMB_DATA[1]: High data byte to be sent.

SMB_PRTCL: Write 0x0C to initiate the process call protocol, or 0x8C to initiate the process call protocol with PEC.

Data Returned:

SMB_DATA[0]: Low data byte received.

SMB_DATA[1]: High data byte received.

SMB_STS: Status code for transaction.

SMB_PRTCL: 0x00 to indicate command completion.

13.9.2.12 Block Write-Block Read Process Call

Data Sent:

SMB_ADDR:	Address of SMBus device.
SMB_CMD:	Command byte to be sent.
SMB_DATA[0-31]:	Data bytes to write (1-31).
SMB_BCNT:	Number of data bytes (1-31) to be sent.
SMB_PRTCL:	Write 0x0D to initiate the write block-read block process call protocol, or 0x8D to initiate the write block-read block process call protocol with PEC.

Data Returned:

SMB_BCNT:	Number of data bytes (1-31) received.
SMB_DATA[0-31]:	Data bytes received (1-31).
SMB_STS:	Status code for transaction.
SMB_PRTCL:	0x00 to indicate command completion.

Note: The following restrictions apply: The aggregate data length of the write and read blocks must not exceed 32 bytes and each block (write and read) must contain at least 1 byte of data.

13.9.3 SMBus Register Set

The register set for the SMB-HChas the following format. All registers are 8 bit.

Table 13-4 SMB EC Interface

LOCATION	REGISTER NAME	DESCRIPTION
BASE+0	SMB_PRTCL	Protocol register
BASE+1	SMB_STS	Status register
BASE+2	SMB_ADDR	Address register
BASE+3	SMB_CMD	Command register
BASE+4	SMB_DATA[0]	Data register zero
BASE+5	SMB_DATA[1]	Data register one
BASE+6	SMB_DATA[2]	Data register two
BASE+7	SMB_DATA[3]	Data register three
BASE+8	SMB_DATA[4]	Data register four
BASE+9	SMB_DATA[5]	Data register five
BASE+10	SMB_DATA[6]	Data register six
BASE+11	SMB_DATA[7]	Data register seven
BASE+12	SMB_DATA[8]	Data register eight
BASE+13	SMB_DATA[9]	Data register nine
BASE+14	SMB_DATA[10]	Data register ten
BASE+15	SMB_DATA[11]	Data register eleven

Table 13-4 SMB EC Interface *(continued)*

BASE+16	SMB_DATA[12]	Data register twelve
BASE+17	SMB_DATA[13]	Data register thirteen
BASE+18	SMB_DATA[14]	Data register fourteen
BASE+19	SMB_DATA[15]	Data register fifteen
BASE+20	SMB_DATA[16]	Data register sixteen
BASE+21	SMB_DATA[17]	Data register seventeen
BASE+22	SMB_DATA[18]	Data register eighteen
BASE+23	SMB_DATA[19]	Data register nineteen
BASE+24	SMB_DATA[20]	Data register twenty
BASE+25	SMB_DATA[21]	Data register twenty-one
BASE+26	SMB_DATA[22]	Data register twenty-two
BASE+27	SMB_DATA[23]	Data register twenty-three
BASE+28	SMB_DATA[24]	Data register twenty-four
BASE+29	SMB_DATA[25]	Data register twenty-five
BASE+30	SMB_DATA[26]	Data register twenty-six
BASE+31	SMB_DATA[27]	Data register twenty-seven
BASE+32	SMB_DATA[28]	Data register twenty-eight
BASE+33	SMB_DATA[29]	Data register twenty-nine
BASE+34	SMB_DATA[30]	Data register thirty
BASE+35	SMB_DATA[31]	Data register thirty-one
BASE+36	SMB_BCNT	Block Count Register
BASE+37	SMB_ALRM_ADDR	Alarm address
BASE+38	SMB_ALRM_DATA[0]	Alarm data register zero
BASE+39	SMB_ALRM_DATA[1]	Alarm data register one

13.10 SMBus Devices

The embedded controller interface provides the system with a standard method to access devices on the SMBus. It does not define the data and/or access protocol(s) used by any particular SMBus device. Further, the embedded controller can (and probably will) serve as a gatekeeper to prevent accidental or malicious access to devices on the SMBus.

Some SMBus devices are defined by their address and a specification that describes the data and the protocol used to access that data. For example, the Smart Battery System devices are defined by a series of specifications including:

- Smart Battery Data specification
- Smart Battery Charger specification
- Smart Battery Selector specification
- Smart Battery System Manager specification

The embedded controller can also be used to emulate (in part or totally) any SMBus device.

13.10.1 SMBus Device Access Restrictions

In some cases, the embedded controller interface will not allow access to a particular SMBus device. Some SMBus devices can and do communicate directly between themselves. Unexpected accesses can interfere with their normal operation and cause unpredictable results.

13.10.2 SMBus Device Command Access Restriction

There are cases where part of an SMBus device's commands are public while others are private. Extraneous attempts to access these commands might cause interference with the SMBus device's normal operation.

The Smart Battery and the Smart Battery Charger are good examples of devices that should not have their entire command set exposed. The Smart Battery commands the Smart Battery Charger to supply a specific charging voltage and charging current. Attempts by anyone to alter these values can cause damage to the battery or the mobile system. To protect the system's integrity, the embedded controller interface can restrict access to these commands by returning one of the following error codes: Device Command Access Denied (0x12) or Device Access Denied (0x17).

13.11 Defining an Embedded Controller Device in ACPI Namespace

An embedded controller device is created using the named device object. The embedded controller's device object requires the following elements:

Table 13-5 Embedded Controller Device Object Control Methods

Object	Description
_CRS	Named object that returns the Embedded Controller's current resource settings. Embedded Controllers are considered static resources; hence only return their defined resources. The embedded controller resides only in system I/O or memory space. The first address region returned is the data port, and the second address region returned is the status/command port for the embedded controller. CRS is a standard device configuration control method defined in section 6.2.1, "_CRS (Current Resource Settings."
_HID	Named object that provides the Embedded Controller's Plug and Play identifier. This value is set to PNP0C09. _HID is a standard device configuration control method defined in section 6.1.4, "_HID (Hardware ID)."
_GPE	Named Object that evaluates to either an integer or a package. If _GPE evaluates to an integer, the value is the bit assignment of the SCI interrupt within the GPEx_STS register of a GPE block described in the FADT that the embedded controller will trigger. If _GPE evaluates to a package, then that package contains two elements. The first is an object reference to the GPE Block device that contains the GPE register that will be triggered by the embedded controller. The second element is numeric (integer) that specifies the bit assignment of the SCI interrupt within the GPEx_STS register of the GPE Block device referenced by the first element in the package. This control method is specific to the embedded controller.

13.11.1 Example: EC Definition ASL Code

Example ASL code that defines an embedded controller device is shown below:

```
Device(EC0) {
    // PnP ID
    Name(_HID, EISAID(“PNP0C09”))
    // Returns the “Current Resources” of EC
    Name(_CRS,
        ResourceTemplate(){                   // port 0x62 and 0x66
            IO(Decode16, 0x62, 0x62, 0, 1),
            IO(Decode16, 0x66, 0x66, 0, 1)
            }
        )

// Define that the EC SCI is bit 0 of the GP_STS register
    Name(_GPE, 0)

    OperationRegion(ECOR, EmbeddedControl, 0, 0xFF)
    Field(ECOR, ByteAcc, Lock, Preserve) {
        // Field definitions go here
        }
    }
```

13.12 Defining an EC SMBus Host Controller in ACPI Namespace

An EC-SMB-HC device is defined using the named device object. The EC-SMB- HC’s device object requires the following elements:

Table 13-6 EC SMBus HC Device Objects

Object	Description
_HID	Named object that provides the EC-SMB- HC’s Plug and Play identifier. This value is be set to ACPI0001. _HID is a standard device configuration control method defined in section 6.1.4, “_HID (Hardware ID).”
_EC	Named object that evaluates to a WORD that defines the SMBus attributes needed by the SMBus driver. _EC is the Embedded Controller Offset Query Control Method. The most significant byte is the address offset in embedded controller space of the SMBus controller; the least significant byte is the query value for all SMBus events.

13.12.1Example: EC SMBus Host Controller ASL-Code

Example ASL code that defines an SMB-HC from within an embedded controller device is shown below:

```
Device(EC0)
{
    Name(_HID, EISAID("PNP0C09"))
    Name(_CRS, ResourceTemplate()
        {
            IO(Decode16, 0x62, 0x62, 0, 1),  // Status port
            IO(Decode16, 0x66, 0x66, 0, 1)   // command port
        })
    Name(_GPE, 0)

    Device (SMB0)
    {
        Name(_HID, "ACPI0001")                // EC-SMB-HC
        Name(_UID, 0)                         // Unique device identifier
        Name(_EC, 0x2030)                     // EC offset 0x20, query bit 0x30
            :
    }

    Device (SMB1)
    {
        Name(_HID, "ACPI0001")                // EC-SMB-HC
        Name(_UID, 1)                         // Unique device identifier
        Name(_EC, 0x8031)                     // EC offset 0x80, query bit 0x31
            :
    }
}               // end of EC0
```

14 ACPI System Management Bus Interface Specification

This section describes the System Management Bus (SMBus) generic address space and the use of this address space to access SMBus devices from AML.

Unlike other address spaces, SMBus operation regions are inherently *non-linear*, where each offset within an SMBus address space represents a variable-sized (from 0 to 32 bytes) field. Given this uniqueness, SMBus operation regions include restrictions on their field definitions and require the use of an SMBus-specific data buffer for all transactions.

The SMBus interface presented in this section is intended for use with any hardware implementation compatible with the SMBus specification. SMBus hardware is broadly classified as either non-EC–based or EC-based. EC-based SMBus implementations comply with the standard register set defined in section 13, ACPI Embedded Controller Interface Specification."

Non-EC SMBus implementations can employ any hardware interface and are typically used for their cost savings when SMBus security is not required. Non–EC-based SMBus implementations require the development of hardware specific drivers for each OS implementation. See section 14.2, "Declaring SMBus Host Controller Objects," for more information.

Support of the SMBus generic address space by ACPI-compatible operating systems is optional. As such, the Smart Battery System Implementer's Forum (SBS-IF) has defined an SMBus interface based on a standard set of control methods. This interface is documented in the *SMBus Control Method Interface Specification*, available from the SBS-IF Web site at: http://www.sbs-forum.org/.

14.1 SMBus Overview

SMBus is a two-wire interface based upon the I²C protocol. The SMBus is a low-speed bus that provides positive addressing for devices, as well as bus arbitration. For more information, refer to the complete set of SMBus specifications published by the SBS-IF.

14.1.1 SMBus Slave Addresses

Slave addresses are specified using a 7-bit non-shifted notation. For example, the slave address of the Smart Battery Selector device would be specified as 0x0A (1010b), not 0x14 (10100b) as might be found in other documents. These two different forms of addresses result from the format in which addresses are transmitted on the SMBus.

During transmission over the physical SMBus, the slave address is formatted in an 8-bit block with bits 7-1 containing the address and bit 0 containing the read/write bit. ASL code, on the other hand, presents the slave address simply as a 7-bit value making it the responsibility of the OS (driver) to shift the value if needed. For example, the ASL value would have to be shifted left 1 bit before being written to the SMB_ADDR register in the EC based SMBus as described in section 13.9.1.3, "Address Register, SMB_ADDR."

14.1.2 SMBus Protocols

There are six possible *command protocols* for any given SMBus slave device, and a device may use any or all of the protocols to communicate. The protocols and associated access type indicators are listed below. Notice that the protocols values are similar to those defined for the EC-based SMBus in section 13.9.1.2, "Protocol Register, SMB_PRTCL," except that protocol pairs (for example, Read Byte, Write Byte) have been joined.

Table 14-1 SMBus Protocol Types

Value	Type	Description
0x02	SMBQuick	SMBus Read/Write Quick Protocol
0x04	SMBSendReceive	SMBus Send/Receive Byte Protocol
0x06	SMBByte	SMBus Read/Write Byte Protocol
0x08	SMBWord	SMBus Read/Write Word Protocol
0x0A	SMBBlock	SMBus Read/Write Block Protocol
0x0C	SMBProcessCall	SMBus Process Call Protocol
0x0D	SMBBlockProcessCall	SMBus Write Block-Read Block Process Call Protocol

All other protocol values are reserved.

Notice that bit 7 of the protocol value is used by this interface to indicate to the SMB-HC whether or not packet error checking (PEC) should be employed for a transaction. Packet error checking is described in section 7.4 of the *System Management Bus Specification, Version 1.1.* This highly desirable capability improves the reliability and robustness of SMBus communications.

The bit encoding of the protocol value is shown below. For example, the value 0x86 would be used to specify the PEC version of the SMBus Read/Write Byte protocol.

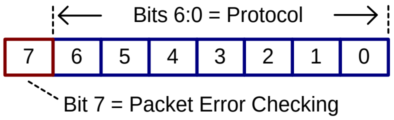

Figure 14-1 Bit Encoding Example

Notice that bit 0 of the protocol value is always zero (even number hexadecimal values). In a manner similar to the slave address, software that implements the SMBus interface is responsible for setting this bit to indicate whether the transaction is a read (for example, Read Byte) or write (for example, Write Byte) operation.

For example, software implanting this interface for EC-SMBus segments would set bit 0 for read transactions. For the SMBByte protocol (0x06), this would result in the value 0x07 being placed into the SMB_PRTCL register (or 0x87 if PEC is requested).

14.1.3 SMBus Status Codes

The use of status codes helps AML determine whether an SMBus transaction was successful. In general, a status code of zero indicates success, while a non-zero value indicates failure. The SMBus interface uses the same status codes defined for the EC-SMBus (see section 13.9.1.1, "Status Register, SMB_STS").

14.1.4 SMBus Command Values

SMBus devices may optionally support up to 256 device-specific commands. For these devices, each *command value* supported by the device is modeled by this interface as a separate *virtual register.* Protocols that do not transmit a command value (for example, Read/Write Quick and Send/Receive Byte) are modeled using a single virtual register (with a command value = 0x00).

14.2 Declaring SMBus Host Controller Objects

EC-based SMBus 1.0-compatible HCs should be modeled in the ACPI namespace as described in section 13.12, “Defining an Embedded Controller SMBus Host Controller in ACPI Namespace.” An example definition is given below. Using the HID value “ACPI0001” identifies that this SMB-HC is implemented on an embedded controller using the standard SMBus register set defined in section 13.9, SMBus Host Controller Interface via Embedded Controller.”

```
Device (SMB0)
{
    Name(_HID, "ACPI0001")        // EC-based SMBus 1.0 compatible Host Controller
    Name(_EC, 0x2030)             // EC offset 0x20, query bit 0x30
      :
}
```

EC-based SMBus 2.0-compatible host controllers should be defined similarly in the name space as follows:

```
Device (SMB0)
{
    Name(_HID, "ACPI0005")        // EC-based SMBus 2.0 compatible Host Controller
    Name(_EC, 0x2030)             // EC offset 0x20, query bit 0x30
      :
}
```

Non–EC-based SMB-HCs should be modeled in a manner similar to the EC-based SMBus HC. An example definition is given below. These devices use a vendor-specific hardware identifier (HID) to specify the type of SMB-HC (do not use “ACPI0001” or “ACPI0005”). Using a vendor-specific HID allows the correct software to be loaded to service this segment’s SMBus address space.

```
Device(SMB0)
{
    Name(_HID, "<Vendor-Specific HID>")  // Vendor-Specific HID
      :
}
```

Regardless of the type of hardware, some OS software element (for example, the SMBus HC driver) must register with OSPM to support all SMBus operation regions defined for the segment. This software allows the generic SMBus interface defined in this section to be used on a specific hardware implementation by translating between the conceptual (for example, SMBus address space) and physical (for example, process of writing/reading registers) models. Because of this linkage, SMBus operation regions must be defined immediately within the scope of the corresponding SMBus device.

14.3 Declaring SMBus Devices

The SMBus, as defined by the SMBus 1.0 Specification, is not an enumerable bus. As a result, an SMBus 1.0-compatible SMB-HCdriver cannot discover child devices on the SMBus and load the appropriate corresponding device drivers. As such, SMBus 1.0-compatible devices are declared in the ACPI namespace, in like manner to other motherboard devices, and enumerated by OSPM.

The SMBus 2.0 specification adds mechanisms enabling device enumeration on the bus while providing compatibility with existing devices. ACPI 2.0 defines and associates the “ACPI0005” HID value with an EC-based SMBus 2.0-compatible host controller. OSPM will enumerate SMBus 1.0-compatible devices when declared in the namespace under an SMBus 2.0-compatible host controller.

The responsibility for the definition of ACPI namespace objects, required by an SMBus 2.0-compatible host controller driver to enumerate non–bus-enumerable devices, is relegated to the SBS-IF in ACPI 2.0. The definition of these objects is documented in the *SMBus ACPI Namespace Device Definition Specification*, available from the SBS-IF Web site at: http://www.sbs-forum.org/.

ACPI 2.0 uses _ADR to associate SMBus devices with their lowest SMBus slave address.

14.4 Declaring SMBus Operation Regions

Each SMBus operation region definition identifies a single SMBus slave address. Operation regions are defined only for those SMBus devices that need to be accessed from AML. As with other regions, SMBus operation regions are only accessible via the **Field** term (see section 14.5, “Declaring SMBus Fields”).

This interface models each SMBus device as having a 256-byte linear address range. Each byte offset within this range corresponds to a single command value (for example, byte offset 0x12 equates to command value 0x12), with a maximum of 256 command values. By doing this, SMBus address spaces appear linear and can be processed in a manner similar to the other address space types.

The syntax for the **OperationRegion** term (from section 16.2.3.3.1.14, “OperationRegion [Declare Operation Region]”) is described below.

```
OperationRegion(
    RegionName, //NameString
    RegionSpace,    //RegionSpaceKeyword
    Offset,     //TermArg=>Integer
    Length      //TermArg=>Integer
)
```

Where:

- *RegionName* specifies a name for this slave device (for example, “SBD0”).
- *RegionSpace* must be set to **SMBus** (operation region type value 0x04).
- *Offset* is a word-sized value specifying the slave address and initial command value offset for the target device. The slave address is stored in the high byte and the command value offset is stored in the low byte. For example, the value 0x4200 would be used for an SMBus device residing at slave address 0x42 with an initial command value offset of zero (0).
- *Length* is set to the 0x100 (256), representing the maximum number of possible command values, for regions with an initial command value offset of zero (0). The difference of these two values is used for regions with non-zero offsets. For example, a region with an *Offset* value of 0x4210 would have a corresponding *Length* of 0xF0 (0x100 minus 0x10).

For example, the Smart Battery Subsystem (illustrated below) consists of the Smart Battery Charger at slave address 0x09, the Smart Battery System Manager at slave address 0x0A, and one or more batteries (multiplexed) at slave address 0x0B. (Notice that Figure 14-1 represents the logical connection of a Smart Battery Subsystem. The actual physical connections of the Smart Battery(s) and the Smart Battery Charger are made through the Smart Battery System Manager.) All devices support the Read/Write Word protocol. Batteries also support the Read/Write Block protocol.

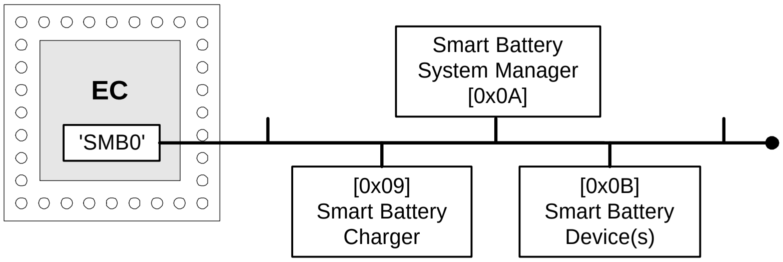

Figure 14-2 Smart Battery Subsystem Devices

The following ASL code shows the use of the OperationRegion term to describe these SMBus devices:

```
Device (SMB0)
{
    Name(_HID, "ACPI0001")                // EC-SMBus Host Controller
    Name(_EC, 0x2030)                     // EC offset 0x20, query bit 0x30

    OperationRegion(SBC0, SMBus, 0x0900, 0x100)      // Smart Battery Charger
    OperationRegion(SBS0, SMBus, 0x0A00, 0x100)      // Smart Battery Selector
    OperationRegion(SBD0, SMBus, 0x0B00, 0x100)      // Smart Battery Device(s)
      :
}
```

Notice that these operation regions in this example are defined within the immediate context of the ‘owning’ EC-SMBus device. Each definition corresponds to a separate slave address (device), and happens to use an initial command value offset of zero (0).

14.5 Declaring SMBus Fields

As with other regions, SMBus operation regions are only accessible via the Field term. Each field element is assigned a unique command value and represents a virtual register on the targeted SMBus device.

The syntax for the **Field** term (from section 16.2.3.3.1.10, “Event [Declare Event Synchronization Object]”) is described below.

```
Field(
    RegionName, //NameString=>OperationRegion
    AccessType, //AccessTypeKeyword
    LockRule,   //LockRuleKeyword
    UpdateRule  //UpdateRuleKeyword – ignored
) {FieldUnitList}
```

Where:

- *RegionName* specifies the operation region name previously defined for the device.
- *AccessType* must be set to **BufferAcc**. This indicates that access to field elements will be done using a region-specific data buffer. For this access type, the field handler is not aware of the data buffer’s contents which may be of any size. When a field of this type is used as the source argument in an operation it simply evaluates to a buffer. When used as the destination, however, the buffer is passed bi-directionally to allow data to be returned fromwrite operations. The modified buffer then becomes the execution result of that operation. This is slightly different than the normal case in which the execution result is the same as the value written to the destination. Note that the source is never changed, since it could be a read only object (see section 14.6, “Declaring an SMBus Data Buffer” and section 16.2.3.4, “Opcode Terms”).
- *LockRule* indicates if access to this operation region requires acquisition of the Global Lock for synchronization. This field should be set to **Lock** on system with firmware that may access the SMBus, and **NoLock** otherwise.
- *UpdateRule* is not applicable to SMBus operation regions since each virtual register is accessed in its entirety. This field is ignored for all SMBus field definitions.

SMBus operation regions require that all field elements be declared at command value granularity. This means that each virtual register cannot be broken down to its individual bits within the field definition.

Access to sub-portions of virtual registers can be done only outside of the field definition. This limitation is imposed both to simplify the SMBus interface and to maintain consistency with the physical model defined by the SMBus specification.

SMBus protocols are assigned to field elements using the AccessAs term within the field definition. The syntax for this term (from section 16.1.3, “ASL Language and Terms”) is described below.

```
AccessAs(
    AccessType, //AccessTypeKeyword
    AccessAttribute    //Nothing | ByteConst | AccessAttribKeyword
)
```

Where:

- *AccessType* must be set to **BufferAcc**.
- *AccessAttribute* indicates the SMBus protocol to assign to command values that follow this term. See section 14.1.2, “SMBus Protocols,” for a listing of the SMBus protocol types and values.

An AccessAs term must appear as the first entry in a field definition to set the initial SMBus protocol for the field elements that follow. A maximum of one SMBus protocol may be defined for each field element. Devices supporting multiple protocols for a single command value can be modeled by specifying multiple field elements with the same offset (command value), where each field element is preceded by an AccessAs term specifying an alternate protocol.

For example, the register at command value 0x08 for a Smart Battery device (illustrated below) represents a word value specifying the battery temperature (in degrees Kelvin), while the register at command value 0x20 represents a variable-length (0 to 32 bytes) character string specifying the name of the company that manufactured the battery.

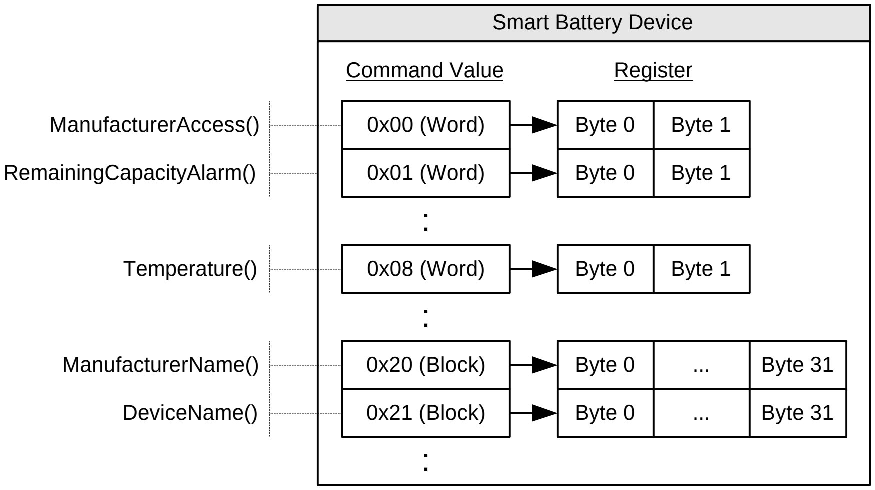

Figure 14-3 Smart Battery Device Virtual Registers

The following ASL code shows the use of the OperationRegion, Field, AccessAs, and Offset terms to represent these Smart Battery device virtual registers:

```
    OperationRegion(SBD0, SMBus, 0x0B00, 0x0100)
    Field(SBD0, BufferAcc, NoLock, Preserve)
    {
        AccessAs(BufferAcc, SMBWord)  // Use the SMBWord protocol for the following…
        MFGA, 8,                      // ManufacturerAccess() [command value 0x00]
        RCAP, 8,                      // RemainingCapacityAlarm() [command value 0x01]
        Offset(0x08)                  // Skip to command value 0x08…
        BTMP, 8,                      // Temperature() [command value 0x08]
        Offset(0x20)                  // Skip to command value 0x20…
        AccessAs(BufferAcc, SMBBlock) // Use the SMBBlock protocol for the following…
        MFGN, 8,                      // ManufacturerName() [command value 0x20]
        DEVN, 8                       // DeviceName() [command value 0x21]
    }
```

Notice that command values are equivalent to the field element’s byte offset (for example, MFGA=0, RCAP=1, BTMP=8). The **AccessAs** term indicates which SMBus protocol to use for each command value.

14.6 Declaring and Using an SMBus Data Buffer

The use of a data buffer for SMBus transactions allows AML to receive status and data length values, as well as making it possible to implement the Process Call protocol. As previously mentioned, the **BufferAcc** access type is used to indicate to the field handler that a region-specific data buffer will be used.

For SMBus operation regions, this data buffer is defined as a fixed-length 34-byte buffer that, if represented using a ‘C’-styled declaration, would be modeled as follows:

```
typedef struct
{
    BYTE        Status;         // Byte 0 of the data buffer
    BYTE        Length;         // Byte 1 of the data buffer
    BYTE[32]    Data;           // Bytes 2 through 33 of the data buffer
}
```

Where:

- *Status* (byte 0) indicates the status code of a given SMBus transaction. See section 14.1.3, “SMBus Status Code,” for more information.
- *Length* (byte 1) specifies the number of bytes of valid data that exists in the data bufferUse of this field is only defined for the Read/Write Block protocol, where valid *Length* values are 0 through 32. For other protocols—where the data length is implied by the protocol—this field is reserved.
- *Data* (bytes 2-33) represents a 32-byte buffer, and is the location where actual data is stored.

For example, the following ASL shows the use of the SMBus data buffer for performing transactions to a Smart Battery device. This code is based on the example ASL presented in section 14.5, “Declaring SMBus Fields,” which lists the operation region and field definitions for the Smart Battery device.

```
/* Create the SMBus data buffer */
Name(BUFF, Buffer(34){})              // Create SMBus data buffer as BUFF
CreateByteField(BUFF, 0x00, OB1)      // OB1 = Status (Byte)
CreateByteField(BUFF, 0x01, OB2)      // OB2 = Length (Byte)
CreateWordField(BUFF, 0x02, OB3)      // OB3 = Data (Word – Bytes 2 & 3)
CreateField(BUFF, 0x10, 256, OB4)     // OB4 = Data (Block – Bytes 2-33)

/* Read the battery temperature */
Store(BTMP, BUFF)                     // Invoke Read Word transaction
If(LEqual(OB1, 0x00))                 // Successful?
{
    // OB3 = Battery temperature in 1/10th degrees Kelvin
}

/* Read the battery manufacturer name */
Store(MFGN, BUFF)                     // Invoke Read Block transaction
If(LEqual(OB1, 0x00))                 // Successful?
{
    // OB2 = Length of the manufacturer name
    // OB4 = Manufacturer name (as a counted string)
}
```

Notice the use of the **CreateField** primitives to access the data buffer’s sub-elements (*Status*, *Length*, and *Data*), where *Data* (bytes 2-33) is ‘typecast’ as both word (OB3) and block (OB4) data.

The example above demonstrates the use of the Store() operator to invoke a Read Block transaction to obtain the name of the battery manufacturer. Evaluation of the *source* operand (MFGN) results in a 34-byte buffer that gets copied by Store() to the *destination* buffer (BUFF).

Capturing the results of a write operation, for example to check the status code, requires an additional Store() operator, as shown below.

```
Store(Store(BUFF, MFGN), BUFF)        // Invoke Write Block transaction
If(LEqual(OB1, 0x00)) {…}             // Transaction successful?
```

Note that the outer Store() copies the results of the Write Block transaction back into BUFF. This is the nature of BufferAcc's bi-directionality described in section 14.5, "Declaring SMBus Fields" It should be noted that storing (or parsing) the result of an SMBus Write transaction is not required although useful for ascertaining the outcome of a transaction.

SMBus Process Call protocols require similar semantics due to the fact that only destination operands are passed bi-directionally. These transactions require the use of the double-Store() semantics to properly capture the return results.

14.7 Using the SMBus Protocols

This section provides information and examples on how each of the SMBus protocols can be used to access SMBus devices from AML.

14.7.1 Read/Write Quick (SMBQuick)

The SMBus Read/Write Quick protocol (SMBQuick) is typically used to control simple devices using a device-specific binary command (for example, ON and OFF). Command values are not used by this protocol and thus only a single element (at offset 0) can be specified in the field definition. This protocol transfers no data.

The following ASL code illustrates how a device supporting the Read/Write Quick protocol should be accessed:

```
OperationRegion(SMBD, SMBus, 0x4200, 0x100)  // SMBus device at slave address 0x42
Field(SMBD, BufferAcc, NoLock, Preserve)
{
    AccessAs(BufferAcc, SMBQuick)         // Use the SMBus Read/Write Quick protocol
    FLD0, 8                               // Virtual register at command value 0.
}

/* Create the SMBus data buffer */

Name(BUFF, Buffer(34){})              // Create SMBus data buffer as BUFF
CreateByteField(BUFF, 0x00, OB1)      // OB1 = Status (Byte)

/* Signal device (e.g. OFF) */
Store(FLD0, BUFF)                     // Invoke Read Quick transaction
If(LEqual(OB1, 0x00)) {…}             // Successful?

/* Signal device (e.g. ON) */
Store(BUFF, FLD0)                     // Invoke Write Quick transaction
```

In this example, a single field element (FLD0) at offset 0 is defined to represent the protocol's read/write bit. Access to FLD0 will cause an SMBus transaction to occur to the device. Reading the field results in a Read Quick, and writing to the field results in a Write Quick. In either case data is not transferred—access to the register is simply used as a mechanism to invoke the transaction.

14.7.2 Send/Receive Byte (SMBSendReceive)

The SMBus Send/Receive Byte protocol (SMBSendReceive) transfers a single byte of data. Like Read/Write Quick, command values are not used by this protocol and thus only a single element (at offset 0) can be specified in the field definition.

The following ASL code illustrates how a device supporting the Send/Receive Byte protocol should be accessed:

```
OperationRegion(SMBD, SMBus, 0x4200, 0x100)  // SMBus device at slave address 0x42
Field(SMBD, BufferAcc, NoLock, Preserve)
{
    AccessAs(BufferAcc, SMBSendReceive)   // Use the SMBus Send/Receive Byte protocol
    FLD0, 8                               // Virtual register at command value 0.
}
```

```
/* Create the SMBus data buffer */

Name(BUFF, Buffer(34){})              // Create SMBus data buffer as BUFF
CreateByteField(BUFF, 0x00, STAT)     // STAT = Status (Byte)
CreateByteField(BUFF, 0x02, DATA)     // DATA = Data (Byte)

/* Receive a byte of data from the device */
Store(FLD0, BUFF)                     // Invoke a Receive Byte transaction
If(LEqual(STAT, 0x00))                // Successful?
{
    // DATA = Received byte…
}

/* Send the byte ‘0x16’ to the device */
Store(0x16, DATA)                     // Save 0x16 into the data buffer
Store(BUFF, FLD0)                     // Invoke a Send Byte transaction
```

In this example, a single field element (FLD0) at offset 0 is defined to represent the protocol’s data byte. Access to FLD0 will cause an SMBus transaction to occur to the device. Reading the field results in a Receive Byte, and writing to the field results in a Send Byte.

14.7.3 Read/Write Byte (SMBByte)

The SMBus Read/Write Byte protocol (SMBByte) also transfers a single byte of data. But unlike Send/Receive Byte, this protocol uses a command value to reference up to 256 byte-sized virtual registers.

The following ASL code illustrates how a device supporting the Read/Write Byte protocol should be accessed:

```
OperationRegion(SMBD, SMBus, 0x4200, 0x100)  // SMBus device at slave address 0x42
Field(SMBD, BufferAcc, NoLock, Preserve)
{
    AccessAs(BufferAcc, SMBByte)          // Use the SMBus Read/Write Byte protocol
    FLD0, 8,                              // Virtual register at command value 0.
    FLD1, 8,                              // Virtual register at command value 1.
    FLD2, 8                               // Virtual register at command value 2.
}

/* Create the SMBus data buffer */
Name(BUFF, Buffer(34){})              // Create SMBus data buffer as BUFF
CreateByteField(BUFF, 0x00, STAT)     // STAT = Status (Byte)
CreateByteField(BUFF, 0x02, DATA)     // DATA = Data (Byte)

/* Read a byte of data from the device using command value 1 */
Store(FLD1, BUFF)                     // Invoke a Read Byte transaction
If(LEqual(STAT, 0x00))                // Successful?
{
    // DATA = Byte read from FLD1…
}

/* Write the byte ‘0x16’ to the device using command value 2 */
Store(0x16, DATA)                     // Save 0x16 into the data buffer
Store(BUFF, FLD2)                     // Invoke a Write Byte transaction
```

In this example, three field elements (FLD0, FLD1, and FLD2) are defined to represent the virtual registers for command values 0, 1, and 2. Access to any of the field elements will cause an SMBus transaction to occur to the device. Reading FLD1 results in a Read Byte with a command value of 1, and writing to FLD2 results in a Write Byte with command value 2.

14.7.4 Read/Write Word (SMBWord)

The SMBus Read/Write Word protocol (SMBWord) transfers 2 bytes of data. This protocol also uses a command value to reference up to 256 word-sized virtual device registers.

The following ASL code illustrates how a device supporting the Read/Write Word protocol should be accessed:

```
OperationRegion(SMBD, SMBus, 0x4200, 0x100)  // SMBus device at slave address 0x42
Field(SMBD, BufferAcc, NoLock, Preserve)
{
    AccessAs(BufferAcc, SMBWord)          // Use the SMBus Read/Write Word protocol
    FLD0, 8,                              // Virtual register at command value 0.
    FLD1, 8,                              // Virtual register at command value 1.
    FLD2, 8                               // Virtual register at command value 2.
}

/* Create the SMBus data buffer */
Name(BUFF, Buffer(34){})              // Create SMBus data buffer as BUFF
CreateByteField(BUFF, 0x00, STAT)     // STAT = Status (Byte)
CreateWordField(BUFF, 0x02, DATA)     // DATA = Data (Word)

/* Read two bytes of data from the device using command value 1 */
Store(FLD1, BUFF)                     // Invoke a Read Word transaction
If(LEqual(STAT, 0x00))                // Successful?
{
    // DATA = Word read from FLD1…
}
/* Write the word ‘0x5416’ to the device using command value 2 */
Store(0x5416, DATA)                   // Save 0x5416 into the data buffer
Store(BUFF, FLD2)                     // Invoke a Write Word transaction
```

In this example, three field elements (FLD0, FLD1, and FLD2) are defined to represent the virtual registers for command values 0, 1, and 2. Access to any of the field elements will cause an SMBus transaction to occur to the device. Reading FLD1 results in a Read Word with a command value of 1, and writing to FLD2 results in a Write Word with command value 2.

Notice that although accessing each field element transmits a word (16 bits) of data, the fields are listed as 8 bits each. The actual data size is determined by the protocol. Every field element is declared with a length of 8 bits so that command values and byte offsets are equivalent.

14.7.5 Read/Write Block (SMBBlock)

The SMBus Read/Write Block protocol (SMBBlock) transfers variable-sized (0-32 bytes) data. This protocol uses a command value to reference up to 256 block-sized virtual registers.

The following ASL code illustrates how a device supporting the Read/Write Block protocol should be accessed:

```
OperationRegion(SMBD, SMBus, 0x4200, 0x100)  // SMBus device at slave address 0x42
Field(SMBD, BufferAcc, NoLock, Preserve)
{
    AccessAs(BufferAcc, SMBBlock)         // Use the SMBus Read/Write Block protocol
    FLD0, 8,                              // Virtual register at command value 0.
    FLD1, 8,                              // Virtual register at command value 1.
    FLD2, 8                               // Virtual register at command value 2.
}

/* Create the SMBus data buffer */
Name(BUFF, Buffer(34){})              // Create SMBus data buffer as BUFF
CreateByteField(BUFF, 0x00, STAT)     // STAT = Status (Byte)
CreateByteField(BUFF, 0x01, SIZE)     // SIZE = Length (Byte)
CreateField(BUFF, 0x10, 256, DATA)    // DATA = Data (Block)

/* Read block data from the device using command value 1 */
Store(FLD1, BUFF)                     // Invoke a Read Block transaction
If(LEqual(STAT, 0x00))                // Successful?
{
    // SIZE = Size (number of bytes) of the block data read from FLD1…
    // DATA = Block data read from FLD1…
}

/* Write the block ‘TEST’ to the device using command value 2 */
Store(“TEST”, DATA)                   // Save “TEST” into the data buffer
Store(4, SIZE)                        // Length of valid data in the data buffer
Store(BUFF, FLD2)                     // Invoke a Write Word transaction
```

In this example, three field elements (FLD0, FLD1, and FLD2) are defined to represent the virtual registers for command values 0, 1, and 2. Access to any of the field elements will cause an SMBus transaction to occur to the device. Reading FLD1 results in a Read Block with a command value of 1, and writing to FLD2 results in a Write Block with command value 2.

14.7.6 Word Process Call (SMBProcessCall)

The SMBus Process Call protocol (SMBProcessCall) transfers 2 bytes of data bi-directionally (performs a Write Word followed by a Read Word as an atomic transaction). This protocol uses a command value to reference up to 256 word-sized virtual registers.

