Advanced Configuration and Power Interface Specification

Intel
Microsoft
Toshiba
Revision 1.0
December 22, 1996

Contents

1. Introduction

The Advanced Configuration and Power Interface (ACPI) specification is the key element in Operating System Directed Power Management (OSPM). OSPM and ACPI both apply to all classes of computers, explicitly including desktop, mobile, home, and server machines.

ACPI evolves the existing collection of power management BIOS code, APM APIs, PNPBIOS APIs, and so on into a well-specified power management and configuration mechanism. It provides support for an orderly transition from existing (legacy) hardware to ACPI hardware, and it allows for both mechanisms to exist in a single machine and 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.

This document describes the structures and mechanisms necessary to move to operating system (OS) directed power management and enable advanced configuration architectures—that is, the structures and mechanisms necessary to implement ACPI-compatible hardware and to use that hardware to implement OSPM support.

1.1 Principal Goals

ACPI is the key element in implementing OSPM. ACPI is 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 PCs to implement motherboard configuration and power management functions, using appropriate cost/function tradeoffs.
    - PCs include mobile, desktop, workstation, server, and home 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 opportunities for miscoordination 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.
    - The hardware can evolve independently from the OS, decoupling hardware ship cycles from OS ship cycles and allowing new ACPI-compatible hardware to work well with prior ACPI-compatible operating systems.
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.

- Today, power management only exists on a subset of PCs. This inhibits application vendors from supporting or exploiting it.
  - Moving power management functionality into the OS makes it available on every machine that the OS is installed on. The level of functionality (power savings, etc) will vary 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.
- Today, power management algorithms are restricted by the information available to the BIOS that implements them. This limits the functionality that can be implemented.
  - Centralizing power management information and directives from the user, applications, and hardware in the OS allows implementation of more powerful functionality. For example, an OS could 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 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 up 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 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 are loaded when docks, and so on are connected to the system, 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.
- 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's command methods allow 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.

Planned future versions of the Microsoft® Windows 95® and Windows NT® operating systems are examples of ACPI-compatible operating systems categorized in the right-most column of the previous table. Future ACPI-compatible versions of Windows 95 will provide the same legacy support as the current version of Windows 95.

1.4 OEM Implementation Strategy

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

- An OEM can adopt the OS vendor-provided ACPI driver 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 off or reset without OS consent is also needed to deal with various rare, but problematic, situations.

1.6 ACPI Specification and the Structure Of ACPI

This specification defines the ACPI interfaces; that is, the interfaces between the OS software, the hardware, and BIOS software. This specification also defines the semantics of these interfaces.

Figure 1-1 lays out the software and hardware components relevant to ACPI and how they relate to each other. This specification describes the *interfaces between* components, the contents of the ACPI Tables, and the related semantics of the other ACPI components. Note that the ACPI Tables, which describe a particular platform's hardware, are at heart of the ACPI implementation and the role of the ACPI BIOS is primarily to supply the ACPI Tables (rather than an 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.

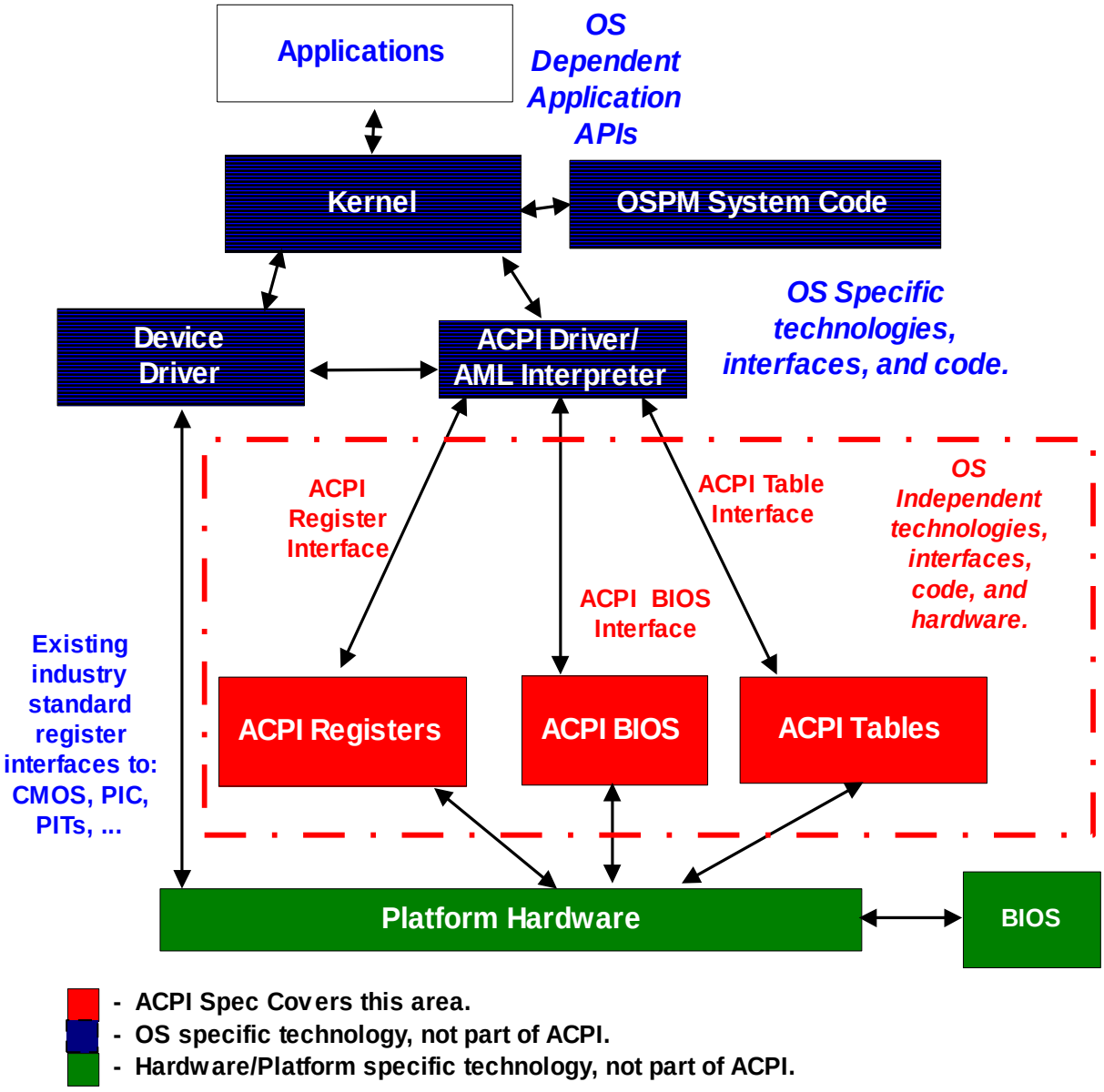


Figure 1-1 OSPM/ACPI Global System

There are three runtime components to ACPI:

- ACPI Tables - These 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 can make use of a p-code type of language, the interpretation of which is performed by the OS. That is, the OS contains and uses an AML interpreter that executes procedures encoded in AM and stored in the ACPI tables; ACPI Machine Language (AML) is a compact, tokenized, abstract kind of machine language.
- ACPI Registers - The constrained part of the hardware interface, described (at least in location) by the ACPI Tables.
- ACPI BIOS - 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 BIOS. Note that in the figure above, the boxes labeled "BIOS" and "ACPI BIOS" refer to the same component on a platform; the box labeled "ACPI BIOS" is broken out to emphasize that a portion of the BIOS is compatible with the ACPI specifications.

1.7 Minimum Requirements for OSPM/ACPI Systems

The minimum requirements for an OSPM/ACPI-compatible system are:

- A power-management timer (for more information, see section 4.7.2.1).
- A power or sleep button (for more information, see section 4.7.2.2).
- A real time clock wakeup alarm, (for more information, see section 4.7.2.4).
- Implementation of at least one system sleep state (for more information, see section 9.1).
- Interrupt events generate System Control Interrupts (SCIs) and the GP_STS hardware registers are implemented (for more information, see section 4.7.4.3).
- A Description Table provided in firmware (in the ACPI BIOS) for the platform system (main) board. For more information, see section 5.2)
- A user accessible fail-safe mechanism to either unconditionally reset or turn off the machine.

The minimum requirements for an OSPM/ACPI-compatible operating system are:

- Support for the following interfaces.
  - Interfaces specific to the IA platform:
    - The ACPI extended E820 memory reporting interface (for more information, see section 14).
  - Smart Battery, Selector, and Charger specifications.
  - All ACPI devices defined within this specification (for more information, see section 5.6.4).
  - The ACPI thermal model.
  - The power button as implemented in the fixed feature space (for more information, see section 4.6.2.2.1).
- ACPI AML interpreter.
- Plug and Play configuration support.
- OS-driven power management support (device drivers are responsible for restoring device context as described by the Device Power Management Class Specifications).
- Support the S1-S3 system sleeping states.

1.8 Target Audience

This specification is intended for the following users:

- OEMs who will be building ACPI-compatible hardware.
- Suppliers of ACPI-compatible operating systems, device drivers, and so on.
- Builders of ACPI descriptor tables and builders of tools to aid in constructing such tables.
- Authors of BIOS and Firmware codes.
- CPU and chip set vendors.
- Peripheral vendors.

1.9 Document Organization

The ACPI specification document is organized into four parts.

- The first part of the specification (sections 1, 2, and 3) introduces ACPI and provides an executive overview.
- The second part of the specification (sections 4 and 5) defines the ACPI hardware and software programming models.
- The third part (sections 6 through 13) specifies the ACPI implementation details; this part of the specification is primarily for developers.
- The fourth part (sections 14 through 16) are technical reference sections; section 15 is the ACPI Source Language (ASL) reference, parts of which are referred to by most of the other sections in the document.

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: Fully Off (D3), D2, D1, and Fully-On (D0).
- 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. Hardware: Defines a set of hardware interfaces that meet the goals of this specification.
- Section 5. Software: 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 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. Implementing Waking/Sleeping: Defines in detail the transitions between system working and sleeping states and their relationship to wake-up events. Refers to the reserved objects defined in sections 6, 7, and 8.
- Section 10: ACPI-Specific Devices: 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. Embedded Controller and SMBus: Defines the interfaces between an ACPI-compatible OS and an embedded controller and between an ACPI-compatible OS and an SMBus controller.

1.9.4 Technical Reference

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

- Section 14. Query System Address Map. 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.
- Section 15. ACPI Source Language (ASL) Reference. Defines the syntax of all the ASL statements that can be used to write ACPI control methods, along with example syntax usage.

- Section 16. ACPI Machine Language (AML) Specification: Defines the grammar of the language of the ACPI virtual machine language. An ASL translator (compiler) outputs AML.

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

Other specifications relevant to the ACPI specification are:

- *Smart Battery Charger Specification*, Revision 1.0, Duracell/Intel, Inc., June, 1996
- *Smart Battery Data Specification*, Revision 1.0, Duracell/Intel, Inc., February, 1995
- *Smart Battery System Windows Programming Interface*, Revision 1.0, Intel Inc., February, 1995
- *System Management Bus BIOS Interface Specification*, Revision 1.0, February, 1995
- *System Management Bus Specification*, Revision 1.0, Intel, Inc., February, 1995
- *System Management Bus Windows Programming Interface*, Revision 1.0, Intel Inc., February, 1995
- *The I2C-Bus and How To Use It* (includes the specification), Philips Semiconductors, January 1992

Documentation and specifications for the "On Now" power management initiative available from http://www.microsoft.com/hwdev/onnow.htm:

- *Toward the "On Now" Machine: The Evolution of the PC Platform.*
- Device Class Power Management Specifications:
  - *Device Class Power Management Reference Specification: Audio Device Class*
  - *Device Class Power Management Reference Specification: Communications Device Class*
  - *Device Class Power Management Reference Specification: Display Device Class*
  - *Device Class Power Management Reference Specification: Input Device Class*
  - *Device Class Power Management Reference Specification: Network Device Class*
  - *Device Class Power Management Reference Specification: PC Card Controller Device Class*
  - *Device Class Power Management Reference Specification: Storage Device Class*

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 (the definitions are presented as an alphabetical list).
- 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

ACPI:

Advanced Configuration and Power Interface - as defined in this document, 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 Name Space:

The ACPI Name Space is 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, etc. The OS dynamically changes the contents of the Name Space 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 Name Space comes from the Differentiated System Description Table, which contains the Differentiated Definition Block, and one or more other definition blocks.

AML:

ACPI control method ***M***achine ***L***anguage. Pseudocode for a virtual machine supported by an ACPI-compatible operating system and in which ACPI control methods are written. The AML encoding definition is provided in section 16.

ASL:

ACPI control method ***S***ource ***L***anguage. The programming language equivalent for ***AML***. ASL is compiled into AML images. The ASL statements are defined in section 15.

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 by including a different set of control methods for each chip set revision.

CPU, or processor:

The central processor unit (CPU), or processor, is 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 clock speed and cooling controls. The ACPI specification defines a working state, labeled G0, in which the processor executes instructions. Processor low power states, labeled C1 through C3, are also defined. In the low power states the processor executes no instructions, thus reducing power consumption and, potentially, operating temperatures. For more information, see section 8.

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 Name Space. Other definition blocks, which the OS can dynamically insert and remove from the active ACPI Name Space, can contain references to the Differentiated Definition Block. For more information, see section 5.2.7.

Device:

Hardware components outside the core chip set of a platform. Examples of devices are LCD panels, video adapters, IDE CD-ROM and hard disk controllers, COM ports, etc. In the ACPI scheme of power management, buses are devices. For more information, see section 3.3.2.

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), 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:

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

Embedded Controller:

Embedded controllers are 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:

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 (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:

The Firmware ACPI Control Structure (FACS) is a structure in read/write memory that the BIOS uses for handshaking between the firmware and the OS, and is passed to an ACPI-compatible OS via the Fixed ACPI Description Table (FACP). The FACS contains the system's hardware signature at last boot, the firmware waking vector, and the global lock.

Fixed ACPI Description Table:

An OEM must provide a Fixed ACPI Description Table (FACP) to an ACPI-compatible OS in the Root System Description Table. The FACP 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 Differentiated System Description Table (DSDT) that contains other platform implementation and configuration details. The OS always inserts the name space information defined in the Differentiated Definition Block in the DSDT into the ACPI Name Space 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 the ACPI driver 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. While a fixed feature event occurs an SCI is raised. For ACPI fixed-feature events, the ACPI driver (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 IO address space. ACPI defines *register blocks* for fixed features (each register block gets a separate pointer from the FACP ACPI table). For more information, see section 4.6.

General Purpose Event (GPE) 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.

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 that 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 486 instruction set and having an industry-standard PC architecture.

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

Multiple APIC Description Table:
The Multiple APIC Description Table (APIC) is used on systems supporting the APIC to describes the APIC implementation. Following the Multiple APIC Description Table is a list of APIC structures that declare the APIC features of the machine.

Object:
The nodes of the ACPI Name Space 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, etc. Package objects refer to other objects. Objects also have type, size, and relative name.

Object name:
Object names are part of the ACPI Name Space. There is a set of rules for naming objects.

OSPM:
OS-Directed Power Management is 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.

Persistent System Description Table:
Persistent System Description Tables are Definition Blocks, similar to Secondary System Description Tables, except a Persistent System Description Table can be saved by the OS and automatically loaded at every boot.

Power 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/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 tradeoffs among system speed, noise, battery life, processing speed, and AC power consumption. Power management is required for some system functions, such as appliance (e.g. answering machine, furnace control) operations.

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

Power Sources:
The battery and AC adapter that supply power to a platform.

P-Code:
P-code is a kind of simple "virtual machine language" that ACPI uses to describe control methods. Its principal advantages are that it is portable, compact, and powerful. There are many kinds of p-code; ACPI defines its own for reasons of simplicity. The ACPI specification defines an ACPI Source Language (ASL) and an ACPI Machine Language (AML). Control methods are written in ASL, for which there is a relatively simple specification. A compiler converts the ASL form of the p-code to the AML form. The ACPI-compatible OS contains a p-code interpreter for the AML form of the language.

Register Grouping:
A *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 affects. ACPI drivers are designed such that they will write zeros to all reserved bits in enable and status registers and preserve bits in control registers.

Root System Description Pointer:
An ACPI compatible system must provide a Root System Description Pointer in the systems low address space. This structure's only purpose is to provide the physical address of the Root System Description Table.

Root System Description Table:
The Root System Description Table starts with the signature 'RSDT,' followed by an array of physical pointers to the other System Description Tables that provide various information on other standards that are defined on the current system. The OS locates that Root System Description Table by following the pointer in the Root System Description Pointer structure.

Secondary System Description Table:
Secondary System Description Tables are a continuation of the Differentiated System Description Table. Multiple Secondary System Description Tables can be used as part of a platform description. After the Differentiated System Description Table is loaded into ACPI name space, 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: --battery, charger, selector list—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 levels 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.

SMBus:
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.

SMBus Interface:
ACPI defines a standard hardware and software communications interface between an OS bus driver and an SMBus Controller via an embedded controller.

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 a 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 a System Control Interrupt (SCI) when the temperature in a thermal zone moves above or below the trip point temperature.

2.2 Global System State Definitions

Global system states (Gx 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 (e.g. turning off the system's power through the movement of a large red switch). This operating mode is required by various government agencies and countries. It is implied by the entry of this off state through a mechanical means that the no electrical current is running through the circuitry and it can be worked on without damaging the hardware or endangering the

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.

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, etc.). Latency for returning to the Working state varies on the wakeup environment selected prior to entry of this state (for example, should the system answer phone calls, etc.). 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, devices (peripherals) are dynamically having their power state changed. The user will be able to select (through some user interface) 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:

S4 Non-Volatile Sleep (NVS) is 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 non-volatile storage file 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 NVS data set is found, certain aspects of the configuration of the machine has 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.

Note 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 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

Note 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. These states are defined very generically here. 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*, which are separate documents from 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 section 1.10.

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 addresses 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 *class* of device; it may not be defined by many classes of devices. 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 loose some context (i.e., 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 *class* of device; it may not be defined by many classes of devices. 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 init and load

Note: Devices often have different power modes within a given state. Devices can use these modes as long as they can automatically switch between these modes transparently from the software, without violating the rules for the current D*x* state the device is in. Low power modes that affect performance (i.e., 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 (Sx states) are types of sleeping states within the global sleeping state, G1. The Sx states are briefly defined below. For a detailed definition of the system behavior within each Sx state, see section 7.5.2. For a detailed definition of the transitions between each of the Sx states, see section 9.1.