The following ASL code illustrates how a device supporting the Process Call protocol should be accessed:

```
OperationRegion(SMBD, SMBus, 0x4200, 0x100)  // SMBus device at slave address 0x42
Field(SMBD, BufferAcc, NoLock, Preserve)
{
    AccessAs(BufferAcc, SMBProcessCall)   // Use the SMBus Process Call protocol
    FLD0, 8,                              // Virtual register at command value 0.
    FLD1, 8,                              // Virtual register at command value 1.
    FLD2, 8                               // Virtual register at command value 2.
}

/* Create the SMBus data buffer */
Name(BUFF, Buffer(34){})              // Create SMBus data buffer as BUFF
CreateByteField(BUFF, 0x00, STAT)     // STAT = Status (Byte)
CreateWordField(BUFF, 0x02, DATA)     // DATA = Data (Word)

/* Process Call with input value ‘0x5416’ to the device using command value 1 */
Store(0x5416, DATA)                   // Save 0x5416 into the data buffer
Store(Store(BUFF, FLD1), BUFF)        // Invoke a Process Call transaction
If(LEqual(STAT, 0x00))                // Successful?
{
    // DATA = Word returned from FLD1…
}
```

In this example, three field elements (FLD0, FLD1, and FLD2) are defined to represent the virtual registers for command values 0, 1, and 2. Access to any of the field elements will cause an SMBus transaction to occur to the device. Reading *or* writing FLD1 results in a Process Call with a command value of 1. Notice that unlike other protocols, Process Call involves both a write and read operation in a single atomic transaction. This means that the *Data* element of the SMBus data buffer is set with an input value before the transaction is invoked, and holds the output value following the successful completion of the transaction.

14.7.7 Block Process Call (SMBBlockProcessCall)

The SMBus Block Write-Read Block Process Call protocol (SMBBlockProcessCall) transfers a block of data bi-directionally (performs a Write Block followed by a Read Block as an atomic transaction). The maximum aggregate amount of data that may be transferred is limited to 32 bytes. This protocol uses a command value to reference up to 256 block-sized virtual registers.

The following ASL code illustrates how a device supporting the Process Call protocol should be accessed:

```
OperationRegion(SMBD, SMBus, 0x4200, 0x100) // SMbus device at slave address 0x42
Field(SMBD, BufferAcc, NoLock, Preserve)
{
    AccessAs(BufferAcc, SMBBlockProcessCall) // Use the Block Process Call protocol
    FLD0, 8,   // Virtual register representing a command value of 0
    FLD1, 8 // Virtual register representing a command value of 1
}

/* Create the SMBus data buffer as BUFF */
Name(BUFF, Buffer(34)())              // Create SMBus data buffer as BUFF
CreateByteField(BUFF, 0x00, STAT)     // STAT = Status (Byte)
CreateByteField(BUFF, 0x01, SIZE)     // SIZE = Length (Byte)
CreateField(BUFF, 0x10, 256, DATA)    // Data (Block)

/* Process Call with input value "ACPI 2.0" to the device using command value 1 */

Store("ACPI 2.0", DATA)           // Fill in outgoing data
Store(8, SIZE)                    // Length of the valid data
Store(Store(BUFF, FLD1), BUFF)    // Execute the PC
if (LEqual(STAT, 0x00))           // Test the status
{
 /* BUFF now contains information returned from PC */
 /* SIZE now equals size of data returned */
}
```

15 System Address Map Interfaces

This section explains how an ACPI-compatible system conveys its memory resources/type mappings to OSPM. There are three ways for the system to convey memory resources /mappings to OSPM. The first is an INT 15 BIOS interface that is used in IA-PC–based systems to convey the system's initial memory map. EFI enabled systems use the EFI defined **GetMemoryMap()** boot services function to convey memory resources to the OS loader. These resources must then be conveyed by the OS loader to OSPM. See the EFI specification for more information on EFI services.

Lastly, if memory resources may be added or removed dynamically, memory devices are defined in the ACPI Namespace conveying the resource information described by the memory device (see section 10.12, "Memory Devices").

ACPI defines four address range types; AddressRangeMemory, AddressRangeACPI, AddressRangeNVS, and AddressRangeReserved as described in the table below:

Table 15-1 Address Range Types

Value	Mnemonic	Description
1	AddressRangeMemory	This range is available RAM usable by the operating system.
2	AddressRangeReserved	This range of addresses is in use or reserved by the system and must not be used by the operating system.
3	AddressRangeACPI	ACPI Reclaim Memory. This range is available RAM usable by the OS after it reads the ACPI tables.
4	AddressRangeNVS	ACPI NVS Memory. This range of addresses is in use or reserve by the system and must not be used by the operating system. This range is required to be saved and restored across an NVS sleep.
Other	Undefined	Undefined. Reserved for future use. OSPM must treat any range of this type as if the type returned was AddressRangeReserved.

The BIOS can use the *AddressRangeReserved* address range type to block out various addresses as not suitable for use by a programmable device. Some of the reasons a BIOS would do this are:

- The address range contains system ROM.
- The address range contains RAM in use by the ROM.
- The address range is in use by a memory-mapped system device.
- The address range is, for whatever reason, unsuitable for a standard device to use as a device memory space.

Note: OSPM will not save or restore memory reported as AddressRangeReserved when transitioning to or from the S4 sleeping state.

15.1 INT 15H, E820H - Query System Address Map

This interface is used in real mode only on IA-PC-based systems and provides a memory map for all of the installed RAM, and of physical memory ranges reserved by the BIOS. The address map is returned through successive invocations of this interface; each returning information on a single range of physical addresses. Each range includes a type that indicates how the range of physical addresses is to be treated by the OSPM.

If the information returned from E820 in some way differs from INT-15 88 or INT-15 E801, the information returned from E820 supersedes the information returned from INT-15 88 or INT-15 E801. This replacement allows the BIOS to return any information that it requires from INT-15 88 or INT-15 E801 for compatibility reasons. For compatibility reasons, if E820 returns any AddressRangeACPI or AddressRangeNVS memory ranges below 16 MB, the INT-15 88 and INT-15 E801 functions must return the top of memory below the AddressRangeACPI and AddressRangeNVS memory ranges.

The memory map conveyed by this interface is not required to reflect any changes in available physical memory that have occurred after the BIOS has initially passed control to the operating system. For example, if memory is added dynamically, this interface is not required to reflect the new system memory configuration.

Table 15-2 Input

EAX	Function Code	E820h
EBX	Continuation	Contains the continuation value to get the next range of physical memory. This is the value returned by a previous call to this routine. If this is the first call, EBX must contain zero.
ES:DI	Buffer Pointer	Pointer to an Address Range Descriptor structure that the BIOS fills in.
ECX	Buffer Size	The length in bytes of the structure passed to the BIOS. The BIOS fills in the number of bytes of the structure indicated in the ECX register, maximum, or whatever amount of the structure the BIOS implements. The minimum size that must be supported by both the BIOS and the caller is 20 bytes. Future implementations might extend this structure.
EDX	Signature	‘SMAP’ Used by the BIOS to verify the caller is requesting the system map information to be returned in ES:DI.

Table 15-3 Output

CF	Carry Flag	Non-Carry – Indicates No Error
EAX	Signature	‘SMAP.’ Signature to verify correct BIOS revision.
ES:DI	Buffer Pointer	Returned Address Range Descriptor pointer. Same value as on input.
ECX	Buffer Size	Number of bytes returned by the BIOS in the address range descriptor. The minimum size structure returned by the BIOS is 20 bytes.
EBX	Continuation	Contains the continuation value to get the next address range descriptor. The actual significance of the continuation value is up to the discretion of the BIOS. The caller must pass the continuation value unchanged as input to the next iteration of the E820 call in order to get the next Address Range Descriptor. A return value of zero means that this is the last descriptor. **Note:** the BIOS can also indicate that the last descriptor has already been returned during previous iterations by returning the carry flag set. The caller will ignore any other information returned by the BIOS when the carry flag is set.

Table 15-4 Address Range Descriptor Structure

Offset in Bytes	Name	Description
0	BaseAddrLow	Low 32 Bits of Base Address
4	BaseAddrHigh	High 32 Bits of Base Address
8	LengthLow	Low 32 Bits of Length in Bytes
12	LengthHigh	High 32 Bits of Length in Bytes
16	Type	Address type of this range

The *BaseAddrLow* and *BaseAddrHigh* together are the 64-bit base address of this range. The base address is the physical address of the start of the range being specified.

The *LengthLow* and *LengthHigh* together are the 64-bit length of this range. The length is the physical contiguous length in bytes of a range being specified.

The *Type* field describes the usage of the described address range as defined in Table 15-1.

15.2 E820 Assumptions and Limitations

- The BIOS returns address ranges describing baseboard memory.
- The BIOS does *not* return a range description for the memory mapping of PCI devices, ISA Option ROMs, and ISA Plug and Play cards because the OS has mechanisms available to detect them.
- The BIOS returns chip set-defined address holes that are not being used by devices as reserved.
- Address ranges defined for baseboard memory-mapped I/O devices, such as APICs, are returned as reserved.
- All occurrences of the system BIOS are mapped as reserved, including the areas below 1 MB, at 16 MB (if present), and at end of the 4-GB address space.
- Standard PC address ranges are not reported. For example, video memory at A0000 to BFFFF physical addresses are not described by this function. The range from E0000 to EFFFF is specific to the baseboard and is reported as it applies to that baseboard.
- All of lower memory is reported as normal memory. The OS must handle standard RAM locations that are reserved for specific uses, such as the interrupt vector table (0:0) and the BIOS data area (40:0).

15.3 EFI GetMemoryMap() Boot Services Function

EFI enabled systems use the EFI defined GetMemoryMap() boot services function to convey memory resources to the OS loader. These resources must then be conveyed by the OS loader to OSPM.

The GetMemoryMap interface is only available at boot services time. It is not available as a run-time service after OSPM is loaded. The OS or its loader initiates the transition from boot services to run-time services by calling ExitBootServices(). After the call to ExitBootServices() all system memory map information must be derived from objects in the ACPI Namespace.

The GetMemoryMap() interface returns an array of EFI memory descriptors. These memory descriptors define a system memory map of all the installed RAM, and of physical memory ranges reserved by the firmware. Each descriptor contains a type field that dictates how the physical address range is to be treated by the operating system. Table 15-4 below describes the memory types returned by the EFI GetMemoryMap() interface along with a mapping from EFI memory type to ACPI address range types. See the EFI specification for more information on EFI memory types.

Table 15-5 EFI Memory Types and mapping to ACPI address range types

Type	Mnemonic	Description	ACPI Address Range Type
0	EfiReservedMemoryType	Not used.	AddressRangeReserved
1	EfiLoaderCode	The Loader and/or OS may use this memory as they see fit. **Note:** the OS loader that called ExitBootServices() is executing out of one or more EfiLoaderCode sections.	AddressRangeMemory

Table 15-5 EFI Memory Types and mapping to ACPI address range types *(continued)*

Type	Mnemonic	Description	ACPI Address Range Type
2	EfiLoaderData	The Loader and/or OS may use this memory as they see fit. **Note:** the OS loader that called ExitBootServices() is utilizing out of one or more EfiLoaderData sections.	AddressRangeMemory
3	EfiBootServicesCode	Memory available for general use.	AddressRangeMemory
4	EfiBootServicesData	Memory available for general use.	AddressRangeMemory
5	EfiRuntimeServiceCode	The OS and loader must preserve this memory range in the working and ACPI S1–S3 states.	AddressRangeReserved
6	EfiRuntimeServicesData	The OS and loader must preserve this memory range in the working and ACPI S1–S3 states.	AddressRangeReserved
7	EfiConventionalMemory	Memory available for general use.	AddressRangeMemory
8	EfiACPIReclainMemory	The memory is to be preserved by the loader and OS until ACPI in enabled. Once ACPI is enabled, the memory in this range is available for general use.	AddressRangeACPI
9	EfiACPIMemoryNVS	The OS and loader must preserve this memory range in the working and ACPI S1–S3 states.	AddressRangeNVS
10	EfiMemoryMappedIO	The OS does not use this memory. All system memory-mapped I/O port space information should come from ACPI tables.	AddressRangeReserved
11	EfiMemoryMappedIOPortSpace	The OS does not use this memory. All system memory-mapped I/O port space information should come from ACPI tables.	AddressRangeReserved
12	EfiPalCode	The OS and loader must preserve this memory range in the working and ACPI S1–S3 states.	AddressRangeReserved
13	EfiFirmwareReserved	Memory reserved by system firmware.	AddressRangeReserved

15.4 EFI Assumptions and Limitations

- The firmware returns address ranges describing the current system memory configuration.
- The firmware does *not* return a range description for the memory mapping of PCI devices, ISA Option ROMs, and ISA Plug and Play cards because the OS has mechanisms available to detect them.
- The firmware returns chip set-defined address holes that are not being used by devices as reserved.
- Address ranges defined for baseboard memory-mapped I/O devices, such as APICs, are returned as reserved.
- All occurrences of the system firmware are mapped as reserved, including the areas below 1 MB, at 16 MB (if present), and at end of the 4-GB address space. This can include PAL code on Itanium^{TM}-based systems.
- Standard PC address ranges are not reported. For example, video memory at A0000 to BFFFF physical addresses are not described by this function. The range from E0000 to EFFFF is specific to the baseboard and is reported as it applies to that baseboard.
- All of lower memory is reported as normal memory. The OS must handle standard RAM locations that are reserved for specific uses, such as the interrupt vector table (0:0) and the BIOS data area (40:0).
- EFI contains descriptors for memory mapped I/O and memory mapped I/O port space to allow for virtual mode calls to EFI run-time functions. The OS must never use these regions.

15.5 Example Address Map

This sample address map (for an Intel processor-based system) describes a machine that has 128 MB of RAM, 640 KB of base memory and 127 MB of extended memory. The base memory has 639 KB available for the user and 1 KB for an extended BIOS data area. A 4-MB Linear Frame Buffer (LFB) is based at 12 MB. The memory hole created by the chip set is from 8 MB to 16 MB. Memory-mapped APIC devices are in the system. The I/O Unit is at FEC00000 and the Local Unit is at FEE00000. The system BIOS is remapped to 1 GB–64 KB.

The 639-KB endpoint of the first memory range is also the base memory size reported in the BIOS data segment at 40:13. The following table shows the memory map of a typical system.

Table 15-6 Sample Memory Map

Base (Hex)	Length	Type	Description
0000 0000	639 KB	AddressRangeMemory	Available Base memory. Typically the same value as is returned using the INT 12 function.
0009 FC00	1 KB	AddressRangeReserved	Memory reserved for use by the BIOS(s). This area typically includes the Extended BIOS data area.
000F 0000	64 KB	AddressRangeReserved	System BIOS
0010 0000	7 MB	AddressRangeMemory	Extended memory, which is not limited to the 64-MB address range.
0080 0000	4 MB	AddressRangeReserved	Chip set memory hole required to support the LFB mapping at 12 MB.
0100 0000	120 MB	AddressRangeMemory	Baseboard RAM relocated above a chip set memory hole.
FEC0 0000	4 KB	AddressRangeReserved	I/O APIC memory mapped I/O at FEC00000.
FEE0 0000	4 KB	AddressRangeReserved	Local APIC memory mapped I/O at FEE00000.
FFFF 0000	64 KB	AddressRangeReserved	Remapped System BIOS at end of address space.

15.6 Example: Operating System Usage

The following code segment illustrates the algorithm to be used when calling the Query System Address Map function. It is an implementation example and uses non-standard mechanisms.

```
E820Present = FALSE;
    Reg.ebx = 0;
    do {
        Reg.eax = 0xE820;
        Reg.es  = SEGMENT (&Descriptor);
        Reg.di  = OFFSET  (&Descriptor);
        Reg.ecx = sizeof  (Descriptor);
        Reg.edx = 'SMAP';

        _int( 15, regs );

        if ((Regs.eflags & EFLAG_CARRY)  ||  Regs.eax != 'SMAP') {
            break;
        }

        if (Regs.ecx < 20  ||  Reg.ecx > sizeof (Descriptor) ) {
            // bug in bios - all returned descriptors must be
            // at least 20 bytes long, and cannot be larger then
            // the input buffer.

            break;
        }

        E820Present = TRUE;
            .
            .
            .
        Add address range Descriptor.BaseAddress through
        Descriptor.BaseAddress + Descriptor.Length
        as type Descriptor.Type
            .
            .
            .

    } while (Regs.ebx != 0);

    if (!E820Present) {
        .
        .
        .
    call INT-15 88 and/or INT-15 E801 to obtain old style
    memory information
        .
        .
        .
    }
```

16 ACPI Source Language (ASL) Reference

This section formally defines the ACPI Source Language (ASL). ASL is a source language for defining ACPI objects including writing ACPI control methods. OEMs and BIOS developers define objects and write control methods in ASL and then use a translator tool (compiler) to generate ACPI Machine Language (AML) versions of the control methods. For a formal definition of AML, see the ACPI Machine Language (AML) Specification, section 17, “ACPI Machine Language Specification.”

AML and ASL are ***different languages*** though they are closely related.

Every ACPI-compatible OS must support AML. A given user can define some arbitrary source language (to replace ASL) and write a tool to translate it to AML.

An OEM or BIOS vendor needs to write ASL and be able to single-step AML for debugging. (Debuggers and similar tools are expected to be AML-level tools, not source-level tools.) An ASL translator implementer must understand how to read ASL and generate AML. An AML interpreter author must understand how to execute AML.

This section has two parts:

- The ASL grammar, which is the formal ASL specification and also serves as a quick reference.
- A full ASL reference, which repeats the ASL term syntax and adds information about the semantics of the language.

16.1 ASL Language Grammar

The purpose of this section is to state unambiguously the grammar rules used by the syntax checker of an ASL compiler.

ASL statements declare objects. Each object has three parts, two of which might not be present.

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

FixedList refers to a list, of known length, that supplies data that all instances of a given **ObjectType** must have. A fixed list is written as **(*a* , *b* , *c* , …)** where the number of arguments depends on the specific **ObjectType**, and some elements can be nested objects, that is **(a, b, (q, r, s, t), d)**. Arguments to a **FixedList** can have default values, in which case they can be skipped. Thus, **(a,,c)** will cause the default value for the second argument to be used. Some **ObjectTypes** can have a null **FixedList,** which is simply omitted. Trailing arguments of some object types can be left out of a fixed list, in which case the default value is used.

VariableList refers to a list, not of predetermined length, of child objects that help define the parent. It is written as **{ x, y, z, aa, bb, cc }** where any argument can be a nested object. **ObjectType** determines what terms are legal elements of the **VariableList**. Some **ObjectTypes** may have a null variable list, which is simply omitted.

Other rules for writing ASL statements are the following:

- Multiple blanks are the same as one. Blank, (,), ‘,’ and newline are all token separators.
- // marks the beginning of a comment, which continues from the // to the end of the line.
- /* marks the beginning of a comment, which continues from the /* to the next */.
- **“”** surround an ASCII string.

- Numeric constants can be written in three ways: ordinary decimal, octal (using 0*ddd*) or hexadecimal, using the notation 0x*dd.*
- **Nothing** indicates an empty item. For example, **{ Nothing }** is equivalent to **{}**.

16.1.1 ASL Grammar Notation

The notation used to express the ASL grammar is specified in the following table.

Table 16-1 ASL Grammar Notation

Notation Convention	**Description**	**Example**
Term := Term Term …	The term to the left of := can be expanded into the sequence of terms on the right.	aterm := bterm cterm means that aterm can be expanded into the two-term sequence of bterm followed by cterm.
Angle brackets (< >)	Used to group items.	<a b> | <c d> means either a b or c d.
Arrow (=>))	Indicates required run-time reduction of an ASL argument to an AML data type. Means "reduces to" or "evaluates to" at run-time.	**TermArg=>Integer** means that the argument must be an ASL **TermArg** that must resolve to an **Integer** data type when it is evaluated by an AML interpreter.
Bar symbol (|)	Separates alternatives.	aterm := bterm | <cterm dterm> means the following constructs are possible: bterm cterm dterm aterm := <bterm | cterm> dterm means the following constructs are possible: bterm dterm cterm dterm
Term Term Term	Terms separated from each other by spaces form an ordered list.	N/A
Word in bold.	Denotes the name of a term in the ASL grammar, representing any instance of such a term. ASL terms are not case-sensitive.	In the following ASL term definition: **ThermalZone** (*ZoneName*) {ObjectList} the item in bold is the name of the term.
Word in italics	Names of arguments to objects that are replaced for a given instance.	In the following ASL term definition: **ThermalZone** (*ZoneName*) {ObjectList} the italicized item is an argument. The item that is not bolded or italicized is defined elsewhere in the ASL grammar.
Single quotes (' ')	Indicate constant characters.	'A'
0x*dd*	Refers to a byte value expressed as 2two hexadecimal digits.	0x21 means a value of hexadecimal 21, or decimal 37. Notice that a value expressed in hexadecimal must start with a leading zero (0).
Dash character (**-**)	Indicates a range.	1-9 means a single digit in the range 1 to 9 inclusive.

16.1.2 ASL Names

```
LeadNameChar                  := ‘A’-‘Z’ | ‘a’-‘z’ | ‘_’
DigitChar                     := ‘0’-‘9’
NameChar                      := DigitChar | LeadNameChar
RootChar                      := ‘\’
NameSeg                       := <LeadNameChar NameChar NameChar NameChar> |
                                 <LeadNameChar NameChar NameChar> |
                                 <LeadNameChar NameChar> |
                                 <LeadNameChar>
NameString                    := <RootChar NamePath> | <‘^’ PrefixPath NamePath> |
                                 NonEmptyNamePath
PrefixPath                    := Nothing | <‘^’ PrefixPath>
NamePath                      := Nothing | <NameSeg NamePathTail>
NonEmptyNamePath              := NameSeg | <NameSeg NamePathTail>
NamePathTail                  := Nothing | <‘.’ NameSeg NamePathTail>
```

16.1.3 ASL Language and Terms

```
ASLCode                       := DefinitionBlockTerm

DefinitionBlockTerm           := DefinitionBlock(
                                     AMLFileName,           //StringData
                                     TableSignature,        //StringData
                                     ComplianceRevision,    //ByteConst
                                     OEMID,                 //StringData
                                     TableID,               //StringData
                                     OEMRevision            //DWordConst
                                 ) {ObjectList}

ObjectList                    := Nothing | <Object ObjectList>
Object                        := CompilerDirective | NamedObject | NameSpaceModifier

DataObject                    := BufferData | PackageData | IntegerData | StringData
DataRefObject                 := DataObject | ObjectReference | DDBHandle

ComputationalData             := BufferData | IntegerData | StringData
BufferData                    := Type5Opcode | BufferTerm
PackageData                   := PackageTerm
IntegerData                   := Type3Opcode | Integer | ConstTerm
StringData                    := Type4Opcode | String

NamedObject                   := BankFieldTerm | CreateBitFieldTerm | CreateByteFieldTerm
                                 | CreateDWordFieldTerm | CreateFieldTerm |
                                 CreateQWordFieldTerm | CreateWordFieldTerm |
                                 DataRegionTerm | DeviceTerm | EventTerm | FieldTerm |
                                 IndexFieldTerm | MethodTerm | MutexTerm | OpRegionTerm |
                                 PowerResTerm | ProcessorTerm | ThermalZoneTerm

NameSpaceModifier             := AliasTerm | NameTerm | ScopeTerm

TermList                      := Nothing | <Term TermList>
Term                          := Object | Type1Opcode | Type2Opcode

CompilerDirective             := IncludeTerm | ExternalTerm

UserTerm                      := NameString(                //NameString=>Method
                                     ArgList
                                 ) => Nothing | DataRefObject
ArgList                       := Nothing | <TermArg ArgListTail>
ArgListTail                   := Nothing | <‘,’ TermArg ArgListTail>
TermArg                       := Type2Opcode | DataObject | ArgTerm | LocalTerm |
                                 NameString
Target                        := Nothing | SuperName
```

```
Type1Opcode                  := BreakTerm | BreakPointTerm | ContinueTerm | FatalTerm |
                                IfElseTerm | LoadTerm | NoOpTerm | NotifyTerm |
                                ReleaseTerm | ResetTerm | ReturnTerm | SignalTerm |
                                SleepTerm | StallTerm | SwitchTerm | UnloadTerm |
                                WhileTerm
```

A Type 1 opcode term does not return a value and can only be used standalone on a line of ASL code. Since these opcodes do not return a value they cannot be used as a term in an expression.

```
Type2Opcode                  := AcquireTerm | AddTerm | AndTerm | ConcatTerm |
                                ConcatResTerm | CondRefOfTerm | CopyObjectTerm |
                                DecTerm | DerefOfTerm | DivideTerm |FindSetLeftBitTerm |
                                FindSetRightBitTerm | FromBCDTerm | IncTerm |
                                IndexTerm | LAndTerm | LEqualTerm | LGreaterTerm |
                                LGreaterEqualTerm | LLessTerm | LLessEqualTerm |
                                LNotTerm | LNotEqualTerm | LoadTableTerm | LOrTerm |
                                MatchTerm | MidTerm |ModTerm | MultiplyTerm | NAndTerm |
                                NOrTerm | NotTerm | ObjectTypeTerm | OrTerm |
                                RefOfTerm | ShiftLeftTerm | ShiftRightTerm |
                                SizeOfTerm | StoreTerm | SubtractTerm | ToBCDTerm |
                                ToBufferTerm | ToDecimalStringTerm | ToHexStringTerm |
                                ToIntegerTerm | ToStringTerm | WaitTerm | XorTerm |
                                UserTerm
```

A Type 2 opcode returns a value and can be used in an expression.

```
Type3Opcode                  := AddTerm | AndTerm | DecTerm | DivideTerm | EISAIDTerm |
                                FindSetLeftBitTerm | FindSetRightBitTerm | FromBCDTerm |
                                IncTerm | IndexTerm | LAndTerm | LEqualTerm |
                                LGreaterTerm | LGreaterEqualTerm | LLessTerm |
                                LLessEqualTerm | LNotTerm | LNotEqualTerm | LOrTerm |
                                MatchTerm | ModTerm | MultiplyTerm | NAndTerm |
                                NOrTerm | NotTerm | OrTerm | ShiftLeftTerm |
                                ShiftRightTerm | SubtractTerm | ToBCDTerm |
                                ToIntegerTerm | XorTerm
```

The Type 3 opcodes are a subset of Type 2 opcodes that return an Integer value and can be used in a expression that evaluates to a constant. These opcodes may be evaluated at ASL compile-time. To ensure that these opcodes will evaluate to a constant, the following rules apply: The term cannot have a destination (target) operand, and must have either a Type3Opcode, Type4Opcode, Type5Opcode, ConstExprTerm, Integer, BufferTerm, Package, or String for all arguments.

```
Type4Opcode                  := ConcatTerm | MidTerm | ToDecimalStringTerm |
                                ToHexStringTerm | ToStringTerm
```

The Type 4 opcodes are a subset of Type 2 opcodes that return an String value and can be used in a expression that evaluates to a constant. These opcodes may be evaluated at ASL compile-time. To ensure that these opcodes will evaluate to a constant, the following rules apply: The term cannot have a destination (target) operand, and must have either a Type3Opcode, Type4Opcode, Type5Opcode, ConstExprTerm, Integer, BufferTerm, Package, or String for all arguments.

```
Type5Opcode                  := ConcatTerm | ConcatResTerm | MidTerm |
                                ResourceTemplateTerm | ToBufferTerm | UnicodeTerm
```

The Type 5 opcodes are a subset of Type 2 opcodes that return a Buffer value and can be used in a expression that evaluates to a constant. These opcodes may be evaluated at ASL compile-time. To ensure that these opcodes will evaluate to a constant, the following rules apply: The term cannot have a destination (target) operand, and must have either a Type3Opcode, Type4Opcode, Type5Opcode, ConstExprTerm, Integer, BufferTerm, Package, or String for all arguments.

```
Type6Opcode                  := RefOfTerm | DerefOfTerm | IndexTerm | UserTerm

IncludeTerm                  := Include(
                                    IncFilePathName        //StringData
                                )
```