S1 Sleeping State:

The S1 sleeping state is a low wake-up 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-up latency sleeping state. This state is similar to the S1 sleeping state except 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-up event.

S3 Sleeping State:

The S3 sleeping state is a low wake-up 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-up event.

S4 Sleeping State:

The S4 sleeping state is the lowest power, longest wake-up 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 the OS does not save any context nor enable any devices to wake the system. The system is in the "soft" off state and requires a complete boot when awakened. 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 (Cx states) are processor power consumption and thermal management states within the global working state, G0. The Cx states are briefly defined below. For a more detailed definition of each Cx state from the software perspective, see section 8.2. For a detailed definition of the Cx states from the hardware perspective, see section 4.7.1.12.

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 on this state is required to 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 declared in the FACP Table 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 of the C1 and C2 states. The worst-case hardware latency for this state is declared in the FACP Table, 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.

3. Overview

The ACPI interface gives the operating system (OS) direct control over the power management and Plug and Play functions of a computer. When it starts, the ACPI OS takes over these functions from legacy BIOS interfaces such as the APM BIOS and the PNPBIOS. Having done this, the OS is responsible for handling Plug and Play events as well as controlling power and thermal states based on user settings and application requests. ACPI provides low-level controls so the OS can 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.
- **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, etc. This mechanism is very flexible in that it does not define specifically how events are routed to the core logic chipset.
- **Battery management** - Battery management policy moves from the APM BIOS to the ACPI OS. The OS determines the Low battery and battery warning points, and the OS also calculates the battery remaining capacity and battery remaining life. 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 (CMBatt) interface. A CMBatt 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.
- **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.
- **System Management Bus 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.

3.1 System Power Management

Under OS-directed power management (OSPM), the operating system 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:

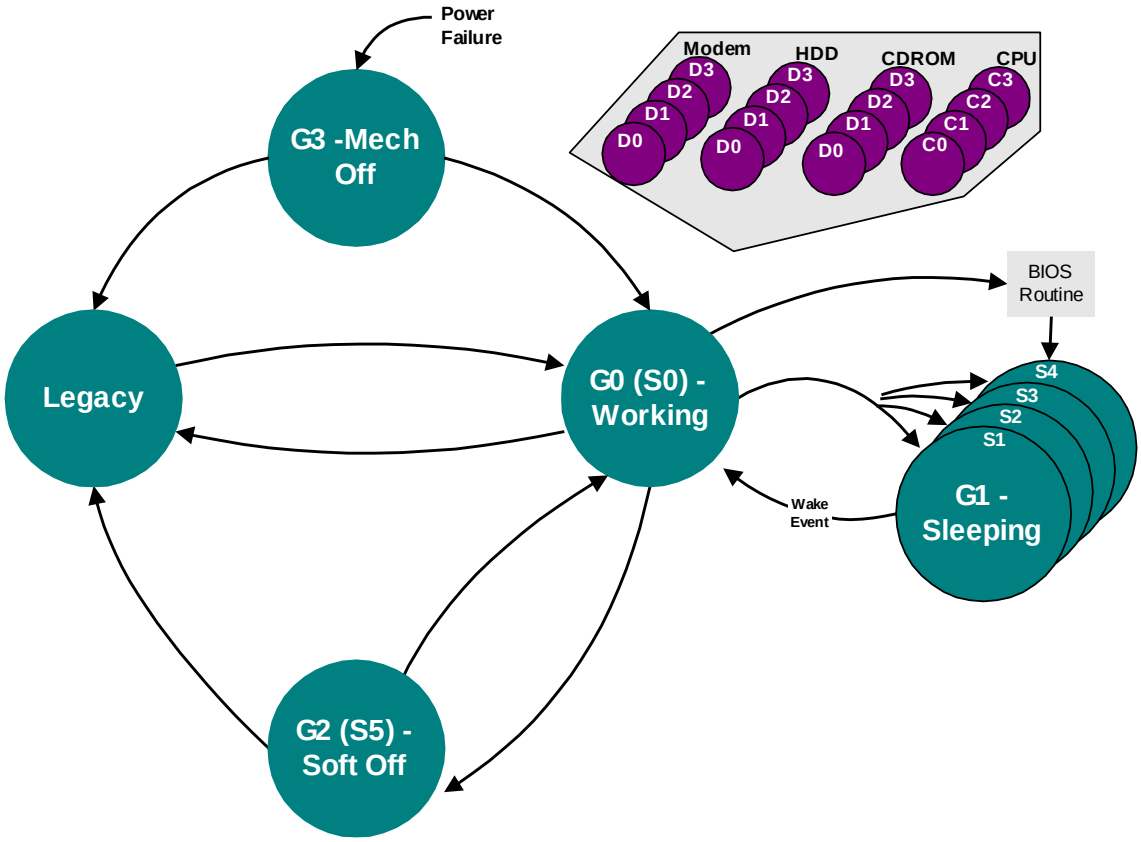


Figure 3-1 Global System Power States and Transitions

(See section 2.2 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 do some 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 up.)

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 tradeoffs between 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 and/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 a non-volatile storage (NVS) file and transition the hardware into a Soft Off 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 (e.g., a RISC system) transitions directly from the Mechanical Off state to the Working state. Users 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 to 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 could be 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 will be in some obvious place (for example, beside the keyboard on a laptop). Unlike today's on/off button, 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) and the embedded controller interface (see section 13).

3.2.2.2 Desktop PCs

Power-managed desktops will really 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, etc. 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-up from the Soft-Off 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 will often get the largest absolute power savings. Why? Because they have the largest hardware configurations, *and* it's not practical for somebody to hit the off switch when they leave at night.

- Day Mode - In day mode, servers will get 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. OS-driven power management 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 sleep and still be able to wake up and answer service requests coming in over the network, phone links, etc, 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 up 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 up 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 Wakeup operation devices use the 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 base-line level of power management support the OS can utilize. 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 (e.g. D3 - off), the OS can put the entire bus into the power supply mode appropriate for that state (e.g. D3 - off).

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

- 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 two 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 can be done by resetting the device).
- Device driver - what the device driver must do to restore the device to full on.
- Restore latency - how long it takes to restore the device to full on.

More specifically, power management specifications for each class of device (e.g., modem, network adapter, hard disk, etc) more precisely define the power states and power policy for the class. See section 2.3 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, etc.).

Device Wake-Up Characteristics. Each class of device has a standard definition of its wake-up policy.

Microsoft's 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 the capabilities of all the devices it controls to the OS. 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 up the machine.

Note: Some devices on the main board are enumerated by other busses. 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.

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 wakeup). The OS can use this information to infer what device and system power states the device can support wakeup from.
- 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.

3.4.2 Setting Device Power States

The Set Power State operation is used by the OS 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 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 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 itself in the D3 state). Or if a particular bus supports a low power supply state, the OS will put the bus into that state if all devices were in the D1 or D2 state. Whatever power state a device is put into, 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 any commands to the device.

The Set Power State operation is also used by the OS to enable power management features like wakeup (described in section 7).

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 will track all the devices on a given power resource. When all the devices on a resource have been turned off, the OS will turn 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 will first turn on the power resource using a control method and then signal 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. Note 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

The Get Power Status operation is used by the OS 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 a System Control Interrupt (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 Intel/Duracell 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 Wakeup operation is used by devices to wake the computer from a sleeping power state. This operation must not depend on the CPU because the CPU will not be powered. When it puts the computer in a sleeping power state, the OS will enable wakeup on those devices that the user's applications need to wake the machine. The OS will also make sure any bridges between the device and the core logic are in the lowest power state in which they can still forward the wakeup signal. When a device with wakeup 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 (e.g. 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 wakeup on those devices. The OS enables the wakeup 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 wakeup feature enabled can wake the machine. The OS will keep track of what power states the wakeup devices are capable of and will keep the machine in a power state in which the wakeup can still wake the machine1 (based on capabilities reported in the Description Table).

When the computer is in the Sleeping power state and a wakeup device decides to wake the machine, it signals to the ACPI chip set. The SCI status bit corresponding to the device waking the machine will be set, and the ACPI chip set will resume the machine. Once the OS is up and running again, it will clear the bit and handle the event that caused the wakeup. The control method for this event then uses the Notify command to tell the OS which device caused the wakeup.

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

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 a very low battery situation.

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 are 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 wakeup policy for the modem is very simple: when the phone rings and wakeup is enabled, wake the machine.

Based on that information, the modem and the COM port it is attached to 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.