The file must contain elements that are grammatically correct in the current scope

```
ExternalTerm                  := External(
                                      ObjName,              //NameString
                                      ObjType               //Nothing | ObjectTypeKeyword
                                  )

BankFieldTerm                 := BankField(
                                      RegionName,           //NameString=>OperationRegion
                                      BankName,             //NameString=>FieldUnit
                                      BankValue,            //TermArg=>Integer
                                      AccessType,           //AccessTypeKeyword
                                      LockRule,             //LockRuleKeyword
                                      UpdateRule            //UpdateRuleKeyword
                                  ) {FieldUnitList}

FieldUnitList                 := Nothing | <FieldUnit FieldUnitListTail>
FieldUnitListTail             := Nothing | <‘,’ FieldUnit FieldUnitListTail>

FieldUnit                     := FieldUnitEntry | OffsetTerm | AccessAsTerm
FieldUnitEntry                := <Nothing | NameSeg> ‘,’ Integer

OffsetTerm                    := Offset(
                                      ByteOffset            //IntegerData
                                  )

AccessAsTerm                  := AccessAs(
                                      AccessType,           //AccessTypeKeyword
                                      AccessAttribute       //Nothing | ByteConstExpr |
                                                            //AccessAttribKeyword
                                  )

CreateBitFieldTerm            := CreateBitField(
                                      SourceBuffer,         //TermArg=>Buffer
                                      BitIndex,             //TermArg=>Integer
                                      BitFieldName          //NameString
                                  )

CreateByteFieldTerm           := CreateByteField(
                                      SourceBuffer,         //TermArg=>Buffer
                                      ByteIndex,            //TermArg=>Integer
                                      ByteFieldName         //NameString
                                  )

CreateDWordFieldTerm          := CreateDWordField(
                                      SourceBuffer,         //TermArg=>Buffer
                                      ByteIndex,            //TermArg=>Integer
                                      DWordFieldName        //NameString
                                  )

CreateFieldTerm               := CreateField(
                                      SourceBuffer,         //TermArg=>Buffer
                                      BitIndex,             //TermArg=>Integer
                                      NumBits,              //TermArg=>Integer
                                      FieldName             //NameString
                                  )

CreateQWordFieldTerm          := CreateQWordField(
                                      SourceBuffer,         //TermArg=>Buffer
                                      ByteIndex,            //TermArg=>Integer
                                      QWordFieldName        //NameString
                                  )

CreateWordFieldTerm           := CreateWordField(
                                      SourceBuffer,         //TermArg=>Buffer
                                      ByteIndex,            //TermArg=>Integer
                                      WordFieldName         //NameString
                                  )

DataRegionTerm                := DataTableRegion(
                                      RegionName,           // NameString
                                      SignatureString,      // TermArg=>String
                                      OemIDString,          // TermArg=>String
```

```
                                     OemTableIDString       // TermArg=>String
                                  )

DeviceTerm                     := Device(
                                     DeviceName             //NameString
                                  ) {ObjectList}

EventTerm                      := Event(
                                     EventName              //NameString
                                  )

FieldTerm                      := Field(
                                     RegionName,            //NameString=>OperationRegion
                                     AccessType,            //AccessTypeKeyword
                                     LockRule,              //LockRuleKeyword
                                     UpdateRule             //UpdateRuleKeyword
                                  ) {FieldUnitList}

IndexFieldTerm                 := IndexField(
                                     IndexName,             //NameString=>FieldUnit
                                     DataName,              //NameString=>FieldUnit
                                     AccessType,            //AccessTypeKeyword
                                     LockRule,              //LockRuleKeyword
                                     UpdateRule             //UpdateRuleKeyword
                                  ) {FieldUnitList}

MethodTerm                     := Method(
                                     MethodName,            //NameString
                                     NumArgs,               //Nothing | ByteConstExpr
                                     SerializeRule,         //Nothing |
                                                            //SerializeRuleKeyword
                                     SyncLevel              //Nothing | ByteConstExpr
                                  ) {TermList}

MutexTerm                      := Mutex(
                                     MutexName,             //NameString
                                     SyncLevel              //ByteConstExpr
                                  )

OpRegionTerm                   := OperationRegion(
                                     RegionName,            //NameString
                                     RegionSpace,           //RegionSpaceKeyword
                                     Offset,                //TermArg=>Integer
                                     Length                 //TermArg=>Integer
                                  )

PowerResTerm                   := PowerResource(
                                     ResourceName,          //NameString
                                     SystemLevel,           //ByteConstExpr
                                     ResourceOrder          //WordConstExpr
                                  ) {ObjectList}

ProcessorTerm                  := Processor(
                                     ProcessorName,         //NameString
                                     ProcessorID,           //ByteConstExpr
                                     PBlockAddress,         //DWordConstExpr|Nothing (=0)
                                     PblockLength           //ByteConstExpr|Nothing (=0)
                                  ) {ObjectList}

ThermalZoneTerm                := ThermalZone(
                                     ThermalZoneName        //NameString
                                  ) {ObjectList}

AliasTerm                      := Alias(
                                     SourceObject,          //NameString
                                     AliasObject            //NameString
                                  )
```

```
NameTerm                      := Name(
                                     ObjectName,           //NameString
                                     Object                //DataObject
                                  )

ScopeTerm                     := Scope(
                                     Location              //NameString
                                  ) {ObjectList}

BreakTerm                     := Break

BreakPointTerm                := BreakPoint

ContinueTerm                  := Continue

FatalTerm                     := Fatal(
                                     Type,                 //ByteConstExpr
                                     Code,                 //DWordConstExpr
                                     Arg                   //TermArg=>Integer
                                  )

IfElseTerm                    := IfTerm ElseTerm

IfTerm                        := If(
                                     Predicate             //TermArg=>Integer
                                  ) {TermList}

ElseTerm                      := Nothing | <Else {TermList}> | <ElseIf (
                                     Predicate             //TermArg=>Integer
                                  ) {TermList} ElseTerm>

LoadTerm                      := Load(
                                     Object,               //NameString
                                     DDBHandle             //SuperName
                                  )

NoOpTerm                      := Noop

NotifyTerm                    := Notify(
                                     Object,               //SuperName=>ThermalZone |
                                                           // Processor | Device
                                     NotificationValue     //TermArg=>Integer
                                  )

ReleaseTerm                   := Release(
                                     SyncObject            //SuperName
                                  )

ResetTerm                     := Reset(
                                     SyncObject            //SuperName
                                  )

ReturnTerm                    := Return(
                                     Arg                   //Nothing |
                                                           // TermArg=>DataRefObject
                                  )

SignalTerm                    := Signal(
                                     SyncObject            //SuperName
                                  )

SleepTerm                     := Sleep(
                                     MilliSecs             //TermArg=>Integer
                                  )

StallTerm                     := Stall(
                                     MicroSecs             //TermArg=>Integer
                                  )

SwitchTerm                    := Switch(
                                  Predicate                //TermArg=>ComputationalData
                                  ) {CaseTermList}

CaseTermList                  := Nothing | CaseTerm | DefaultTerm DefaultTermList |
                                  CaseTerm CaseTermList

DefaultTermList               := Nothing | CaseTerm | CaseTerm DefaultTermList

CaseTerm                      := Case(
                                      Value                 //DataObject
                                  ) {TermList}
DefaultTerm                   := Default {TermList}

UnloadTerm                    := Unload(
                                      DDBHandle             //SuperName
                                  )

WhileTerm                     := While(
                                      Predicate             //TermArg=>Integer
                                  ) {TermList}

AcquireTerm                   := Acquire(
                                      SyncObject,           //SuperName=>Mutex
                                      TimeoutValue          //WordConstExpr
                                  ) => Boolean              // True means timed-out

AddTerm                       := Add(
                                      Addend1,              //TermArg=>Integer
                                      Addend2,              //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

AndTerm                       := And(
                                      Source1,              //TermArg=>Integer
                                      Source2,              //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

ConcatTerm                    := Concatenate(
                                      Source1,              //TermArg=>ComputationalData
                                      Source2,              //TermArg=>ComputationalData
                                      Result                //Target
                                  ) => ComputationalData

ConcatResTerm                 := ConcatenateResTemplate(
                                      Source1,              //TermArg=>Buffer
                                      Source2,              //TermArg=>Buffer
                                      Result                //Target
                                  ) => Buffer

CondRefOfTerm                 := CondRefOf(
                                      Source,               //SuperName
                                      Destination           //Target
                                  ) => Boolean

CopyObjectTerm                := CopyObject(
                                      Source,               //TermArg=>DataRefObject
                                      Result,               //NameString | LocalTerm |
                                                            //  ArgTerm
                                  ) => DataRefObject

DecTerm                       := Decrement(
                                      Addend                //SuperName
                                  ) => Integer

DerefOfTerm                   := DerefOf(
                                      Source                //TermArg=>ObjectReference
                                                            //ObjectReference is an
                                                            //object produced by terms
                                                            //such as Index, RefOf or
                                                            //CondRefOf.
                                  ) => DataRefObject
```

```
DivideTerm                    := Divide(
                                     Dividend,              //TermArg=>Integer
                                     Divisor,               //TermArg=>Integer
                                     Remainder,             //Target
                                     Result                 //Target
                                  ) => Integer              //returns Result

FindSetLeftBitTerm            := FindSetLeftBit(
                                     Source,                //TermArg=>Integer
                                     Result                 //Target
                                  ) => Integer

FindSetRightBitTerm           := FindSetRightBit(
                                     Source,                //TermArg=>Integer
                                     Result                 //Target
                                  ) => Integer

FromBCDTerm                   := FromBCD(
                                     BCDValue,              //TermArg=>Integer
                                     Result                 //Target
                                  ) => Integer

IncTerm                       := Increment(
                                     Addend                 //SuperName
                                  ) => Integer

IndexTerm                     := Index(
                                     Source,                //TermArg=>
                                                            // <String | Buffer |
                                                            //  PackageTerm>
                                     Index,                 //TermArg=>Integer
                                     Destination            //Target
                                  ) => ObjectReference

LAndTerm                      := LAnd(
                                     Source1,               //TermArg=>Integer
                                     Source2                //TermArg=>Integer
                                  ) => Boolean

LEqualTerm                    := LEqual(
                                     Source1,               //TermArg=>ComputationalData
                                     Source2                //TermArg=>ComputationalData
                                  ) => Boolean

LGreaterTerm                  := LGreater(
                                     Source1,               //TermArg=>ComputationalData
                                     Source2                //TermArg=>ComputationalData
                                  ) => Boolean

LGreaterEqualTerm             := LGreaterEqual(
                                     Source1,               //TermArg=>ComputationalData
                                     Source2                //TermArg=>ComputationalData
                                  ) => Boolean

LLessTerm                     := LLess(
                                     Source1,               //TermArg=>ComputationalData
                                     Source2                //TermArg=>ComputationalData
                                  ) => Boolean

LLessEqualTerm                := LLessEqual(
                                     Source1,               //TermArg=>ComputationalData
                                     Source2                //TermArg=>ComputationalData
                                  ) => Boolean

LNotTerm                      := LNot(
                                     Source,                //TermArg=>ComputationalData
                                  ) => Boolean
```

```
LNotEqualTerm                 := LNotEqual(
                                      Source1,              //TermArg=>ComputationalData
                                      Source2               //TermArg=>ComputationalData
                                  ) => Boolean

LoadTableTerm                 := LoadTable(
                                      SignatureString,      //TermArg=>String
                                      OemIDString,          //TermArg=>String
                                      OemTableIDString,     //TermArg=>String
                                      RootPathString,       //Nothing | TermArg=>String
                                      ParameterPathString,  //Nothing | TermArg=>String
                                      ParameterData         //Nothing |
                                                            //   TermArg=>DataRefObject
                                                                    ) => DDBHandle

LOrTerm                       := LOr(
                                      Source1,              //TermArg=>ComputationalData
                                      Source2               //TermArg=>ComputationalData
                                  ) => Boolean

MatchTerm                     := Match(
                                      SearchPackage,        //TermArg=>Package
                                      Op1,                  //MatchOpKeyword
                                      MatchObject1,         //TermArg=>Integer
                                      Op2,                  //MatchOpKeyword
                                      MatchObject2,         //TermArg=>Integer
                                      StartIndex            //TermArg=>Integer
                                  ) => Ones | Integer

MidTerm                       := Mid(
                                      Source,               //TermArg=>Buffer|String
                                      Index,                //TermArg=>Integer
                                      Length,               //TermArg=>Integer
                                      Result                //Target
                                  ) => Buffer|String

ModTerm                       := Mod(
                                      Dividend,             //TermArg=>Integer
                                      Divisor,              //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer              //returns Result


MultiplyTerm                  := Multiply(
                                      Multiplicand,         //TermArg=>Integer
                                      Multiplier,           //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

NAndTerm                      := NAnd(
                                      Source1,              //TermArg=>Integer
                                      Source2               //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

NOrTerm                       := NOr(
                                      Source1,              //TermArg=>Integer
                                      Source2               //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

NotTerm                       := Not(
                                      Source,               //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

ObjectTypeTerm                := ObjectType(
                                      Object                //SuperName
                                  ) => Integer
```

```
OrTerm                        := Or(
                                      Source1,              //TermArg=>Integer
                                      Source2               //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

RefOfTerm                     := RefOf(
                                      Object                //SuperName
                                  ) => ObjectReference

ShiftLeftTerm                 := ShiftLeft(
                                      Source,               //TermArg=>Integer
                                      ShiftCount            //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

ShiftRightTerm                := ShiftRight(
                                      Source,               //TermArg=>Integer
                                      ShiftCount            //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

SizeOfTerm                    := SizeOf(
                                      DataObject            //SuperName=>
                                                            //  String | Buffer | Package
                                  ) => Integer

StoreTerm                     := Store(
                                      Source,               //TermArg=>DataRefObject
                                      Destination           //SuperName
                                  ) => DataRefObject

SubtractTerm                  := Subtract(
                                      Addend1,              //TermArg=>Integer
                                      Addend2,              //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

ToBCDTerm                     := ToBCD(
                                      Value,                //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer

ToBufferTerm                  := ToBuffer(
                                      Data,                 //TermArg=>ComputationalData
                                      Result                //Target
                                  ) => ComputationalData

ToDecimalStringTerm           := ToDecimalString(
                                      Data,                 //TermArg=>ComputationalData
                                      Result                //Target
                                  ) => String

ToHexStringTerm               := ToHexString(
                                      Data,                 //TermArg=>ComputationalData
                                      Result                //Target
                                  ) => String

ToIntegerTerm                 := ToInteger(
                                      Data,                 //TermArg=>ComputationalData
                                      Result                //Target
                                  ) => Integer

ToStringTerm                  := ToString(
                                      Source,               //TermArg=>Buffer
                                      Length,               //Nothing | TermArg=>Integer
                                      Result                //Target
                                  ) => String
```

```
WaitTerm                      := Wait(
                                     SyncObject,            //SuperName=>Event
                                     TimeoutValue           //TermArg=>Integer
                                  ) => Boolean              // True means timed-out

XOrTerm                       := XOr(
                                     Source1,               //TermArg=>Integer
                                     Source2                //TermArg=>Integer
                                     Result                 //Target
                                  ) => Integer

ObjectTypeKeyword             := UnknownObj | IntObj | StrObj | BuffObj | PkgObj |
                                  FieldUnitObj | DeviceObj | EventObj | MethodObj |
                                  MutexObj | OpRegionObj | PowerResObj | ThermalZoneObj |
                                  BuffFieldObj | DDBHandleObj

AccessTypeKeyword             := AnyAcc | ByteAcc | WordAcc | DWordAcc | QWordAcc |
                                  BufferAcc
AccessAttribKeyword           := SMBQuick | SMBSendReceive | SMBByte | SMBWord | SMBBlock
                                  | SMBProcessCall | SMBBlockProcessCall
                                     // Note: AccessAttribKeywords are for
                                     //       SMBus BufferAcc only.

LockRuleKeyword               := Lock | NoLock
UpdateRuleKeyword             := Preserve | WriteAsOnes | WriteAsZeros

RegionSpaceKeyword            := UserDefRegionSpace | SystemIO | SystemMemory |
                                  PCI_Config | EmbeddedControl | SMBus | SystemCMOS |
                                  PciBarTarget
AddressSpaceKeyword           := RegionSpaceKeyword | FFixedHW
UserDefRegionSpace            := IntegerData => 0x80-0xff

SerializeRuleKeyword          := Serialized | NotSerialized

MatchOpKeyword                := MTR | MEQ | MLE | MLT | MGE | MGT

DMATypeKeyword                := Compatibility | TypeA | TypeB | TypeF
BusMasterKeyword              := BusMaster | NotBusMaster
XferTypeKeyword               := Transfer8 | Transfer16 | Transfer8_16

ResourceTypeKeyword           := ResourceConsumer | ResourceProducer
MinKeyword                    := MinFixed | MinNotFixed
MaxKeyword                    := MaxFixed | MaxNotFixed
DecodeKeyword                 := SubDecode | PosDecode
RangeTypeKeyword              := ISAOnlyRanges | NonISAOnlyRanges | EntireRange
MemTypeKeyword                := Cacheable | WriteCombining | Prefetchable | NonCacheable
ReadWriteKeyword              := ReadWrite | ReadOnly
InterruptTypeKeyword          := Edge | Level
InterruptLevel                := ActiveHigh | ActiveLow
ShareTypeKeyword              := Shared | Exclusive
IODecodeKeyword               := Decode16 | Decode10
TypeKeyword                   := TypeTranslation | TypeStatic
TranslationKeyword            := SparseTranslation | DenseTranslation
AddressKeyword                := AddressRangeMemory | AddressRangeReserved |
                                  AddressRangeNVS | AddressRangeACPI

SuperName                     := NameString | ArgTerm | LocalTerm | DebugTerm |
                                  Type6Opcode | UserTerm
ArgTerm                       := Arg0 | Arg1 | Arg2 | Arg3 | Arg4 | Arg5 | Arg6
LocalTerm                     := Local0 | Local1 | Local2 | Local3 | Local4 | Local5 |
                                  Local6 | Local7
DebugTerm                     := Debug

LeadDigitChar                 := ‘1’-‘9’
OctalDigitChar                := ‘0’-‘7’
HexDigitChar                  := DigitChar | ‘A’-‘F’ | ‘a’-‘f’


Integer                       := DecimalConst | OctalConst | HexConst
DecimalConst                  := LeadDigitChar | <DecimalConst DigitChar>
OctalConst                    := ‘0’ | <OctalConst OctalDigitChar>
```

```
HexConst                      := <0x HexDigitChar> | <0X HexDigitChar> | <HexConst
                                  HexDigitChar>
ByteConst                     := Integer => 0x00-0xFF
WordConst                     := Integer => 0x0000-0xFFFF
DWordConst                    := Integer => 0x00000000-0xFFFFFFFF
QWordConst                    := Integer => 0x0000000000000000-0xFFFFFFFFFFFFFFFF

DDBHandle                     := Integer
ObjectReference               := Integer
String                        := ‘”’ AsciiCharList ‘”’
AsciiCharList                 := Nothing | <EscapeSeq AsciiCharList> | <AsciiChar
                                  AsciiCharList>
AsciiChar                     := 0x01-0x21 | 0x23-0x5B | 0x5D-0x7F
EscapeSeq                     := SimpleEscapeSeq | OctalEscapeSeq | HexEscapeSeq
SimpleEscapeSeq               := \' | \" | \a | \b | \f | \n | \r | \t | \v | \\
OctalEscapeSeq                :=  \ OctalDigit |
                                  \ OctalDigit OctalDigit |
                                  \ OctalDigit OctalDigit OctalDigit
OctalDigitChar                := ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’
HexEscapeSeq                  :=  \x HexDigitChar |
                                  \x HexDigitChar HexDigitChar
NullChar                      := 0x00
ConstTerm                     := Zero | One | Ones | Revision
Boolean                       := True | False
True                          := Ones
False                         := Zero

ByteConstExpr                 := <Type3Opcode | ConstExprTerm | Integer> => ByteConst
WordConstExpr                 := <Type3Opcode | ConstExprTerm | Integer> => WordConst
DwordConstExpr                := <Type3Opcode | ConstExprTerm | Integer> => DWordConst
QwordConstExpr                := <Type3Opcode | ConstExprTerm | Integer> => QWordConst
ConstExprTerm                 := Zero | One | Ones


BufferTerm                    := Buffer(
                                      BuffSize              //Nothing |
                                                            //TermArg=>Integer
                                  ) {StringData | ByteList} => Buffer

ByteList                      := Nothing | <ByteConstExpr ByteListTail>
ByteListTail                  := Nothing | <‘,’ ByteConstExpr ByteListTail>

DWordList                     := Nothing | <DWordConstExpr DWordListTail>
DWordListTail                 := Nothing | <‘,’ DWordConstExpr DWordListTail>

PackageTerm                   := Package(
                                      NumElements           //Nothing |
                                                            //ByteConstExpr |
                                                            //TermArg=>Integer
                                  ) {PackageList} => Package

PackageList                   := Nothing | <PackageElement PackageListTail>
PackageListTail               := Nothing | <‘,’ PackageElement PackageListTail>
PackageElement                := DataObject | NameString

EISAIDTerm                    := EISAID(
                                      EISAIDString          //StringData
                                  ) => DWordConst

ResourceTemplateTerm          := ResourceTemplate() {ResourceMacroList} => Buffer

UnicodeTerm                   := Unicode(
                                      ASCIIString           //StringData
                                  ) => Buffer

ResourceMacroList             := Nothing | <ResourceMacroTerm ResourceMacroList>
```

```
ResourceMacroTerm             := DMATerm | DWordIOTerm | DWordMemoryTerm |
                                  EndDependentFnTerm | FixedIOTerm | InterruptTerm |
                                  IOTerm | IRQNoFlagsTerm | IRQTerm | Memory24Term |
                                  Memory32FixedTerm | Memory32Term | QWordIOTerm |
                                  QWordMemoryTerm | RegisterTerm | StartDependentFnTerm |
                                  StartDependentFnNoPriTerm | VendorLongTerm |
                                  VendorShortTerm | WordBusNumberTerm | WordIOTerm

DMATerm                       := DMA(
                                      DMAType,              //DMATypeKeyword (_TYP)
                                      BusMaster,            //BusMasterKeyword (_BM)
                                      XferType,             //XferTypeKeyword (_SIZ)
                                      ResourceTag           //Nothing | NameString
                                  ) {ByteList}              //List of channels (0-7)

DWordIOTerm                   := DWordIO(
                                      ResourceType,         //Nothing (ResourceConsumer)|
                                                            //  ResourceTypeKeyword
                                      MinType,              //Nothing (MinNotFixed) |
                                                            //  MinKeyword (_MIF)
                                      MaxType,              //Nothing (MaxNotFixed) |
                                                            //  MaxKeyword (_MAF)
                                      Decode,               //Nothing (PosDecode) |
                                                            //  DecodeKeyword (_DEC)
                                      RangeType,            //Nothing (EntireRange) |
                                                            //  RangeTypeKeyword (_RNG)
                                      AddressGranularity,   //DWordConstExpr (_GRA)
                                      MinAddress,           //DWordConstExpr (_MIN)
                                      MaxAddress,           //DWordConstExpr (_MAX)
                                      Translation,          //DWordConstExpr (_TRA)
                                      AddressLen,           //DWordConstExpr (_LEN)
                                      ResSourceIndex,       //Nothing | ByteConstExpr
                                      ResSource,            //Nothing | StringData
                                      ResourceTag           //Nothing | NameString
                                      Type                  //Nothing | TypeKeyword
                                      TranslationType       //Nothing |
                                                            //  TranslationKeyword
                                  )

DWordMemoryTerm               := DWordMemory(
                                      ResourceType,         //Nothing (ResourceConsumer)|
                                                            //  ResourceTypeKeyword
                                      Decode,               //Nothing (PosDecode) |
                                                            //  DecodeKeyword (_DEC)
                                      MinType,              //Nothing (MinNotFixed) |
                                                            //  MinKeyword (_MIF)
                                      MaxType,              //Nothing (MaxNotFixed) |
                                                            //  MaxKeyword (_MAF)
                                      MemType,              //Nothing (NonCacheable) |
                                                            //  MemTypeKeyword (_MEM)
                                      ReadWriteType,        //ReadWriteKeyword (_RW)
                                      AddressGranularity,   //DWordConstExpr (_GRA)
                                      MinAddress,           //DWordConstExpr (_MIN)
                                      MaxAddress,           //DWordConstExpr (_MAX)
                                      Translation,          //DWordConstExpr (_TRA)
                                      AddressLen,           //DWordConstExpr (_LEN)
                                      ResSourceIndex,       //Nothing | ByteConstExpr
                                      ResSource,            //Nothing | StringData
                                      ResourceTag           //Nothing | NameString
                                      AddressRange          //Nothing | AddressKeyword
                                                            //         (_MTP)
                                      Type                  //Nothing | TypeKeyword
                                                            //         (_TTP)
                                  )

EndDependentFnTerm            := EndDependentFn()
```

```
FixedIOTerm                   := FixedIO(
                                     AddressBase,           //WordConstExpr (_BAS)
                                     RangeLen,              //ByteConstExpr (_LEN)
                                     ResourceTag            //Nothing | NameString
                                  )

InterruptTerm                 := Interrupt(
                                     ResourceType,          //Nothing (ResourceConsumer)|
                                                            //ResourceTypeKeyword
                                     InterruptType,         //InterruptTypeKeyword
                                                            //  (_LL, _HE)
                                     InterruptLevel,        //InterruptLevelKeyword
                                                            //  (_LL, _HE)
                                     ShareType,             //Nothing (Exclusive)
                                                            //ShareTypeKeyword (_SHR)
                                     ResSourceIndex,        //Nothing | ByteConstExpr
                                     ResSource,             //Nothing | StringData
                                     ResourceTag            //Nothing | NameString
                                  ) {DWordList}             //list of interrupts (_INT)

IOTerm                        := IO(
                                     IODecode,              //IODecodeKeyword (_DEC)
                                     MinAddress,            //WordConstExpr (_MIN)
                                     MaxAddress,            //WordConstExpr (_MAX)
                                     Alignment,             //ByteConstExpr (_ALN)
                                     RangeLen,              //ByteConstExpr (_LEN)
                                     ResourceTag            //Nothing | NameString
                                  )

IRQNoFlagsTerm                := IRQNoFlags(
                                     ResourceTag            //Nothing | NameString
                                  ) {ByteList}              //list of interrupts (0-15)

IRQTerm                       := IRQ(
                                     InterruptType,         //InterruptTypeKeyword
                                                            //  (_LL, _HE)
                                     InterruptLevel,        //InterruptLevelKeyword
                                                            //  (_LL, _HE)
                                     ShareType,             //Nothing (Exclusive)
                                                            //ShareTypeKeyword (_SHR)
                                     ResourceTag            //Nothing | NameString
                                  ) {ByteList}              //list of interrupts (0-15)

Memory24Term                  := Memory24(
                                     ReadWriteType,         //ReadWriteKeyword (_RW)
                                     MinAddress[23:8],      //WordConstExpr (_MIN)
                                     MaxAddress[23:8],      //WordConstExpr (_MAX)
                                     Alignment,             //WordConstExpr (_ALN)
                                     RangeLen,              //WordConstExpr (_LEN)
                                     ResourceTag            //Nothing | NameString
                                  )

Memory32FixedTerm             := Memory32Fixed(
                                     ReadWriteType,         //ReadWriteKeyword (_RW)
                                     AddressBase,           //DWordConstExpr (_BAS)
                                     RangeLen,              //DWordConstExpr (_LEN)
                                     ResourceTag            //Nothing | NameString
                                  )

Memory32Term                  := Memory32(
                                     ReadWriteType,         //ReadWriteKeyword (_RW)
                                     MinAddress,            //DWordConstExpr (_MIN)
                                     MaxAddress,            //DWordConstExpr (_MAX)
                                     Alignment,             //DWordConstExpr (_ALN)
                                     RangeLen,              //DWordConstExpr (_LEN)
                                     ResourceTag            //Nothing | NameString
                                  )
```

```
QWordIOTerm                   := QWordIO(
                                     ResourceType,          //Nothing (ResourceConsumer)|
                                                            //  ResourceTypeKeyword
                                     MinType,               //Nothing (MinNotFixed) |
                                                            //  MinKeyword (_MIF)
                                     MaxType,               //Nothing (MaxNotFixed) |
                                                            //  MaxKeyword (_MAF)
                                     Decode,                //Nothing (PosDecode) |
                                                            //  DecodeKeyword (_DEC)
                                     RangeType,             //Nothing (EntireRange) |
                                                            //  RangeTypeKeyword (_RNG)
                                     AddressGranularity,    //QWordConstExpr (_GRA)
                                     MinAddress,            //QWordConstExpr (_MIN)
                                     MaxAddress,            //QWordConstExpr (_MAX)
                                     Translation,           //QWordConstExpr (_TRA)
                                     AddressLen,            //QWordConstExpr (_LEN)
                                     ResSourceIndex,        //Nothing | ByteConstExpr
                                     ResSource,             //Nothing | StringData
                                     ResourceTag            //Nothing | NameString
                                     Type                   //Nothing | TypeKeyword
                                     TranslationType        //Nothing |
                                                            //  TranslationKeyword)
                                  )

QWordMemoryTerm               := QWordMemory(
                                     ResourceType,          //Nothing (ResourceConsumer)|
                                                            //  ResourceTypeKeyword
                                     Decode,                //Nothing (PosDecode) |
                                                            //  DecodeKeyword (_DEC)
                                     MinType,               //Nothing (MinNotFixed) |
                                                            //  MinKeyword (_MIF)
                                     MaxType,               //Nothing (MaxNotFixed) |
                                                            //  MaxKeyword (_MAF)
                                     MemType,               //Nothing (NonCacheable) |
                                                            //  MemTypeKeyword (_MEM)
                                     ReadWriteType,         //ReadWriteKeyword (_RW)
                                     AddressGranularity,    //QWordConstExpr (_GRA)
                                     MinAddress,            //QWordConstExpr (_MIN)
                                     MaxAddress,            //QWordConstExpr (_MAX)
                                     Translation,           //QWordConstExpr (_TRA)
                                     AddressLen,            //QWordConstExpr (_LEN)
                                     ResSourceIndex,        //Nothing | ByteConstExpr
                                     ResSource,             //Nothing | StringData
                                     ResourceTag            //Nothing | NameString
                                     AddressRange           //Nothing | AddressKeyword
                                                            //         (_MTP)
                                     Type                   //Nothing | TypeKeyword
                                                            //         (_TTP)
                                  )

RegisterTerm                  := Register(
                                     AddressSpaceID,        //AddressSpaceKeyword (_ASI)
                                     RegisterBitWidth,      //ByteConstExpr (_RBW)
                                     RegisterOffset,        //ByteConstExpr (_RBO)
                                     RegisterAddress,       //QWordConstExpr (_ADR)
                                  )

StartDependentFnTerm          := StartDependentFn(
                                     CompatPriority,        //ByteConstExpr (0-2)
                                     PerfRobustPriority     //ByteConstExpr (0-2)
                                  ) {ResourceMacroList}

StartDependentFnNoPriTerm     := StartDependentFnNoPri() {ResourceMacroList}

VendorLongTerm                := VendorLong(
                                     ResourceTag            //Nothing | NameString
                                  ) {ByteList}

VendorShortTerm               := VendorShort(
                                     ResourceTag            //Nothing | NameString
                                  ) {ByteList}              //up to 7 bytes
```

```
WordBusNumberTerm             := WordBusNumber(
                                     ResourceType,         //Nothing (ResourceConsumer)|
                                                           //  ResourceTypeKeyword
                                     MinType,              //Nothing (MinNotFixed) |
                                                           //  MinKeyword (_MIF)
                                     MaxType,              //Nothing (MaxNotFixed) |
                                                           //  MaxKeyword (_MAF)
                                     Decode,               //Nothing (PosDecode) |
                                                           //  DecodeKeyword (_DEC)
                                     AddressGranularity,   //WordConstExpr (_GRA)
                                     MinAddress,           //WordConstExpr (_MIN)
                                     MaxAddress,           //WordConstExpr (_MAX)
                                     Translation,          //WordConstExpr (_TRA)
                                     AddressLen,           //WordConstExpr (_LEN)
                                     ResSourceIndex,       //Nothing | ByteConstExpr
                                     ResSource,            //Nothing | StringData
                                     ResourceTag           //Nothing | NameString
                                 )
```

```
WordIOTerm                    := WordIO(
                                     ResourceType,         //Nothing (ResourceConsumer)|
                                                           //  ResourceTypeKeyword
                                     MinType,              //Nothing (MinNotFixed) |
                                                           //  MinKeyword (_MIF)
                                     MaxType,              //Nothing (MaxNotFixed) |
                                                           //  MaxKeyword (_MAF)
                                     Decode,               //Nothing (PosDecode) |
                                                           //  DecodeKeyword (_DEC)
                                     RangeType,            //Nothing (EntireRange) |
                                                           //  RangeTypeKeyword (_RNG)
                                     AddressGranularity,   //WordConstExpr _GRA)
                                     MinAddress,           //WordConstExpr (_MIN)
                                     MaxAddress,           //WordConstExpr (_MAX)
                                     Translation,          //WordConstExpr (_TRA)
                                     AddressLen,           //WordConstExpr (_LEN)
                                     ResSourceIndex,       //Nothing | ByteConstExpr
                                     ResSource,            //Nothing | StringData
                                     ResourceTag           //Nothing | NameString
                                     Type                  //Nothing | TypeKeyword
                                     TranslationType       //Nothing |
                                                           //  TranslationKeyword
                                 )
```

16.2 Full ASL Reference

This reference section is for developers who are writing ASL code while developing definition blocks for platforms.

16.2.1 ASL Names

This section describes how to encode object names using ASL.

The following table lists the characters legal in any position in an ASL object name. ASL names are not case-sensitive and will be converted to upper case.

Table 16-2 Named Object Reference Encodings

Value	Description	“Title”
0x41-0x5A, 0x5F, 0x61-0x7A	Lead character of name (‘A’–‘Z’, ‘_’ , ‘a’–‘z’)	**LeadNameChar**
0x30-0x39, 0x41-0x5A, 0x5F, 0x61-0x7A	Non-lead (trailing) character of name (‘A’–‘Z’, ‘_’, ‘a’–‘z’, ’0–9’)	**NameChar**

The following table lists the name modifiers that can be prefixed to an ASL name.

Table 16-3 Definition Block Name Modifier Encodings

	Description	NamePrefix :=	Followed by …
5C	Namespace root (‘\’)	RootPrefix	Name
5E	Parent namespace (‘^’)	ParentPrefix	ParentPrefix or Name

16.2.1.1 _T_*x* Reserved Object Names

The ACPI specification reserves object names with the prefix _T_ for internal use by the ASL compiler. The ASL compiler may, for example, use these objects to store temporary values when implementing translating complicated control structures into AML. The ASL compiler must declare _T_*x* objects normally (using Name) and must not define them more than once within the same scope.

16.2.2 ASL Data Types

ASL provides a wide variety of data types and operators that manipulate data. It also provides mechanisms for both explicit and implicit conversion between the data types when used with ASL operators.

The table below describes each of the available data types.

Table 16-4 Summary of ASL Data Types

ASL Data Type	Description
[Uninitialized]	No assigned type or value. This is the type of all control method LocalX variables and unused ArgX variables at the beginning of method execution, as well as all uninitialized Package elements. Uninitialized objects must be initialized (via Store or CopyObject) before they may be used as source operands in ASL expressions.
Buffer	An array of bytes. Uninitialized elements are zero by default.
Buffer Field	Portion of a buffer created using CreateBitField, CreateByteField, CreateWordField, CreateQWordField, CreateField, or returned by the Index operator.
DDB Handle	Definition block handle returned by the Load operator
Debug Object	Debug output object. Formats an object and prints it to the system debug port. Has no effect if debugging is not active.
Device	Device or bus object
Event	Event synchronization object
Field Unit (within an Operation Region)	Portion of an address space, bit-aligned and of one-bit granularity. Created using Field, BankField, or IndexField.
Integer	An *n*-bit little-endian unsigned integer. In ACPI 1.0 this was at least 32-bits. In ACPI 2.0 this is at least 64.bits.
Integer Constant	Created by the ASL terms “Zero”, “One”, “Ones”, and “Revision”.
Method	Control Method (Executable AML function)
Mutex	Mutex synchronization object
Object Reference	Reference to an object created using the RefOf operator
Operation Region	Operation Region (A region within an Address Space)
Package	Collection of ASL objects with a fixed number of elements (up to 255).
Power Resource	Power Resource description object
Processor	Processor description object

ASL Data Type	Description
String	Null-terminated ASCII string with up to 200 characters.
Thermal Zone	Thermal Zone description object

Compatibility Note: The ability to store and manipulate object references is new in ACPI 2.0. In ACPI 1.0 references could not be stored in variables, passed as parameters or returned from functions.

16.2.2.1 Data Type Conversion Overview

ASL provides two mechanisms to convert objects from one data type to another data type at run-time (during execution of the AML interpreter). The first mechanism, ***Explicit Data Type Conversion,*** allows the use of explicit ASL operators to convert an object to a different data type. The second mechanism, ***Implicit Data Type Conversion***, is invoked by the AML interpreter when it is necessary to convert a data object to an expected data type before it is used or stored.

The following general rules apply to data type conversions:

- Input parameters are always subject to implicit data type conversion (also known as implicit source operand conversion) whenever the operand type does not match the expected input type.
- Output (target) parameters for all operators except the explicit data conversion operators are subject to implicit data type conversion (also known as implicit result object conversion) whenever the target is an existing named object or named field that is of a different type than the object to be stored.
- Output parameters for the explicit data conversion operators, as well as output parameters that refer to a method local or argument (LocalX or ArgX) are not subject to implicit type conversion.

Both of these mechanisms (explicit and implicit conversion) are described in detail in the sections that follow.

16.2.2.2 Explicit Data Type Conversions

The following ASL operators are provided to *explicitly* convert an object from one data type to another:

- FromBCD — Convert an Integer to a BCD Integer
- ToBCD — Convert a BCD Integer to a standard binary Integer.
- ToBuffer — Convert an Integer, String, or Buffer to an object of type Buffer
- ToDecimalString — Convert an Integer, String, or Buffer to an object of type String. The string contains the ASCII representation of the decimal value of the source operand.
- ToHexString — Convert an Integer, String, or Buffer to an object of type String. The string contains the ASCII representation of the hexadecimal value of the source operand.
- ToInteger — Convert an Integer, String, or Buffer to an object of type Integer.
- ToString — Copy directly and convert a Buffer to an object of type String.

The following ASL operators are provided to copy and transfer objects:

- CopyObject — Explicitly store a copy of the operand object to the target name. No implicit type conversion is performed. (This operator is used to avoid the implicit conversion inherent in the ASL Store operator.)
- Store — Store a copy of the operand object to the target name. Implicit conversion is performed if the target name is of a fixed data type (see below). However, Stores to method locals and arguments do not perform implicit conversion and are therefore the same as using CopyObject.

16.2.2.3 Implicit Data Type Conversions

Automatic or *Implicit* type conversions can take place at two different times during the execution of an ASL operator. First, it may be necessary to convert one or more of the source operands to the data type(s) expected by the ASL operator. Second, the result of the operation may require conversion before it is stored into the destination. (Many of the ASL operators can store their result optionally into an object specified by the last parameter. In these operators, if the destination is specified, the action is exactly as if a Store operator had been used to place the result in the destination.)

Such data conversions are performed by an AML interpreter during execution of AML code and are known collectively as ***Implicit Operand Conversions***. As described briefly above, there are two different types of implicit operand conversion:

1. Conversion of a source operand from a mismatched data type to the correct data type required by an ASL operator, called ***Implicit Source Conversion***. This conversion occurs when a source operand must be converted to the operand type expected by the operator. Any or all of the source operands may be converted in this manner before the execution of the ASL operator can proceed.
2. Conversion of the result of an operation to the existing type of a target operand before it is stored into the target operand, called ***Implicit Result Conversion***. This conversion occurs when the target is a fixed type such as a named object or a field. When storing to a method Local or Arg, no conversion is required because these data types are of variable type (the store simply overwrites any existing object and the existing type).

16.2.2.3.1 Implicit Source Operand Conversion

During the execution of an ASL operator, each source operand is processed by the AML interpreter as follows:

- If the operand is of the type expected by the operator, no conversion is necessary.
- If the operand type is incorrect, attempt to convert it to the proper type.
- For the Concatenate operator, the data type of the first operand dictates both the required type of the second operand and the type of the result object. (The second operator is converted, if necessary, to match the type of the first operand.)
- If conversion is impossible, abort the running control method and issue a fatal error.

An implicit source conversion will be attempted anytime a source operand contains a data type that is different that the type expected by the operator. For example:

```
Store (“5678”, Local1)
Add (0x1234, Local1, BUF1)
```

In the Add statement above, *Local1* contains a String object and must undergo conversion to an Integer object before the Add operation can proceed.

In some cases, the operator may take more than one type of operand (such as Integer and String). In this case, depending on the type of the operand, the highest priority conversion is applied. Table 16-4 describes the source operand conversions available. For example:

```
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)
```

The Concatenate operator can take an Integer, Buffer or String for its first two parameters and the type of the first parameter determines how the second parameter will be converted. In this example, the first parameter is of type Buffer Field (from the CreateDWordField operator). What should it be converted to: Integer, Buffer or String? According to Table 16-4, the highest priority conversion is to Integer. Therefore, both of the following objects will be converted to Integers:

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

And will then joined together and the resulting type and value will be:

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

16.2.2.3.2 Implicit Result Object Conversion

For all ASL operators that generate and store a result value (including the Store operator), the result object is processed and stored by the AML interpreter as follows:

- If the ASL operator is one of the *explicit* conversion operators (ToString, ToInteger, etc., and the CopyObject operator), no conversion is performed. (In other words, the result object is stored directly to the target and completely overwrites any existing object already stored at the target.)
- If the target is a method local or argument (LocalX or ArgX), no conversion is performed and the result is stored directly to the target.
- If the target is a fixed type such as a named object or field object, an attempt is made to convert the source to the existing target type before storing.
- If conversion is impossible, abort the running control method and issue a fatal error.

An implicit result conversion can occur anytime the result of an operator is stored into an object that is of a fixed type. For example:

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

Since BUF1 is a named object of fixed type *Buffer*, the Integer result of the Add operation must be converted to a Buffer before it is stored into BUF1.

16.2.2.4 Data Types and Type Conversions

The following table lists the available ASL data types and the available data type conversions (if any) for each. The entry for each data type is fully cross-referenced, showing both the types to which the object may be converted as well as all other types that may be converted to the data type.

The allowable conversions apply to both explicit and implicit conversions.

Table 16-4a Data Types and Type Conversions

ASL Data Type	**Can be implicitly or explicitly converted to these Data Types: (In priority order)**	**Can be implicitly or explicitly converted from these Data Types:**
[Uninitialized]	None. Causes a fatal error when used as a source operand in any ASL statement.	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	None. Causes a fatal error when used as a source operand in any ASL statement.	Integer, String, Buffer, Package, Field Unit, Buffer Field, DDB Handle
Device	None	None
Event	None	None
Field Unit (within an Operation Region)	Integer, Buffer, String, Debug Object	Integer, Buffer, String

ASL Data Type	**Can be implicitly or explicitly converted to these Data Types: (In priority order)**	**Can be implicitly or explicitly converted from these Data Types:**
Integer	Buffer, Buffer Field, DDB Handle, Field Unit, String, Debug Object	Buffer, String
Integer Constant	Integer, Debug Object	None. Also, storing any object to a constant is a no-op, not an error.
Method	None	None
Mutex	None	None
Object Reference	None	None
Operation Region	None	None
Package	Debug Object	None
String	Integer, Buffer, Debug Object	Integer, Buffer
Power Resource	None	None
Processor	None	None
Thermal Zone	None	None

16.2.2.5 Data Type Conversion Rules

The following table presents the detailed data conversion rules for each of the allowable data type conversions. These conversion rules are implemented by the AML Interpreter and apply to all conversion types — explicit conversions, implicit source conversions, and implicit result conversions.

Table 16-4b Object Conversion Rules

To convert from an object of this Data Type	**To an object of this Data Type**	**This action is performed by the AML Interpreter:**
Buffer	Buffer Field	The contents of the buffer are copied to the Buffer Field. If the buffer is smaller than the size of the buffer field, it is zero extended. If the buffer is larger than the size of the buffer field, the upper bits are truncated. Compatibility Note: This conversion is new in ACPI 2.0. The behavior in ACPI 1.0 was undefined.
	Debug Object	Each buffer byte is displayed as hexadecimal integer, delimited by spaces and/or commas.
	Field Unit	The entire contents of the buffer are copied to the Field Unit. If the buffer is larger (in bits) than the size of the Field Unit, it is broken into pieces and completely written to the Field Unit, lower chunks first. If the integer (or the last piece of the integer, if broken up) is smaller or equal in size to the Field Unit, then it is zero extended before being written.
	Integer	The contents of the buffer are copied to the Integer, starting with the least-significant bit and continuing until the buffer has been completely copied — up to the maximum number of bits in an Integer (64 in ACPI 2.0).
	String	The entire contents of the buffer are converted to a string of two-character hexadecimal numbers, each separated by a space. A fatal error is generated if greater than two hundred ASCII characters are created.
Buffer Field	[See Rule]	If the Buffer Field is smaller than or equal to the size of an Integer (in bits), it will be treated as an Integer. Otherwise, it will be treated as a Buffer. (See the conversion rules for the Integer and Buffer data types.)
	Debug Object	Each byte is displayed as hexadecimal integer , delimited by spaces and/or commas
DDB Handle	[See Rule]	The object is treated as an Integer (See conversion rules for the Integer data type.)
Field Unit	[See Rule]	If the Field Unit is smaller than or equal to the size of an Integer (in bits), it will be treated as an Integer. If the Field Unit is larger than the size of an Integer, it will be treated as a Buffer. The size of an Integer is indicated by the Definition Block table header's Revision field. A Revision field value less than 2 indicates that the size of an Integer is 32-bits. A value greater than or equal to 2 signifies that the size of an Integer is 64-bits. (See the conversion rules for the Integer and Buffer data types.)
	Debug Object	Each byte is displayed as hexadecimal integer , delimited by spaces and/or commas

To convert from an object of this Data Type	**To an object of this Data Type**	**This action is performed by the AML Interpreter:**
Integer	Buffer	The Integer overwrites the entire Buffer object. If the integer requires more bits than the size of the Buffer, then the integer is truncated before being copied to the Buffer. If the integer contains fewer bits than the size of the buffer, the Integer is zero-extended to fill the entire buffer
	Buffer Field	The Integer overwrites the entire Buffer Field. If the integer is smaller than the size of the buffer field, it is zero-extended. If the integer is larger than the size of the buffer field, the upper bits are truncated. Compatibility Note: This conversion is new in ACPI 2.0. The behavior in ACPI 1.0 was undefined.
	Debug Object	Displayed as a hexadecimal integer.
	Field Unit	The Integer overwrites the entire Field Unit. If the integer is smaller than the size of the buffer field, it is zero-extended. If the integer is larger than the size of the buffer field, the upper bits are truncated.
	String	Creates an ASCII hexadecimal string.
Package	Package	All existing contents (if any) of the target package are deleted, and the contents of the source package are copied into the target package. (In other words, overwrites the same as any other object.)
	Debug Object	Each element of the package is displayed based on its type.
String	Buffer	The string is treated as a Buffer, with each ASCII character copied to one Buffer byte. If the string is longer than the buffer, it is truncated. If the string is shorter than the buffer, the buffer size is reduced
	Buffer Field	The string is treated as a buffer. If this buffer is smaller than the size of the buffer field, it is zero extended. If the buffer is larger than the size of the buffer field, the upper bits are truncated. Compatibility Note: This conversion is new in ACPI 2.0. The behavior in ACPI 1.0 was undefined.
	Debug Object	Each byte displayed as an ASCII character
	Field Unit	Each character of the string is written, starting with the first, to the Field Unit. If the Field Unit is less than eight bits, then the upper bits of each character are lost. If the Field Unit is greater than eight bits, then the additional bits are zeroed.
	Integer	The ASCII string is interpreted as a hexadecimal constant. Starts with the first hexadecimal ASCII character (‘0’-‘9’, ‘A’-‘F’, ‘a’, ‘f’) and ends with the first non-hexadecimal character.

16.2.2.6 Rules for Storing and Copying Objects

The table below lists the actions performed when storing objects to different types of named targets. ASL provides the following types of “store” operations:

- The Store operator is used to explicitly store an object to a location, with implicit conversion support of the source object.

- Many of the ASL operators can store their result optionally into an object specified by the last parameter. In these operators, if the destination is specified, the action is exactly as if a Store operator had been used to place the result in the destination.
- The CopyObject operator is used to explicitly store a copy of an object to a location, with no implicit conversion support.

Table 16-4c Object Storing and Copying Rules

When Storing an object of any data type to this type of Target location	**This action is performed by the Store operator or any ASL operator with a Target operand:**	**This action is performed by the CopyObject operator:**
Method **ArgX** variable	The object is copied to the destination with no conversion applied, with one exception. If the ArgX contains an Object Reference, an automatic de-reference occurs and the object is copied to the target of the Object Reference instead of overwriting the contents of ArgX	
Method **LocalX** variable	The object is copied to the destination with no conversion applied. Even if LocalX contains an Object Reference, it is overwritten.	
Field Unit or Buffer Field	The object is copied to the destination after implicit result conversion is applied	Fields permanently retain their type and cannot be changed. Therefore, CopyObject can only be used to copy an object of type Integer or Buffer to fields.
Named data object	The object is copied to the destination after implicit result conversion is applied to match the existing type of the named location	The object and type are copied to the named location.

16.2.3 ASL Terms

This section describes all the ASL terms and provides sample ASL code that uses the terms.

The ASL terms are grouped into the following categories:

- Definition block terms
- Compiler directive terms
- Object terms
- Opcode terms
- User terms
- Data objects
- Miscellaneous objects

16.2.3.1 Definition Block Term

```
DefinitionBlockTerm           := DefinitionBlock(
                                      AMLFileName,           //String
                                      TableSignature,        //String
                                      ComplianceRevision,    //ByteConstExpr
                                      OEMID,                 //String
                                      TableID,               //String
                                      OEMRevision            //DWordConstExpr
                                  ) {TermList}
```

The **DefinitionBlock** term specifies the unit of data and/or AML code that the OS will load as part of the Differentiated Definition Block or as part of an additional Definition Block. This unit of data and/or AML code describes either the base system or some large extension (such as a docking station). The entire DefinitionBlock will be loaded and compiled by the OS as a single unit, and can be unloaded by the OS as a single unit.

16.2.3.2 Compiler Directive Terms

The compiler directives are:

- Include term
- External term

16.2.3.2.1 Include (Include Another ASL File)

```
IncludeTerm                   := Include(
                                      IncFilePathName        //String
                                  )
```

IncFilePathname is the full OS file system path to another file that contains ASL terms to be included in the current file of ASL terms.

16.2.3.2.2 External (Declare External Objects)

```
ExternalTerm                  := External(
                                      ObjName,               //NameString
                                      ObjType                //Nothing | ObjectTypeKeyword
                                  )
```

The External compiler directive is to let the assembler know that the object is declared external to this table so that the assembler will not complain about the undeclared object. During compiling, the assembler will create the external object at the specified place in the namespace (if a full path of the object is specified), or the object will be created at the current scope of the External term. *ObjType* is optional. If not specified, "**UnknownObj**" type is assumed.

16.2.3.3 Object Terms

Object terms include: Named Object terms and Namespace Modifiers.

16.2.3.3.1 Named Object Terms

The ASL terms that can be used to create named objects in a definition block are listed in the following table.

Table 16-5 Named Object Terms

ASL Statement	Description
BankField	Declares fields in a banked configuration object.
CreateBitField	Declare a bit field object of a buffer object.
CreateByteField	Declares a byte field object of a buffer object.
CreateDWordField	Declares a DWord field object of a buffer object.
CreateField	Declares a field object of any bit length of a buffer object.
CreateQWordField	Declares a QWord field object of a buffer object.
CreateWordField	Declares a Word field object of a buffer object.
DataTableRegion	Declares a Data Table Region.
Device	Declares a bus/device object.
Event	Declares an event synchronization object.
Field	Declares fields of an operation region object.
IndexField	Declares fields in an index/data configuration object.
Method	Declares a control method.
Mutex	Declares a mutex synchronization object.
OperationRegion	Declares an operational region.
PowerResource	Declares a power resource object.
Processor	Declares a processor package.
ThermalZone	Declares a thermal zone package.

16.2.3.3.1.1 BankField (Declare Bank/Data Field)

```
BankFieldTerm                 := BankField(
                                      RegionName,           //NameString=>OperationRegion
                                      BankName,             //NameString=>FieldUnit
                                      BankValue,            //TermArg=>Integer
                                      AccessType,           //AccessTypeKeyword
                                      LockRule,             //LockRuleKeyword
                                      UpdateRule            //UpdateRuleKeyword
                                  ) {FieldUnitList}
```

This statement creates data field objects. The contents of the created objects are obtained by a reference to a bank selection register.

This encoding is used to define named data field objects whose data values are fields within a larger object selected by a bank-selected register. Accessing the contents of a banked field data object will occur automatically through the proper bank setting, with synchronization occurring on the operation region that contains the *BankName* data variable, and on the Global Lock if specified by the *LockRule.*

The *AccessType*, *LockRule*, *UpdateRule*, and *FieldUnitList* are the same format as the Field operator.

The following is a block of ASL sample code using *BankField:*

- Creates a 4-bit bank-selected register in system I/O space.
- Creates overlapping fields in the same system I/O space that are selected via the bank register.

```
// define 256-byte operational region in SystemIO space
// and name it GIO0
OperationRegion (GIO0, SystemIO, 0x125, 0x100)

// create some field in GIO including a 4-bit bank select register
Field (GIO0, ByteAcc, NoLock, Preserve) {
    GLB1, 1,
    GLB2, 1,
    Offset(1),         // move to offset for byte 1
    BNK1, 4
}

// Create FET0 & FET1 in bank 0 at byte offset 0x30
BankField (GIO0, BNK1, 0, ByteAcc, NoLock, Preserve) {
    Offset (0x30),
    FET0, 1,
    FET1, 1
}

// Create BLVL & BAC in bank 1 at the same offset
BankField (GIO0, BNK1, 1, ByteAcc, NoLock, Preserve) {
    Offset (0x30),
    BLVL, 7,
    BAC,  1
}
```

16.2.3.3.1.2 CreateBitField

```
CreateBitFieldTerm            := CreateBitField(
                                      SourceBuffer,         //TermArg=>Buffer
                                      BitIndex,             //TermArg=>Integer
                                      BitFieldName          //NameString
                                  )
```

SourceBuffer is evaluated as a buffer. *BitIndex* is evaluated as an integer. A new buffer field object *BitFieldName* is created for the bit of *SourceBuffer* at the bit index of *BitIndex*. The bit-defined field within *SourceBuffer* must exist.

16.2.3.3.1.3 CreateByteField

```
CreateByteFieldTerm           := CreateByteField(
                                      SourceBuffer,         //TermArg=>Buffer
                                      ByteIndex,            //TermArg=>Integer
                                      ByteFieldName         //NameString
                                  )
```

SourceBuffer is evaluated as a buffer. *ByteIndex* is evaluated as an integer. A new buffer field object *ByteFieldName* is created for the byte of *SourceBuffer* at the byte index of *ByteIndex*. The byte-defined field within *SourceBuffer* must exist.

16.2.3.3.1.4 CreateDWordField

```
CreateDWordFieldTerm          := CreateDWordField(
                                      SourceBuffer,         //TermArg=>Buffer
                                      ByteIndex,            //TermArg=>Integer
                                      DWordFieldName        //NameString
                                  )
```

SourceBuffer is evaluated as a buffer. *ByteIndex* is evaluated as an integer. A new buffer field object *DWordFieldName* is created for the DWord of *SourceBuffer* at the byte index of *ByteIndex*. The DWord-defined field within *SourceBuffer* must exist.

16.2.3.3.1.5 CreateField (Field)

```
CreateFieldTerm              := CreateField(
                                     SourceBuffer,          //TermArg=>Buffer
                                     BitIndex,              //TermArg=>Integer
                                     NumBits,               //TermArg=>Integer
                                     FieldName              //NameString
                                 )
```

SourceBuffer is evaluated as a buffer. *BitIndex* and *NumBits* are evaluated as integers. A new buffer field object *FieldName* is created for the bits of *SourceBuffer* at *BitIndex* for *NumBits.* The entire bit range of the defined field within *SourceBuffer* must exist.

16.2.3.3.1.6 CreateQWordField

```
CreateQWordFieldTerm         := CreateQWordField(
                                     SourceBuffer,          //TermArg=>Buffer
                                     ByteIndex,             //TermArg=>Integer
                                     QWordFieldName         //NameString
                                 )
```

SourceBuffer is evaluated as a buffer. *ByteIndex* is evaluated as an integer. A new buffer field object *QWordFieldName* is created for the QWord of *SourceBuffer* at the byte index of *ByteIndex.* The QWord-defined field within *SourceBuffer* must exist.

16.2.3.3.1.7 CreateWordField

```
CreateWordFieldTerm          := CreateWordField(
                                     SourceBuffer,          //TermArg=>Buffer
                                     ByteIndex,             //TermArg=>Integer
                                     WordFieldName          //NameString
                                 )
```

SourceBuffer is evaluated as a buffer. *ByteIndex* is evaluated as an integer. A new bufferfield object *WordFieldName* is created for the word of *SourceBuffer* at the byte index of *ByteIndex.* The word-defined field within *SourceBuffer* must exist.

16.2.3.3.1.8 DataTableRegion

```
DataRegionTerm               := DataTableRegion(
                                 RegionName,                //NameString
                                     SignatureString,       //TermArg=>String
                                 OemIDString,               //TermArg=>String
                                 OemTableIDString           //TermArg=>String
                             )
```

A Data Table Region is a special Operation Region. Its region space is always memory. The memory referred to by the Data Table Region is the memory that is occupied by the table referenced in XSDT that is identified by SignatureString, OemIDString and OemTableIDString. Any Field object can reference RegionName

The base address of a Data Table region is the address of the first byte of the header of the table identified by SignatureString, OemIDString and OemTableIDString. The length of the region is the length of the table.

Any table referenced by a Data Table Region must be in memory marked by AddressRangeReserved or AddressRangeNVS.

16.2.3.3.1.9 Device (Declare Bus/Device Package)

```
DeviceTerm                    := Device(
                                     DeviceName            //NameString
                                 ) {ObjectList}
```

Creates a Device object, which represents either a bus or a device or any other such entity of use. **Device** opens a name scope.

A Bus/Device Package is one of the basic ways the Differentiated Definition Block describes the hardware devices in the system to the operating software. Each Bus/Device Package is defined somewhere in the hierarchical namespace corresponding to that device's location in the system. Within the namespace of the device are other names that provide information and control of the device, along with any sub-devices that in turn describe sub-devices, and so on.

For any device, the BIOS provides only information that is added to the device in a non-hardware standard manner. This type of value-added function is expressible in the ACPI Definition Block such that operating software can use the function.

The BIOS supplies Device Objects only for devices that are obtaining some system-added function outside the device's normal capabilities and for any Device Object required to fill in the tree for such a device. For example, if the system includes a PCI device (integrated or otherwise) with no additional functions such as power management, the BIOS would not report such a device; however, if the system included an integrated ISA device below the integrated PCI device (device is an ISA bridge), then the system would include a Device Package for the ISA device with the minimum feature being added being the ISA device's ID and configuration information and the parent PCI device, because it is required to get the ISA Device Package placement in the namespace correct.

The following block of ASL sample code shows a nested use of Device objects to describe an IDE controller connected to the root PCI bus.

```
Device (IDE0) {       // primary controller
    Name(_ADR, 0)     // put PCI Address (device/function) here

    // define region for IDE mode register
    OperationRegion (PCIC, PCI_Config, 0x50, 0x10)
    Field (PCIC, AnyAcc, NoLock, Preserve) {
        …
    }
    Device(PRIM) {            //Primary adapter
        Name(_ADR, 0)         //Primary adapter = 0
        …
        Method(_STM, 2){
            …
        }
        Method(_GTM){
            …
        }
        Device(MSTR) {        // master channel
            Name(_ADR, 0)
            Name(_PR0, Package(){0, PIDE})

            Name(_GTF){
                …
            }
        }

        Device(SLAV) {
            Name(_ADR, 1)
            Name(_PR0, Package(){0, PIDE})
            Name(_GTF){
                …
            }
        }
    }
}
```

16.2.3.3.1.10 Event (Declare Event Synchronization Object)

```
EventTerm                      := Event(
                                      EventName              //NameString
                                  )
```

Creates an event synchronization object named *EventName*.

For more information about the uses of an event synchronization object, see the ASL definitions for the Wait, Signal, and Reset function operators.

16.2.3.3.1.11 Field (Declare Field Objects)

```
FieldTerm                      := Field(
                                      RegionName,            //NameString=>OperationRegion
                                      AccessType,            //AccessTypeKeyword
                                      LockRule,              //LockRuleKeyword
                                      UpdateRule             //UpdateRuleKeyword
                                  ) {FieldUnitList}
```

Declares a series of named data objects whose data values are fields within a larger object. The fields are parts of the object named by *RegionName*, but their names appear in the same scope as the **Field** term.

For example, the field operator allows a larger operation region that represents a hardware register to be broken down into individual bit fields that can then be accessed by the bit field names. Extracting and combining the component field from its parent is done automatically when the field is accessed.

Accessing the contents of a field data object provides access to the corresponding field within the parent object. If the parent object supports Mutex synchronization, accesses to modify the component data objects will acquire and release ownership of the parent object around the modification.

In general, accesses within the parent object are performed naturally aligned. If desired, *AccessType* set to a value other than **AnyAcc** can be used to force minimum access width. Notice that the parent object must be able to accommodate the *AccessType* width. For example, an access type of **WordAcc** cannot read the last byte of an odd-length operation region. The exceptions to natural alignment are the access types used for a non-linear SMBus device. These will be discussed in detail below. Not all access types are meaningful for every type of operational region.

The following table relates region types declared with an OperationRegion term to the different access types supported for each region.

Table 16-6 OperationRegion Region Types and Access Types

Region Types	Access Type	Description
SystemMemory	**ByteAcc**	
	WordAcc	
	DWordAcc	
	QWordAcc	
	AnyAcc	Read/Write Byte, Word, DWord, QWord access
SystemIO	**ByteAcc**	
	WordAcc	
	DWordAcc	
	QWordAcc	
	AnyAcc	Read/Write Byte, Word, DWord, QWord access
PCI_Config	**ByteAcc**	
	WordAcc	
	DWordAcc	
	QWordAcc	
	AnyAcc	Read/Write Byte, Word, DWord, QWord access
EmbeddedControl	**ByteAcc**	
SMBus	**BufferAcc**	Reads and writes to this operation region involve the use of a region specific data buffer. See section 14, “ACPI System Management Bus Interface Specification,” for more information.
CMOS	**ByteAcc**	
PciBarTarget	**ByteAcc**	
	WordAcc	
	DWordAcc	
	QWordAcc	
	AnyAcc	Read/Write Byte, Word, DWord, QWord access

If *LockRule* is set to **Lock**, accesses to modify the component data objects will acquire and release the Global Lock. If both types of locking occur, the Global Lock is acquired after the parent object Mutex.

UpdateRule is used to specify how the unmodified bits of a field are treated. For example, if a field defines a component data object of 4 bits in the middle of a **WordAcc** region, when those 4 bits are modified the UpdateRule specifies how the other 12 bits are treated.

The named data objects are provided in FieldList as a series of names and bit widths. Bits assigned no name (or NULL) are skipped. The ASL compiler supports an **Offset(**ByteOffset**)** macro within a FieldList to skip to the bit position of the supplied byte offset.

SMBus regions are inherently non-linear, where each offset within an SMBus address space represents a variable sized (0 to 32 bytes) field. Given this uniqueness, SMBus operation regions include restrictions on their field definitions and require the use of an SMBus-specific data buffer when initiating transactions. See section 14, “ACPI System Management Bus Interface Specification,” for more information.

16.2.3.3.1.11.1 CMOS Protocols

This section describes how CMOS can be accessed from ASL. Most computers contain an RTC/CMOS device that can be represented as a linear array of bytes of non-volatile memory. There is a standard mechanism for accessing the first 64 bytes of non-volatile RAM in devices that are compatible with the Motorola RTC/CMOS device that was in the IBM PC/AT. But today’s RTC/CMOS devices usually contain more than 64 bytes of non-volatile RAM, and there is no standard for access to these extensions. To solve this problem, new PnP IDs are presented here for each type of extension.

All bytes of CMOS that are related to the current time, day, date, month, year and century are read-only.

16.2.3.3.1.11.1.1 PC/AT-compatible RTC/CMOS Devices (PNP0B00)

The standard PC/AT-compatible RTC/CMOS device is denoted by the PnP ID PNP0B00. If an ACPI platform uses a device that is compatible with this device, it may describe this in its ACPI namespace. ASL may then read and write this as a linear 64-byte array. If PNP0B00 is used, ASL and ACPI operating systems may not assume that any extensions to the CMOS exist.

Note: This means that the CENTURY field in the Fixed ACPI Description Table may only contain values between 0 and 63.

This is an example of how this device could be described:

```
Device (RTC0) {
    Name(_HID, EISAID("PNP0B00"))

    Name(_CRS, ResourceTemplate() {
        IO(Decode16, 0x70, 0x70, 0x1, 0x2)
    }

    OperationRegion(CMS1, CMOS, 0, 0x40)

    Field(CMS1, ByteAcc, NoLock, Preserve) {
        AccessAs(ByteAcc, 0),
        CM00,   8,
        ,256,
        CM01,   8,
        CM02,   16,
        , 216,
        CM03,   8
    }
```

16.2.3.3.1.11.1.2 Intel PIIX4-compatible RTC/CMOS Devices (PNP0B01)

The Intel PIIX4 contains an RTC/CMOS device that is compatible with the one in the PC/AT. But it contains 256 bytes of non-volatile RAM. The first 64 bytes are accessed via the same mechanism as the 64 bytes in the PC/AT. The upper 192 bytes are accessed through an interface that is only used on Intel chips. (See 82371AB PCI-TO-ISA / IDEXCELERATOR (PIIX4) for details.)

Any platform containing this device or one that is compatible with it may use the PNP ID PNP0B01. This will allow an ACPI-compatible OS to recognize the RTC/CMOS device as using the programming interface of the PIIX4. Thus, the array of bytes that ASL can read and write with this device is 256 bytes long.

Note: This also means that the CENTURY field in the Fixed ACPI Description Table may contain values between 0 and 255.

This is an example of how this device could be described:

```
Device (RTC0) {
    Name(_HID, EISAID("PNP0B01"))

    Name(_CRS, ResourceTemplate() {
        IO(Decode16, 0x70, 0x70, 0x1, 0x2)
        IO(Decode16, 0x72, 0x72, 0x1, 0x2)
    }

    OperationRegion(CMS1, CMOS, 0, 0x100)

    Field(CMS1, ByteAcc, NoLock, Preserve) {
        AccessAs(ByteAcc, 0),
        CM00,   8,
        ,256,
        CM01,   8,
        CM02,   16,
        , 224,
        CM03,   8,
        , 184,
        CENT,   8
    }
```

16.2.3.3.1.11.1.3 Dallas Semiconductor-compatible RTC/CMOS Devices (PNP0B02)

Dallas Semiconductor RTC/CMOS devices are compatible with the one in the PC/AT, but they contain 256 bytes of non-volatile RAM or more. The first 64 bytes are accessed via the same mechanism as the 64 bytes in the PC/AT. The upper bytes are accessed through an interface that is only used on Dallas Semiconductor chips.

Any platform containing this device or one that is compatible with it may use the PNP ID PNP0B02. This will allow an ACPI-compatible OS to recognize the RTC/CMOS device as using the Dallas Semiconductor programming interface. Thus, the array of bytes that ASL can read and write with this device is 256 bytes long.

Description of these devices is similar to the PIIX4 example above, and the CENTURY field of the FADT may also contain values between 0 and 255.

16.2.3.3.1.11.2 PCI Device BAR Target Protocols

This section describes how PCI devices' control registers can be accessed from ASL. PCI devices each have an address space associated with them called the Configuration Space. At offset 0x10 through offset 0x27, there are as many as six Base Address Registers, (BARs). These BARs contain the base address of a series of control registers (in I/O or Memory space) for the PCI device. Since a Plug and Play OS may change the values of these BARs at any time, ASL cannot read and write from these deterministically using I/O or Memory operation regions. Furthermore, a Plug and Play Play OS will automatically assign ownership of the I/O and Memory regions associated with these BARs to a device driver associated with the PCI device. An ACPI OS (which must also be a Plug and Play operating system) will not allow ASL to read and write regions that are owned by native device drivers.

If a platform uses a PCI BAR Target operation region, an ACPI OS will not load a native device driver for the associated PCI function. For example, if any of the BARs in a PCI function are associated with a PCI BAR Target operation region, then the OS will assume that the PCI function is to be entirely under the control of the ACPI BIOS. No driver will be loaded. Thus, a PCI function can be used as a platform controller for some task (hot-plug PCI, and so on) that the ACPI BIOS performs.

16.2.3.3.1.11.2.1 Declaring a PCI BAR Target Operation Region

PCI BARs contain the base address of an I/O or Memory region that a PCI device's control registers lie within. Each BAR implements a protocol for determining whether those control registers are within I/O or Memory space and how much address space the PCI device decodes. (See the PCI Specification for more details.)

PCI BAR Target operation regions are declared by providing the offset of the BAR within the PCI device's PCI configuration space. The BAR determines whether the actual access to the device occurs through an I/O or Memory cycle, not by the declaration of the operation region. The length of the region is similarly implied.

In the term `OperationRegion(PBAR, PciBarTarget, 0x10, 0x4)`, the offset is the offset of the BAR within the configuration space of the device. This would be an example of an operation region that uses the first BAR in the device.

16.2.3.3.1.11.2.2 PCI Header Types and PCI BAR Target Operation Regions

PCI BAR Target operation regions may only be declared in the scope of PCI devices that have a PCI Header Type of 0. PCI devices with other header types are bridges. The control of PCI bridges is beyond the scope of ASL.

16.2.3.3.1.12 IndexField (Declare Index/Data Fields)

```
IndexFieldTerm              := IndexField(
                                    IndexName,            //NameString=>FieldUnit
                                    DataName,             //NameString=>FieldUnit
                                    AccessType,           //AccessTypeKeyword
                                    LockRule,             //LockRuleKeyword
                                    UpdateRule            //UpdateRuleKeyword
                                ) {FieldUnitList}
```

Creates a series of named data objects whose data values are fields within a larger object accessed by an index/data-style reference to *IndexName* and *DataName*.

This encoding is used to define named data objects whose data values are fields within an index/data register pair. This provides a simple way to declare register variables that occur behind a typical index and data register pair.

Accessing the contents of an indexed field data object will automatically occur through the *DataName* object by using an *IndexName* object aligned on an *AccessType* boundary, with synchronization occurring on the operation region that contains the index data variable, and on the Global Lock if specified by *LockRule*.

AccessType, *LockRule*, *UpdateRule*, and *FieldList* are the same format as the **Field** term.

The following is a block of ASL sample code using **IndexField***:*

Creates an index/data register in system I/O space made up of 8-bit registers.

- Creates a FET0 field within the indexed range.

```
Method(EX1){
    // define 256-byte operational region in SystemIO space
    // and name it GIO0
    OperationRegion (GIO0, 1, 0x125, 0x100)
    // create field named Preserve structured as a sequence
    // of index and data bytes
    Field (GIO0, ByteAcc, NoLock, WriteAsZeros) {
        IDX0, 8,
        DAT0, 8,
            .
            .
            .
        }
    // Create an IndexField within IDX0 & DAT0 which has
    // FETs in the first two bits of indexed offset 0,
    // and another 2 FETs in the high bit on indexed
    // 2f and the low bit of indexed offset 30
    IndexField (IDX0, DAT0, ByteAcc, NoLock, Preserve) {
        FET0, 1,
        FET1, 1,
        Offset(0x2f),      // skip to byte offset 2f
        , 7,               // skip another 7 bits
        FET3, 1,
        FET4, 1
    }
    // Clear FET3 (index 2f, bit 7)
    Store (Zero, FET3)
} // End EX1
```

16.2.3.3.1.13 Method (Declare Control Method)

```
MethodTerm                    := Method(
                                      MethodName,           //NameString
                                      NumArgs,              //Nothing | ByteConstExpr
                                      SerializeRule,        //Nothing |
                                                            //  SerializeRuleKeyword
                                      SyncLevel             //Nothing |
                                                            //  ByteConstExpr
                                  ) {TermList}
```

Declares a named package containing a series of object references that collectively represent a control method, which is a procedure that can be invoked to perform computation. **Method** opens a name scope.

System software executes a control method by referencing the objects in the package in order. For more information on control method execution, see section 5.5.3, “Control Method Execution.”

The current namespace location used during name creation is adjusted to be the current location on the namespace tree. Any names created within this scope are “below” the name of this package. The current namespace location is assigned to the method package, and all namespace references that occur during control method execution for this package are relative to that location.

If a method is declared as **Serialized**, an implicit mutex associated with the method object is acquired at the specified *SyncLevel*. If no *SyncLevel* is specified, *SyncLevel 0* is assumed. The serialize rule can be used to prevent reentering of a method. This is especially useful if the method creates namespace objects. Without the serialize rule, the reentering of a method will fail when it attempts to create the same namespace object.

Also notice that all namespace objects created by a method have temporary lifetime. When method execution exits, the created objects will be destroyed.

The following block of ASL sample code shows a use of **Method** for defining a control method that turns on a power resource.

```
Method(_ON) {
    Store (One, GIO.IDEP)         // assert power
    Sleep (10)                    // wait 10ms
    Store (One, GIO.IDER)         // de-assert reset#
    Stall (10)                    // wait 10us
    Store (Zero, GIO.IDEI)        // de-assert isolation
}
```

16.2.3.3.1.14 Mutex (Declare Synchronization/Mutex Object)

```
MutexTerm                     := Mutex(
                                      MutexName,           //NameString
                                      SyncLevel            //ByteConstExpr
                                  )
```

Creates a data mutex synchronization object named *MutexName*, with level from 0 to 15 specified by *SyncLevel.*

A synchronization object provides a control method with a mechanism for waiting for certain events. To prevent deadlocks, wherever more than one synchronization object must be owned, the synchronization objects must always be released in the order opposite the order in which they were acquired. The *SyncLevel* parameter declares the logical nesting level of the synchronization object. All **Acquire** terms must refer to a synchronization object with an equal or greater *SyncLevel* to current level, and all **Release** terms must refer to a synchronization object with equal or lower *SyncLevel* to the current level.

Mutex synchronization provides the means for mutually exclusive ownership. Ownership is acquired using an **Acquire** term and is released using a **Release** term. Ownership of a Mutex must be relinquished before completion of any invocation. For example, the top-level control method cannot exit while still holding ownership of a Mutex. Acquiring ownership of a Mutex can be nested.

The SyncLevel of a thread before acquiring any mutexes is zero. The SyncLevel of the Global Lock (_GL) is zero.

16.2.3.3.1.15 OperationRegion (Declare Operation Region)

```
OpRegionTerm                  := OperationRegion(
                                      RegionName,          //NameString
                                      RegionSpace,         //RegionSpaceKeyword
                                      Offset,              //TermArg=>Integer
                                      Length               //TermArg=>Integer
                                  )
```

Declares an operation region. *Offset* is the offset within the selected *RegionSpace* at which the region starts (byte-granular), and *Length* is the length of the region in bytes.

An Operation Region is a type of data object where read or write operations to the data object are performed in some hardware space. For example, the Definition Block can define an Operation Region within a bus, or system I/O space. Any reads or writes to the named object will result in accesses to the I/O space.

Operation regions are regions in some space that contain hardware registers for *exclusive* use by ACPI control methods. In general, no hardware register (at least byte-granular) within the operation region accessed by an ACPI control method can be shared with any accesses from any other source, with the exception of using the Global Lock to share a region with the firmware. The entire Operation Region can be allocated for exclusive use to the ACPI subsystem in the host OS.

Operation Regions that are defined within the scope of a method are the exception to this rule. These Operation Regions are known as “Dynamic” since the OS has no idea that they exist or what registers they use until the control method is executed. Using a Dynamic SystemIO or SystemMemory Operation Region is not recommended since the OS cannot *guarantee* exclusive access. All other types of Operation Regions may be Dynamic.

Operation Regions have “virtual content” and are only accessible via **Field** objects Operation Region objects may be defined down to actual bit controls using **Field** data object definitions. The actual bit content of a **Field** is comprised of bits from within a larger **Buffer** that are normalized for that field (in other words, shifted down and masked to the proper length), and as such the data type of a **Field** is **Buffer**. Therefore fields that are 32 bits or less in size may be read and stored as Integers.

An Operation Region object implicitly supports Mutex synchronization. Updates to the object, or a **Field** data object for the region**,** will automatically synchronize on the Operation Region object; however, a control method may also explicitly synchronize to a region to prevent other accesses to the region (from other control methods). Notice that, according to the control method execution model, control method execution is non-preemptive. Because of this, explicit synchronization to an Operation Region needs to be done only in cases where a control method blocks or yields execution and where the type of register usage requires such synchronization.

There are seven predefined Operation Region types specified in ACPI:

0 – SystemMemory

1 – SystemIO

2 – PCI_Config

3 – EmbeddedControl

4 – SMBus

5 – CMOS

6 – PCIBARTarget

In addition, OEMs may define Operation Regions types 0x80 to 0xFF.

The following example ASL code shows the use of **OperationRegion** combined with **Field** to describe IDE 0 and 1 controlled through general I/O space, using one FET.

```
OperationRegion (GIO, SystemIO, 0x125, 0x1)
Field (GIO, ByteAcc, NoLock, Preserve) {
    IDEI,   1,      // IDEISO_EN    - isolation buffer
    IDEP,   1,      // IDE_PWR_EN   - power
    IDER,   1       // IDERST#_EN   - reset#
}
```

16.2.3.3.1.16 PowerResource (Declare Power Resource)

```
PowerResTerm                  := PowerResource(
                                      ResourceName,          //NameString
                                      SystemLevel,           //ByteConstExpr
                                      ResourceOrder          //WordConstExpr
                                 ) {ObjectList}
```

Declares a power resource. **PowerResource** opens a name scope.

For a definition of the **PowerResource** term, see section 7.1, “Declaring a Power Resource Object.”

16.2.3.3.1.17 Processor (Declare Processor)

```
ProcessorTerm                  := Processor(
                                       ProcessorName,         //NameString
                                       ProcessorID,           //ByteConstExpr
                                       PBlockAddress,         //DWordConstExpr
                                       PblockLength           //ByteConstExpr
                                   ) {ObjectList}
```

Declares a named processor object. **Processor** opens a name scope. Each processor is required to have a unique *ProcessorID* value that is unique from any other *ProcessorID* value.

For each processor in the system, the ACPI BIOS declares one processor object in the namespace anywhere within the _SB scope. For compatibility with operating systems implementing ACPI 1.0, the processor object may also be declared under the _PR scope. An ACPI 2.0-compatible namespace may define Processor objects in either the _SB or _PR scope but not both.

PBlockAddress provides the system I/O address for the processors register block. Each processor can supply a different such address. *PBlockLength* is the length of the processor register block, in bytes and is either 0 (for no P_BLK) or 6. With one exception, all processors are required to have the same *PBlockLength.* The exception is that the boot processor can have a non-zero *PBlockLength* when all other processors have a zero *PBlockLength.*

The following block of ASL sample code shows a use of the **Processor** term.

```
Processor(
    \_PR.CPU0,      // namespace name
    1,
    0x120,          // PBlk system IO address
    6               // PBlkLen
    )
    {ObjectList}
```

The ObjectList is an optional list that may contain an arbitrary number of ASL Objects. Processor-specific objects that may be included in the ObjectList include _PTC, _CST, _PCT, _PSS, and _PPC. These processor-specific objects can only be specified when the processor object is declared within the _SB scope. For a full definition of these objects, see section 8, “Processor Control.”

16.2.3.3.1.18 ThermalZone (Declare Thermal Zone)

```
ThermalZoneTerm                := ThermalZone(
                                       ThermalZoneName        //NameString
                                   ) {ObjectList}
```

Declares a named Thermal Zone object. **ThermalZone** opens a name scope.

Each use of a **ThermalZone** term declares one thermal zone in the system. Each thermal zone in a system is required to have a unique *ThermalZoneName*.

A thermal zone may be declared in the namespace anywhere within the _SB scope. For compatibility with operating systems implementing ACPI 1.0, a thermal zone may also be declared under the _TZ scope. An ACPI 2.0-compatible namespace may define Thermal Zone objects in either the _SB or _TZ scope but not both.

For sample ASL code that uses a ThermalZone statement, see section 12, “Thermal Management.”

16.2.3.3.2 Namespace Modifiers

The namespace modifiers are as follows:

Table 16-7 Namespace Modifiers

ASL Statement	Description
Alias	Defines a name alias.
Name	Defines a global name and attaches a buffer, literal data item, or package to it.
Scope	Declares the placement of one or more object names in the ACPI namespace when the definition block that contains the Scope statement is loaded.

16.2.3.3.2.1 Alias (Declare Name Alias)

```
AliasTerm                     := Alias(
                                      SourceObject,          //NameString
                                      AliasObject            //NameString
                                  )
```

Creates a new name, *AliasObject*, which refers to and acts exactly the same as *SourceObject.*

AliasObject is created as an alias of *SourceObject* in the namespace. The *SourceObject* name must already exist in the namespace. If the alias is to a name within the same definition block, the *SourceObject* name must be logically ahead of this definition in the block. The following example shows use of an **Alias** term:

```
       Alias(\SUS.SET.EVEN, SSE)
```

16.2.3.3.2.2 Name (Declare Named Object)

```
NameTerm                      := Name(
                                      ObjectName,            //NameString
                                      Object                 //DataObject
                                  )
```

Attaches *Object* to *ObjectName* in the Global ACPI namespace.

This encoding is to create *ObjectName* in the namespace, which references the *Object*.

The following example creates the name PTTX in the root of the namespace that references a package.

```
    Name(\PTTX,               // Port to Port Translate Table
        Package() { Package() { 0x43, 0x59 }, Package() { 0x90, 0xff }}
    )
```

The following example creates the name CNT in the root of the namespace that references an integer data object with the value 5.

```
    Name(\CNT, 5)
```

16.2.3.3.2.3 Scope (Open Named Scope)

```
ScopeTerm                     := Scope(
                                     Location              //NameString
                                 ) {ObjectList}
```

Opens and assigns a base namespace scope to a collection of objects. All object names defined within the scope act relative to *Location*. Notice that *Location* does not have to be below the surrounding scope, but can refer to any location within the namespace. The **Scope** term itself does not create objects, but only locates objects in the namespace; the located objects are created by other ASL terms.

The object referred to by *Location* must already exist in the namespace and be one of the following object types that has a namespace scope associated with it:

Predefined scope such as: \ (root), _SB, \GPE, _PR, _TZ, _SI, etc.
Device
Processor
Thermal Zone
Power Resource

The **Scope** term alters the current namespace location to the existing *Location*. This causes the defined objects within *ObjectList* to occur relative to this new location in the namespace.

The following example ASL code places the defined objects in the ACPI namespace as shown:

```
Scope(\PCI0) {
    Name(X, 3)
    Scope(\) {
        Method(RQ) { Return(0) }
    }
    Name(^Y, 4)
}
```

places the defined objects in the ACPI namespace as shown:

```
\PCI0.X
\RQ
\Y
```

16.2.3.4 Opcode Terms

There are two types of ASL opcode terms: Type 1 opcodes and Type 2 opcodes.

A Type1 opcode term can only be used standing alone on a line of ASL code; because these types of terms do not return a value, they cannot be used as a term in an expression.

A Type2 opcode term can be used in an expression because these types of terms return a value. When used in an expression, the argument that names the object in which to store the result is optional.

Notice that in the opcode definitions below, when the definition says “result is stored in” this literally means that the **Store** operator is assumed and the “execution result” is the *Source* operand to the **Store** opcode. If the optional argument in which the result is to be stored is an operation region field with an *AccessType* of **BufferAcc**, the execution result of the entire opcode (the execution result of the implied **Store**) may be different than the data written to the *Destination* of the implied **Store**, just as it would be in an explicit Store operation. See section 16.2.3.4.2.43 “Store (Store)”.

16.2.3.4.1 Type 1 Opcodes

```
Type1Opcode                   := BreakTerm | BreakPointTerm | ContinueTerm | FatalTerm |
                                 IfElseTerm | LoadTerm | NoOpTerm | NotifyTerm |
                                 ReleaseTerm | ResetTerm | ReturnTerm | SignalTerm |
                                 SleepTerm | StallTerm | SwitchTerm | UnloadTerm |
                                 WhileTerm
```

The Type 1 opcodes are listed in the following table.

Table 16-8 Type 1 Opcodes

ASL Statement	Description
Break	Continue immediately following the innermost enclosing **While** scope
BreakPoint	Used for debugging. Stops execution in the debugger.
Continue	Continue innermost enclosing While loop where condition is evaluated
Else	Else
ElseIf	ElseIf
Fatal	Fatal check
If	If
Load	Load differentiating definition block
Noop	No operation
Notify	Notify the OS that the specified notification event occurred for the specified object.
Release	Release a synchronization object
Reset	Reset a synchronization object
Return	Return from a control method, optionally setting a return value
Signal	Signal a synchronization object
Sleep	Sleep *n* milliseconds (yields the processor)
Stall	Delay *n* microseconds (does not yield the processor)
Switch	Select code to execute based on expression value
Unload	Unload definition block
While	While

16.2.3.4.1.1 Break (Break)

```
BreakTerm                    := Break
```

Break causes execution to continue immediately following the innermost enclosing **While** scope, in the current Method. If there is no enclosing **While** within the current Method, a fatal error is generated.

Compatibility Note: In ACPI 1.0, the Break command continued immediately following the innermost “code package.” In ACPI 2.0, the Break command has been changed to exit the innermost “While” package. This should have no impact on existing code, since the ACPI 1.0 definition was, in practice, useless.

16.2.3.4.1.2 BreakPoint (BreakPoint)

```
BreakPointTerm               := BreakPoint
```

Used for debugging, the **Breakpoint** opcode stops the execution and enters the AML debugger. In the retail version of the interpreter, **BreakPoint** is equivalent to **Noop**.

16.2.3.4.1.3 Continue – Continue Innermost Enclosing While

```
ContinueTerm                  := Continue
```

Continue causes execution to continue at the start of the innermost enclosing **While** scope, in the current Method, at the point where the condition is evaluated. If there is no enclosing **While** within the current Method, a fatal error is generated.

16.2.3.4.1.4 Else/ElseIf (Else Operator)

```
ElseTerm                      := Nothing | <Else {TermList}>|
                              <ElseIf (predicate) {TermList} ElseTerm
```

If *Predicate* evaluates to 0 in an **If** statement, then control is transferred to the Else portion, which can consist of zero or more **ElseIf** statements followed by zero or one **Else** statements. If the *Predicate* of any **ElseIf** statement evaluates to non-zero, the statements in its term list are executed and then control is transferred past the end of the final Else term. If no *Predicate* evaluates to non-zero, then the statements in the **Else** term list are executed.

The following example checks Local0 to be zero or non-zero. On non-zero, CNT is incremented; otherwise, CNT is decremented.

```
    If (LGreater(Local0,5) {
        Increment (CNT)
    } Else If (Local0) {
        Add(CNT,5,CNT)
    }
    Else {
        Decrement (CNT)
    }
```

Compatibility Note: The **ElseIf** operator is new in ACPI 2.0, but is backward compatible with the ACPI 1.0 specification. The ACPI 2.0 compiler must synthesize **ElseIf** from the If..Else opcodes available in 1.0. For example:

```
If (predicate1) {
    …statements1…
}
ElseIf (predicate2) {
    …statements2…
}
Else {
    …statements3…
}
```

is translated to the following:

```
If(predicate1) {
    …statements1…
}
Else {
    If (predicate2) {
        …statements2…
    }
    Else {
        …statements3…
    }
}
```

16.2.3.4.1.5 Fatal (Fatal Check)

```
FatalTerm                      := Fatal(
                                       Type,                 //ByteConstExpr
                                       Code,                 //DWordConstExpr
                                       Arg                   //TermArg=>Integer
                                   )
```

This operation is used to inform the OS that there has been an OEM-defined fatal error. In response, the OS must log the fatal event and perform a controlled OS shutdown in a timely fashion.

16.2.3.4.1.6 If (If Operator)

```
IfTerm  := If(
                                       Predicate             //TermArg=>Integer
                                   ) {TermList}
```

Predicate is evaluated as an integer. If the integer is non-zero, the term list of the **If** term is executed.

The following examples all check for bit 3 in **Local0** being set, and clear it if set.

```
    // example 1
    If (And(Local0, 4)) {
        XOr (Local0, 4, Local0)
    }
    // example 2
    Store(4, Local2)
    If (And(Local0, Local2)) {
        XOr (Local0, Local2, Local0)
    }
```

16.2.3.4.1.7 Load (Load Definition Block)

```
LoadTerm                       := Load(
                                       Object,               //NameString
                                       DDBHandle             //SuperName
                                   )
```

Performs a run-time load of a Definition Block. The *Object* parameter can either refer to an operation region field or an operation region directly. If the object is an operation region, the operation region must be in SystemMemory space. The Definition Block should contain a DESCRIPTION_HEADER of type SSDT. The Definition Block must be totally contained within the supplied operation region or operation region field. OSPM reads this table into memory, the checksum is verified, and then it is loaded into the ACPI namespace. The *DDBHandle* parameter is the handle to the Definition Block that can be used to unload the Definition Block at a future time.

The OS can also check the OEM Table ID and Revision ID against a database for a newer revision Definition Block of the same OEM Table ID and load it instead.

The default namespace location to load the Definition Block is relative to the current namespace. The new Definition Block can override this by specifying absolute names or by adjusting the namespace location using the **Scope** operator.

Loading a Definition Block is a synchronous operation. Upon completion of the operation, the Definition Block has been loaded. The control methods defined in the Definition Block are not executed during load time.

16.2.3.4.1.8 Noop Code (No Operation)

```
NoOpTerm                       := Noop
```

This operation has no effect.

16.2.3.4.1.9 Notify (Notify)

```
NotifyTerm                    := Notify(
                                      Object,               //SuperName=>ThermalZone |
                                                            //  Processor | Device
                                      NotificationValue     //TermArg=>ByteConstExpr
                                  )
```

Notifies the OS that the *NotificationValue* for the *Object* has occurred. *Object* must be a reference to a device, processor, or thermal zone object.