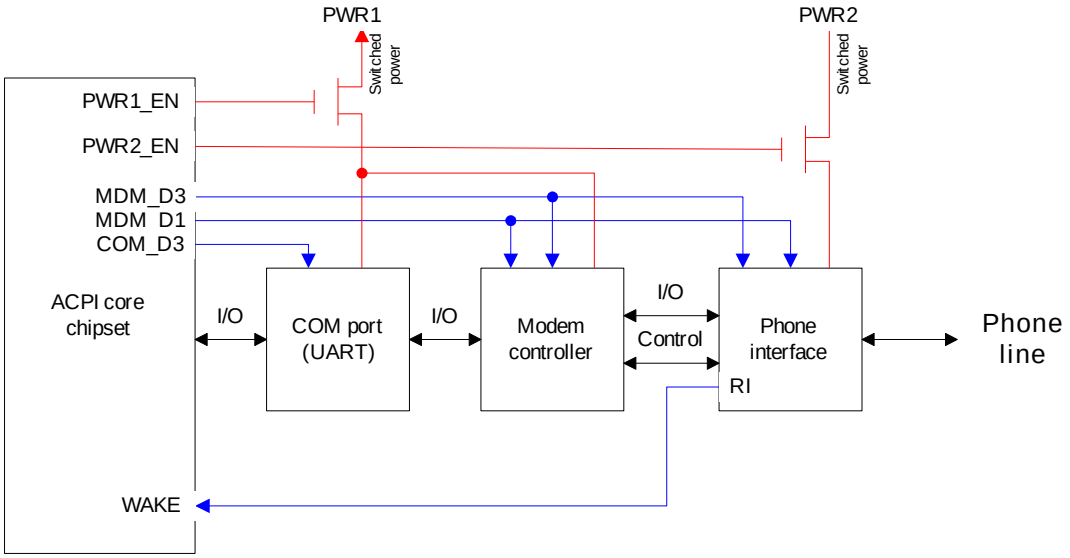


Figure 3-2 Example Modem and COM Port Hardware

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 will switch the modem to different power states according to the power policy defined for modems.

When an application opens the COM port, the OS will turn on the modem by putting it in the D0 state. Then if the application puts the modem in answer mode, the OS will put 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 ACPI driver 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, the ACPI driver 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.

The ACPI driver does not always turn off power resources when a given device is put into a lower power state. For example, assume that the PWR1 power plane also powers an LPT port that is active. Suppose the user terminates the modem application causing the COM port to be closed, therefore causing the modem to be shut off (state D3). As always, the ACPI driver checks to see which power resources are no longer needed. Because the LPT port is still active, PWR1 is in use. The ACPI driver will not turn off the PWR1 resource. It will continue the state transition process by running the modem's control method to switch the device to the D3 power state. The control method will cause 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 now closed, the same sequence of events would take place to put it into the D3 state. Note that these registers might not be in the device itself. For example, the control method could read the register that controls MDM_D3.

3.4.6 Getting the Modem's Power Status

Being an integrated modem, the device has 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), the ACPI driver runs a control method (_PSC) supplied in the modem's entry in the Differentiated Definition Block. This control method reads from whatever registers are necessary to determine the modem's power state.

3.4.6.1 Waking the Computer

As indicated in the capabilities, this modem can wake the machine from any device power state. Before putting the computer in a sleep state, the OS will enable wakeup on any devices that applications have requested to wake the machine. Then, it will choose the lowest sleeping state that can still provide the power resources necessary to allow all enabled wakeup 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 would put the modem in the D3 state because it supports wake up from that state. Finally, the OS saves a resume vector and puts the machine to sleep 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-up event. The chip set then awakens the system and the hardware will eventually pass control back to the OS (the waking mechanism differs depending on the sleeping state). Once the OS is running, it will put the device in the D0 state and begin 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 pre-empt 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 lower power states (which vary in power and latency) when it enters its idle loop.

The CPU states are defined in detail in section 8.

3.6 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 description tables as well as devices that are controlled through other standards.

3.6.1 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 Differentiated Description Table. This table will have control methods that tell 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, the ACPI driver 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.

3.7 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 would set the status bit and fire 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.

3.8 Battery Management

Battery management policy moves from the APM BIOS to the ACPI-compatible OS. The OS determines the low battery point and battery warning point. The OS also calculates the remaining battery capacity and remaining battery life.

An ACPI-compatible battery device needs either a Smart Battery subsystem interface or a Control Method Battery (CMBatt) 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.
- **CMBatt** 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 battery device control methods, see section 11.2.2.

This section describes how a CMBatt interface works and what kind of AML code interface is needed .

3.8.1 CMBatt Diagram

CMBatt is accessed by an AML code interface so a system hardware designer can choose any communication interface at the hardware level. One example is shown in Figure 3-3. The battery has built-in information and can communicate with embedded controller (EC) using the I^2C interface. The AML code interface returns the battery information stored in the RAM of the EC. The OS can set the battery trip point at which an SCI will be generated.

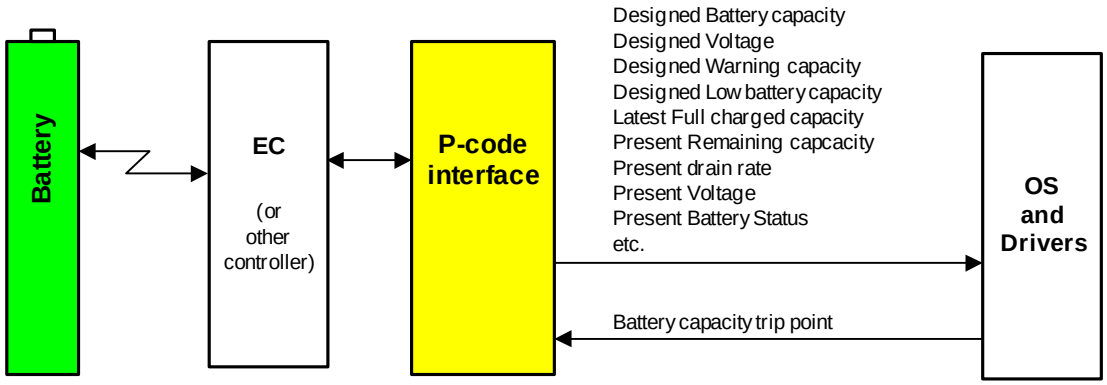


Figure 3-3 Control Method Battery Diagram

3.8.2 Battery Events

The AML code that handles an SCI for a battery event notifies the system of the batteries upon which the status might have changed.

When a battery device is inserted into the system or removed from the system, the hardware asserts a GP event. The AML code handler for this event will issue a Notify(, 0x00) on the battery device to initiate the standard device Plug and Play actions.

When the present state of the battery has changed or when the trip point set by the _BTP control method is crossed, the hardware will assert a GP event. The AML code handler for this event issues a Notify(,0x80) on the battery device.

3.8.3 Battery Capacity

CMBatt reports the designed capacity, the latest full-charged capacity, and the present remaining capacity. Battery remaining capacity decreases during usage, and it also changes depending on the environment. Therefore, the OS must use latest full-charged capacity to calculates the battery percentage.

A system must use either [mA] or [mW] for the unit of battery information calculation and reporting. Mixing [mA] and [mW] is not allowed on a system.

CMBatt reports the OEM-designed initial warning capacity and OEM-designed initial low capacity . An ACPI-compatible OS determines independent warning and low battery capacity based on these initial capacities.

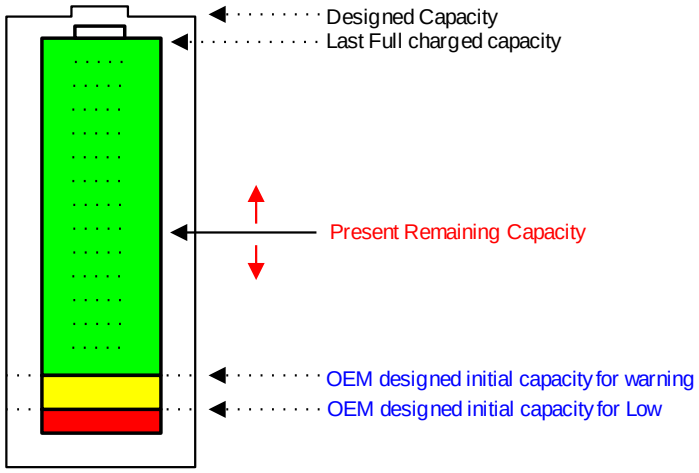


Figure 3-4 Reporting Battery Capacity

3.8.4 Battery Gas Gauge

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

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

CMBatt 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:

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

Note 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 CMBatt can report the percentage directly to OS. Reporting the “Last Full Charged capacity =100” and “BatteryPresentRate=0xFFFFFFFF” means that ”Battery remaining capacity” is a battery percentage and the its value should be in the range 0 through 100 as follows.

$$\text{Remaining Battery Percentage[\%]} = \frac{\text{Battery Remaining Capacity [=0 ~ 100]}}{\text{Last Full Charged Capacity [=100]}} * 100$$

$$\text{Remaining Battery Life [h]} = \frac{\text{Battery Remaining Capacity [mAh/mWh]}}{\text{Battery Present Rate [=0xFFFFFFFF]}} = \text{unknown}$$

CMBatt have an OEM-designed initial capacity for warning and initial capacity for low. An ACPI-compatible OS can determine independent warning and low battery capacity values based on the designed warning capacity and designed low capacity shown in Figure 3-5 and Table 3-1.