Object type determines the notification values. For example, the notification values for a thermal zone object are different from the notification values used for a device object. Undefined notification values are treated as reserved and are ignored by the OS.

For lists of defined Notification values, see section 5.6.3, “Device Object Notifications.”

16.2.3.4.1.10 Release (Release a Mutex Synchronization Object)

```
ReleaseTerm                   := Release(
                                      SyncObject            //SuperName=>Mutex
                                  )
```

SynchObject must be a mutex synchronization object. If the mutex object is owned by the current invocation, ownership for the Mutex is released once. It is fatal to release ownership on a Mutex unless it is currently owned. A Mutex must be totally released before an invocation completes.

16.2.3.4.1.11 Reset (Reset an Event Synchronization Object)

```
ResetTerm                     := Reset(
                                      SyncObject            //SuperName=>Event
                                  )
```

SynchObject must be an Event synchronization object. This encoding is used to reset an event synchronization object to a non-signaled state. See also the Wait and Signal function operator definitions.

16.2.3.4.1.12 Return (Return)

```
ReturnTerm                    := Return(
                                      Arg                   //Nothing |
                                                            // TermArg=>DataRefObject
                                  )
```

Returns control to the invoking control method, optionally returning a copy of the object named in *Arg*.

16.2.3.4.1.13 Signal (Signal a Synchronization Event)

```
SignalTerm                    := Signal(
                                      SyncObject            //SuperName=>Event
                                  )
```

SynchObject must be an Event synchronization object. The Event object is signaled once, allowing one invocation to acquire the event.

16.2.3.4.1.14 Sleep (Sleep)

```
SleepTerm                     := Sleep(
                                      MilliSecs             //TermArg=>Integer
                                  )
```

The **Sleep** term is used to implement long-term timing requirements. Execution is delayed for at least the required number of milliseconds. The implementation of **Sleep** is to round the request up to the closest sleep time supported by the OS and relinquish the processor.

16.2.3.4.1.15 Stall (Stall for a Short Time)

```
StallTerm                     := Stall(
                                     MicroSecs             //TermArg=>Integer
                                  )
```

The **Stall** term is used to implement short-term timing requirements. Execution is delayed for at least the required number of microseconds. The implementation of **Stall** is OS-specific, but must not relinquish control of the processor. Because of this, delays longer than 100 microseconds must use **Sleep** instead of **Stall**.

16.2.3.4.1.16 Switch – Select Code To Execute Based On Expression

```
SwitchTerm                    := Switch(
                                     Predicate             // TermArg=>ComputationalData
                                  ) {CaseTermList}

DefaultTermList               := Nothing | CaseTerm | CaseTerm DefaultTermList

CaseTermList                  := Nothing | CaseTerm | DefaultTerm DefaultTermList |
                                  CaseTerm CaseTermList

CaseTerm                      := Case(DataObject) {TermList}
DefaultTerm                   := Default {TermList}
```

The **Switch**, **Case** and **Default** statements help simplify the creation of conditional and branching code. The **Switch** statement transfers control to a statement within its body.

If the **Case** *value* is an Integer, Buffer or String, then control passes to the statement that matches the value of **Switch(***Predicate***)**. If the **Case** value is a Package, then control passes if any member of the package matches the **Switch**(*Predicate*). The **Switch** *CaseTermList* can include any number of **Case** instances, but no two **Case** *value*s (or members of a *value*, if *value* is a Package) within the same **Switch** statement can contain the same value.

Execution of the statement body begins at the selected statement's TermList and proceeds until the end of the body or until an **ExitSwitch** (or other valid **Exit**x) statement transfers control out of the body.

Use of the **Switch** statement usually looks something like this:

```
Switch ( expression )
{
    Case ( value ) {
        Statements executed if Lequal(expression, value)

    }
    Case ( Package() {value,value,value}) {
        Statements executed if Lequal(expression, any value in package)
    Default {
        statements executed if expression does not equal
        any case constant-expression
    }
}
```

The **Default** statement is executed if no **Case** *value* matches the value of **switch (** *expression* **)**. If the **Default** statement is omitted, and no **Case** match is found, none of the statements in the **Switch** body are executed. There can be at most one **Default** statement. The **Default** statement need not come at the end; it can appear anywhere in the body of the **Switch** statement.

A **Case** or **Default** term can only appear inside a **Switch** statement. Switch statements can be nested.

Compatibility Note: The **Switch, Case,** and **Default** terms are new to ACPI 2.0. However, their implementation is backward compatible with ACPI 1.0 AML interpreters.

Compiler Note: The following example demonstrates how the Switch statement should be translated into ACPI 1.0-compatible AML:

```
Switch (Add(ABCD(),1)
{
    Case(1) {
    …statements1…
    }
    Case(Package() {4,5,6}) {
    …statements2…
    }
    Default {
    …statements3…
    }
}
```

is translated as:

```
While(One)
{
    Name(_T_I,0)                  // Create Integer temporary variable for result
    Store(Add(ABCD(),1),_T_I)
    If (LEqual(_T_I,1)) {
        …statements1…
    }
    Else {
    If (LNotEqual(Match(Package() {4,5,6},MEQ,_T_I,MTR,0,0),Ones)) {
        …statements2…
    }
    Else {
        …statements3…
    }
    }
    Break
}
```

Note: If the compiler is unable to determine the type of the expression, then it should generate a warning and assume integer type. The warning should indicate that the ASL should use one of the type conversion operators (ToInteger, ToBuffer, ToDecimalString or ToHexString). For example:

```
Switch(ABCD())        // Can’t determine the type because methods can return anything.
{
    …case statements…
}
```

will generate a warning and the following code:

```
Name(_T_I,0)
Store(ABCD(),_T_I)
```

To remove the warning, the code should be:

```
Switch(Int(ABCD()))
{
    …case statements…
}
```

16.2.3.4.1.17 Unload (Unload Definition Block)

```
UnloadTerm                    := Unload(
                                      DDBHandle              //TermArg=>DDBHandle
                                 )
```

Performs a run-time unload of a Definition Block that was loaded using a **Load** term. Loading or unloading a Definition Block is a synchronous operation, and no control method execution occurs during the function. On completion of the **Unload** operation, the Definition Block has been unloaded (all the namespace objects created as a result of the corresponding **Load** operation will be removed from the namespace).

16.2.3.4.1.18 While (While)

```
WhileTerm                     := While(
                                     Predicate             //TermArg=>Integer
                                 ) {TermList}
```

Predicate is evaluated as an integer. If the integer is non-zero, the list of terms in TermList is executed. The operation repeats until the *Predicate* evaluates to zero.

16.2.3.4.2 Type 2 Opcodes

```
Type2Opcode                   := AcquireTerm | AddTerm | AndTerm | ConcatTerm |
                                 ConcateResTerm | CondRefOfTerm | DecTerm | DerefOfTerm |
                                 DivideTerm | FindSetLeftBitTerm | FindSetRightBitTerm |
                                 FromBCDTerm | IncTerm | IndexTerm | LAndTerm |
                                 LEqualTerm | LGreaterTerm | LGreaterEqualTerm |
                                 LLessTerm | LLessEqualTerm | LNotTerm | LNotEqualTerm |
                                 LoadTableTerm | LOrTerm | MatchTerm | MidTerm |
                                 ModTerm | MultiplyTerm | NAndTerm | NOrTerm | NotTerm |
                                 ObjectTypeTerm | OrTerm | RefOfTerm | ShiftLeftTerm |
                                 ShiftRightTerm | SizeOfTerm | StoreTerm | SubtractTerm |
                                 ToBCDTerm | ToBufferTerm | ToDecimalStringTerm |
                                 ToHexStringTerm | ToIntegerTerm | ToStringTerm |
                                 WaitTerm | XorTerm | UserTerm
```

The ASL terms for Type 2 Opcodes are listed in the following table.

Table 16-9 Type 2 Opcodes

ASL Statement	Description
Acquire	Acquire a mutex
Add	Add two values
And	Bitwise And
Concatenate	Concatenate two strings, integers or buffers
ConcatenateResTemplate	Concatenate two resource templates
CondRefOf	Conditional reference to an object
Decrement	Decrement a value
DerefOf	Dereference an object reference
Divide	Divide
FindSetLeftBit	Index of first least significant bit set
FindSetRightBit	Index of first most significant bit set
FromBCD	Convert from BCD to numeric
Increment	Increment a value
Index	Reference the nth element/byte/character of a package, buffer or string
LAnd	Logical And

Table 16-9 Type 2 Opcodes *(continued)*

ASL Statement	Description
LEqual	Logical Equal
LGreater	Logical Greater
LGreaterEqual	Logical Not less
LLess	Logical Less
LLessEqual	Logical Not greater
LNot	Logical Not
LNotEqual	Logical Not equal
LoadTable	Load Table from RSDT/XSDT
LOr	Logical Or
Match	Search for match in package array
Mid	Returns a portion of buffer or string
Mod	Modulo
Multiply	Multiply
NAnd	Bitwise Nand
NOr	Bitwise Nor
Not	Bitwise Not
ObjectType	Type of object
Or	Bitwise Or
RefOf	Reference to an object
ShiftLeft	Shift value left
ShiftRight	Shift value right
SizeOf	Get the size of a buffer, string, or package
Store	Store value
Subtract	Subtract values
ToBCD	Convert numeric to BCD
ToBuffer	Convert data type to buffer
ToDecimalString	Convert data type to decimal string
ToHexString	Convert data type to hexadecimal string
ToInteger	Convert data type to integer
ToString	Copy ASCII string from buffer
Wait	Wait
Xor	Bitwise Xor

16.2.3.4.2.1 Acquire (Acquire a Mutex)

```
AcquireTerm                   := Acquire(
                                     SyncObject,            //SuperName=>Mutex
                                     TimeoutValue           //WordConstExpr
                                 ) => Boolean               //True means timed-out
```

SynchObject must be a mutex synchronization object. It refers to the mutex to be acquired.

Ownership of the Mutex is obtained. If the Mutex is already owned by a different invocation, the processor is relinquished until the owner of the Mutex releases it or until at least *TimeoutValue* milliseconds have elapsed. A Mutex can be acquired more than once by the same invocation.

This operation returns True if a timeout occurred and the mutex ownership was not acquired. A *TimeoutValue* of 0xFFFF indicates that there is no time out and the operation will wait indefinitely.

16.2.3.4.2.2 Add (Add)

```
AddTerm                       := Add(
                                     Addend1,               //TermArg=>Integer
                                     Addend2,               //TermArg=>Integer
                                     Result                 //Target
                                 ) => Integer
```

Addend1 and *Addend2* are evaluated as integer data types and are added, and the result is optionally stored into *Result*. Overflow conditions are ignored and the result of overflows simply loses the most significant bits.

16.2.3.4.2.3 And (Bitwise And)

```
AndTerm                       := And(
                                     Source1,               //TermArg=>Integer
                                     Source2,               //TermArg=>Integer
                                     Result                 //Target
                                 ) => Integer
```

Source1 and *Source2* are evaluated as integer data types, a bitwise AND is performed, and the result is optionally stored into *Result.*

16.2.3.4.2.4 ToBuffer (Convert Data Type to Buffer)

```
ToBufferTerm                  := ToBuffer(
                                     Data,                  //TermArg=>ComputationalData
                                     Result                 //Target
                                 ) => Buffer
```

Data must be evaluated to integer, string, or buffer. *Data* is then converted to buffer type and the result is optionally stored into *Result*. If *Data* was an integer, it is converted into n bytes of buffer (where n is 4 if the definition block has defined integers as 32-bits or 8 if the definition block has defined integers as 64-bits as indicated by the Definition Block table header’s Revision field), taking the least significant byte of integer as the first byte of buffer. If *Data* is a buffer, no conversion is performed.

16.2.3.4.2.5 Concatenate (Concatenate)

```
ConcatTerm                    := Concatenate(
                                     Source1,               //TermArg=>ComputationalData
                                     Source2,               //TermArg=>ComputationalData
                                     Result                 //Target
                                 ) => ComputationalData
```

Source1 and *Source2* are evaluated. *Source1* and *Source2* must be of the same data type (that is, both integers, both strings, or both buffers). *Source2* is concatenated to *Source1* and the result data is optionally stored into *Result.*

Table 16-10 Concatenate Data Types

Source1 Data Type	Source2 Data Type	Result Data Type
Integer	Integer	Buffer
String	String	String
Buffer	Buffer	Buffer

16.2.3.4.2.6 ConcatenateResTemplate (Concatenate Resource Templates)

```
ConcatResTerm                 := ConcatenateResTemplate(
                                     Source1,             //TermArg=>Buffer
                                     Source2,             //TermArg=>Buffer
                                     Result               //Target
                                 ) => Buffer
```

Source1 and *Source2* are evaluated as resource template buffers. The resource descriptors from *Source2* are appended to the resource descriptors from *Source1*. Then a new end tag and checksum are appended and the result is stored in *Result*, if specified. If either *Source1* or *Source2* is exactly 1 byte in length, a run-time error occurs. An empty buffer is treated as a resource template with only an end tag.

16.2.3.4.2.7 CondRefOf (Conditional Reference Of)

```
CondRefOfTerm                 := CondRefOf(
                                     Source,              //SuperName
                                     Destination          // Target
                                 ) => Boolean
```

Attempts to set *Destination* to refer to *Source*. The *Source* of this operation can be any object type (for example, data package, device object, and so on). On success, the *Destination* object is set to refer to *Source* and the execution result of this operation is the value **True**. On failure, *Destination* is unchanged and the execution result of this operation is the value **False**. This can be used to reference items in the namespace that may appear dynamically (for example, from a dynamically loaded differentiation definition block).

CondRefOf is equivalent to **RefOf** except that if the *Source* object does not exist, it is fatal for **RefOf** but not for **CondRefOf**.

16.2.3.4.2.8 CopyObject (Copy Object)

```
CopyObjectTerm                := CopyObject(
                                     Source,              //SuperName=>DataRefObject
                                     Destination          //NameString | LocalTerm |
                                                          // ArgTerm
                                 ) => DataRefObject
```

Converts the contents of the *Source* to a DataRefObject using the conversion rules in 16.2.2 and then copies the results without conversion to the object referred to by *Destination*. If *Destination* is already an initialized object of type DataRefObject, the original contents of *Destination* are discarded and replaced with *Source*. Otherwise, a fatal error is generated.

Compatibility Note: The CopyObject operator is new in ACPI 2.0.

16.2.3.4.2.9 Decrement (Decrement)

```
DecTerm                      := Decrement(
                                    Addend                //SuperName
                                ) => Integer
```

This operation decrements the *Addend* by one and the result is stored back to *Addend*. Equivalent to Subtract(Addend,1,Addend). Underflow conditions are ignored and the result is 1s.

16.2.3.4.2.10 ToDecimalString (Convert Data Type to Decimal String)

```
ToDecimalStringTerm          := ToDecimalString(
                                    Data,                 //TermArg=>ComputationalData
                                    Result                //Target
                                ) => String
```

Data must be evaluated to integer, string, or buffer. *Data* is then converted to a decimal string, and the result is optionally stored into *Result*. If *Data* is already a string, no action is performed. If *Data* is a buffer, it is converted to a string of decimal values separated by commas. (Each byte of the buffer is converted to a single decimal value.)

16.2.3.4.2.11 DerefOf (Dereference Of Operator)

```
DerefOfTerm                  := DerefOf(
                                    Source                //TermArg=>ObjectReference |
                                    //  String
                                ) => Object
```

Returns the object referred by the *Source* object reference. If the *Source* evaluates to an object reference, the actual contents of the object referred to are returned. If the *Source* evaluates to a string, the string is evaluated as an ASL name (relative to the current scope) and the contents of that object are returned. If the object specified by *Source* does not exist then a fatal error is generated.

Compatibility Note: The use of a String with DerefOf is new in ACPI 2.0.

16.2.3.4.2.12 Divide (Divide)

```
DivideTerm                   := Divide(
                                    Dividend,             //TermArg=>Integer
                                    Divisor,              //TermArg=>Integer
                                    Remainder,            //Target
                                    Result                //Target
                                ) => Integer              //returns Result
```

Dividend and *Divisor* are evaluated as integer data. *Dividend* is divided by *Divisor*, then the resulting remainder is optionally stored into *Remainder* and the resulting quotient is optionally stored into *Result*. Divide-by-zero exceptions are fatal.

16.2.3.4.2.13 FindSetLeftBit (Find Set Left Bit)

```
FindSetLeftBitTerm           := FindSetLeftBit(
                                    Source,               //TermArg=>Integer
                                    Result                //Target
                                ) => Integer
```

Source is evaluated as integer data type, and the one-based bit location of the first MSb (most significant set bit) is optionally stored into *Result*. The result of 0 means no bit was set, 1 means the left-most bit set is the first bit, 2 means the left-most bit set is the second bit, and so on.

16.2.3.4.2.14 FindSetRightBit (Find Set Right Bit)

```
FindSetRightBitTerm           := FindSetRightBit(
                                      Source,               //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer
```

Source is evaluated as integer data type, and the one-based bit location of the most LSb (least significant set bit) is optionally stored in *Result.* The result of 0 means no bit was set, 32 means the first bit set is the thirty-second bit, 31 means the first bit set is the thirty-first bit, and so on.

16.2.3.4.2.15 FromBCD (Convert from BCD)

```
FromBCDTerm                   := FromBCD(
                                      BCDValue,             //TermArg=>Integer
                                      Result                //Target
                                  ) => Integer
```

The **FromBCD** operation is used to convert *BCDValue* to a numeric format and store the numeric value into *Result*.

16.2.3.4.2.16 ToHexString (Convert Data Type to Hexadecimal String)

```
ToHexStringTerm               := ToHexString(
                                      Data,                 //TermArg=>ComputationalData
                                      Result                //Target
                                  ) => String
```

Data must be evaluated to integer, string, or buffer. *Data* is then converted to a hexadecimal string, and the result is optionally stored into *Result*. If *Data* is already a string, no action is performed. If *Data* is a buffer, it is converted to a string of hexadecimal values separated by commas.

16.2.3.4.2.17 Increment (Increment)

```
IncTerm                       := Increment(
                                      Addend                //SuperName
                                  ) => Integer
```

Add one to the Addend and place the result back in Addend. Equivalent to **Add**(*Addend*, 1, *Addend*). Overflow conditions are ignored and the result of an overflow is zero.

16.2.3.4.2.18 Index (Index)

```
IndexTerm                     := Index(
                                      Source,               //TermArg= Buffer |
                                                            //  String |  Package>
                                      Index,                //TermArg=>Integer
                                      Destination           //Target
                                  ) => ObjectReference
```

Source is evaluated to a buffer, string, or package data type. *Index* is evaluated to an integer. The reference to the nth object (where n = *Index*) within *Source* is optionally stored as a reference into *Destination.* When *Source* evaluates to a Buffer, Index returns a reference to a Buffer Field containing the nth byte in the buffer. When *Source* evaluates to a String, Index returns a reference to a Buffer Field containing the nth character in the string. When *Source* evaluates to a Package, Index returns a reference to the nth object in the package.

16.2.3.4.2.18.1 Index with Packages

The following example ASL code shows a way to use the **Index** term to store into a local variable the sixth element of the first package of a set of nested packages:

```
Name(IO0D, Package() {
     Package() {
     0x01, 0x03F8, 0x03F8, 0x01, 0x08, 0x01,
     0x25, 0xFF, 0xFE, 0x00, 0x00
     },
     Package() {
     0x01, 0x02F8, 0x02F8, 0x01, 0x08, 0x01,
     0x25, 0xFF, 0xBE, 0x00, 0x00
     },
     Package() {
     0x01, 0x03E8, 0x03E8, 0x01, 0x08, 0x01,
     0x25, 0xFF, 0xFA, 0x00, 0x00
     },
     Package() {
     0x01, 0x02E8, 0x02E8, 0x01, 0x08, 0x01,
     0x25, 0xFF, 0xBA, 0x00, 0x00
     },
     Package() {
     0x01, 0x0100, 0x03F8, 0x08, 0x08, 0x02,
     0x25, 0x20, 0x7F, 0x00, 0x00,
     }
})

//Get the 6th element of the first package
Store(DeRefOf(Index(DeRefOf(Index(IO0D, 0)), 5)), Local0)
```

Note: DeRefOf is necessary in the first operand of the Store command in order to get the actual object, rather than just a reference to the object. If DeRefOf were not used, then Local0 would contain an object reference to the sixth element in the first package rather than the number 1.

16.2.3.4.2.18.2 Index with Buffers

The following example ASL code shows a way to store into the third byte of a buffer:

```
Name(BUFF, Buffer() {0x01, 0x02, 0x03, 0x04, 0x05})

//Store 0x55 into the third byte of the buffer
Store(0x55, Index(BUFF, 2))
```

The Index operator returns a reference to an 8-bit Buffer Field (similar to that created using CreateByteField).

If *Source* is evaluated to a buffer data type, the *ObjectReference* refers to the byte at *Index* within *Source.* If *Source* is evaluated to a buffer data type, a **Store** operation will only change the byte at *Index* within *Source.*

The following example ASL code shows the results of a series of **Store** operations:

```
Name(SRCB, Buffer() {0x10, 0x20, 0x30, 0x40})

Name(BUFF, Buffer() {0x1, 0x2, 0x3, 0x4})
```

The following will store 0x78 into the 3rd byte of the destination buffer:

```
Store (0x12345678, Index(BUFF, 2))
```

The following will store 0x10 into the 2nd byte of the destination buffer:

```
Store (SRCB, Index(BUFF, 1))
```

The following will store 0x41 (an ‘A’) into the 4th byte of the destination buffer:

```
    Store(“ABCDEFGH”, Index(BUFF, 3))
```

Compatibility Note: New in ACPI 2.0. In ACPI 1.0, the behavior of storing data larger than 8-bits into a buffer using Index was undefined.

16.2.3.4.2.18.3 Index with Strings

The following example ASL code shows a way to store into the 3rd character in a string:

```
    Name(STR, “ABCDEFGHIJKL”)

    // Store ‘H’ (0x48) into the third character to the string
    Store(“H”, Index(STR,2))
```

The Index operator returns a reference to an 8-bit Buffer Field (similar to that created using CreateByteField).

Compatibility Note: New in ACPI 2.0.

16.2.3.4.2.19 ToInteger (Convert Data Type to Integer)

```
ToIntegerTerm                  := ToInteger(
                                       Data,                 //TermArg=>ComputationalData
                                       Result                //Target
                                   ) => Integer
```

Data must be evaluated to integer, string, or buffer. *Data* is then converted to integer type and the result is optionally stored into *Result*. If *Data* was a string, it must be either a decimal or hexadecimal numeric string (in other words, prefixed by “0x”) and the value must not exceed the maximum of an integer value. If the value is exceeding the maximum, the result of the conversion is unpredictable. If *Data* was a Buffer, the first 8 bytes of the buffer are converted to an integer, taking the first byte as the least significant byte of the integer. If *Data* was an integer, no action is performed.

16.2.3.4.2.20 LAnd (Logical And)

```
LAndTerm                       := LAnd(
                                       Source1,              //TermArg=>Integer
                                       Source2               //TermArg=>Integer
                                   ) => Boolean
```

Source1 and *source2* are evaluated as integers. If both values are non-zero, True is returned: otherwise, False is returned.

16.2.3.4.2.21 LEqual (Logical Equal)

```
LEqualTerm                     := LEqual(
                                       Source1,              //TermArg=>ComputationalData
                                       Source2               //TermArg=>ComputationalData
                                   ) => Boolean
```

Source1 and *Source2* must be evaluated to the same data type as integers, strings, or buffers. If the values are equal, True is returned; otherwise, False is returned.

16.2.3.4.2.22 LGreater (Logical Greater)

```
LGreaterTerm                   := LGreater(
                                       Source1,              //TermArg=>ComputationalData
                                       Source2               //TermArg=>ComputationalData
                                   ) => Boolean
```

Source1 and *Source2* must be evaluated to the same data type as integers, strings, or buffers. If *Source1* is greater than *Source2*, True is returned; otherwise, False is returned.

16.2.3.4.2.23 LGreaterEqual (Logical Greater Than Or Equal)

```
LGreaterEqualTerm              := LGreaterEqual(
                                       Source1,              //TermArg=>ComputationalData
                                       Source2               //TermArg=>ComputationalData
                                   ) => Boolean
```

Source1 and *Source2* must be evaluated to the same data type as integers, strings, or buffers. If *Source1* is greater than or equal to *Source2,* True is returned; otherwise, False is returned.

16.2.3.4.2.24 LLess (Logical Less)

```
LLessTerm                      := LLess(
                                       Source1,              //TermArg=>ComputationalData
                                       Source2               //TermArg=>ComputationalData
                                   ) => Boolean
```

Source1 and *Source2* must be evaluated to the same data type as integers, strings, or buffers. If *Source1* is less than *Source2,* True is returned; otherwise, False is returned.

16.2.3.4.2.25 LLessEqual (Logical Less Than Or Equal)

```
LLessEqualTerm                 := LLessEqual(
                                       Source1,              //TermArg=>ComputationalData
                                       Source2               //TermArg=>ComputationalData
                                   ) => Boolean
```

Source1 and *Source2* must be evaluated to the same data type as integers, strings, or buffers. If *Source1* is less than or equal to *Source2,* True is returned; otherwise, False is returned.

16.2.3.4.2.26 LNot (Logical Not)

```
LNotTerm                       := LNot(
                                       Source,               //TermArg=>Integer
                                   ) => Boolean
```

Source1 is evaluated as an integer. If the value is zero True is returned; otherwise, False is returned.

16.2.3.4.2.27 LNotEqual (Logical Not Equal)

```
LNotEqualTerm                  := LNotEqual(
                                       Source1,              //TermArg=>ComputationalData
                                       Source2               //TermArg=>ComputationalData
                                   ) => Boolean
```

Source1 and *Source2* must be evaluated to the same data type as integers, strings, or buffers. If *Source1* is not equal to *Source2*, True is returned; otherwise, False is returned.

16.2.3.4.2.28 LoadTable (Load Definition Block From XSDT)

```
LoadTableTerm                  := LoadTable(
                                   SignatureString,          //TermArg=>String
                                   OEMIDString,              //TermArg=>String
                                   OEMTableIDString,         //TermArg=>String
                                   RootPathString,           //Nothing | TermArg=>String
                                   ParameterPathString,      //Nothing | TermArg=>String
                                   ParameterData,            //Nothing |
                                                             //TermArg=>DataRefObject
                               ) => DDBHandle
```

Performs a run-time load of a Definition Block from the XSDT. The XSDT is searched for a table where the Signature field matches *SignatureString*, the OEM ID field matches *OEMIDString*, and the OEM Table ID matches *OEMTableIDString*. All comparisons are case sensitive. If the *SignatureString* is greater than four characters, the *OEMIDString* is greater than six characters, or the *OEMTableID* is greater than eight characters, a run-time error is generated. The OS can also check the OEM Table ID and Revision ID against a database for a newer revision Definition Block of the same OEM Table ID and load it instead.

The *RootPathString* specifies the root of the Definition Block. It is evaluated using normal scoping rules, assuming that the scope of the **LoadTable** instruction is the current scope. The new Definition Block can override this by specifying absolute names or by adjusting the namespace location using the **Scope** operator. If *RootPathString* is not specified, "\" is assumed

If *ParameterPathString* and *ParameterData* are specified, the data object specified by *ParameterData* is stored into the object specified by *ParameterPathString* after the table has been added into the namespace. If the first character of *ParameterPathString* is a backslash ('\') or caret ('^') character, then the path of the object is *ParameterPathString*. Otherwise, it is *RootPathString.ParameterPathString*. If the specified object does not exist, a run-time error is generated.

The handle of the loaded table is returned. If no table matches the specified signature, then 0 is returned.

Any table referenced by **Load Table** must be in memory marked by AddressRangeReserved or AddressRangeNVS.

Loading a Definition Block is a synchronous operation. Upon completion of the operation, the Definition Block has been loaded. The control methods defined in the Definition Block are not executed during load time.

For example:

```
    Store(LoadTable("OEM1", "MYOEM", "TABLE1", "\\_SB.PCI0","MYD",
                    Package(){0,"\\_SB.PCI0"}),
                    Local0)
```

This command would search through the RSDT or XSDT for a table with the signature "OEM1," the OEM ID of "MYOEM," and the table ID of "TABLE1." If not found, it would store **Zero** in Local0. Otherwise, it will store a package containing 0 and "\\_SB.PCI0" into the variable at _SB.PCI0.MYD.

16.2.3.4.2.29 LOr (Logical Or)

```
LOrTerm := LOr(
                                         Source1,               //TermArg=>Integer
                                         Source2                //TermArg=>Integer
                                     ) => Boolean
```

Source1 and *Source2* are evaluated as integers. If either value is non-zero, True is returned; otherwise, False is returned.

16.2.3.4.2.30 Match (Find Object Match)

```
MatchTerm                      := Match(
                                         SearchPackage,         //TermArg=>Package
                                         Op1,                   //MatchOpKeyword
                                         MatchObject1,          //TermArg=>Integer
                                         Op2,                   //MatchOpKeyword
                                         MatchObject2,          //TermArg=>Integer
                                         StartIndex             //TermArg=>Integer
                                     ) => Ones | Integer
```

SearchPackage is evaluated to a package object and is treated as a one-dimension array. A comparison is performed for each element of the package, starting with the index value indicated by *StartIndex* (0 is the first element). If the element of *SearchPackage* being compared against is called *P[i]***,** then the comparison is:

if (*P[i] Op1 MatchObject1***) and (***P[i] Op2 MatchObject2***) then Match** => i is returned.

If the comparison succeeds, the index of the element that succeeded is returned; otherwise, the constant object **ONES** is returned.

Op1 and *Op2* have the values and meanings listed in the Table 16-13.

Table 16-11 Match Term Operator Meanings

Operator	Encoding	Macro
TRUE – A don't care, always returns TRUE	0	**MTR**
EQ – Returns TRUE if P[i] == MatchObject	1	**MEQ**
LE – Returns TRUE if P[i] <= MatchObject	2	**MLE**
LT – Returns TRUE if P[i] < MatchObject	3	**MLT**
GE – Returns TRUE if P[i] >= MatchObject	4	**MGE**
GT – Returns TRUE if P[i] > MatchObject	5	**MGT**

Following are some example uses of **Match**:

```
Name(P1,
Package() {1981, 1983, 1985, 1987, 1989, 1990, 1991, 1993, 1995, 1997, 1999, 2001}
)

// match 1993 == P1[i]
Match(P1, MEQ, 1993, MTR, 0, 0)   // -> 7, since P1[7] == 1993

// match 1984 == P1[i]
Match(P1, MEQ, 1984, MTR, 0, 0)   // -> ONES (not found)

// match P1[i] > 1984 and P1[i] <= 2000
Match(P1, MGT, 1984, MLE, 2000, 0) // -> 2, since P1[2]>1984 and P1[2]<=2000

// match P1[i] > 1984 and P1[i] <= 2000, starting with 3rd element
Match(P1, MGT, 1984, MLE, 2000, 3) // -> 3, first match at or past Start
```

16.2.3.4.2.31 Mid (Retrieve Portion of Buffer or String)

```
MidTerm                      := Mid(
                                    Source,                //TermArg=>Buffer|String
                                    Index,                 //TermArg=>Integer
                                    Length,                //TermArg=>Integer
                                    Result                 //Target
                                ) => Buffer|String
```

Source is evaluated as either a Buffer or String.

If *Source* is a buffer, then *Length* bytes, starting with the *Index*th byte (zero-based) are optionally copied into *Result*. If *Index* is greater than or equal to the length of the buffer, then the result is an empty buffer. Otherwise, if *Index* + *Length* is greater than or equal to the length of the buffer, then only bytes up to an including the last byte are included in the result.

If *Source* is a string, then *Length* characters, starting with the *Index*th character (zero-based) are optionally copied into *Result.* If *Index* is greater than or equal to the length of the buffer, then the result is an empty string. Otherwise, if *Index* + *Length* is greater than or equal to the length of the string, then only bytes up to an including the last character are included in the result.

16.2.3.4.2.32 Mod (Modulo)

```
ModTerm                      := Mod(
                                    Dividend,              //TermArg=>Integer
                                    Divisor,               //TermArg=>Integer
                                    Result                 //Target
                                ) => Integer               //returns Result
```

Dividend and *Divisor* are evaluated as integer data. *Dividend* is divided by *Divisor*, then the resulting remainder is optionally stored into *Result*. If *Divisor* evaluates to zero, a fatal exception is generated.

16.2.3.4.2.33 Multiply (Multiply)

```
MultiplyTerm                 := Multiply(
                                    Multiplicand,          //TermArg=>Integer
                                    Multiplier,            //TermArg=>Integer
                                    Result                 //Target
                                ) => Integer
```

Multiplicand and *Multiplier* are evaluated as integer data types. *Multiplicand* is multiplied by *Multiplier* and the result is optionally stored into *Result.* Overflow conditions are ignored and results are undefined.

16.2.3.4.2.34 NAnd (Bitwise Nand)

```
NAndTerm                     := NAnd(
                                    Source1,               //TermArg=>Integer
                                    Source2                //TermArg=>Integer
                                    Result                 //Target
                                ) => Integer
```

Source1 and *Source2* are evaluated as integer data types, a bitwise **NAND** is performed, and the result is optionally stored in *Result.*

16.2.3.4.2.35 NOr (Bitwise Nor)

```
NOrTerm                      := NOr(
                                    Source1,               //TermArg=>Integer
                                    Source2                //TermArg=>Integer
                                    Result                 //Target
                                ) => Integer
```

Source1 and *Source2* are evaluated as integer data types, a bitwise **NOR** is performed, and the result is optionally stored in *Result.*

16.2.3.4.2.36 Not (Not)

```
NotTerm                      := Not(
                                    Source,                //TermArg=>Integer
                                    Result                 //Target
                                ) => Integer
```

Source1 is evaluated as an integer data type, a bitwise **NOT** is performed, and the result is optionally stored in *Result.*

16.2.3.4.2.37 ObjectType (Object Type)

```
ObjectTypeTerm                := ObjectType(
                                     Object                //SuperName
                                 ) => Integer
```

The execution result of this operation is an integer that has the numeric value of the object type for *Object*. The object type codes are listed in Table 16-12. Notice that if this operation is performed on an object reference such as one produced by the **Alias, Index,** or **RefOf** statements, the object type of the base object is returned. For typeless objects such as pre-defined scope names (in other words, _SB, _GPE, and so on), the type value 0 (**Uninitialized**) is returned.

Table 16-12 Values Returned By the ObjectType Operator

Value	Meaning
0	Uninitialized
1	Integer
2	String
3	Buffer
4	Package
5	Field Unit
6	Device
7	Event
8	Method
9	Mutex
10	Operation Region
11	Power Resource
12	Processor
13	Thermal Zone
14	Buffer Field
15	DDB Handle
16	Debug Object
>16	Reserved

16.2.3.4.2.38 Or (Bit-wise Or)

```
OrTerm                        := Or(
                                     Source1,              //TermArg=>Integer
                                     Source2               //TermArg=>Integer
                                     Result                //Target
                                 ) => Integer
```

Source1 and *Source2* are evaluated as integer data types, a bitwise **OR** is performed, and the result is optionally stored in *Result.*

16.2.3.4.2.39 RefOf (Reference Of)

```
RefOfTerm                     := RefOf(
                                       Object                //SuperName
                                   ) => ObjectReference
```

Returns an object reference to *Object*. *Object* can be any object type (for example, a package, a device object, and so on).

If the *Object* does not exist, the result of a **RefOf** operation is fatal. Use the **CondRefOf** term in cases where the *Object* might not exist.

The primary purpose of **RefOf()** is to allow an object to be passed to a method as an argument to the method without the object being evaluated at the time the method was loaded.

16.2.3.4.2.40 ShiftLeft (Shift Left)

```
ShiftLeftTerm                 := ShiftLeft(
                                       Source,               //TermArg=>Integer
                                       ShiftCount            //TermArg=>Integer
                                       Result                //Target
                                   ) => Integer
```

Source and *ShiftCount* are evaluated as integer data types. *Source* is shifted left with the least significant bit zeroed *ShiftCount* times. The result is optionally stored into *Result.*

16.2.3.4.2.41 ShiftRight (Shift Right)

```
ShiftRightTerm                := ShiftRight(
                                       Source,               //TermArg=>Integer
                                       ShiftCount            //TermArg=>Integer
                                       Result                //Target
                                   ) => Integer
```

Source and *ShiftCount* are evaluated as integer data types. *Source* is shifted right with the most significant bit zeroed *ShiftCount* times. The result is optionally stored into *Result.*

16.2.3.4.2.42 SizeOf (SizeOf Data Object)

```
SizeOfTerm                    := SizeOf(
                                       Object                //SuperName=>
                                                             //Buffer|String|Package

                                   ) => Integer
```

Returns the size of a buffer, string, or package data object. For a buffer, it returns the size in bytes of the data. For a string, it returns the size in bytes of the string, not counting the trailing NULL. For a package, it returns the number of elements. For an object reference, the size of the referenced object is returned. Other data types cause a fatal run-time error.

16.2.3.4.2.43 Store (Store)

```
StoreTerm                     := Store(
                                       Source,               //TermArg=>DataRefObject
                                       Destination           //SuperName=>ObjectReference
                                   ) => DataRefObject
```

This operation evaluates *Source* converts to the data type of *Destination* and writes the results into *Destination.* For information on automatic data-type conversion, see section 16.2.2, “ASL Data Types.” Stores to Operational Region Field data types may relinquish the processor depending on the region type.

All stores (of any type) to the constant **Zero**, constant **One**, or constant **Ones** object are not allowed. Stores to read-only objects are fatal. The execution result of the operation depends on the type of *Destination.* For any type other than an operation region field, the execution result is the same as the data written to

Destination. For operation region fields with an *AccessType* of **ByteAcc**, **WordAcc**, **DWordAcc**, **QWordAcc** or **AnyAcc**, the execution result is the same as the data written to *Destination* as in the normal case, but when the *AccessType* is **BufferAcc**, the operation region handler may modify the data when it is written to the *Destination* so that the execution result contains modified data.

The following example creates the name CNT that references an integer data object with the value 5 and then stores CNT to Local0. After the Store operation, Local0 is an integer object with the value 5.

```
Name(CNT, 5)
Store(CNT, Local0)
```

16.2.3.4.2.44 ToString (Create ASCII String From Buffer)

```
ToStringTerm                := ToString(
                                 Source,                 //TermArg=>Buffer
                                 Length,                 //Nothing | TermArg=>Integer
                                 Result                  //Target
                               ) => String
```

Source is evaluated as a buffer. Starting with the first byte, the contents of the buffer are copied into the string until the number of characters specified by *Length* is reached or a null (0) character is found. If *Length* is not specified or is **Ones**, then the contents of the buffer are copied until a null (0) character is found. In any case, a fatal error will be generated if the number of characters copied exceeds 200 (not including the terminating null). If the source buffer has a length of zero, a zero length (null terminator only) string will be created. The result is copied into the *Result*.

16.2.3.4.2.45 Subtract (Subtract)

```
SubtractTerm                := Subtract(
                                     Addend1,            //TermArg=>Integer
                                     Addend2,            //TermArg=>Integer
                                     Result              //Target
                                 ) => Integer
```

Addend1 and *Addend2* are evaluated as integer data types. *Addend2* is subtracted from *Addend1*, and the result is optionally stored into *Result*. Underflow conditions are ignored and the result simply loses the most significant bits.

16.2.3.4.2.46 ToBCD (Convert to BCD)

```
ToBCDTerm                   := ToBCD(
                                     Value,              //TermArg=>Integer
                                     Result              //Target
                                 ) => Integer
```

The **ToBCD** operation is used to convert *Value* from a numeric format to a BCD format and optionally store the numeric value into *Result*.

16.2.3.4.2.47 Wait (Wait for a Synchronization Event)

```
WaitTerm                    := Wait(
                                     SyncObject,         //SuperName=>Event
                                     TimeoutValue        //TermArg=>Integer
                                 ) => Boolean
```

SynchObject must be an event synchronization object. The calling method blocks waiting for the event to be signaled.

The pending signal count is decremented. If there is no pending signal count, the processor is relinquished until a signal count is posted to the Event or until at least *TimeoutValue* milliseconds have elapsed.

This operation returns a non-zero value if a timeout occurred and a signal was not acquired. A *TimeoutValue* of 0xFFFF indicates that there is no time out and the operation will wait indefinitely.

16.2.3.4.2.48 XOr (Bitwise Xor)

```
XOrTerm                      := XOr(
                                    Source1,              //TermArg=>Integer
                                    Source2               //TermArg=>Integer
                                    Result                //Target
                                ) => Integer
```

Source1 and *Source2* are evaluated as integer data types, a bitwise **XOR** is performed, and the result is optionally stored into *Result.*

16.2.3.5 User Terms

```
UserTerm                     := NameString(               //NameString=>Method
                                    ArgList
                                ) => Nothing | DataRefObject
```

NameString must refer to an existing Method in the namespace. If the Method is not present, a fatal error is generated. It can either be an absolute namespace path or else it must be accessible at the current scope of invocation. The number of arguments in *ArgList* must match the number of arguments declared in the method object.

16.2.3.6 Data Objects

There are four different types of data objects:

- Buffer terms
- Package terms
- Literal data terms
- Data macros

16.2.3.6.1 Buffer (Declare Buffer Object)

```
BufferTerm                   := Buffer(
                                    BuffSize              //Nothing |
                                                          //  TermArg=>Integer
                                ) {String | ByteList} => Buffer
```

Declares a Buffer, of size *BuffSize* and initial value of *Initializer (ByteList).*

The optional *BuffSize* parameter specifies the size of the buffer and the initial value is specified in *Initializer* ByteList. If *BuffSize* is not specified, it defaults to the size of initializer. If the count is too small to hold the value specified by initializer, initializer size is used. For example, all four of the following examples generate the same data in namespace, although they have different ASL encodings:

```
Buffer(10) {“P00.00A”}
Buffer(Arg0) {0x50 0x30 0x30 0x2e 0x30 0x30 0x41}
Buffer(10) {0x50 0x30 0x30 0x2e 0x30 0x30 0x41 0x00 0x00 0x00}
Buffer() {0x50 0x30 0x30 0x2e 0x30 0x30 0x41 0x00 0x00 0x00}
```

16.2.3.6.2 Package (Declare Package Object)

```
PackageTerm                  := Package(
                                    NumElements           //Nothing |
                                                          //  ByteConstExpr |
                                                          //  TermArg=>Integer
                                ) {PackageList} => Package
```

Declares an unnamed aggregation of data items, constants, and/or references to control methods. The size of the package is *NumElements. PackageList* contains the list data items, constants, and/or control method references used to initialize the package. If *NumElements* is absent, it is set to match the number of elements in the PackageList. If *NumElements* is present and greater than the number of elements in the PackageList, the default entry of type Uninitialized (see **ObjectType**) is used to initialize the package elements beyond those initialized from the PackageList. Evaluating an undefined element will yield an error, but elements can be assigned values to make them defined. It is an error for *NumElements* to be less than the number of elements in the PackageList. It is an error for *NumElements* to exceed 255.

There are two types of package elements in the PackageList: data objects and references to control methods.

Note: If non-method code-package objects are implemented in an ASL compiler, evaluations of these objects are performed within the scope of the invoking method, and are performed when the containing definition block is loaded. This means that the targets of all stores, loads, and references to the locals, arguments, or constant terms are in the same name scope as the invoking method.

Example 1: Note

```
Package () {
    3,
    9,
    “ACPI 1.0 COMPLIANT”,
    Package () {
        “CheckSum=>”,
        Package () {
            7,
            9
        }
    },
    0
}
```

Example 2: This example defines and initializes a two-dimensional array.

```
Package () {
    Package () {11, 12, 13},
    Package () {21, 22, 23}
}
```

Example 3: This example is a legal encoding, but of no apparent use.

```
Package (){}
```

Example 4: This encoding allocates space for ten things to be defined later (see the **Name** and **Index** term definitions).

```
Package (10) {}
```

Note: The ability to create variable-sized packages is new in ACPI 2.0. ACPI 1.0 only allowed fixed-size packages with up to 255 elements.

16.2.3.6.3 Literal Data Terms

Literal Data terms include:

- Integers
- Strings
- Constant data terms

16.2.3.6.3.1 Integers

```
LeadDigitChar                 := ‘1’-‘9’
OctalDigitChar                := ‘0’-‘7’
HexDigitChar                  := DigitChar | ‘A’-‘F’ | ‘a’-‘f’


Integer                       := DecimalConst | OctalConst | HexConst
DecimalConst                  := LeadDigitChar | <DecimalConst DigitChar>
OctalConst                    := ‘0’ | <OctalConst OctalDigitChar>
HexConst                      := <0x HexDigitChar> | <0X HexDigitChar> | <HexConst
                                  HexDigitChar>
ByteConst                     := Integer=>0x00-0xff
WordConst                     := Integer=>0x0000-0xffff
DWordConst                    := Integer=>0x00000000-0xffffffff
QWordConst                    := Integer=>0x0000000000000000-0xffffffffffffffff
```

Numeric constants can be specified in decimal, octal, or hexadecimal. Octal constants are preceded by a leading zero (0), and hexadecimal constants are preceded by a leading zero and either a lower or upper case ‘x’. In some cases, the grammar specifies that the number must evaluate to an integer within a limited range, such as 0x00–0xFF, and so on.

16.2.3.6.3.2 Strings

```
String                        := ‘”’ AsciiCharList ‘”’
AsciiCharList                 := Nothing | <EscapeSequence AsciiCharList> | <AsciiChar
                                  AsciiCharList>
AsciiChar                     := 0x01-0x21 | 0x23-0x5B | 0x5D-0x7F
EscapeSeq                     := SimpleEscapeSeq | OctalEscapeSeq | HexEscapeSeq
SimpleEscapeSeq               :=  \' | \" | \a | \b | \f | \n | \r | \t | \v | \\
OctalEscapeSeq                :=  \ OctalDigit |
                                  \ OctalDigit OctalDigit |
                                  \ OctalDigit OctalDigit OctalDigit
HexEscapeSeq                  :=  \x HexDigit |
                                  \x HexDigit HexDigit
NullChar                      := 0x00
```

String literals consist of zero or more ASCII characters surrounded by double quotation marks ("). A String may not exceed 200 characters. A string literal represents a sequence of characters that, taken together, form a null-terminated string. After all adjacent strings in the constant have been concatenated, NullChar is appended.

Since String Literals are defined to contain only non-null ASCII values, both Hex and Octal escape sequence values must be non-null values in the ASCII range 0x01 through 0x7F. For arbitrary byte data (outside the range of ASCII values), the **Buffer** object should be used instead.

Since the backslash is used as the escape character and also the namespace root prefix, any string literals that are to contain a fully qualified namepath from the root of the namespace must use the double backslash to indicate this:

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

The double backslash is only required within quoted string literals.

Since double quotation marks are used close a string, a special escape sequence (\") is used to allow quotation marks within strings. Other escape sequences are listed in the table below:

Escape Sequence	ASCII Character
\a	0x07 (BEL)
\b	0x08 (BS)
\f	0x0C (FF)

Escape Sequence	ASCII Character
\n	0x0A (LF)
\r	0x0D (CR)
\t	0x09 (TAB)
\v	0x0B (VT)
\"	0x22 (")
\'	0x27 (')
\\	0x5C (\)

Since literal strings are read-only constants, the following ASL statement (for example) is not supported:

```
Store(“ABC”, ”DEF”)
```

However, the following sequence of statements is supported:

```
Name(STR, ”DEF”)
...

Store(“ABC”, STR)
```

16.2.3.6.3.3 Constant Data Terms

```
ConstTerm                      := Zero | One | Ones | Revision
```

The constant declaration terms are **Zero**, **One**, **Ones**, and **Revision**.

16.2.3.6.3.3.1 Zero (Constant Zero Object)

The constant **Zero** object is an object of type Integer that will always read as all bits clear. Writes to this object are not allowed.

16.2.3.6.3.3.2 One (Constant One Object)

The constant **One** object is an object of type Integer that will always read the LSB as set and all other bits as clear (that is, the value of 1). Writes to this object are not allowed.

16.2.3.6.3.3.3 Ones (Constant Ones Object)

The constant **Ones** object is an object of type Integer that will always read as all bits set. Writes to this object are not allowed.

16.2.3.6.3.3.4 Revision (Constant Revision Object)

The constant **Revision** object is an object of type Integer that will always read as the revision of the AML interpreter.

16.2.3.6.4 Data Macros

The data macros are:

EISAID terms

ResourceTemplate terms

Unicode term

16.2.3.6.4.1 EISAID Macro (Convert EISA ID String To Integer)

```
EISAIDTerm                    := EISAID(
                                     EISAIDString          //String
                                 ) => DWordConst
```

Converts *EISAIDString*, a 7-character text string argument, into its corresponding 4-byte numeric EISA ID encoding. It can be used when declaring IDs for devices that have EISA IDs. The *EISAIDString* must be of the form “UUUNNNN”, where “U” is an uppercase letter and “N” is a hexadecimal digit. No asterisks or other characters are allowed in the string. For example, EISAID(“PNP0C09”) is a valid invocation of the macro.

16.2.3.6.4.2 ResourceTemplate Macro (Convert Resource To Buffer)

```
ResourceTemplateTerm          := ResourceTemplate() {ResourceMacroList} => Buffer
```

For a full definition of the ResourceTemplateTerm macro, see section 6.4.1, “ASL Macros for Resource Descriptors.”

16.2.3.6.4.3 Unicode Macro (Convert Ascii String To Unicode)

```
UnicodeTerm                   := Unicode(
                                     ASCIIString           //String
                                 ) => Buffer
```

This macro will convert an ASCII string to a Unicode string contained in a buffer. The format of the Unicode string is 16 bits per character, with a 16-bit null terminator.

16.2.3.7 Miscellaneous Objects

Miscellaneous objects include:

- Debug objects
- ArgX objects
- LocalX objects

16.2.3.7.1 Debug Data Object

```
DebugTerm                     := Debug
```

The debug data object is a virtual data object. Writes to this object provide debugging information. On at least debug versions of the interpreter, any writes into this object are appropriately displayed on the system’s native kernel debugger. All writes to the debug object are otherwise benign. If the system is in use without a kernel debugger, then writes to the debug object are ignored. The following table relates the ASL term types that can be written to the Debug object to the format of the information on the kernel debugger display.

Table 16-13 Debug Object Display Formats

ASL Term Type	Display Format
Numeric data object	All digits displayed in hexadecimal format.
String data object	String is displayed.
Object reference	Information about the object is displayed (for example, object type and object name), but the object is not evaluated.

The Debug object is a write-only object; attempting to read from the debug object is not supported.

16.2.3.7.2 Arg*x* (Method Argument Data Objects)

```
ArgTerm                        = Arg0 | Arg1 | Arg2 | Arg3 | Arg4 | Arg5 | Arg6
```

Up to 7 argument-object references can be passed to a control method. On entry to a control method, only the argument objects that are passed are usable.

16.2.3.7.3 Local*x* (Method Local Data Objects)

```
LocalTerm                      := Local0 | Local1 | Local2 | Local3 | Local4 | Local5 |
                                   Local6 | Local7
```

Up to 8 local objects can be referenced in a control method. On entry to a control method, these objects are uninitialized and cannot be used until some value or reference is stored into the object. Once initialized, these objects are preserved in the scope of execution for that control method.

16.2.4 ASL Macros for Resource Descriptors

ASL includes some macros for creating resource descriptors. The ResourceTemplate macro creates a Buffer in which resource descriptor macros can be listed. The ResourceTemplate macro automatically generates an End descriptor and calculates the checksum for the resource template. The format for the ResourceTemplate macro is as follows:

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

The following is an example of how these macros can be used to create a resource template that can be returned from a _PRS control method:

```
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()
})
```

Occasionally, it is necessary to change a parameter of a descriptor in an existing resource template. To facilitate this, the descriptor macros optionally include a name declaration that can be used later to refer to the descriptor. When a name is declared with a descriptor, the ASL compiler will automatically create field names under the given name to refer to individual fields in the descriptor.

The offset returned by a reference to a resource descriptor field name is either in units of bytes (for 8-, 16-, 32-, and 64-bit field widths) or in bits (for all other field widths). In all cases, the returned offset is the integer offset (in either bytes or bits) of the name from the first byte (offset 0) of the parent resource template.

For example, given the above resource template, the following code changes the minimum and maximum addresses for the I/O descriptor named IO2:

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

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

The resource template macros for each of the resource descriptors are listed below, after the table that defines the resource descriptor. The resource template macros are formally defined in section 15, “Memory.”

The reserved names (such as _MIN and _MAX) for the fields of each resource descriptor are defined in the appropriate table entry of the table that defines that resource descriptor.

16.2.4.1 ASL Macro for IRQ Descriptor

The following macro generates a short IRQ descriptor with optional IRQ Information byte:

```
IRQ(
    Edge | Level,                 // _LL, _HE
    ActiveHigh | ActiveLow,       // _LL, _HE
    Shared | Exclusive | Nothing, // _SHR, Nothing defaults to Exclusive
    NameString | Nothing          // A name to refer back to this resource
    )
    {
        ByteConstExpr [, ByteConstExpr ...]  // List of IRQ numbers (valid values: 0-15)
    }
```

The following macro generates a short IRQ descriptor without optional IRQ Information byte:

```
IRQNoFlags(
    NameString | Nothing          // A name to refer back to this resource
    )
    {
        ByteConstExpr [, ByteConstExpr ...]  // list of IRQ numbers (valid values: 0-15)
    }
```

16.2.4.2 ASL Macro for DMA Descriptor

The following macro generates a short DMA descriptor.

```
DMA(
    Compatibility | TypeA | TypeB | TypeF,       // _TYP, DMA channel speed
    BusMaster | NotBusMaster,                    // _BM, Nothing defaults to BusMaster
    Transfer8 | Transfer16 | Transfer8_16        // _SIZ, Transfer size
    NameString | Nothing                         // A name to refer back to this resource
    )
    {
    ByteConstExpr [, ByteConstExpr ...]                  // List of channel numbers
                                                 //(valid values: 0-7)
    }
```

16.2.4.3 ASL Macro for Start-Dependent Function Descriptor

The following macro generates a Start-Dependent Function descriptor with the optional Priority byte:

```
StartDependentFn(
    ByteConstExpr,     // Compatibility priority (valid values: 0-2)
    ByteConstExpr      // Performance/Robustness priority  (valid values: 0-2)
    )
    {
    // List of descriptors for this dependent function
    }
```

The following macros generates a Start Dependent Function descriptor without the optional Priority byte:

```
StartDependentFnNoPri(
    )
    {
    Descriptors
    )
```

16.2.4.4 ASL Macro for End-Dependent Functions descriptor

The following macro generates an End-Dependent Functions descriptor:

```
EndDependentFn(
)
```

16.2.4.5 ASL Macro for I/O Port Descriptor

The following macro generates a short I/O descriptor:

```
IO(
    Decode16 | Decode10,          // _DEC
    WordConstExpr,                // _MIN, Address minimum
    WordConstExpr,                // _MAX, Address max
    ByteConstExpr,                // _ALN, Base alignment
    ByteConstExpr                 // _LEN, Range length
    NameString | Nothing          // A name to refer back to this resource
)
```

16.2.4.6 ASL Macro for Fixed I/O Port Descriptor

The following macro generates a short Fixed I/O descriptor:

```
FixedIO(
    WordConstExpr,            // _BAS, Address base
    ByteConstExpr             // _LEN, Range length
    NameString | Nothing      // A name to refer back to this resource
    )
```

16.2.4.7 ASL Macro for Short Vendor-Defined Descriptor

The following macro generates a short Vendor-Defined descriptor:

```
VendorShort(
    NameString | Nothing          // A name to refer back to this resource
    )
    {
    ByteConstExpr [, ByteConstExpr ...]   // List of bytes, up to 7 bytes
    }
```

16.2.4.8 ASL Macro for 24-bit Memory Descriptor

The following macro generates a long 24-bit Memory descriptor:

```
Memory24(
    ReadWrite | ReadOnly,         // _RW
    WordConstExpr,                // _MIN, Minimum base memory address [23:8]
    WordConstExpr,                // _MAX, Maximum base memory address [23:8]
    WordConstExpr,                // _ALN, Base alignment
    WordConstExpr                 // _LEN, Range length
    NameString | Nothing          // A name to refer back to this resource
)
```

16.2.4.9 ASL Macro for Long Vendor-Defined Descriptor

The following macro generates a long Vendor-Defined descriptor:

```
VendorLong(
   NameString | Nothing              // A name to refer back to this resource
   )
   {
   ByteConstExpr [, ByteConstExpr ...]   // List of bytes
   }
```

16.2.4.10 ASL Macro for 32-Bit Memory Descriptor

The following macro generates a long 32-bit Memory descriptor:

```
Memory32(
    ReadWrite | ReadOnly,         // _RW
    DWordConstExpr,               // _MIN, Minimum base memory address
    DWordConstExpr,               // _MAX, Maximum base memory address
    DWordConstExpr,               // _ALN, Base alignment
    DWordConstExpr                // _LEN, Range length
    NameString | Nothing          // A name to refer back to this resource
    )
```

16.2.4.11 ASL Macro for 32-bit Fixed Memory Descriptor

The following macro generates a long 32-bit Fixed Memory descriptor:

```
Memory32Fixed(
    ReadWrite | ReadOnly,         // _RW
    DWordConstExpr,               // _BAS, Range base
    DWordConstExpr                // _LEN, Range length
    NameString | Nothing          // A name to refer back to this resource
    )
```

16.2.4.12 ASL Macros for QWORD Address Space Descriptor

The following macro generates a QWORD Address descriptor with ResourceType = Memory:

```
QWordMemory(
    ResourceConsumer | ResourceProducer | Nothing,   // Nothing=>ResourceConsumer
    SubDecode | PosDecode | Nothing,                 // _DEC, Nothing=>PosDecode
    MinFixed | MinNotFixed | Nothing,                // _MIF, Nothing=>MinNotFixed
    MaxFixed | MaxNotFixed | Nothing,                // _MAF, Nothing=>MaxNotFixed
    Cacheable | WriteCombining | Prefetchable |
    NonCacheable | Nothing,                          // _MEM, Nothing=>NonCacheable
    ReadWrite | ReadOnly,                            // _RW, Nothing == ReadWrite
    QWordConstExpr,                                  // _GRA, Address granularity
    QWordConstExpr,                                  // _MIN, Address range minimum
    QWordConstExpr,                                  // _MAX, Address range max
    QWordConstExpr,                                  // _TRA, Translation
    QWordConstExpr,                                  // _LEN, Address range length
    ByteConstExpr | Nothing,                         // Resource Source Index;
                                                     // if Nothing, not generated
    StringData | Nothing                             // Resource Source;
                                                     // if Nothing, not generated
    NameString | Nothing                             // A name to refer back
                                                     // to this resource
    AddressRangeMemory | AddressRangeReserved |
    AddressRangeACPI | AddressRangeNVS | Nothing,    // _MTP, Nothing=>AddressRangeMemory
    TypeTranslation | TypeStatic | Nothing,          // _TTP, Nothing=>TypeStatic
    )
```

The following generates a QWORD Address descriptor with ResourceType = I/O:

```
QWORDIO(
    ResourceConsumer | ResourceProducer | Nothing,   // Nothing == ResourceConsumer
    MinFixed | MinNotFixed | Nothing,                // _MIF, Nothing => MinNotFixed
    MaxFixed | MaxNotFixed | Nothing,                // _MAF, Nothing => MaxNotFixed
    SubDecode | PosDecode | Nothing,                 // _DEC, Nothing => PosDecode
    ISAOnlyRanges | NonISAOnlyRanges |
    EntireRange | Nothing,                           // _RNG, Nothing => EntireRange
    QWordConstExpr,                                  // _GRA: Address granularity
    QWordConstExpr,                                  // _MIN: Address range minimum
    QWordConstExpr,                                  // _MAX: Address range max
    QWordConstExpr,                                  // _TRA: Translation
    QWordConstExpr,                                  // _LEN, Address range length
    ByteConstExpr | Nothing,                         // Resource Source Index;
                                                     // if Nothing, not generated
    StringData | Nothing                             // Resource Source;
                                                     // if Nothing, not generated
    NameString | Nothing                             // A name to refer back to this
                                                     // resource
    TypeTranslation | TypeStatic | Nothing,          // _TTP, Nothing=>TypeStatic
    SparseTranslation | DenseTranslation | Nothing   // _TRS, Nothing=>DenseTranslation
    )
```

16.2.4.13 ASL Macros for DWORD Address Space Descriptor

The following macro generates a DWORD Address descriptor with ResourceType = Memory:

```
DWordMemory(
    ResourceConsumer | ResourceProducer | Nothing,   // Nothing=>ResourceConsumer
    SubDecode | PosDecode | Nothing,                 // _DEC, Nothing=>PosDecode
    MinFixed | MinNotFixed | Nothing,                // _MIF, Nothing=>MinNotFixed
    MaxFixed | MaxNotFixed | Nothing,                // _MAF, Nothing=>MaxNotFixed
    Cacheable | WriteCombining | Prefetchable |
    NonCacheable | Nothing,                          // _MEM, Nothing=>NonCacheable
    ReadWrite | ReadOnly,                            // _RW, Nothing == ReadWrite
    DWordConstExpr,                                  // _GRA, Address granularity
    DWordConstExpr,                                  // _MIN, Address range minimum
    DWordConstExpr,                                  // _MAX, Address range max
    DWordConstExpr,                                  // _TRA, Translation
    DWordConstExpr,                                  // _LEN, Address range length
    ByteConstExpr | Nothing,                         // Resource Source Index;
                                                     // if Nothing, not generated
    StringData | Nothing                             // Resource Source;
                                                     // if Nothing, not generated
    NameString | Nothing                             // A name to refer back
                                                     // to this resource
    AddressRangeMemory | AddressRangeReserved |
    AddressRangeACPI | AddressRangeNVS | Nothing,    // _MTP, Nothing=>AddressRangeMemory
    TypeTranslation | TypeStatic | Nothing,          // _TTP, Nothing=>TypeStatic
    )
```

The following generates a DWORD Address descriptor with ResourceType = I/O:

```
DWordIO(
    ResourceConsumer | ResourceProducer | Nothing,   // Nothing == ResourceConsumer
    MinFixed | MinNotFixed | Nothing,                // _MIF, Nothing => MinNotFixed
    MaxFixed | MaxNotFixed | Nothing,                // _MAF, Nothing => MaxNotFixed
    SubDecode | PosDecode | Nothing,                 // _DEC, Nothing => PosDecode
    ISAOnlyRanges | NonISAOnlyRanges |
    EntireRange | Nothing,                           // _RNG, Nothing => EntireRange
    DWordConstExpr,                                  // _GRA: Address granularity
    DWordConstExpr,                                  // _MIN: Address range minimum
    DWordConstExpr,                                  // _MAX: Address range max
    DWordConstExpr,                                  // _TRA: Translation
    DWordConstExpr,                                  // _LEN, Address range length
    ByteConstExpr | Nothing,                         // Resource Source Index;
                                                     // if Nothing, not generated
    StringData | Nothing                             // Resource Source;
                                                     // if Nothing, not generated
    NameString | Nothing                             // A name to refer back to this
                                                     // resource
    TypeTranslation | TypeStatic | Nothing,          // _TTP, Nothing=>TypeStatic
    SparseTranslation | DenseTranslation | Nothing   // _TRS, Nothing=>DenseTranslation
    )
```

16.2.4.14 ASL Macros for WORD Address Descriptor

The following macro generates a WORD Address descriptor with ResourceType = I/O

```
WORDIO(
    ResourceConsumer | ResourceProducer | Nothing,       // Nothing=>ResourceConsumer
    MinFixed | MinNotFixed | Nothing,                    // _MIF, Nothing=>MinNotFixed
    MaxFixed | MaxNotFixed | Nothing,                    // _MAF, Nothing=>MaxNotFixed
    SubDecode | PosDecode | Nothing,                     // _DEC, Nothing=>PosDecode
    ISAOnlyRanges | NonISAOnlyRanges | EntireRange,      // _RNG
    WordConstExpr,                                       // _GRA: Address granularity
    WordConstExpr,                                       // _MIN: Address range minimum
    WordConstExpr,                                       // _MAX: Address range max
    WordConstExpr,                                       // _TRA: Translation
    WordConstExpr,                                       // _LEN, Address range length
    ByteConstExpr | Nothing,                             // Resource Source Index;
                                                         // if Nothing, not generated
    StringData | Nothing                                 // Resource Source;
                                                         // if Nothing, not generated
    NameString | Nothing                                 // A name to refer back
                                                         // to this resource
    TypeTranslation | TypeStatic | Nothing,              // _TTP, Nothing=>TypeStatic
    SparseTranslation | DenseTranslation | Nothing       // _TRS,
                                                         // Nothing=>DenseTranslation
    )
```

The following macros generates a WORD Address descriptor with ResourceType = BusNumber:

```
WordBusNumber(
    ResourceConsumer | ResourceProducer | Nothing,      // Nothing=>ResourceConsumer
    MinFixed | MinNotFixed | Nothing,                   // _MIF, Nothing=>MinNotFixed
    MaxFixed | MaxNotFixed | Nothing,                   // _MAF, Nothing=>MaxNotFixed
    SubDecode | PosDecode | Nothing,                    // _DEC, Nothing=>PosDecode
    WordConstExpr,                                      // _GRA, Address granularity
    WordConstExpr,                                      // _MIN, Address range minimum
    WordConstExpr,                                      // _MAX, Address range max
    WordConstExpr,                                      // _TRA: Translation
    WordConstExpr,                                      // _LEN, Address range length
    ByteConstExpr | Nothing,                            // Resource Source Index;
                                                        // if Nothing, not generated
    StringData | Nothing                                // Resource Source;
                                                        // if Nothing, not generated
    NameString | Nothing                                // A name to refer back
                                                        // to this resource
    )
```

16.2.4.15 ASL Macro for Extended Interrupt Descriptor

The following macro generates an Extended Interrupt descriptor:

```
Interrupt(
    ResourceConsumer | ResourceProducer | Nothing,  // Nothing=>ResourceConsumer
    Edge | Level,                                   // _HE
    ActiveHigh | ActiveLow ,                        // __LL
    Shared | Exclusive | Nothing,                   // _SHR: Nothing=>Exclusive
    ByteConstExpr | Nothing,                        // Resource Source Index;
                                                    //   if Nothing, not generated
    StringData | Nothing                            // Resource Source;
                                                    //   if Nothing, not generated
    NameString | Nothing                            // A name to refer back
                                                    // to this resource
    )
    {
    DWordConstExpr [, DWordConstExpr ...]           // _INT, list of interrupt numbers
    }
```

16.2.4.16 ASL Macro for Generic Register Descriptor

The following macro generates a Generic Register descriptor:

```
Register(
    AddressSpaceKeyword,  // _ASI, Address Space ID
    ByteConstExpr,        // _RBW, Register Bit Width
    ByteConstExpr,        // _RBO, Register Bit Offset
    QWordConstExpr        // _ADR, Register Address
)
```

17 ACPI Machine Language (AML) Specification

This section formally defines the ACPI Control Method Machine Language (AML) language. AML is the ACPI Control Method virtual machine language, machine code for a virtual machine that is supported by an ACPI-compatible OS. ACPI control methods can be written in AML, but humans ordinarily write control methods in ASL.

AML is the language processed by the ACPI AML interpreter. It is primarily a declarative language. It's best not to think of it as a stream of code, but rather as a set of declarations that the ACPI AML interpreter will compile into the ACPI Namespace at definition block load time. For example, notice that DefByte allocates an anonymous integer variable with a byte-size initial value in ACPI namespace, and passes in an initial value. The byte in the AML stream that defines the initial value is *not* the address of the variable's storage location.

An OEM or BIOS vendor needs to write ASL and be able to single-step AML for debugging. (Debuggers and other ACPI control method language tools are expected to be AML-level tools, not source-level tools.) An ASL translator implementer must understand how to read ASL and generate AML. An AML interpreter author must understand how to execute AML.

AML and ASL are ***different languages*** though they are closely related.

All ACPI-compatible operating systems must support AML. A given user can define some arbitrary source language (to replace ASL) and write a tool to translate it to AML. However, the ACPI group will support a single translator for a single language, ASL.

17.1 Notation Conventions

The notation conventions in the table below help the reader to interpret the AML formal grammar.

Table 17-1 AML Grammar Notation Conventions

Notation Convention	Description	Example
0x*dd*	Refers to a byte value expressed as 2 hexadecimal digits.	0x21
Number in bold.	Denotes the encoding of the AML term.	
Term => Evaluated Type	Shows the resulting type of the evaluation of Term.	
Single quotes (' ')	Indicate constant characters.	'A' => 0x41
Term := Term Term …	The term to the left of := can be expanded into the sequence of terms on the right.	aterm := bterm cterm means that aterm can be expanded into the two-term sequence of bterm followed by cterm.
Term Term Term …	Terms separated from each other by spaces form an ordered list.	

Table 17-1 AML Grammar Notation Conventions *(continued)*

Notation Convention	Description	Example
Angle brackets (< >)	Used to group items.	<a b> | <c d> means either a b or c d.
Bar symbol (|)	Separates alternatives.	aterm := bterm | [cterm dterm] means the following constructs are possible: bterm cterm dterm aterm := [bterm | cterm] dterm means the following constructs are possible: bterm dterm cterm dterm
Dash character (**-**)	Indicates a range.	1**-**9 means a single digit in the range 1 to 9 inclusive.
Parenthesized term following another term.	The parenthesized term is the repeat count of the previous term.	aterm(3) means aterm aterm aterm. bterm(*n*) means *n* number of bterms.

17.2 AML Grammar Definition

This section defines the byte values that make up an AML byte stream.

```
AMLCode := DefBlockHdr TermList
DefBlockHdr                   := TableSig TableLen SpecCompliance CheckSum OemID
                                  OemTableID OemRev CreatorID CreatorRev
TableSig                      := DWordConst
                                  //As defined in section 5.2.3.
TableLen                      := DwordConst
                                  //Length of the table in bytes including the block
                                  //header.
SpecCompliance                := ByteConst
                                  //The revision of the structure.
CheckSum                      := ByteConst
                                  //Byte checksum of the entire table.
OemID   := ByteConst(6)

                                  //OEM ID of up to 6 characters. If the OEM ID is
                                  //shorter than 6 characters, it can be terminated
                                  //with a NULL character.
OemTableID                    := ByteConst(8)
                                  //OEM Table ID of up to 8 characters. If the OEM
                                  //Table ID is shorter than 8 characters, it can be
                                  //terminated with a NULL character.
OemRev  := DWordConst

                                  //OEM Table Revision.
CreatorID                     := DWordConst
                                  //Vendor ID of the ASL compiler.
CreatorRev                    := DWordConst
                                  //Revision of the ASL compiler.
```

The AML encoding can be categorized in the following groups:

- Name objects encoding
- Data objects encoding
- Package length encoding
- Term objects encoding
- Miscellaneous objects encoding

17.2.1 Name Objects Encoding

```
LeadNameChar                  := ‘A’-‘Z’ | ‘_’
DigitChar                     := ‘0’-‘9’
NameChar                      := DigitChar | LeadNameChar
RootChar                      := ‘\’
ParentPrefixChar              := ‘^’

‘A’-‘Z’                       := 0x41-0x5a
‘_’                           := 0x5f
‘0’-‘9’                       := 0x30-0x39
‘\’                           := 0x5c
‘^’                           := 0x5e

NameSeg                       := <LeadNameChar NameChar NameChar NameChar>
                                   // Notice that NameSegs shorter than 4 characters are
                                   // filled with trailing ‘_’s.
NameString                    := <RootChar NamePath> | <PrefixPath NamePath>
PrefixPath                    := Nothing | <‘^’ PrefixPath>
NamePath                      := NameSeg | DualNamePath | MultiNamePath | NullName

DualNamePath                  := DualNamePrefix NameSeg NameSeg
DualNamePrefix                := 0x2e
MultiNamePath                 := MultiNamePrefix SegCount NameSeg(SegCount)
MultiNamePrefix               := 0x2f
SegCount                      := ByteData
                                   // SegCount can be from 1 to 255.
                                   // MultiNamePrefix(35) => 0x2f 0x23
                                   // and following by 35 NameSegs.
                                   // So, the total encoding length
                                   // will be 1 + 1 + 35*4 = 142.
                                   // Notice that:
                                   //   DualNamePrefix NameSeg NameSeg
                                   // has a smaller encoding than the
                                   // equivalent encoding of:
                                   //    MultiNamePrefix(2) NameSeg NameSeg
SimpleName                    := NameString | ArgObj | LocalObj
SuperName                     := SimpleName | DebugObj | Type6Opcode
NullName                      := 0x00
Target                        := SuperName | NullName
```

17.2.2 Data Objects Encoding

```
ComputationalData             := ByteConst | WordConst | DwordConst | QwordConst | String
                                  | ConstObj | RevisionOp | DefBuffer
DataObject                    := ComputationalData | DefPackage | DefVarPackage
DataRefObject                 := DataObject | ObjectReference | DDBHandle

ByteConst                     := BytePrefix ByteData
BytePrefix                    := 0x0a
WordConst                     := WordPrefix WordData
WordPrefix                    := 0x0b
DWordConst                    := DWordPrefix DWordData
DWordPrefix                   := 0x0c
QWordConst                    := QWordPrefix QWordData
QWordPrefix                   := 0x0e
String                        := StringPrefix AsciiCharList NullChar
StringPrefix                  := 0x0d
ConstObj                      := ZeroOp | OneOp | OnesOp
ByteList                      := Nothing | <ByteData ByteList>
ByteData                      := 0x00-0xff
WordData                      := ByteData[0:7] ByteData[8:15]
                                  // 0x0000-0xffff
DWordData                     := WordData[0:15] WordData[16:31]
                                  // 0x00000000-0xffffffff
QWordData                     := DwordData[0:31] DwordData[32:63]
                                  // 0x0000000000000000-0xffffffffffffffff
AsciiCharList                 := Nothing | <AsciiChar AsciiCharList>
AsciiChar                     := 0x01-0x7f
NullChar                      := 0x00
ZeroOp                        := 0x00
OneOp                         := 0x01
OnesOp                        := 0xff
RevisionOp                    := ExtOpPrefix 0x30
ExtOpPrefix                   := 0x5b
```

17.2.3 Package Length Encoding

```
PkgLength                     := PkgLeadByte |
                                  <PkgLeadByte ByteData> |
                                  <PkgLeadByte ByteData ByteData> |
                                  <PkgLeadByte ByteData ByteData ByteData>
PkgLeadByte                   := <bit 7-6: follow ByteData count>
                                  <bit 5-4: reserved>
                                  <bit 3-0: least significant package length byte>
                                  // Note: The high 2 bits of the first byte reveal how
                                  //  many follow bytes are in the PkgLength. If the
                                  //  PkgLength has only one byte, bit 0 through 5 are
                                  //  used to encode the package length (in other words,
                                  //  values 0-63).
                                  //  If the package length value is more than
                                  //  63, more than one byte must be used for the
                                  //  encoding in which case bit 5 and 4 of the
                                  //  PkgLeadByte are reserved and must be zero. If
                                  //  multiple bytes encoding is used, bits 3-0 of the
                                  //  PkgLeadByte become the least significant 4 bits
                                  //  of the resulting package length value. The next
                                  //  ByteData will become the next least significant
                                  //  8 bits of the resulting value and so on.
```

17.2.4 Term Objects Encoding

```
TermObj                       := NameSpaceModifierObj | NamedObj | Type1Opcode |
                                  Type2Opcode
TermList                      := Nothing | <TermObj TermList>

TermArg                       := Type2Opcode | DataObject | ArgObj | LocalObj
UserTermObj                   := NameString TermArgList
TermArgList                   := Nothing | <TermArg TermArgList>

ObjectList                    := Nothing | <Object ObjectList>
Object                        := NameSpaceModifierObj | NamedObj
```

17.2.4.1 Namespace Modifier Objects Encoding

```
NameSpaceModifierObj          := DefAlias | DefName | DefScope

DefAlias                      := AliasOp NameString NameString
AliasOp                       := 0x06

DefName                       := NameOp NameString DataRefObject
NameOp                        := 0x08

DefScope                      := ScopeOp PkgLength NameString TermList
ScopeOp                       := 0x10
```