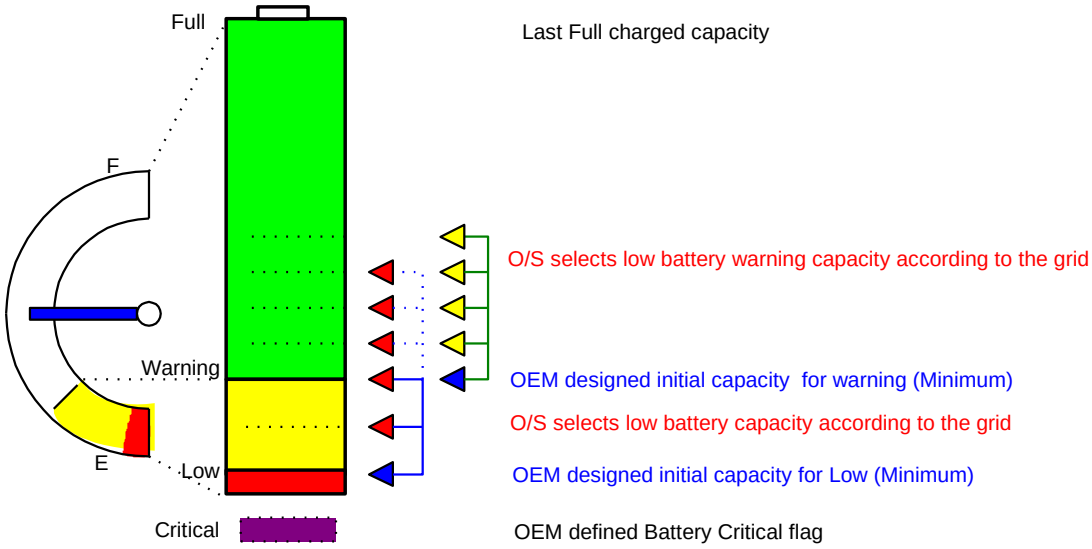


Figure 3-5 Low Battery and Warning

CMBatt and an ACPI-compatible OS manage the three battery level shown in Table 3-1.

Table 3-1 Low Battery Levels

Level	Description
Warning	The battery is approaching and is close to the Low level. This is an early warning; the battery is not yet in the Low capacity. The OS can determine a built-in low battery warning point that will not fall below the OEM-defined initial remaining-capacity for warning. The OS will use this warning level to notify the user via UI.
Low	The Battery is low. The OS determines a built-in low battery level that will not fall below the OEM-defined initial remaining-capacity for low. At this level , the OS will transition the system to a user defined state (i.e., a sleep state , shutdown). If the remaining capacity is not accurate and hardware detects the low battery before the remaining capacity reaches the OS-specified low level, CMBatt can report the remaining-capacity as same as (or less than) OEM-designed initial capacity to alert the OS that the battery is low.
Critical	Battery is fully discharged and cannot supply any more power to the system. This level does not mean battery failure. The system cannot use the battery until it has been re-charged or replaced. The system reports this condition by setting the “Critical” flag in the Battery State field of the _BST (battery status) object. This is an emergency situation because there is not

Level	Description
	enough time for a normal shutdown procedure. Therefore, the OS runs its emergency shutdown at this point. Critical battery level is defined by the OEM. Note: The amount of time taken to complete its emergency shutdown procedure depends on the OS and the system configuration.

If any battery in a system reaches a critical state (and it is a secondary battery) and is also discharging (as reported by the _BST control method), the OS will initiate an orderly but critical shutdown of the system. If there are multiple batteries in the system, the OS will continue to run even if one or more batteries reach critical so long as a critical battery device is not also discharging.

3.9 Thermal Management

ACPI moves the hardware cooling policies from the firmware to the OS. With the operating software watching over the system temperature, new cooling decisions can be made based on application load on the CPU as well as the thermal heuristics of the system. The OS will also be able to 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 thermal zones if necessary. Figure 3-6 is an example mobile PC diagram that depicts a standard 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*.

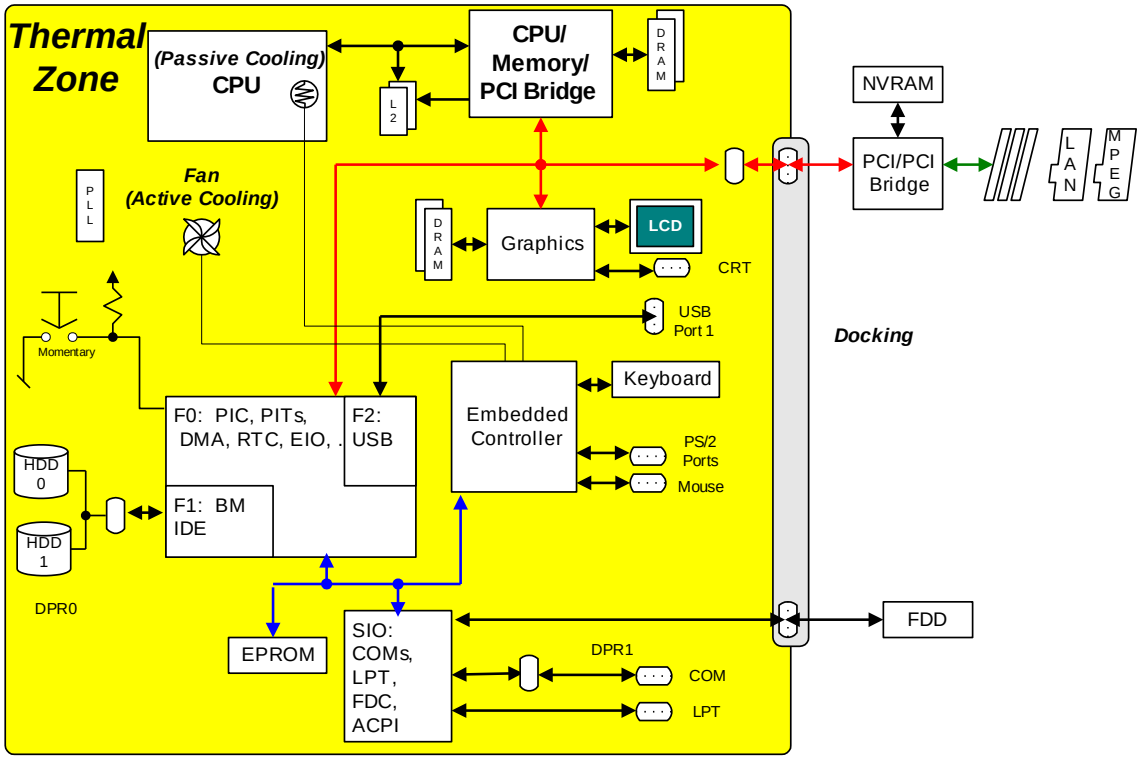


Figure 3-6 Thermal Zone

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.

3.9.1 Active and Passive Cooling

ACPI defines two cooling methods, Active and Passive:

- **Passive cooling**: OS reduces the power consumption of the processor to reduce the thermal output of the machine.
- **Active cooling**: OS takes a direct action such as turning on a fan.

Cooling method is a user-defined function that can be set in the OS through a control panel. These two cooling methods 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 CPU performance, but it creates fan noise, while Passive cooling reduces system performance, but it is quiet. (Note: Exceptions can be made. For example a battery charger, although it reduces the power to reduce heat, can be implemented as an active cooling device. For more information, see section 12. The significance of allowing the user to choose energy utilization is most critical to the operator of a mobile computer where battery charge preservation often has higher priority over maximum system performance. A mobile PC user is also more likely to be in a locale where quietness of the system is preferable over CPU performance. With these two cooling methods a PC user will be able to have a choice of *performance* versus *quietness* and some control over the rate of battery drain.

3.9.2 Performance vs. Silence

An ACPI-compatible OS offers a cooling choice to the end user at run-time that allows the user to adjust the rate of battery discharge between maximum and less than maximum. This flexibility is most important to a mobile PC user. For example, if a user is taking notes on her PC in a quiet environment, such as a library or a corporate meeting, she might want to set the cooling mode to *Silence*. This will sacrifice CPU speed, but it will turn off the fan to make the system quiet. Since the user is using the CPU to edit text, high CPU performance is probably not needed. On the other hand, another user might be in a lab running a graphics-intensive application and will need to set the cooling mode to *Performance* to utilize the maximum CPU bandwidth. 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 will run the CPU at maximum speed and keep the fan turned off.

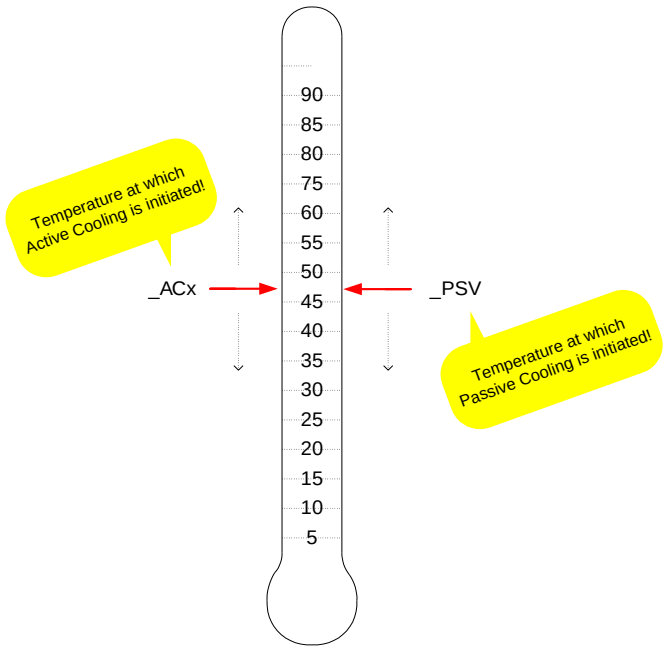


Figure 3-7 Active and Passive Policy Settings

To design a balanced thermal implementation, ACPI reserves the _AC*x* and _PSV objects to handle the two separate cooling modes. An OEM must choose the temperature value for each object so the OS will initiate the cooling policies at the desired target temperatures. (The ACPI specification defines Kelvin as the standard for temperature. All thermal control methods and 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 $1/10^{th}$ of a degree (e.g, 310.5). Kelvin is expressed as $\theta/K = T/°C + 273.2$.)