17.2.4.2 Named Objects Encoding

```
NamedObj                      := DefBankField | DefCreateBitField | DefCreateByteField |
                                  DefCreateDWordField | DefCreateField |
                                  DefCreateQWordField | DefCreateWordField |
                                  DefDataRegion | DefDevice | DefEvent | DefField |
                                  DefIndexField | DefMethod | DefMutex | DefOpRegion |
                                  DefPowerRes | DefProcessor | DefThermalZone

DefBankField                  := BankFieldOp PkgLength NameString NameString BankValue
                                  FieldFlags FieldList
BankFieldOp                   := ExtOpPrefix 0x87
BankValue                     := TermArg=>Integer
FieldFlags                    := ByteData
                                  // bit 0-3: AccessType
                                  //    0: AnyAcc
                                  //    1: ByteAcc
                                  //    2: WordAcc
                                  //    3: DWordAcc
                                  //    4: QWordAcc
                                  //    5: BufferAcc
                                  //    6: Reserved
                                  //    7-15: Reserved
                                  // bit 4: LockRule
                                  //    0: NoLock
                                  //    1: Lock
                                  // bit 5-6: UpdateRule
                                  //    0: Preserve
                                  //    1: WriteAsOnes
                                  //    2: WriteAsZeros
                                  // bit 7: reserved (must be 0)
FieldList                     := Nothing | <FieldElement FieldList>
FieldElement                  := NamedField | ReservedField | AccessField
NamedField                    := NameSeg PkgLength
ReservedField                 := 0x00 PkgLength
AccessField                   := 0x01 AccessType AccessAttrib
AccessType                    := ByteData
                                  // Same as AccessType bits of FieldFlags
```

```
AccessAttrib                  := ByteData
                                  // If AccessType is BufferAcc for the SMB OpRegion,
                                  // AccessAttrib can be one of the following values:
                                  //   0x02 – SMBQuick
                                  //   0x04 – SMBSendReceive
                                  //   0x06 – SMBByte
                                  //   0x08 – SMBWord
                                  //   0x0a – SMBBlock
                                  //   0x0c – SMBProcessCall
                                  //   0x0d - SMBBlockProcessCall

DefCreateBitField             := CreateBitFieldOp SourceBuff BitIndex NameString
CreateBitFieldOp              := 0x8d
SourceBuff                    := TermArg=>Buffer
BitIndex                      := TermArg=>Integer

DefCreateByteField            := CreateByteFieldOp SourceBuff ByteIndex NameString
CreateByteFieldOp             := 0x8c
ByteIndex                     := TermArg=>Integer

DefCreateDWordField           := CreateDWordFieldOp SourceBuff ByteIndex NameString
CreateDWordFieldOp            := 0x8a

DefCreateField                := CreateFieldOp SourceBuff BitIndex NumBits NameString
CreateFieldOp                 := ExtOpPrefix 0x13
NumBits                       := TermArg=>Integer

DefCreateQWordField           := CreateQWordFieldOp SourceBuff ByteIndex NameString
CreateQWordFieldOp            := 0x8f

DefCreateWordField            := CreateWordFieldOp SourceBuff ByteIndex NameString
CreateWordFieldOp             := 0x8b

DefDataRegion                 := DataRegionOp NameString TermArg TermArg TermArg
DataRegionOp                  := ExOpPrefix 0x88

DefDevice                     := DeviceOp PkgLength NameString ObjectList
DeviceOp                      := ExtOpPrefix 0x82

DefEvent                      := EventOp NameString
EventOp                       := ExtOpPrefix 0x02

DefField                      := FieldOp PkgLength NameString FieldFlags FieldList
FieldOp                       := ExtOpPrefix 0x81

DefIndexField                 := IndexFieldOp PkgLength NameString NameString FieldFlags
                                  FieldList
IndexFieldOp                  := ExtOpPrefix 0x86

DefMethod                     := MethodOp PkgLength NameString MethodFlags TermList
MethodOp                      := 0x14
MethodFlags                   := ByteData
                                  // bit 0-2: ArgCount (0-7)
                                  // bit 3: SerializeFlag
                                  //    0: NotSerialized
                                  //    1: Serialized
                                  // bit 4-7: SyncLevel (0x00-0x0f)

DefMutex                      := MutexOp NameString SyncFlags
MutexOp                       := ExtOpPrefix 0x01
SyncFlags                     := ByteData
                                  // bit 0-3: SyncLevel (0x00-0x0f)
                                  // bit 4-7: reserved (must be 0)
```

```
DefOpRegion                   := OpRegionOp NameString RegionSpace RegionOffset RegionLen
OpRegionOp                    := ExtOpPrefix 0x80
RegionSpace                   := ByteData
                                  // 0x00: SystemMemory
                                  // 0x01: SystemIO
                                  // 0x02: PCI_Config
                                  // 0x03: EmbeddedControl
                                  // 0x04: SMBus
                                  // 0x05: CMOS
                                  // 0x06: PciBarTarget
                                  // 0x80-0xff: user defined
RegionOffset                  := TermArg=>Integer
RegionLen                     := TermArg=>Integer

DefPowerRes                   := PowerResOp PkgLength NameString SystemLevel
                                  ResourceOrder ObjectList
PowerResOp                    := ExtOpPrefix 0x84
SystemLevel                   := ByteData
ResourceOrder                 := WordData

DefProcessor                  := ProcessorOp PkgLength NameString ProcID PblkAddr PblkLen
                                  ObjectList
ProcessorOp                   := ExtOpPrefix 0x83
ProcID                        := ByteData
PblkAddr                      := DwordData
PblkLen                       := ByteData

DefThermalZone                := ThermalZoneOp PkgLength NameString ObjectList
ThermalZoneOp                 := ExtOpPrefix 0x85
```

17.2.4.3 Type 1 Opcodes Encoding

```
Type1Opcode                   := DefBreak | DefBreakPoint | DefContinue | DefFatal |
                                  DefIfElse | DefLoad | DefNoop | DefNotify | DefRelease |
                                  DefReset | DefReturn | DefSignal | DefSleep | DefStall |
                                  DefUnload | DefWhile

DefBreak                      := BreakOp
BreakOp                       := 0xa5

DefBreakPoint                 := BreakPointOp
BreakPointOp                  := 0xcc

DefContinue                   := ContinueOp
ContinueOp                    := 0x9f

DefElse                       := Nothing | <ElseOp PkgLength TermList>
ElseOp                        := 0xa1

DefFatal                      := FatalOp FatalType FatalCode FatalArg
FatalOp                       := ExtOpPrefix 0x32
FatalType                     := ByteData
FatalCode                     := DwordData
FatalArg                      := TermArg=>Integer

DefIfElse                     := IfOp PkgLength Predicate TermList DefElse
IfOp                          := 0xa0
Predicate                     := TermArg=>Integer

DefLoad                       := LoadOp NameString DDBHandleObject
LoadOp                        := ExtOpPrefix 0x20
DDBHandleObject               := SuperName

DefNoop                       := NoopOp
NoopOp                        := 0xa3

DefNotify                     := NotifyOp NotifyObject NotifyValue
NotifyOp                      := 0x86
NotifyObject                  := SuperName=>ThermalZone|Processor|Device
NotifyValue                   := TermArg=>Integer
```

```
DefRelease                    := ReleaseOp MutexObject
ReleaseOp                     := ExtOpPrefix 0x27
MutexObject                   := SuperName

DefReset                      := ResetOp EventObject
ResetOp                       := ExtOpPrefix 0x26
EventObject                   := SuperName

DefReturn                     := ReturnOp ArgObject
ReturnOp                      := 0xa4
ArgObject                     := TermArg=>DataRefObject

DefSignal                     := SignalOp EventObject
SignalOp                      := ExtOpPrefix 0x24

DefSleep                      := SleepOp MsecTime
SleepOp                       := ExtOpPrefix 0x22
MsecTime                      := TermArg=>Integer

DefStall                      := StallOp UsecTime
StallOp                       := ExtOpPrefix 0x21
UsecTime                      := TermArg=>ByteData

DefUnload                     := UnloadOp DDBHandleObject
UnloadOp                      := ExtOpPrefix 0x2a

DefWhile                      := WhileOp PkgLength Predicate TermList
WhileOp                       := 0xa2
```

17.2.4.4 Type 2 Opcodes Encoding

```
Type2Opcode                   := DefAcquire | DefAdd | DefAnd | DefBuffer | DefConcat |
                                  DefConcatRes | DefCondRefOf | DefCopyObject |
                                  DefDecrement | DefDerefOf | DefDivide |
                                  DefFindSetLeftBit | DefFindSetRightBit | DefFromBCD |
                                  DefIncrement | DefIndex | DefLAnd | DefLEqual |
                                  DefLGreater | DefLGreaterEqual | DefLLess |
                                  DefLLessEqual | DefMid | DefLNot | DefLNotEqual |
                                  DefLoadTable | DefLOr | DefMatch | DefMod |
                                  DefMultiply | DefNAnd | DefNOr | DefNot |
                                  DefObjectType | DefOr | DefPackage | DefVarPackage |
                                  DefRefOf | DefShiftLeft | DefShiftRight | DefSizeOf |
                                  DefStore | DefSubtract | DefToBCD | DefToBuffer |
                                  DefToDecimalString | DefToHexString | DefToInteger |
                                  DefToString | DefWait | DefXOr | UserTermObj
Type6Opcode                   := DefRefOf | DefDerefOf | DefIndex | UserTermObj

DefAcquire                    := AcquireOp MutexObject Timeout
AcquireOp                     := ExtOpPrefix 0x23
Timeout                       := WordData

DefAdd                        := AddOp Operand Operand Target
AddOp                         := 0x72
Operand                       := TermArg=>Integer

DefAnd                        := AndOp Operand Operand Target
AndOp                         := 0x7b

DefBuffer                     := BufferOp PkgLength BufferSize ByteList
BufferOp                      := 0x11
BufferSize                    := TermArg=>Integer

DefConcat                     := ConcatOp Data Data Target
ConcatOp                      := 0x73
Data                          := TermArg=>ComputationalData
```

```
DefConcatRes                  := ConcatResOp BufData BufData Target
ConcatResOp                   := 0x84
BufData                       := TermArg=>Buffer

DefCondRefOf                  := CondRefOfOp SuperName Target
CondRefOfOp                   := ExtOpPrefix 0x12

DefCopyObject                 := CopyObjectOp TermArg SimpleName
CopyObjectOp                  := 0x9d

DefDecrement                  := DecrementOp SuperName
DecrementOp                   := 0x76

DefDerefOf                    := DerefOfOp ObjReference
DerefOfOp                     := 0x83
ObjReference                  := TermArg=>ObjectReference|String
                                  //ObjectReference is an object produced by terms
                                  //such as Index, RefOf or CondRefOf.

DefDivide                     := DivideOp Dividend Divisor Remainder Quotient
DivideOp                      := 0x78
Dividend                      := TermArg=>Integer
Divisor                       := TermArg=>Integer
Remainder                     := Target
Quotient                      := Target

DefFindSetLeftBit             := FindSetLeftBitOp Operand Target
FindSetLeftBitOp              := 0x81


DefFindSetRightBit            := FindSetRightBitOp Operand Target
FindSetRightBitOp             := 0x82

DefFromBCD                    := FromBCDOp BCDValue Target
FromBCDOp                     := ExtOpPrefix 0x28
BCDValue                      := TermArg=>Integer

DefIncrement                  := IncrementOp SuperName
IncrementOp                   := 0x75

DefIndex                      := IndexOp BuffPkgStrObj IndexValue Target
IndexOp := 0x88
BuffPkgStrObj                 := TermArg=>Buffer, Package or String
IndexValue                    := TermArg=>Integer

DefLAnd                       := LandOp Operand Operand
LandOp                        := 0x90

DefLEqual                     := LequalOp Operand Operand
LequalOp                      := 0x93

DefLGreater                   := LgreaterOp Operand Operand
LgreaterOp                    := 0x94

DefLGreaterEqual              := LgreaterEqualOp Operand Operand
LgreaterEqualOp               := LnotOp LlessOp

DefLLess                      := LlessOp Operand Operand
LlessOp := 0x95

DefLLessEqual                 := LlessEqualOp Operand Operand
LlessEqualOp                  := LnotOp LgreaterOp
```

```
DefLNot                       := LnotOp Operand
LnotOp                        := 0x92

DefLNotEqual                  := LnotEqualOp Operand Operand
LnotEqualOp                   := LnotOp LequalOp

DefLoadTable                  := LoadTableOp TermArg TermArg TermArg TermArg TermArg
                                  TermArg
LoadTableOp                   := ExtOpPrefix 0x1F

DefLOr                        := LorOp Operand Operand
LorOp                         := 0x91

DefMatch                      := MatchOp SearchPkg MatchOpcode Operand MatchOpcode
                                  Operand StartIndex
MatchOp                       := 0x89
SearchPkg                     := TermArg=>Package
MatchOpcode                   := ByteData
                                  // 0: MTR
                                  // 1: MEQ
                                  // 2: MLE
                                  // 3: MLT
                                  // 4: MGE
                                  // 5: MGT

StartIndex                    := TermArg=>Integer

DefMid                        := MidOp MidObj TermArg TermArg Target
MidOp                         := 0x9E
MidObj                        := TermArg=>Buffer|String

DefMod                        := ModOp Dividend Divisor Target
ModOp                         := 0x85

DefMultiply                   := MultiplyOp Operand Operand Target
MultiplyOp                    := 0x77

DefNAnd                       := NandOp Operand Operand Target
NandOp                        := 0x7c

DefNOr                        := NorOp Operand Operand Target
NorOp                         := 0x7e

DefNot                        := NotOp Operand Target
NotOp                         := 0x80

DefObjectType                 := ObjectTypeOp SuperName
ObjectTypeOp                  := 0x8e

DefOr                         := OrOp Operand Operand Target
OrOp                          := 0x7d

DefPackage                    := PackageOp PkgLength NumElements PackageElementList
PackageOp                     := 0x12
DefVarPackage                 := VarPackageOp PkgLength VarNumElements PackageElementList
VarPackageOp                  := 0x13
NumElements                   := ByteData
VarNumElements                := TermArg=>Integer
PackageElementList            := Nothing | <PackageElement PackageElementList>
PackageElement                := DataRefObject | NameString

DefRefOf                      := RefOfOp SuperName
RefOfOp                       := 0x71

DefShiftLeft                  := ShiftLeftOp Operand ShiftCount Target
ShiftLeftOp                   := 0x79
ShiftCount                    := TermArg=>Integer

DefShiftRight                 := ShiftRightOp Operand ShiftCount Target
ShiftRightOp                  := 0x7a
```

```
DefSizeOf                     := SizeOfOp SuperName
SizeOfOp                      := 0x87

DefStore                      := StoreOp TermArg SuperName
StoreOp                       := 0x70

DefSubtract                   := SubtractOp Operand Operand Target
SubtractOp                    := 0x74

DefToBCD                      := ToBCDOp Operand Target
ToBCDOp                       := ExtOpPrefix 0x29

DefToBuffer                   := ToBufferOp Operand Target
ToBufferOp                    := 0x96

DefToDecimalString            := ToDecimalStringOp Operand Target
ToDecimalStringOp             := 0x97

DefToHexString                := ToHexStringOp Operand Target
ToHexStringOp                 := 0x98

DefToInteger                  := ToIntegerOp Operand Target
ToIntegerOp                   := 0x99

DefToString                   := ToStringOp TermArg LengthArg Target
LengthArg                     := TermArg=>Integer
ToStringOp                    := 0x9c

DefWait                       := WaitOp EventObject Operand
WaitOp                        := ExtOpPrefix 0x25

DefXOr                        := XorOp Operand Operand Target
XorOp                         := 0x7f
```

17.2.5 Miscellaneous Objects Encoding

Miscellaneous objects include:

- Arg objects
- Local objects
- Debug objects

17.2.5.1 Arg Objects Encoding

```
ArgObj                        := Arg0Op | Arg1Op | Arg2Op | Arg3Op | Arg4Op | Arg5Op |
                                  Arg6Op
Arg0Op                        := 0x68
Arg1Op                        := 0x69
Arg2Op                        := 0x6a
Arg3Op                        := 0x6b
Arg4Op                        := 0x6c
Arg5Op                        := 0x6d
Arg6Op                        := 0x6e
```

17.2.5.2 Local Objects Encoding

```
LocalObj                      := Local0Op | Local1Op | Local2Op | Local3Op | Local4Op |
                                  Local5Op | Local6Op | Local7Op
Local0Op                      := 0x60
Local1Op                      := 0x61
Local2Op                      := 0x62
Local3Op                      := 0x63
Local4Op                      := 0x64
Local5Op                      := 0x65
Local6Op                      := 0x66
Local7Op                      := 0x67
```

17.2.5.3 Debug Objects Encoding

```
DebugObj                     := DebugOp
DebugOp                      := ExtOpPrefix 0x31
```

17.3 AML Byte Stream Byte Values

The following table lists all the byte values that can be found in an AML byte stream and the meaning of each byte value. This table is useful for debugging AML code.

Table 17-2 AML Byte Stream Byte Values

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x00	ZeroOp	Data Object	—	—
0x01	OneOp	Data Object	—	—
0x02-0x05	—	—	—	—
0x06	AliasOp	Term Object	NameString NameString	—
0x07	—	—	—	—
0x08	NameOp	Term Object	NameString DataRefObject	—
0x09	—	—	—	—
0x0A	BytePrefix	Data Object	ByteData	—
0x0B	WordPrefix	Data Object	WordData	—
0x0C	DWordPrefix	Data Object	DWordData	—
0x0D	StringPrefix	Data Object	AsciiCharList NullChar	—
0x0E	QWordPrefix	Data Object	QWordData	—
0x0F	—	—	—	—
0x10	ScopeOp	Term Object	NameString	TermList
0x11	BufferOp	Term Object	TermArg	ByteList
0x12	PackageOp	Term Object	ByteData	PackageTermList
0x13	VarPackageOp	Term Object	TermArg	PackageTermList
0x14	MethodOp	Term Object	NameString ByteData	TermList
0x15-0x2D	—	—	—	—
0x2E (‘.’)	DualNamePrefix	Name Object	NameSeg NameSeg	—
0x2F (‘/’)	MultiNamePrefix	Name Object	ByteData NameSeg(N)	—
0x30-0x40	—	—	—	—

Table 17-2 AML Byte Stream Byte Values *(continued)*

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x41-0x5A (‘A’-‘Z’)	NameChar	Name Object	—	—
0x5B (‘[’)	ExtOpPrefix	—	ByteData	—
0x5B 0x01	MutexOp	Term Object	NameString ByteData	—
0x5B 0x02	EventOp	Term Object	NameString	—
0x5B 0x12	CondRefOfOp	Term Object	SuperName SuperName	—
0x5B 0x13	CreateFieldOp	Term Object	TermArg TermArg TermArg NameString	—
0x5B 0x1F	LoadTableOp	Term Object	TermArg TermArg TermArg TermArg TermArg TermArg	—
0x5B 0x20	LoadOp	Term Object	NameString SuperName	—
0x5B 0x21	StallOp	Term Object	TermArg	—
0x5B 0x22	SleepOp	Term Object	TermArg	—
0x5B 0x23	AcquireOp	Term Object	SuperName WordData	—
0x5B 0x24	SignalOp	Term Object	SuperName	—
0x5B 0x25	WaitOp	Term Object	SuperName TermArg	—
0x5B 0x26	ResetOp	Term Object	SuperName	—
0x5B 0x27	ReleaseOp	Term Object	SuperName	—
0x5B 0x28	FromBCDOp	Term Object	TermArg Target	—
0x5B 0x29	ToBCD	Term Object	TermArg Target	—
0x5B 0x2A	UnloadOp	Term Object	SuperName	—
0x5B 0x30	RevisionOp	Data Object	—	—

Table 17-2 AML Byte Stream Byte Values *(continued)*

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x5B 0x31	DebugOp	Debug Object	—	—
0x5B 0x32	FatalOp	Term Object	ByteData DWordData TermArg	—
0x5B 0x80	OpRegionOp	Term Object	NameString ByteData TermArg TermArg	—
0x5B 0x81	FieldOp	Term Object	NameString ByteData	FieldList
0x5B 0x82	DeviceOp	Term Object	NameString	ObjectList
0x5B 0x83	ProcessorOp	Term Object	NameString ByteData DWordData ByteData	ObjectList
0x5B 0x84	PowerResOp	Term Object	NameString ByteData WordData	ObjectList
0x5B 0x85	ThermalZoneOp	Term Object	NameString	ObjectList
0x5B 0x86	IndexFieldOp	Term Object	NameString NameString ByteData	FieldList
0x5B 0x87	BankFieldOp	Term Object	NameString NameString TermArg ByteData	FieldList
0x5B 0x88	DataRegionOp	Term Object	NameString TermArg TermArg TermArg	—
0x5C (‘\’)	RootChar	Name Object	—	—
0x5D	—	—	—	—
0x5E (‘^’)	ParentPrefixChar	Name Object	—	—
0x5F(‘_’)	NameChar—	Name Object	—	—
0x60 (‘`’)	Local0Op	Local Object	—	—
0x61 (‘a’)	Local1Op	Local Object	—	—
0x62 (‘b’)	Local2Op	Local Object	—	—
0x63 (‘c’)	Local3Op	Local Object	—	—
0x64 (‘d’)	Local4Op	Local Object	—	—

Table 17-2 AML Byte Stream Byte Values *(continued)*

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x65 (‘e’)	Local5Op	Local Object	—	—
0x66 (‘f’)	Local6Op	Local Object	—	—
0x67 (‘g’)	Local7Op	Local Object	—	—
0x68 (‘h’)	Arg0Op	Arg Object	—	—
0x69 (‘i’)	Arg1Op	Arg Object	—	—
0x6A (‘j’)	Arg2Op	Arg Object	—	—
0x6B (‘k’)	Arg3Op	Arg Object	—	—
0x6C (‘l’)	Arg4Op	Arg Object	—	—
0x6D (‘m’)	Arg5Op	Arg Object	—	—
0x6E (‘n’)	Arg6Op	Arg Object	—	—
0x6F	—	—	—	—
0x70	StoreOp	Term Object	TermArg SuperName	—
0x71	RefOfOp	Term Object	SuperName	—
0x72	AddOp	Term Object	TermArg TermArg Target	—
0x73	ConcatOp	Term Object	TermArg TermArg Target	—
0x74	SubtractOp	Term Object	TermArg TermArg Target	—
0x75	IncrementOp	Term Object	SuperName	—
0x76	DecrementOp	Term Object	SuperName	—
0x77	MultiplyOp	Term Object	TermArg TermArg Target	—
0x78	DivideOp	Term Object	TermArg TermArg Target Target	—
0x79	ShiftLeftOp	Term Object	TermArg TermArg Target	—
0x7A	ShiftRightOp	Term Object	TermArg TermArg Target	—
0x7B	AndOp	Term Object	TermArg TermArg Target	—

Table 17-2 AML Byte Stream Byte Values *(continued)*

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x7C	NandOp	Term Object	TermArg TermArg Target	—
0x7D	OrOp	Term Object	TermArg TermArg Target	—
0x7E	NorOp	Term Object	TermArg TermArg Target	—
0x7F	XorOp	Term Object	TermArg TermArg Target	—
0x80	NotOp	Term Object	TermArg Target	—
0x81	FindSetLeftBitOp	Term Object	TermArg Target	—
0x82	FindSetRightBitOp	Term Object	TermArg Target	—
0x83	DerefOfOp	Term Object	TermArg	—
0x84	ConcatResOp	Term Object	TermArg TermArg Target	—
0x85	ModOp	Term Object	TermArg TermArg Target	—
0x86	NotifyOp	Term Object	SuperName TermArg	—
0x87	SizeOfOp	Term Object	SuperName	—
0x88	IndexOp	Term Object	TermArg TermArg Target	—
0x89	MatchOp	Term Object	TermArg ByteData TermArg ByteData TermArg TermArg	—
0x8A	CreateDWordFieldOp	Term Object	TermArg TermArg NameString	—
0x8B	CreateWordFieldOp	Term Object	TermArg TermArg NameString	—
0x8C	CreateByteFieldOp	Term Object	TermArg TermArg NameString	—
0x8D	CreateBitFieldOp	Term Object	TermArg TermArg NameString	—
0x8E	ObjectTypeOp	Term Object	SuperName	—

Table 17-2 AML Byte Stream Byte Values *(continued)*

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0x8F	CreateQWordFieldOp	Term Object	TermArg TermArg NameString	—
0x90	LandOp	Term Object	TermArg TermArg	—
0x91	LorOp	Term Object	TermArg TermArg	—
0x92	LnotOp	Term Object	TermArg	—
0x92 0x93	LNotEqualOp	Term Object	TermArg TermArg	—
0x92 0x94	LLessEqualOp	Term Object	TermArg TermArg	—
0x92 0x95	LGreaterEqualOp	Term Object	TermArg TermArg	—
0x93	LEqualOp	Term Object	TermArg TermArg	—
0x94	LGreaterOp	Term Object	TermArg TermArg	—
0x95	LLessOp	Term Object	TermArg TermArg	—
0x96	ToBufferOp	Term Object	TermArg Target	—
0x97	ToDecimalStringOp	Term Object	TermArg Target	—
0x98	ToHexStringOp	Term Object	TermArg Target	—
0x99	ToIntegerOp	Term Object	TermArg Target	—
0x9A-0x9B	—	—	—	—
0x9C	ToStringOp	Term Object	TermArg TermArg Target	—
0x9D	CopyObjectOp	Term Object	TermArg SimpleName	—
0x9E	MidOp	Term Object	TermArg TermArg TermArg Target	—
0x9F	ContinueOp	Term Object	—	—
0xA0	IfOp	Term Object	TermArg	TermList

Table 17-2 AML Byte Stream Byte Values *(continued)*

Encoding Value	Encoding Name	Encoding Group	Fixed List Arguments	Variable List Arguments
0xA1	ElseOp	Term Object	—	TermList
0xA2	WhileOp	Term Object	TermArg	TermList
0xA3	NoopOp	Term Object	—	—
0xA4	ReturnOp	Term Object	TermArg	—
0xA5	BreakOp	Term Object	—	—
0xA6-0xCB	—	—	—	—
0xCC	BreakPointOp	Term Object	—	—
0xCD-0xFE	—	—	—	—
0xFF	OnesOp	Data Object	—	—

17.4 AML Encoding of Names in the Namespace

Assume the following namespace exists:

```
\
        S0
            MEM
                SET
                GET
        S1
            MEM
                SET
                GET
            CPU
                SET
                GET
```

Assume further that a definition block is loaded that creates a node \S0.CPU.SET, and loads a block using it as a root. Assume the loaded block contains the following names:

```
    STP1
    ^GET
    ^^PCI0
    ^^PCI0.SBS
    \S2
    \S2.ISA.COM1
    ^^^S3
    ^^^S2.MEM
    ^^^S2.MEM.SET
    Scope(\S0.CPU.SET.STP1) {
        XYZ
        ^ABC
        ^ABC.DEF
    }
```

This will be encoded in AML as:

```
'STP1'
ParentPrefixChar 'GET_'
ParentPrefixChar ParentPrefixChar 'PCI0'
ParentPrefixChar ParentPrefixChar DualNamePrefix 'PCI0' 'SBS_'
RootChar 'S2__'
RootChar MultiNamePrefix 3 'S2__'  'ISA_'  'COM1'
ParentPrefixChar ParentPrefixChar ParentPrefixChar 'S3__'
ParentPrefixChar ParentPrefixChar ParentPrefixChar DualNamePrefix 'S2__'  'MEM_'
ParentPrefixChar ParentPrefixChar ParentPrefixChar MultiNamePrefix 3 'S2__'  'MEM_'
'SET_'
```

After the block is loaded, the namespace will look like this (names added to the namespace by the loading operation are shown in bold):

```
    \
        S0
            MEM
                SET
                GET
            CPU
                SET
                    STP1
                        XYZ
                    ABC
                        DEF
                GET
            PCI0
                SBS
        S1
            MEM
                SET
                GET
            CPU
                SET
                GET
        S2
            ISA
                COM1
            MEM
                SET
        S3
```

APPENDIX A: Device Class Specifications

A Device Class PM Specifications

This section defines the behavior of devices as that behavior relates to power management and, specifically, to the four device power states defined by ACPI. The goal is enabling device vendors to design power-manageable products that meet the basic needs of OSPM and can be utilized by any ACPI-compatible operating system.

A.1 Overview

The power management of individual devices is the responsibility of a *policy owner* in the operating system. This software element will implement a power management policy that is appropriate for the type (or *class*) of device being managed. Device power management policy typically operates in conjunction with a global system power policy implemented in the operating system.

In general, the device-class power management policy strives to reduce power consumption while the system is working by transitioning among various available power states according to device usage. The challenge facing policy owners is to minimize power consumption without adversely impacting the system's usability. This balanced approach provides the user with both power savings and good performance.

Because the policy owner has very specific knowledge about when a device is in use or potentially in use, there is no need for hardware timers or such to determine when to make these transitions. Similarly, this level of understanding of device usage makes it possible to use fewer device power states. Generally, intermediate states attempt to draw a compromise between latency and consumption because of the uncertainty of actual device usage. With the increased knowledge in the OS, good decisions can be made about whether the device is needed at all. With this ability to turn devices off more frequently, the benefit of having intermediate states diminishes.

The policy owner also determines what class-specific events can cause the system to transition from sleeping to working states, and enables this functionality based on application or user requests. Notice that the definition of the wake events that each class supports will influence the system's global power policy in terms of the level of power management a system sleeping state can attain while still meeting wake latency requirements set by applications or the user.

A.2 Device Power States

The following definitions apply to devices of all classes:

- **D0**. State in which device is on and running. It is receiving full power from the system and is delivering full functionality to the user.
- **D1**. Class-specific low-power state (defined in the following section) in which device context may or may not be lost. Buses in D1 cannot do anything to the bus that would force devices on that bus to lose context.
- **D2**. Class-specific low-power state (defined in the following section) in which device context may or may not be lost. Attains greater power savings than D1. Buses in D2 can cause devices on that bus to lose some context (for example, the bus reduces power supplied to the bus). Devices in D2 must be prepared for the bus to be in D2 or higher.
- **D3**. State in which device is off and not running. Device context is lost. Power can be removed from the device.

Device power-state transitions are typically invoked through bus-specific mechanisms (for example, ATA Standby, USB Suspend, and so on). In some cases, bus-specific mechanisms are not available and device-specific mechanisms must be used. Notice that the explicit command for entering the D3 state might be the removal of power.

It is the responsibility of the policy owner (or other software) to restore any lost device context when returning to the D0 state.

A.2.1 Bus Power Management

Policy owners for bus devices (for example, PCI, USB, Small Computer System Interface [SCSI]) have the additional responsibility of tracking the power states of all devices on the bus and for transitioning the bus itself to only those power states that are consistent with those of its devices. This means that the bus state can be no lower than the highest state of one of its devices. However, enabled wake events can affect this as well. For example, if a particular device is in the D2 state and set to wake the system and the bus can only forward wake requests while in the D1 state, then the bus must remain in the D1 state even if all devices are in a lower state.

Below are summaries of relevant bus power management specifications with references to the sources.

A.2.2 Display Power Management

Refer to the *Display Power Management Signaling Specification (DPMS)*, available from:

Video Electronics Standards Association (VESA)
2150 North First Street
Suite 440
San Jose, CA 95131-2029

A DPMS-compliant video controller and DPMS-compliant monitor use the horizontal and vertical sync signals to control the power mode of the monitor. There are 4 modes of operation: normal, standby, suspend and off. DPMS-compliant video controllers toggle the sync lines on or off to select the power mode.

A.2.3 PCMCIA/PCCARD/CardBus Power Management

Refer to the PCMCIA (Personal Computer Memory Card International Association) Web site,at http://www.pc-card.com/.

PCMCIA and PCCARD devices do not have device power states defined. The only power states available are on and off, controlled by the host bus controller. The CardBus specification is a superset of the PCCARD specification, incorporating the power management specification for PCI bus. Power management capabilities query, state transition commands and wake event reporting are identical.

A.2.4 PCI Power Management

Refer to the PCI Special Interest Group (PCISIG) Web site, at http://www.pcisig.com/.

- **PCI Bus Power Management Capabilities Query**. PCI Bus device capabilities are reported via the optional Capabilities List registers, which are accessed via the Cap_Ptr.
- **PCI Bus Power Management State Transition Commands**. PCI Bus device power states are controlled and queried via the standard Power Management Status/Control Register (PMCSR).
- **PCI Bus Wakeup Event Reporting**. PCI wake events are reported on the optional PME# signal, with setting of the Wake_Int bit in the PMCSR. Wake event reporting is controlled by the Wake_En bit in the PMCSR register.

A.2.5 USB Power Management

Refer to the Universal Serial Bus Implementers Forum (USB-IF) Web site, at http://www.usb.org/.

- **USB Power Management Capabilities Query**. USB device capabilities are reported to the USB Host via the standard Power Descriptors. These address power consumption, latency time, wake support, and battery support and status notification.
- **USB Power Management State Transition Commands**. USB device power states are controlled by the USB Host via the standard SET_FEATURE command. USB device power states are queried via the standard USB GET_STATUS command.
- **USB Wakeup Event Reporting**. USB wake event reporting is controlled using the SET_FEATURE command, with value DEVICE_REMOTE_WAKEUP. USB wake events are reported by sending remote wake resume signaling.

A.2.6 Device Classes

Below is a list of the class-specific device power management definitions available in this specification. Notice that there exists a default device class definition that applies to all devices, even if there is a separate, class-specific section that adds additional requirements.

- **Audio Device Class**. Applies to audio devices.
- **COM Port Device Class**. Applies to COM ports devices.
- **Display Device Class**. Applies to CRT monitors, LCD panels, and video controllers for those devices.
- **Input Device Class**. Applies to standard types of input devices such as keyboards, keypads, mice, pointing devices, joysticks, and game pads, plus new types of input devices such as virtual reality devices.
- **Modem Device Class**. Applies to modem and modem-like (for example, ISDN terminal adapters) devices.
- **Network Device Class**. Applies specifically to Ethernet and token ring adapters. ATM and ISDN adapters are not supported by this specification.
- **PC Card Controller Device Class**. Applies to PC Card controllers and slots.
- **Storage Device Class**. Applies specifically to ATA hard disks, floppy disks, ATAPI and SCSI CD-ROMs, and the IDE channel.

A.3 Default Device Class

The requirements expressed in this section apply to all devices, even if there is a separate, class-specific power management definition that identifies additional requirements.

A.3.1 Default Power State Definitions

State	Definition
D0	Device is on and running. It is receiving full power from the system, and is delivering full functionality to the user.
D1	This state is not defined and not used by the default device class.
D2	This state is not defined and not used by the default device class.
D3	Device is off and not running. Device context is assumed lost, and there is no need for any of it to be preserved in hardware. This state should consume the minimum power possible. Its only requirement is to recognize a bus-specific command to re-enter D0. Power can be removed from the device while in D3. If power is removed, the device will receive a bus-specific hardware reset upon reapplication of power, and should initialize itself as in a normal power on.

A.3.2 Default Power Management Policy

Present State	Next State	Cause
D0	D3	Device determined by the OS to not be needed by any applications or the user. System enters a sleeping state.
D3	D0	Device determined by the OS to be needed by some application or the user.

A.3.3 Default Wake Events

There are no default wake events, because knowledge of the device is implicit in servicing such events. Devices can expose wake capabilities to OSPM, and device-specific software can enable these, but there is no generic application-level or OS-wide support for undefined wake events.

A.3.4 Minimum Power Capabilities

All devices must support the D0 and D3 states. Functionality available in D0 must be available after returning to D0 from D3 without requiring a system reboot or any user intervention. This requirement applies whether or not power is removed from the device during D3.

A.4 Audio Device Class

The requirements expressed in this section apply to audio devices.

A.4.1 Power State Definitions

State	Status	Definition
D0	*Required*	Power is on. Device is operating.
D1	*Optional*	Power consumption is less than D0 state. Device must be able to transition between D0 and D1 states within 100 ms. No audio samples may be lost by entering and leaving this state.
D2	*Required*	Power consumption is less than D0 state. Device must be able to transition between D0 and D2 states within 100 ms. Audio samples may be lost by entering and leaving this state.
D3	*Required*	The device is completely off or drawing minimal power. For example, a stereo will be off, but a light-emitting diode (LED) may be on and the stereo may be listening to IR commands.

If a device is in the D1 or D2 state it must resume within 100 ms. A device in the D3 state may take as long as it needs to power up. It is the responsibility of the policy owner to advertise to the system how long a device requires to power up.

All audio devices must be capable of D0, D2 and D3 states. It is desirable that an audio device be capable of D1 state. The difference between D1 and D2 is that a device capable of D1 can maintain complete state information in reduced power mode. The policy owner or other software must save all states for D2-capable devices. Some audio samples may be lost in transitioning into and out of the D2 state.

Notice that the D1 state was added to allow digital signal processor (DSP)-equipped audio hardware to exploit low-power modes in the DSP. For example, a DSP may be used to implement Dolby AC-3 Decode. When paused it stops playing audio, but the DSP may contain thousands of bytes worth of state information. If the DSP supports a low-power state, it can shut down and later resume from exactly the audio sample where it paused without losing state information.

A.4.2 Power Management Policy

For the purpose of the following state transition policy, the following device-specific operational states are defined:

- **Playing**. Audio is playing.
- **Recording:**
  Foreground. Normal application is recording. Recording is considered foreground unless specifically designated low priority.
  Background. Speech recognition or speech activity detection is running. Recording may be preempted by foreground recording or playing. Any audio recording may be designated as background.
- **Full Duplex**. Device is simultaneously playing and recording.
- **Paused**. File handle is open. Only devices that are playing, foreground recording or in full duplex operation may be paused. Background recording may not be paused. State is static and never lost. The paused state assumes that a device must transition to the resumed state rapidly. Playing or recording must be resumed within 100 ms. No audio samples may be lost between the device is paused and later resumed.
- **Closed**. No file handle is open.

Present State	Next State	Cause
D3	D0	Audio device moves from closed to open state or paused when the device receives the resume command.
D0	D1	Audio device receives pause command. If device is D1 capable, this state is preferred. If not, the device driver will preserve context, and the device will be set to D2.
D2/D1	D0	Audio device receives a resume command.
D0	D2	Audio device is closed. Audio inactivity timer started.
D2	D3	Audio inactivity timer expires.
D0	D3	Audio device is in background record mode and receives power-down command.

When an audio device is in the D0 state it will refuse system requests to transition to D3 state unless it is in background record mode. When an audio device is paused (D1 or D2) and it receives a request to transition to the D3 state, it will save the state of the audio device and transition to the D3 state.

Since multimedia applications often open and close audio files in rapid succession, it is recommended that an inactivity timer be employed by the policy owner to prevent needless shutdowns (D3 transitions) of the audio hardware. For example, frequent power cycling may damage audio devices powered by vacuum tubes.

A.4.3 Wake Events

An audio device may be a wake device. For example, a USB microphone designed for security applications might use the USB wake mechanism to signal an alarm condition.

A.4.4 Minimum Power Capabilities

All audio devices must be capable of D0, D2 and D3 power states. If the device is capable of maintaining context while in a low-power state it should advertise support for D1. Transitional latency for the D2 or D3 states must be less than 100 ms. There are no latency restrictions for D3 transitions, but the policy owner should advertise the amount of time required.

A.5 COM Port Device Class

The requirements expressed in this section apply to Universal Asynchronous Receiver/Transmitters (UARTs) such as the common NS16550 buffered serial port and equivalents.

The two required states for any power-managed COM Port are full on (D0) and full off (D3). This in turn requires that the COM port hardware be power-manageable by ACPI control methods for COM ports that are on system boards, or by standard bus power management controls for COM ports that are on add-in cards (for example, PCI). Because of this, ISA-based COM port add-in cards will not be able to meet this requirement, and therefore cannot be compliant with this specification.

A.5.1 Power State Definitions

State	Status	Definition
D0	*Required*	Line drivers are on. UART context is preserved.
D1	*N/A*	This state is not defined for COM Ports. Use the D3 state instead.
D2	*N/A*	This state is not defined for COM Ports. Use the D3 state instead.
D3	*Required*	Line drivers are off (unpowered; outputs isolated from devices attached to the port). UART context is lost. Latency to return to D0 is less than 1 second.

A.5.2 Power Management Policy

Present State	Next State	Cause
D3	D0	Power-on reset COM port opened by an application
D0	D3	COM port closed System enters sleeping state while wake is disabled on this device. System enters sleeping state while wake is enabled on this device and the device is capable of generating wake to the system from state D3.

A.5.3 Wake Events

If the COM port is capable of generating wake events, asserting the “ring indicator” line (V.24 circuit 125) will cause the COM port to assert a wake event. There are two common mechanisms that may be employed (either one or both) for performing machine wake using COM ports.

The first provides a solution that is capable of waking the PC whether the UART is powered (D0) or not (D3). Here, the “ring indicator” line (from V.24 circuit 125) is commonly connected directly to the system wake device in addition to being connected to the UART. While this implementation is normative for COM ports located on system motherboards (see the ACPI specification), it could also be done by add-in cards with COM ports that reside on buses supporting system wake from devices in D3 (for example, PME# signal on PCI).

The second mechanism requires that the UART be powered (D0) to use the UART’s interrupt output pin to generate the wake event instead. When using this method, the OS COM port policy owner or power management control methods are expected to configure the UART. Although any UART interrupt source (for example, ‘data ready’) could theoretically be used to wake the system, these methods are beyond the scope of this document.

A.5.4 Minimum Power Capabilities

A COM port conforming to this specification must support the D0 and D3 states.

A.6 Display Device Class

The requirements expressed in this section apply to CRT monitors, LCD panels, and video controllers.

A.6.1 Power State Definitions

A.6.1.1 CRT Monitors and LCD Panels

State	**Status**	**Definition**
D0	*Required*	This state is equivalent to the “On” state defined in the VESA DPMS specification (see Related Documents) and is signaled to the display using the DPMS method. Display is fully on Video image is active
D1	*Optional*	This state is equivalent to the “Standby” state defined in the VESA DPMS and is signaled to the display using the DPMS method. Display is functional but may be conserving energy Video image is blank Latency to return to D0 must be less than 5 seconds
D2	*Required*	This state is equivalent to the “Suspend” state defined in the VESA DPMS specification and is signaled to the display using the DPMS method. Display is functional and conserving energy Video image is blank Latency to return to D0 is less than 10 seconds
D3	*Required*	This state is equivalent to the “Off” state defined in the VESA DPMS specification and is signaled to the display using the DPMS method. Display is non-functional Video image is blank

A.6.1.2 Video Controllers

State	Status	Definition
D0	*Required*	Back-end is on Video controller context is preserved Video memory contents are preserved
D1	*Optional*	Back-end is off, except for monitor/LCD control signaling (DPMS) Video controller context is preserved Video memory contents is preserved Latency to return to D0 is less than 1 second
D2	*Optional*	Back-end is off, except for monitor/LCD control signaling (DPMS) Video controller context is lost Video memory contents is lost Latency to return to D0 is less than 5 second
D3	*Required*	Back-end is off Video controller context is lost (power removed) Video memory contents is lost (power removed)

A.6.2 Power Management Policy

Present State	Next State	Cause
D0	D1	User inactivity for a period of time (T1)
D1	D2	User inactivity for a period of time (T2 > T1)
D2	D3	User inactivity for a period of time (T3 > T2)
D1/D2/D3	D0	User activity or application UI change (for example, dialog pop-up)

These state transition definitions apply to both the monitor/ LCD and the video controller. However, the control of the two devices is independent, except that a video controller will never be put into a lower power state than its monitor/LCD(s).

Transitions for the video controller are commanded via the bus-specific control mechanism for device states. Monitor/LCD transitions are commanded by signaling from the video controller (DPMS) and are only generated as a result of explicit commands from the policy-owner. Monitor/LCD power control is functionally independent from any other interface the monitor may provide (such as USB). For instance, Hubs and HID devices in the monitor enclosure may be power-managed by their driver over the USB bus, but the Monitor/LCD device itself may not; it must be power-managed by DPMS from the video controller.

A.6.3 Wake Events

Display devices incorporating a system power switch should generate a wake event when the switch is pressed while the system is sleeping.

A.6.4 Minimum Power Capabilities

A CRT monitor or LCD panel conforming to this specification must support the D0, D2, and D3 states. Support for the D1 state is optional. Transitional latencies for the D1 state must be less than 5 seconds, and less than 10 seconds for the D2 state.

A video controller conforming to this specification must support the D0 and D3 states. Support for the D1 and D2 states is optional. Transitional latencies for the D1 state must be less than 1 second, and less than 5 seconds for the D2 state.

A.7 Input Device Class

The requirements expressed in this section apply to standard types of input devices such as keyboards, keypads, mice, pointing devices, joysticks, game pads, to devices that combine these kinds of input functionality (composite devices, and so on), and to new types of input devices such as virtual reality devices, simulation devices, and so on.

A.7.1 Power State Definitions

State	Status	Definition
D0	*Required*	Device is receiving full power from its power source, delivering full functionality to the user, and preserving applicable context and state information.
D1	*Optional*	Input device power consumption is greatly reduced. In general, device is in a power management state and is not delivering any functionality to the user except wake functionality if applicable. Device status, state, or other information indicators (for example, LEDs, LCD displays, and so on) are turned off to save power. The following device context and state information should be preserved by the policy owner or other software: **Keyboard**. Num, caps, scroll lock states (and Compose and Kana states if applicable) and associated LED/indicator states, repeat delay, and repeat rate. **Joystick**. Forced feedback effects (if applicable). **Any input device**. All context and state information that cannot be preserved by the device when it's conserving power.
D2	*N/A*	This state is not defined for input devices, use D1 as the power management state instead.
D3	*Required*	Input device is off and not running. In general, the device is not delivering any functionality to the user except wake functionality if applicable. Device context and state information is lost.

A.7.2 Power Management Policy

Present State	Next State	Cause
D3	D0	Requested by the system
D0	D1/D3*	Requested by the system (for example, system goes to sleep with wake enabled)
D0/D1	D3	Requested by the system (for example, system goes to sleep with wake disabled) Power is removed
D1/D3	D0	Device with enabled wake capability requests transition by generating a wake event Requested by the system

*Depends on capability of device (if it features D1 or D3 wake capability or not); device will be put in state with the lowest possible power consumption.

A.7.3 Wake Events

It is recommended, but not required, that input devices implement and support bus-specific wake mechanisms if these are defined for their bus type. This is recommended because a user typically uses an input device of some kind to wake the system when it is in a power management state (for example, when the system is sleeping).

The actual input data (particular button or key pressed) that's associated with a wake event should never be discarded by the device itself, but should always be passed along to the policy owner or other software for further interpretation. This software implements a policy for how this input data should be interpreted, and decides what should be passed along to higher-level software, and so on.

It is recommended that the device button(s) or key(s) used for power management purposes are clearly labeled with text and/or icons. This is recommended for keyboards and other input devices on which all buttons or keys are typically labeled with text and/or icons that identify their usage.

For example, a keyboard could include a special-purpose power management button (for example, “Power”) that, when pressed during a system sleeping state, generates a wake event. Alternatively, the button(s) on mice and other pointing devices could be used to trigger a wake event.

Examples of more advanced wake events include keyboard wake signaling when any key is pressed, mouse wake signaling on detection of X/Y motion, joystick wake signaling on X/Y motion, and so on. However, in order to avoid accidental or unintentional wake of the system, and to give the user some control over which input events will result in a system wake, it's suggested that more advanced types of wake events are implemented as features that can be turned on or off by the user (for example, as part of the OSPM user interface).

A.7.4 Minimum Power Capabilities

An input device conforming to this specification must support the D0 and D3 states. Support for the D1 state is optional.

A.8 Modem Device Class

The requirements expressed in this section apply to modems and similar devices, such as USB controlled ISDN Terminal Adapters (“digital modems”) and computer-connected telephone devices ("CT phones"). This specification will refer to these devices as “modems; the same considerations apply to digital modems and CT phones unless explicitly stated otherwise.

The scope of this section is further restricted to modems that support power management using methods defined by the relevant PC-modem connection bus. These include PCI, USB, PCCARD (PCMCIA), CardBus, and modems on the system motherboard described by ACPI BIOS control methods. The scope does not include bus-specific means for devices to alert the host PC (for example, how to deliver a ”ringing”’ message), nor does it address how those alerting operations are controlled.

A.8.1 Technology Overview

Modems are traditionally serial devices, but today modems may be attached to a PC by many different means. Further, many new modems expose a software serial interface, where the modem controller function is implemented in software. This specification addresses three different connection types:

- Traditional connections without power-managed connections (for example, COM, LPT, ISA)
- Power managed connections (for example, PCCARD, CardBus, PCI, USB)
- Motherboard modems

For some of the above modem connection types mentioned, there are three different modem architectures possible:

- Traditional modem (DAA, DSP, and controller in hardware)
- Controller-less design (DAA and DSP in hardware)
- "Soft modem" design (DAA and CODEC only in hardware)

The hardware components of the modem shall be controlled by the relevant bus commands, where applicable (USB, PCI, CardBus). The software components are dependent on the power state of the CPU.

A.8.1.1 Traditional Connections

In older methods (COM, LPT, ISA) the modem is controlled primarily by serialized ASCII command strings (for example, V.25ter) and traditional V.24 (RS-232) out-of-band leads. In these legacy devices, there are no common means for power management other than the power switch for the device, or the entire system unit.

An external modem connected to a COM port or LPT port typically has its own power supply. An LPT port modem might run from the current on the LPT port +5V supply. For COM or LPT port modems, power is typically controlled by a user switch.

The most common modem type is an ISA card with an embedded COM port. From a software standpoint, they are logically identical to external modems, but the modems are powered by the PC system unit. Power is drawn from the ISA bus without independent power switching.

A.8.1.2 Power-Managed Connections

PCMCIA, PCCARD and CardBus slots are powered and power-managed by the system, using means defined in the relevant bus specifications. For PCMCIA and PCCARD devices, only D0 and D3 states are available, via Socket Services in the OS and/or ACPI BIOS. CardBus adds intermediate states, using the same mechanisms defined for PCI Bus.

PCI bus slots are powered and power-managed by the system, using means defined in the PCI specification.

USB devices may be powered by the USB itself (100mA or 500mA), or have their own external power supply. All USB devices are power-managed by the USB bus master, using means defined in the USB specification.

A.8.1.3 Motherboard Modems

A modem embedded in the motherboard is powered by controls on the motherboard. It should be power-managed by using control methods exposed via ACPI BIOS tables.

A.8.2 Power State Definitions

State	Status	Definition
D0	*Required*	Phone interface is on (may be on or off hook) Speaker is on Controller Context is preserved
D1	*N/A*	Not defined (do not use)
D2	*Optional*	Phone interface is not powered by the host (on hook) Speaker is off Controller context is preserved 2 seconds maximum restore time
D3	*Required*	Phone interface is not powered by host (on hook) Speaker is off Controller context may be lost 5 seconds maximum restore time

A.8.3 Power Management Policy

Present State	Next State	Cause
D2/D3	D0	System issues a bus command to enter the D0 state (for example, an application is answering or originating a call).
D0	D2	System issues a bus command to enter the D2 state. (for example, an application is listening for an incoming call).
D0	D3	System issues a bus command to enter the D3 state (for example, all applications have closed the Modem device).

A.8.4 Wake Events

For any type of modem device, wake events (if supported and enabled) are only generated in response to detected “ringing” from an incoming call. All other events associated with modems (V.8bis messages, and so on) require that the PC be in the “working” state to capture them. The methods and signals used to generate the wake may vary as a function of the modem connection (bus) type and modem architecture.

Machine wake is allowed from any modem power state (D0, D2, and D3), and is accomplished by methods described in the appropriate bus power management specification (PCI, USB, PCCARD), or by ACPI system board control methods (for Modem on Motherboard implementations).

If the specific modem implementation or connection type does not enable it to assert system wake signaling, these modems will not be able to wake the machine. The OS modem policy owner will have to retain the PC in the “working” state to perform *all* types of event detection (including ringing).

A.8.5 Minimum Power Capabilities

A modem or similar device conforming to this specification must support the D0 and D3 states. Support of the D2 state is optional.

A.9 Network Device Class

The requirements expressed in this section apply to Ethernet and token ring adapters. ATM and ISDN adapters are not supported by this specification.

A.9.1 Power State Definitions

For the purpose of the following state definitions “no bus transmission” means that transmit requests from the host processor are not honored, and “no bus reception” means that received data are not transferred to host memory.

State	Status	Definition
D0	*Required*	Device is on and running and is delivering full functionality and performance to the user Device is fully compliant with the requirements of the attached network
D1	*Optional*	No bus transmission allowed No bus reception allowed No interrupts can occur Device context may be lost
D2	*Optional*	No bus transmission allowed No bus reception allowed No interrupts can occur Device context may be lost
D3	*Required*	Device context is assumed to be lost No bus transmission allowed No bus reception allowed No interrupts can occur

This document does not specify maximum power and maximum latency requirements for the sleeping states because these numbers are very different for different network technologies. The device must meet the requirements of the bus that it attaches to.

Although the descriptions of states D1 and D2 are the same, the choice of whether to implement D1 or D2 or both may depend on bus services required, power requirements, or time required to restore the physical layer. For example, a device designed for a particular bus might include state D1 because it needs a bus service such as a bus clock to support Magic Packet™ wake, and that service is available in the bus device’s D1 power state but not in D2. Also, a device might include both state D1 and state D2 to provide a choice between lower power and lower latency.

A.9.2 Power Management Policy

Present State	Next State	Cause
D0	D*x*	System enters sleep state. If wake is enabled, D*x* is the lowest power state (for example, D1, D2, D3) from which the network device supports system wake. An appropriate time-out has elapsed after a "link down" condition was detected. D*x* is the lowest power state in which the network device can detect "link up."
D0	D3	System initiated network shutdown. System enters sleep state and wake is either not enabled or the network device is capable of waking from D3.
D1/D2/D3	D0	System wake (transition to S0), including a wake caused by a network wake event.

A.9.3 Wake Events

Network wake events are generally the result of either a change in the *link status* or the reception of a *wake frame* from the network.

A.9.3.1 Link Status Events

Link status wake events are useful to indicate a change in the network's availability, particularly when this change may impact the level at which the system should re-enter the sleeping state. For example, a transition from "link off" to "link on" may trigger the system to re-enter sleep at a higher level (for example, S2 versus S3) so that wake frames can be detected. Conversely, a transition from "link on" to "link off" may trigger the system to re-enter sleep at a deeper level (for example, S3 versus S2) since the network is not currently available. The network device should implement an internal delay to avoid unnecessary transitions when the link status toggles on or off momentarily.

A.9.3.2 Wake Frame Events

Wake frame events are used to wake the system whenever *meaningful* data is presented to the system over the network. Examples of meaningful data include the reception of a Magic Packet™, a management request from a remote administrator, or simply network traffic directly targeted to the local system. In all of these cases the network device was pre-programmed by the policy owner or other software with information on how to identify wake frames from other network traffic. The details of how this information is passed between software and network device depend on the OS and therefore are not described in this specification.

A.9.4 Minimum Power Capabilities

A network device conforming to this specification must support the D0 and D3 states. Support for the D1 and D2 states is optional.

A.10 PC Card Controller Device Class

The requirements expressed in this section apply to PC Card controller devices and the PC Card slots.

Power management of *PC Cards* is not defined by this specification. PC Card power management is defined by the relevant power management specification for the card's device class (for example, network, modem, and so on), in conjunction with the PC Card standard (for 16-bit cards) or the PCI Power Management Specification (for CardBus cards).

A.10.1 Power State Definitions

State	Status	Definition
D0	*Required*	Card status change interrupts are fully functional. Card functional interrupts are fully functional. Controller context (for example, memory, I/O windows) is fully functional. Controller interface is fully functional (processor can access cards). Power to cards (slots) is available (may be on or off under software control). The controller is at its highest power consumption level. Bus command response time is at its fastest level. PC Cards can be in any D*x* power state (D0-D3). **Note:** In D0 state, CSTSCHG interrupts can be passed to a system from a powered down PC Card (for more detail, refer to section 5.2.11.2 of PC Card Standard, Electrical Specification).
D1	*Optional*	Card status change interrupts are disabled. CSTSCHG interrupt events are still detectable by the controller and cause the bus-specific wake signal to be asserted if wake is enabled on the controller. Card functional interrupts are disabled. Controller context is preserved (all register contents must be maintained but memory and I/O windows need not be functional). Controller interface is non-functional (processor cannot access cards). Power to cards (slots) is available (may be on or off; retains power setting it had at time of entry to D1). Power-level consumption for the controller is high but less than D0. The time required to restore the function from the D1 state to the D0 state is quicker than resumption from D3. Bus command response time is equal to or slower than in D0. PC Cards can be in the D1, D2, or D3 power states (not D0). **Note:** In D1 state, CSTSCHG interrupts can be passed to a system from a powered-down PC Card (for more detail, refer to section 5.2.11.2 of PC Card Standard, Electrical Specification).
D2	*Optional*	• Functionally the same as D1 (may be implemented instead of D1 in order to allow bus and/or system to enter a lower-power state).

(continued)

State	Status	Definition
D3	*Required*	Card status change interrupt: Disabled and need not be detected. Card functional interrupt: Disabled and need not be detected. Controller context (for example, memory, I/O windows): Lost. Controller interface: Non-functional (processor can not access cards). Clock to controller: Off. Power to cards (slots): Off (card context lost). **Note:** If Vcc is removed (for example, PCI Bus B3) while the device is in the D3 state, a bus-specific reset (for example, PCI RST#) must be asserted when power is restored and functions will then return to the D0 state with a full power-on reset sequence. Whenever the transition from D3 to D0 is initiated through assertion of a bus-specific reset, the power-on defaults will be restored to the function by hardware just as at initial power up. The function must then be fully initialized and reconfigured by software.

A.10.2 Power Management Policy

The PC Card controller is a bus controller. As such, its power state is dependent on the devices plugged into the bus (child devices). OSPM will track the state of all devices on the bus and will put the bus into the best possible power state based on the current device requirements on that bus. For example, if the PC Card cards are all in the D1 state, OSPM will put the PC Card controller in the D1 state.

Present State	Next State	Cause
D2/D3	D0	Any card in any slot needing to transition to state D0 due to a wake event or because of system usage.
D0	D1	No card in any slot is in state D0.
D0	D2	No card in any slot is in state D0 or D1.
D0	D3	All cards in all slots are in state D3.

A.10.3 Wake Events

A wake event is any event that would normally assert the controller’s status change interrupt (for example, card insertion, card battery state change, card ReqAttn event, and so on) or ring-indicate signal.

A.10.4 Minimum Power Capabilities

A PC Card controller device conforming to this specification must support the D0 and D3 states. Support for the D1 or D2 states is optional.

A.11 Storage Device Class

The requirements expressed in this section apply to ATA hard disks, floppy disks, ATAPI and SCSI CD-ROMs, and the IDE channel.

A.11.1 Power State Definitions

A.11.1.1 Hard Disk, CD-ROM and IDE/ATAPI Removable Storage Devices

State	Status	Definition
D0	*Required*	Drive controller (for example, interface and control electronics) is functional. Interface mode context (for example, communications timings) is programmed.
D1	*Optional*	Drive controller (for example, interface and control electronics) is functional. Interface mode context (for example, communications timings) is preserved. Drive motor (for example, spindle) is stopped, with fast-start mode enabled, if available. Laser (if any) is off. Recommended latency to return to D0 is less than 5 seconds. Power consumption in D1 should be no more than 80% of power consumed in D0. **Note:** For ATA devices, this state is invoked by the Standby Immediate command.
D2	*N/A*	This state is not defined for storage devices.
D3	*Required*	Drive controller (for example, interface and control electronics) is not functional; context is lost. Interface mode (for example, communications timings) is not preserved. Drive motor (for example, spindle) is stopped. Laser (if any) is off. Power consumption in D3 is no more than 10% of power consumed in D0. **Note:** For ATA devices, this state is invoked by the “sleep” command.

A.11.1.2 Floppy Disk Devices

State	Status	Definition
D0	*Required*	Drive controller (for example, interface and control electronics) is functional. Drive motor (for example, spindle) is turning.
D1	*N/A*	This state is not defined for floppy disk drives.
D2	*N/A*	This state is not defined for floppy disk drives.
D3	*Required*	Drive controller (for example, interface and control electronics) is not functional; context is lost. Drive motor (for example, spindle) is stopped.

A.11.1.3 IDE Channel Devices

State	Status	Definition
D0	*Required*	Adapter is functional. Adapter interface mode (for example, communications timings) is programmed. Power is applied to the bus (and all devices connected to it).
D1	*N/A*	This state is not defined for the IDE Channel.
D2	*N/A*	This state is not defined for the IDE Channel.
D3	*Required*	Adapter is non-functional. Adapter interface mode (for example, communications timings) is not preserved. Power to the bus (and all devices connected to it) may be off.

A.11.2 Power Management Policy

A.11.2.1 Hard Disk, Floppy Disk, CD-ROM and IDE/ATAPI Removable Storage Devices

Present State	Next State	Cause
D3	D0	Device usage (high-priority I/O).
D0	D1*	Device inactivity (no high-priority I/O) for some period of time (T1).
D0	D3	Device inactivity (no high-priority I/O) for a period of time (T2=>T1). System enters sleeping state.
D1*	D0	Device usage (High-priority I/O).

* If supported.

Note: For ATA, the D3-to-D0 transition requires a reset of the IDE channel. This means that both devices on a channel must be placed into D3 at the same time.

A.11.2.2 IDE Channel Devices

Present State	Next State	Cause
D3	D0	Any device on the channel needing to transition to a state other than state D3.
D0	D3	All devices on the channel in state D3.

A.11.3 Wake Events

Storage devices with removable media can, optionally, signal wake upon insertion of media using their bus-specific notification mechanism. There are no other wake events defined for Storage devices.

A.11.4 Minimum Power Capabilities

A hard disk, CD-ROM or IDE/ATAPI removable storage device conforming to this specification must support the D0 and D3 states. Support for the D1 state is optional.

A floppy disk and IDE channel device conforming to this specification must support the D0 and D3 states.

APPENDIX B: Video Extensions

B ACPI Extensions for Display Adapters

B.1 Introduction

This section of the document describes a number of specialized ACPI methods to support motherboard graphics devices.

In many cases, system manufacturers need to add special support to handle multiple output devices such as panels and TV-out capabilities, as well as special power management features. This is particularly true for notebook manufacturers. The methods described here have been designed to enable interaction between the system BIOS, video driver, and OS to smoothly support these features.

Systems containing a built-in display adapter are required to implement the ACPI Extensions for Display Adapters.

Table B-1 Video Extension Object Requirements

Method	Description	
_DOS	Enable/Disable output switching	Required if system supports display switching or LCD brightness levels
_DOD	Enumerate all devices attached to display adapter	Required if integrated controller supports output switching
_ROM	Get ROM Data	Required if ROM image is stored in proprietary format
_GPD	Get POST Device	Required if _VPO is implemented
_SPD	Set POST Device	Required if _VPO is implemented
_VPO	Video POST Options	Required if system supports changing post VGA device
_ADR	Return the unique ID for this device	Required
_BCL	Query list of brightness control levels supported	Required if embedded LCD supports brightness control
_BCM	Set the brightness level	Required if _BCL is implemented
_DDC	Return the EDID for this device	Required if embedded LCD does not support return of EDID via standard interface
_DCS	Return status of output device	Required if the system supports display switching (via hotkey)
_DGS	Query graphics state	Required if the system supports display switching (via hotkey
_DSS	Device state set	Required if the system supports display switching (via hotkey).

B.2 Definitions

- **Built-in display adapter**. This is a graphics chip that is built into the motherboard and cannot be replaced. ACPI information is valid for such built-in devices.
- **Add-in display adapter**. This is a graphics chip or board that can be added to or removed from the computer. Because the system BIOS cannot have specific knowledge of add-in boards, ACPI information is not available for add-in devices.
- **Boot-up display adapter**. This is the display adapter programmed by the system BIOS during machine power-on self-test (POST). It is the device upon which the machine will show the initial operating system boot screen, as well as any system BIOS messages.
- The system can change the boot-up display adapter, and it can switch between the built-in adapter and the add-in adapter.
- **Display device**. This is a synonym for the term display adapter discussed above.
- **Output device**. This is a device, which is a recipient of the output of a display device. For example, a CRT or a TV is an output device.

B.3 ACPI Namespace

This is an example of the display-related namespace on an ACPI system:

```
GPE                // ACPI General-purpose HW event
    _L0x              // Notify(VGA, 0x80) to tell OSPM of the event, when user presses
// the hot key to switch the output status of the monitor.
// Notify(VGA, 0x81) to tell the event to OSPM, when there are any
// changes on the sub-devices for the VGA controller

SB
|- PCI
    |- VGA            // Define the VGA controller in the namespace
        |- _PS0 / PR0
        |- _PS1 / PR1
        |- _PS3
        |- _DOS        // Method to control display output switching
        |- _DOD        // Method to retrieve information about child output devices
        |- _ROM        // Method to retrieve the ROM image for this device
        |- _GPD        // Method for determining which VGA device will post
        |- _SPD        // Method for controlling which VGA device will post
        |- _VPO        // Method for determining the post options
        |- CRT         // Child device CRT
            |- _ADR    // Hardware ID for this device
            |- _DDC    // Get EDID information from the monitor device
            |- _DCS    // Get current hardware status
            |- _DGS    // Query desired hardware active \ inactive state
            |- _DSS    // Set hardware active \ inactive state
            |- _PS0   \
            |- _PS1    - Power methods
            |- _PS2    - for the output device
            |- _PS3   /
        |- LCD         // Child device LCD
            |- _ADR    // Hardware ID for this device
            |- _DDC    // Get EDID information from the monitor device
            |- _DCS    // Get current hardware status
            |- _DGS    // Query desired hardware active \ inactive state
            |- _DSS    // Set hardware active \ inactive state
            |- _BCL    // Brightness control levels
            |- _BCM    // Brightness control method
            |- _PS0   \
            |- _PS1    - Power methods
            |- _PS2    - for the output device
            |- _PS3   /
        |- TV          // Child Device TV
            |- _ADR    // Hardware ID for this device
            |- _DDC    // Get EDID information from the monitor device
            |- _DCS    // Get current hardware status
            |- _DGS    // Query desired hardware active \ inactive state
            |- _DSS    // Set hardware active \ inactive state
```

The LCD device represents the built-in output device. Mobile PCs will always have a built-in LCD display, but desktop systems that have a built-in graphics adapter generally don't have a built-in output device.

Notify(VGA, 0x80) is an event that should be generated whenever the state of one of the output devices attached to the VGA controller has been switched or toggled. This event will, for example, be generated when the user presses a hotkey to switch the active display output from the LCD panel to the CRT.

Notify(VGA, 0x81) is an event that should be generated whenever the state of any output devices attached to the VGA controller has been changed. This event will, for example, be generated when the user plugs-in or remove a CRT from the VGA port. In this case, OSPM will re-enumerate all devices attached to VGA controller.

The event number is standardized because the event will be handled by the OS directly under certain circumstances (see _DOS method later in this specification).

B.4 Display-specific Methods

The methods described in this section are all associated with specific display devices. This device-specific association is represented in the namespace example in the previous section by the positioning of these methods in a device tree.

B.4.1 _DOS (Enable/Disable Output Switching)

Many ACPI machines currently reprogram the active display output automatically when the user presses the display toggle switch on the keyboard. This is done because most video device drivers are currently not capable of being notified synchronously of such state changes. However, this behavior violates the ACPI specification, because the system modifies some graphics device registers.

The existence of the _DOS method indicates that the system BIOS is capable of automatically switching the active display output or controlling the brightness of the LCD. If it exists at all, the _DOS method must be present for all display output devices. This method is required if the system supports display switching or LCD brightness control.

Arguments:

Bit 1:0

0: The system BIOS should not automatically switch (toggle) the active display output, but instead just save the desired state change for the display output devices in variables associated with each display output, and generate the display switch event. OSPM can query these state changes by calling the _DGS method.

1: The system BIOS should automatically switch (toggle) the active display output, with no interaction required on the OS part. The display switch event should not be generated in this case.

2: The _DGS values should be locked. It's highly recommended that the system BIOS do nothing when hotkey pressed. No switch, no notification.

3: Reserved

Bit 2

0: The system BIOS should automatically control the brightness level of the LCD when the power changes from AC to DC.

1: The system BIOS should not automatically control the brightness level of the LCD when the power changes from AC to DC.

Return Value:

None

The _DOS method controls this automatic switching behavior. This method should do so by saving the parameter passed to this method in a global variable somewhere in the BIOS data segment. The system BIOS then checks the value of this variable when doing display switching. This method is also used to control the generation of the display switching **Notify**(VGA, 0x80/0x81).

The system BIOS, when doing switching of the active display, must verify the state of the variable set by the _DOS method. The default value of this variable must be 1.

B.4.2 _DOD (Enumerate All Devices Attached to the Display Adapter)

This method is used to enumerate devices attached to the display adapter. This method is required if integrated controller supports output switching.

On many laptops today, a number of devices can be connected to the graphics adapter in the machine. These devices are on the motherboard and generally are not directly enumerable by the video driver; for this reason, all motherboard VGA attached devices are listed in the ACPI namespace.

These devices fall into two categories:

- **Video output devices**. For example, a machine with a single display device on the motherboard can have three possible output devices attached to it, such as a TV, a CRT, or a panel.
- **Non-video output devices**. For example, TV Tuner, DVD decoder, Video Capture. They just attach to VGA and their power management closely relates to VGA.

Both ACPI and the video driver have the ability to program and configure output devices. This means that both ACPI and the video driver must enumerate the devices using the same IDs. Because there are no standard configurations for display output devices, no standard ID generation mechanism can be used.

To solve this problem, the _DOD method returns a list of devices attached to the graphics adapter, along with device-specific configuration information. This information will allow the cooperation between ACPI components and the video driver.

Every child device enumerated in the ACPI namespace under the graphics adapter must be specified in this list of devices.

Arguments:

None

Return Value:

A buffer containing an array of video device attributes as described in the table below.

Sample Code:

```
Method (_DOD, 0) {
Return (package(){
0x00010100,    // CRT, detectable by BIOS
0x00010110,        // LCD panel, detectable by BIOS
0x00000200,        // TV, not detectable by the BIOS
0x00020000})       // empty(unknown) device, attached to VGA device
}
```

Table B-2 Video Output Device Attributes

Bits	Definition
15:0	Device ID. The device ID must match the IDs specified by Video Chip Vendors. They must also be unique under VGA namespace.
16	BIOS. Can detect the device.
17	Non-VGA output device whose power is related to the VGA device. This can be used when specifying devices like TV Tuner, DVD decoder, Video Capture, and so on.
20:18	For VGA multiple-head devices, this specifies head ID.
31:21	Reserved (must be 0)

Table B-3 Commonly-used Device IDs

Bits	Definition
0x0100	Monitor
0x0110	Panel
0x0200	TV
0	Other

Please contact the Video Chip vendors for other IDs.

B.4.3 _ROM (Get ROM Data)

This method is used to get a copy of the display devices' ROM data. This method is required when the ROM image is stored in a proprietary format such as stored in the system BIOS ROM. This method is not necessary if the ROM image can be read through standard PCI interface (using ROM BAR).

The video driver can use the data returned by this method to program the device. The format of the data returned by this function is a large linear buffer limited to 4 KB. The content of the buffer is defined by the graphics independent hardware vendor (IHV) that builds this device. The format of this ROM data will traditionally be compatible with the ROM format of the normal PCI video card, which will allow the video driver to program its device, independently of motherboard versus add-in card issues.

The data returned by the _ROM method is implementation-specific data that the video driver needs to program the device. This method is defined to provide this data as motherboard devices typically don't have a dedicated option ROM. This method will allow a video driver to get the key implementation specific data it needs so that it can fully control and program the device without BIOS support.

Arguments:

Arg0: Offset of the display device ROM data.

Arg1: Size of the buffer to fill in (up to 4K).

Output:

Buffer of bytes

B.4.4 _GPD (Get POST Device)

This method is required if the _VPO method is implemented.

This method is used as a mechanism for the OS to query a CMOS value that determines which VGA device will be posted at boot. A zero return value indicates the motherboard VGA will be posted on the next boot, a 1 indicates a PCI VGA device will be posted, and a 2 indicates an AGP VGA device will be posted.

Arguments:

None

Return Value:

A 32-bit value

Bit 1:0

00 – Post the motherboard VGA device

01 – Post an add-in PCI VGA device

10 – Post an add-in AGP VGA device

11 – Reserved

Bit 31:2

Reserved (must be 0)

B.4.5 _SPD (Set POST Device)

This method is required if the _VPO method is implemented.

This method is used as a mechanism for the OS to update a CMOS value that determines which video device will be posted at boot. A zero argument will cause the “motherboard” to be posted on the next boot, a 1 will cause an add-in PCI device to be posted, and a 2 will cause an add-in AGP device to be posted.

Arguments:

Bit 1:0

00 – Post the motherboard VGA device

01 – Post an add-in PCI VGA device

10 – Post an add-in AGP VGA device

11 – Reserved

Bit 31:2

Reserved (must be 0)

Return Value:

A 32-bit value

0 – Success

non-zero – Failure

Sample Code:

```
Method (_SPD, 1)   { // Make the motherboard device the device to post }
```

B.4.6 _VPO (Video POST Options)

This method is required for systems with video devices built onto the motherboard and support changing post-VGA device.

This method is used as a mechanism for the OS to determine what options are implemented. This method will be used in conjunction with _GPD and _SPD.

Arguments:

None

Return Value:

A 32-bit integer

Bit 0: Posting the motherboard VGA device is an option. (Bit 0 should always be set)

Bit 1: Posting a PCI VGA device is an option.

Bit 2: Posting an AGP VGA device is an option.

Bits 31:3: Reserved (must be zero)

B.5 Output Device-specific Methods

The methods in this section are methods associated with the display output device.

B.5.1 _ADR (Return the Unique ID for this Device)

This method returns a unique ID representing the display output device. All output devices must have a unique hardware ID. This method is required for all The IDs returned by this method will appear in the list of hardware IDs returned by the _DOD method.

Arguments:

None

Return Value:

32-bit device ID

Sample Code:

```
Method (_ADR, 0) {
        return(0x0100)  // device ID for this CRT
}
```

This method is required for all output display devices.

B.5.2 _BCL (Query List of Brightness Control Levels Supported)

This method allows the OS to query a list of brightness level supported by built-in display output devices. (This method in not allowed for externally connected displays.) This method is required if an integrated LCD is present and supports brightness levels.

Each brightness level is a number between 0 and 100, and can be thought of as a percentage. For example, 50 can be 50% power consumption or 50% brightness, as defined by the OEM.

Arguments:

None

Return Value:

Package of bytes

Sample Code:

```
Method (_BCL, 0) {
        // List of supported brightness levels
        Return (Package(7){
            80,     // level when machine has full power
            50,     // level when machine is on batteries
                    // other supported levels
            20, 40, 60, 80, 100}
}
```

The first number in the package is the level of the panel when full power is connected to the machine. The second number in the package is the level of the panel when the machine is on batteries. All other numbers are treated as a list of levels OSPM will cycle through when the user toggles (via a keystroke) the brightness level of the display.

These levels will be set using the _BCM method described in the following section.

B.5.3 _BCM (Set the Brightness Level)

This method allows OSPM to set the brightness level of the built-in display output device.

The OS will only set levels that were reported via the _BCL method. This method is required if _BCL is implemented.

Arguments:

Arg0: Desired brightness level

Return Value:

None

Sample Code:

```
Method (_BCM, 1) { // Set the requested level }
```

The method will be called in response to a power source change or at the specific request of the end user, for example, when the user presses a function key that represents brightness control.

B.5.4 _DDC (Return the EDID for this Device)

This method returns an EDID structure that represents the display output device. This method is required for integrated LCDs that do not have another standard mechanism for returning EDID data.

Arguments:

Arg0: Requested data length in bytes

0x01 – 128 bytes

0x02 – 256 bytes

Return Value:

0 – Failure, invalid parameter

non-zero – Requested data, 128 or 256 bytes of data

Sample Code:

```
Method (_DDC, 2) {
            If (LEqual (Arg0, 1)) { Return (Buffer(128){ ,,,, }) }
        If (LEqual (Arg0, 2)) { Return (Buffer(256){ ,,,, }) }
      Return (0)
  }
```

The buffer will later be interpreted as an EDID data block. The format of this data is defined by the VESA EDID specification.

B.5.5 _DCS (Return the Status of Output Device)

This method is required if hotkey display switching is supported.

Arguments:

None

Return Value:

32-bit device status

Table B-4 Device Status

Bits	Definition
0	Output connector exists in the system now
1	Output is activated
2	Output is ready to switch
3	Output is not defective (it is functioning properly)
4	Device is attached (this is optional)
5-31	Reserved (must be zero)

Example:

- If the output signal is activated by _DSS, _DCS returns 0x1F or 0x0F.
- If the output signal is inactivated by _DSS, _DCS returns 0x1D or 0x0D.
- If the device is not attached or cannot be detected, _DCS returns 0x0xxxx and should return 0x1xxxx if it is attached.
- If the output signal cannot be activated, _ DCS returns 0x1B or 0x0B.
- If the output connector does not exist (when undocked), _DCS returns 0x00.

B.5.6 _DGS (Query Graphics State)

This method is used to query the state (active or inactive) of the output device. This method is required if hotkey display switching is supported.

Arguments:

None

Return Value:

A 32-bit device state

Table B-5 Device State

Bits	Definition
0	0 – next desired state is inactive 1 – means next desired state is active
1-31	Reserved (must be zero)

The desired state represents what the user wants to activate or deactivate, based on the special function keys the user pressed. OSPM will query the desired state when it receives the display toggle event (described earlier).

B.5.7 _DSS – Device Set State

OSPM will call this method when it determines the outputs can be activated or deactivated. OSPM will manage this to avoid flickering as much as possible. This method is required if hotkey display switching is supported.

Arguments:
A 32-bit device state

Return Value:
None

Table B-6 Device Status

Bits	Definition
0	0 – Set output device to inactive state 1 – Set output device to active state
30	0 – Do whatever Bit31 requires to do 1 – Don’t do actual switching, but need to change _DGS to next state
31	0 – Don’t do actual switching, just cache the change 1 – If Bit30=0, commit actual switching, including any _DSS with MSB=0 called before If Bit30=1, don’t do actual switching, change _DGS to next state
1-29	Reserved (must be zero)

Example Usage:

OS may call in such an order to turn off CRT, and turn on LCD

CRT._DSS(0);

LCD._DSS(80000001L);

or

LCD._DSS(1);

CRT._DSS(80000000L);

OS may call in such an order to force BIOS to make _DGS jump to next state without actual CRT, LCD switching

CRT._DSS(40000000L);

LCD._DSS(C0000001L);

B.6 Note on State Changes

It is possible to have any number of simultaneous active output devices. It is possible to have 0, 1, 2 ... and so on active output devices. For example, it is possible for both the LCD device and the CRT device to be active simultaneously. It is also possible for all display outputs devices to be inactive (this could happen in a system where multiple graphics cards are present).

The state of the output device is separate from the power state of the device. The “active” state represents whether the image being generated by the graphics adapter would be sent to this particular output device. A device can be powered off or in a low-power mode but still be the active output device. A device can also be in an off state but still be powered on.

Example of the display-switching mechanism:

The laptop has three output devices on the VGA adapter. At this moment in time, the panel and the TV are both active, while the CRT is inactive. The automatic display-switching capability has been disabled by OSPM by calling _DOS(0), represented by global variable display_switching = 0.

The system BIOS, in order to track the state of these devices, will have three global variable to track the state of these devices. There are currently initialized to:

crt_active – 0

panel_active – 1

tv_active – 1

The user now presses the display toggle switch, which would switch the TV output to the CRT.

The system BIOS first updates three temporary variables representing the desired state of output devices:

want_crt_active – 1

want_panel_active – 1

want_tv_active – 0

Then the system BIOS checks the display_switching variable. Because this variable is set to zero, the system BIOS does not do any device reprogramming, but instead generates a **Notify**(VGA, 0x80/0x81) event for the display. This event will be sent to OSPM.

OSPM will call the _DGS method for each enumerated output device to determine which devices should now be active. OSPM will determine whether this is possible, and will reconfigure the internal data structure of the OS to represent this state change. The graphics modes will be recomputed and reset.

Finally, OSPM will call the _DSS method for each output device it has reconfigured.

Note: OSPM may not have called the _DSS routines with the same values and the _DGS routines returned, because the user may be overriding the default behavior of the hardware-switching driver or operating system-provided UI. The data returned by the _DGS method (the want_XXX values) are only a hint to the OS as to what should happen with the output devices.

If the display-switching variable is set to 1, then the BIOS would not send the event, but instead would automatically reprogram the devices to switch outputs. Any legacy display notification mechanism could also be performed at this time.

Index