As shown in Figure 3-7, both control methods can return any temperature value that the OEM designates. But most importantly, the OEM can create each of the Performance and Silence modes by assigning different temperatures to each control method. Generally, if _ACx is set lower than _PSV, then it effectively becomes a Performance cooling mode. Conversely, if _PSV is set lower than _ACx, then it becomes a Silence cooling mode.

3.9.2.1 Cooling Mode: Performance

Figure 3-8 is an example of a performance-centric cooling model on an optimally implemented hardware. Besides setting the _ACx as the initial cooling policy, this system notifies the OS of a temperature change by raising an SCI every 5 degrees.

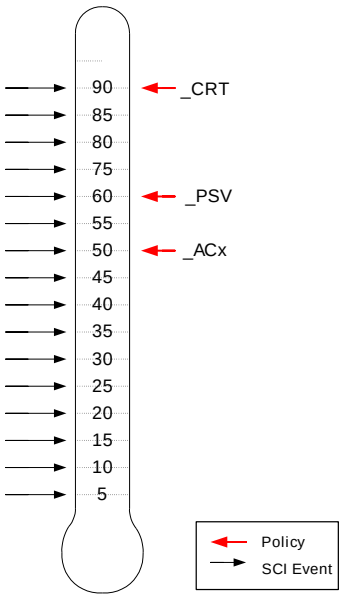


Figure 3-8 Performance Mode Example

This example turns the fan on when the OS receives an SCI at 50 degrees. If for some reason the fan does not reduce the system temperature, then at 60 degrees the OS will start throttling the CPU while running the fan. If the temperature continues to climb, the OS will be notified of a critical temperature at 90 degrees, at which point it will quickly shutdown the system.

3.9.2.2 Cooling Mode: Silence

Figure 3-9 is an example of a cooling model where quietness is the desired behavior of the system. The _PSV is set as the initial cooling policy. In this example, the OS is notified of a temperature change by raising an SCI every 5 degrees.

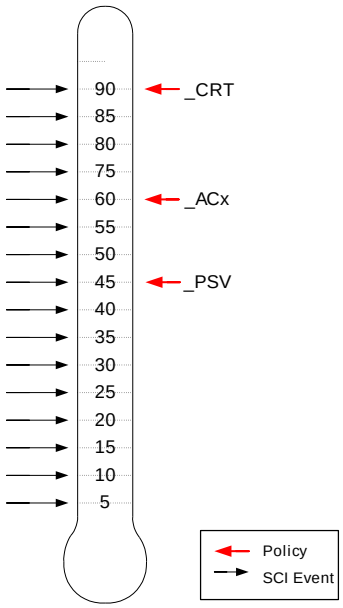


Figure 3-9 Silence Mode Example

This example initiates system cooling by CPU throttling when the OS receives an SCI at 45 degrees. If the throttling is not enough to reduce the heat, the OS will turn the fan on at 60 degrees while throttling the CPU. If the temperature continues to climb, the OS will be notified of a critical temperature at 90 degrees, at which point it will quickly shutdown the system.

3.9.3 Other Thermal Implementations

The ACPI thermal control model allows flexibility in thermal event design. An OEM that needs a less elaborate thermal implementation might consider some other design. For example, Figure 3-10 shows three other possibilities for implementing a thermal feedback design. These are only examples; many other designs are possible.

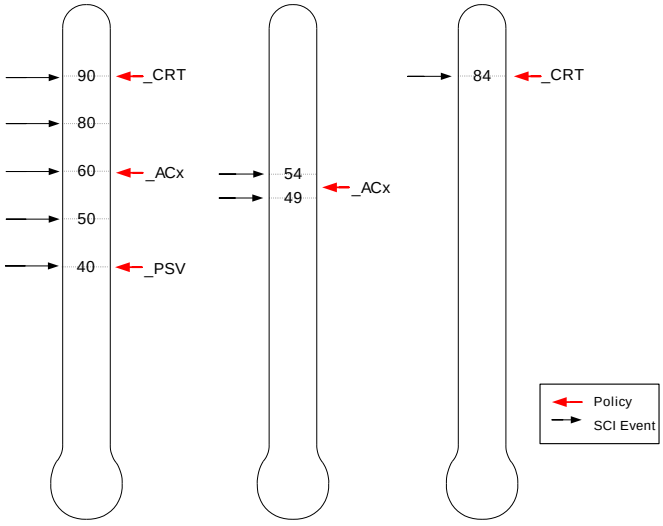


Figure 3-10 Example Thermal Cooling Implementations

3.9.4 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 the OS 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 the OS must shut down the entire system.

4. ACPI Hardware Specification

ACPI defines a standard mechanism for an ACPI-compatible OS to communicate to 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 features that go into fixed space by the following criteria:

- Performance sensitive features.
- Features drivers require during wakeup.
- Features that enable catastrophic failure recovery.

CPU clock control and the power management timer are in fixed space 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 space by the ACPI driver allows the ACPI driver to control the wakeup process without having to load the entire OS. For example, if a PCI configuration space access is needed, the bus enumerator is loaded with all drivers used by the enumerator. Having this hardware in the fixed space at addresses with which the OS can communicate without any other driver's assistance, allows the ACPI driver to gather information prior to making a decision as to whether it continues loading the entire OS or puts it back to sleep.

When the system has crashed, the ACPI driver can only access address spaces that need no driver support. In such a situation, the ACPI driver will attempt to honor fixed power button requests to transition the system to the G2 state.

4.2 Generic Programming Model

Although the fixed programming model requires registers to be defined at specified address locations, the generic programming model allows registers to reside in most address spaces. The ACPI driver directly accesses the fixed feature set registers, but ACPI relies on OEM-provided "pseudo code" (ASL-code) to access generic register space.

ASL code is written by the OEM in the ACPI System Language (ASL) to control generic feature control and event logic. The ASL language enables a number of things:

- Abstracts the hardware from the ACPI driver.
- 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 the ACPI driver 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 ASL code, which is associated with control methods, to be executed by the ACPI driver. By providing ASL-code, generic hardware can take on almost any form.

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

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 ACPI Machine Language (AML), which is a very compact machine language that the ACPI AML code interpreter executes.
The generic feature model is represented in the following block diagram. In this model the generic feature is described to the ACPI driver through AML code. This description takes the form of an object that sits in ACPI name space associated with the hardware that it is adding value to.

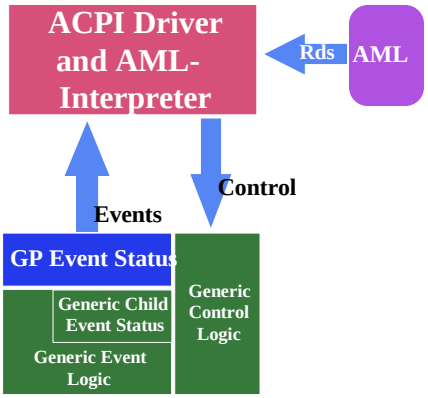


Figure 4-1 Generic Feature Model

As an example of a generic control feature, a platform might be designed such that the IDE HDD's D3 state has valued-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 name space, and associated with that object would be control methods that the ACPI driver calls to control the D3 state of the drive:

- _ON A control method to sequence the IDE drive to the D0 state
- _OFF A control method to sequence the IDE drive to the D3 state
- _STA 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 the OS and the hardware. The OS understands how to control power planes (turn them on or off or to get their status) through its defined power resource 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 _OFF control method. This enables the following sequence:

1. When the OS 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).
2. When the driver returns control, the OS calls the ACPI driver to place the drive in the D3 state.
3. The ACPI driver finds the object associated with the HDD and then finds within that object any AML code associated with the D3 state.
4. The ACPI driver executes the appropriate _OFF 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 a System Control Interrupt, or 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:

1. The ACPI driver 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.
2. The AML-code event handler collects the appropriate information and then executes an AML Notify operation to indicate to the ACPI driver that a particular bus needs re-enumeration.

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

- Which hardware is required or optional.
- How to design fixed features.

- How to design generic 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 a HIGH value is programmed to it. 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 writing this bit HIGH or LOW will result in the bit being read as HIGH or LOW (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 represents a bit set by a hardware signal's HIGH level (this bit is set by the level of the signal, not an edge). The bit is only cleared by software writing a one 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 routine that will be scheduled to be executed 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 is provided to allow the OS and hardware to sequence the platform between the various global system states (G0-G3) as illustrated in the following figure. Upon first power-up 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 HIGH (for more information, see section 4.7.2.5). 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 LOW). In either case, a transition out of the G3 state requires a total boot of the OS.

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. The OS

performs this transition by writing the ACPI_ENABLE value to the SMI_CMD which generates an event to the hardware to transition the platform to its ACPI mode. When hardware has finished the transition it sets the SCI_EN bit HIGH and returns control back to the OS. While in the G0 "working state," the OS 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).

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, the OS 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 the OS decides to enter a sleeping state it picks the most appropriate sleeping state supported by the hardware (OS policy examines what devices have enabled wakeup events and what sleeping these support). The OS initiates the sleeping transition by enabling the appropriate wakeup events and then programming the SLP_TYPx field with the desired sleeping state and then setting the SLP_ENx bit HIGH. The system will then enter a sleeping state; when one of the enabled wakeup events occurs, it will transition the system back to the working state (for more information, see section 9).

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 the OS 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 the OS 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 then setting the SLP_ENx bit HIGH.

The G1 sleeping state is represented by five possible sleeping states that the hardware can support. Each sleeping state has different power and wakeup 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 wakeup 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 wakeup sequences (for more information, see section 9).

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_ENx bit HIGH initiates the sequence). Exiting the G2 soft-off state requires rebooting the OS. In this case, an ACPI-only machine will re-enter the G0 state directly (hardware returns the SCI_EN bit HIGH), while an ACPI/Legacy machine transitions to the Legacy state (SCI_EN bit is LOW).

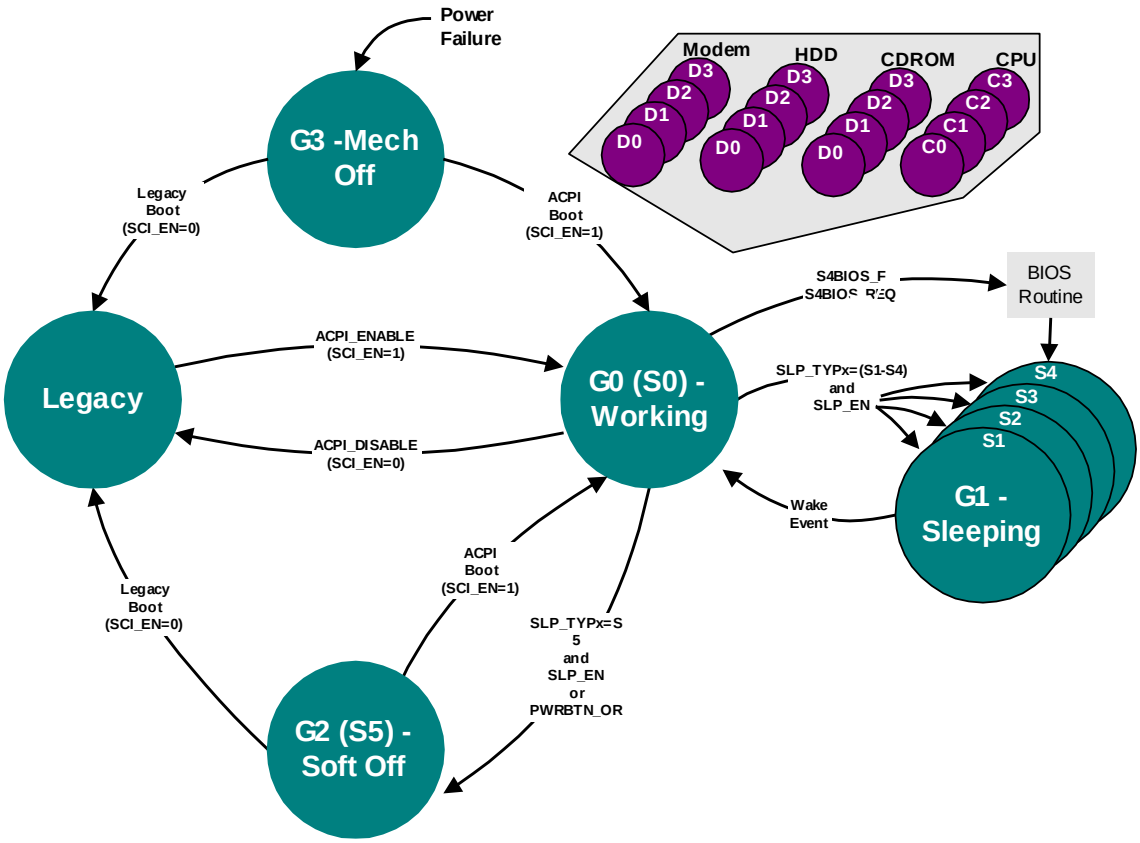


Figure 4-2 Global States and Their Transitions

The ACPI architecture defines mechanisms for hardware to generate events and control logic to implement this behavior model. Events are used to notify the OS that some action is needed, and control logic is used by the OS 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 wakeup 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 10-1).

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, the ACPI driver or an ACPI-aware driver acts as the event handler. For generic logic events the ACPI driver will schedule the execution of an OEM-supplied AML handler associated with the event.

For legacy systems, an event normally generates an OS-transparent interrupt, such as an 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.

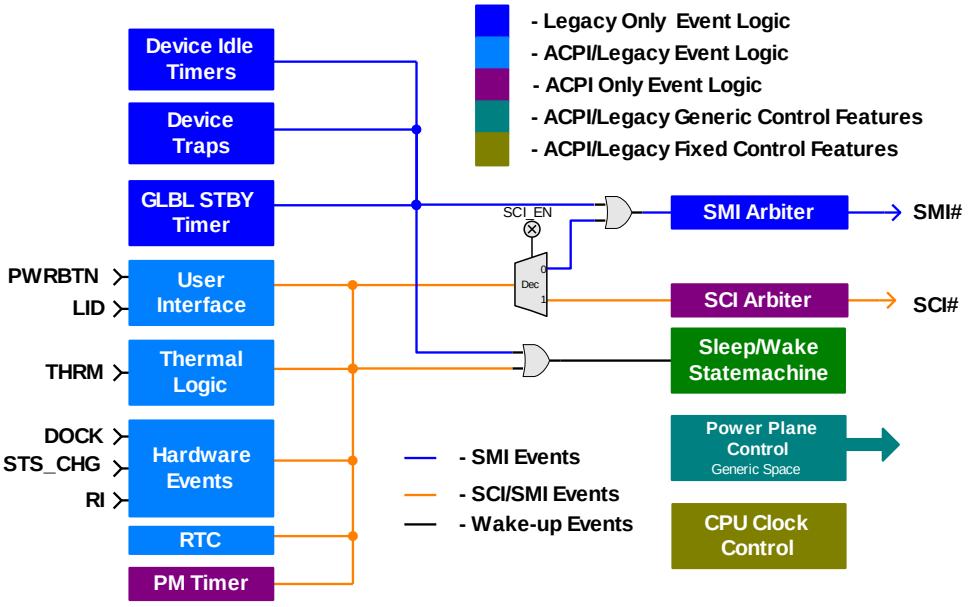


Figure 4-3 Example Event Structure for a Legacy/ACPI Compatible Event Model

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:

1. **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).
2. **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.
3. **Hardware events.** These events are used to trigger the hardware to initiate some hardware sequence such as waking-up, 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 the OS. 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.

This traditional idle timers, trap monitors, and global standby timer are not used by the OS 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 the OS 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 this function is now handled by the drivers. 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 requires some dedicated hardware not required 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 features are implemented as described in this specification so that the ACPI driver can directly access the fixed feature registers.
Generic location features are manipulated by ACPI control methods principally residing in the ACPI name space. These bits are made to be very flexible; however, their use is limited by the defined ACPI control methods (for more information, see section 10). These bits are normally associated with output bits that control power planes, buffer isolation, and device reset resources. Additionally, "child" interrupt status bits can reside in generic address space; however, they have a "parent" interrupt status bit in the GP_STS register. ACPI defines five address spaces that these feature bits can reside in the following:

- System I/O space
- System memory space
- PCI configuration space
- Embedded controller space
- SMBus device space

Generic location feature bit space is described in the ACPI BIOS programming model. These power management features can be implemented by 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 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. ACPI drivers are designed such that they will 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 sections give two examples of cross device dependencies:

4.5.4.1 Example 1

This example illustrates a cross device dependency where a device interferes with the proper operation of other unrelated devices. A system has two unrelated devices A and B. 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 an 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

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 Features

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

- Fixed Features
- Generic Features

Fixed location features reside in system I/O space at the locations described by the ACPI programming model. Generic location 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 name space.
Fixed features have exact definitions for their implementation. Although many fixed features are optional, if implemented they must be implemented as described. This is required because a standard OS driver is talking to these registers and expects the defined behavior.
Generic feature implementation is flexible. This logic is controlled by OEM-supplied ASL/AML-code (for more information, see section 5), 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 the OS from the generic event handler 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.
Note 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	Requirements	Programming Model
Power Management Timer	24-bit/32-bit free running timer.	Required for ACPI compatibility.	Fixed Feature Control Logic.
Power Button	User pushes button to switch the system between the working and sleeping states.	Must have either a power button or a sleep button.	Fixed Feature Event and Control Logic or Generic Event and Logic
Sleep Button	User pushes button to switch the system between the working and sleeping state.	Must have either a power button or a sleep button.	Fixed Feature Event and Control Logic or Generic Event and Logic.
Power Button Over-ride	User sequence (press the power button for 4 seconds) to turn off	This or a similar function required.	

Feature Name	Description	Requirements	Programming Model
	a hung system.		
Real Time Clock Alarm	Programmed time to wake-up the system.	Required for ACPI compatibility (for S1-S3; optional for S4).	Optional Fixed Feature Event[2]
Sleep/Wake Control Logic	Logic used to transition the system between the sleeping and working states.	Required for ACPI compatibility. At least one sleeping state needs to be supported.	Fixed Feature Control and Event Logic.
Embedded Controller Interface	ACPI Embedded Controller protocol and interface, as described in section 13.	Optional.	Generic Event Logic, must reside in the general purpose register block.
Legacy/ACPI Select	Status bit to indicates the system is using the legacy or ACPI power management model (SCI_EN).	Required. Status bit indicates the mode of a legacy/ACPI platform.	Fixed feature Control Logic.
Lid switch	Button used to indicate whether the system's lid is open or closed (mobile systems only).	Optional, strongly recommended for mobile systems.	Generic Event Feature.
C1 Power State	Processor instruction to place the processor into a low-power state.	This is a required feature for all IA-PC platforms.	Processor ISA.
C2 Power Control	Logic to place the processor into a C2 power state.	Optional, strongly recommended for mobile systems.	Fixed Feature Control Logic.
C3 Power Control	Logic to place the processor into a C3 power state.	Optional, strongly recommended for mobile systems.	Fixed Feature Control Logic.
Thermal Control	Logic to generate thermal events at specified trip points.	Optional	Generic Event and Control Logic. See description of thermal logic in section 3.9.
Device Power Management	Control logic for switching between different device power states.	Optional, strongly recommended for mobile systems.	Generic control logic.
AC Adapter	Logic to detect the insertion and removal of the AC adapter.	Optional	Generic event logic
Docking/device insertion and removal	Logic to detect device insertion and removal events	Optional	Generic event logic

4.7 ACPI Register Model

ACPI hardware resides in one of five I/O spaces:

- System I/O
- System memory
- PCI configuration
- SMBus
- Embedded controller space

Different implementations will result in different address spaces being used for different functions; however, all ACPI implementations are required to support system I/O space (the other address spaces are optional). The ACPI specification consists of "fixed registers" and general purpose registers. The fixed register space is

[2] RTC wake-up alarm is required, the fixed feature status bit is optional.

required to be implemented by all ACPI-compatible hardware. The general purpose register space is required for any events generated by value-added hardware.
ACPI defines a register block. An ACPI-compatible system will have an ACPI table (the FACP, built in memory at boot-up) that has a list of 32-bit pointers to the different register blocks used by the ACPI driver. 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:

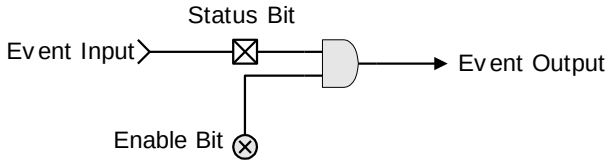


Figure 4-4 Block Diagram of a Status/Enable Cell

Note that the status bit, which hardware sets by the Event Input being HIGH 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 high 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 chip partitioning and providing the programming model with a single register grouping bit structure.
The ACPI driver treats a register grouping as a single register; but located in multiple places. To read a register grouping, the ACPI driver 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 affects for writes (which is a requirement).
The SLP_TYPx field can be different for each register grouping. The respective sleeping object _Sx 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. The ACPI driver 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, the ACPI driver does not read the SLP_TYPx value but throws it away.

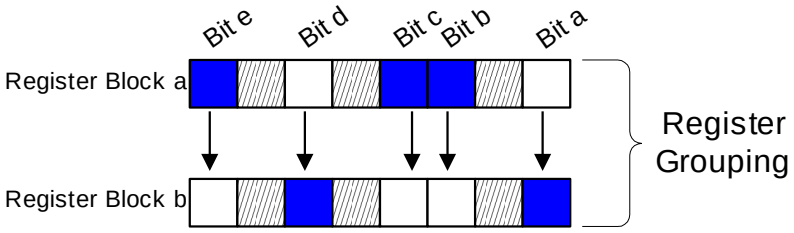


Figure 4-5 Example Fixed Feature Register Grouping

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, software will read register block a, followed by reading register block b. Software then does a logical OR of the two registers and then operates on the results.

When writing to this register grouping, software will write the desired value to register group a followed by writing the same value to register group b.

ACPI defines the following register blocks for fixed features. Each register block gets a separate pointer from the FACP ACPI table. 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:

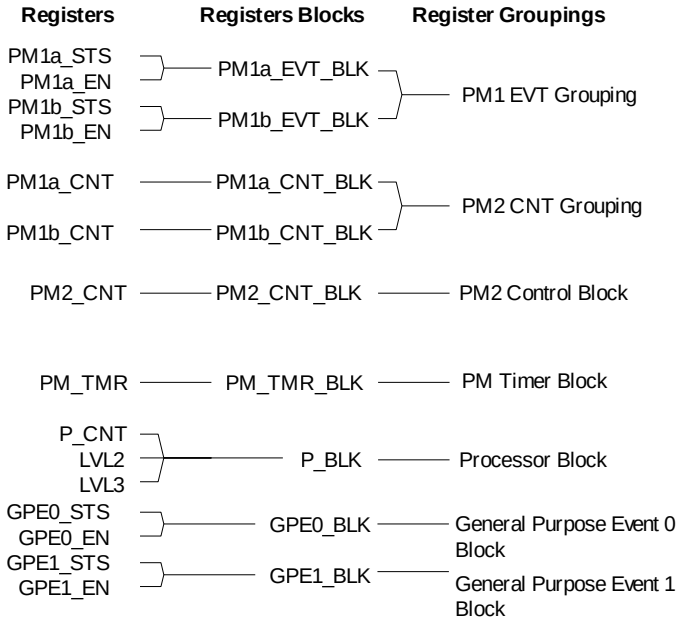


Figure 4-6 Register Blocks versus Register Groupings

The PM1 EVT grouping consists of the PM1a_EVT and PM1b_EVT register blocks, which contain the fixed 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, the OS 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, the OS 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 feature control bits and consist 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. Note that it is possible to have N levels of general-purpose events prior to hitting the GPE event status.

The general-purpose event register space is contained in two register blocks: The GPE0_BLK or the GPE1_BLK. Each register block has a separate 32-bit pointer within the FACP ACPI table. 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 follows the event model for the fixed-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	32	<P_BLK>
P_LVL2	8	<P_BLK>+4h
P_LVL3	8	<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 that must be supported, and the PM1b_EVT_BLK is an optional register block. Each register block has a unique 32-bit pointer in the Fixed ACPI Table (FACP) 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 FACP table.

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 FACP table, 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 that must be supported, and the PM1b_CNT_BLK is an optional register block. Each register block has a unique 32-bit pointer in the Fixed ACPI Table (FACP) 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 FACP table. 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 FACP table, 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 FACP table 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. This register block contains the register that returns the running value of the power management timer. The FACP table 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

There is an optional processor control register block for each processor in the system. This is a homogeneous feature, so 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 (a 32-bit clock control configuration register) and the P_LVL2 and P_LVL3 clock control register. The 32-bit register controls the behavior of the processor clock logic for that processor, the P_LVL2 register is used to force the CPU into the C2 state, and the P_LVL3 register is used to force 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 FACP table, 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 FACP table contain zeros. The GPE0_LEN and GPE1_LEN do not need to have the same size.

4.7.2 Required Fixed Features

This section describes the ACPI required fixed features. These features are required in every ACPI-compatible system.

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 the ACPI driver, 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:

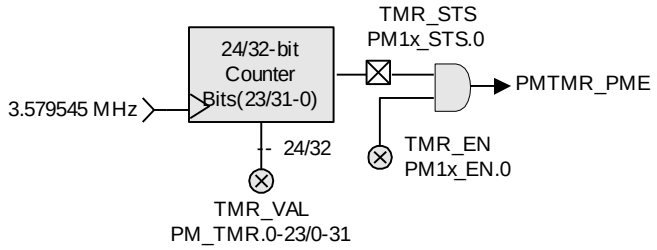


Figure 4-7 Power Management Timer

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 FACP table 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 HIGH to LOW or LOW to HIGH. 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.
The ACPI uses the read-only TMR_VAL field (in the PM TMR register grouping) to read the current value of the timer. The OS never assumes an initial value of the TMR_VAL field; instead, it reads an initial TMR_VAL upon loading the OS and assumes that the timer is counting. 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 feature to increase the accuracy of reading the timer.

4.7.2.2 Buttons

ACPI defines user-initiated events to request the OS 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 the OS 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 function of the button is implied by the type of button.

Control of these button events is either through the fixed programming model or the generic programming model (control method based). The fixed programming model has the advantage that the OS can access the button at any time, including when the system is crashed. In a crashed system with a fixed-feature power button, the OS 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 the OS 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:

Indicated Support	PWR_BUTTON Flag	Power Button Device Object
No power button	Set HIGH	Absent
Fixed feature power button	Set LOW	Absent
Control method power button	Set HIGH	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 the OS event handler is no longer able to respond to power button events, the power button over-ride 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 over-ride event.

4.7.2.2.1.1 Fixed Power Button

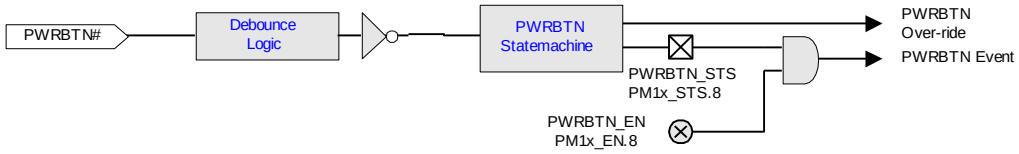


Figure 4-8 Fixed Power Button Logic

The fixed 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. The ACPI driver 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 transistioned the system into the sleeping state unconditionally sets the power button status bit and awakens the system, regardless of the value of the power button enable bit. The ACPI driver responds by clearing the power button status bit and awakening the system.

4.7.2.2.1.2 Control Method Power Button

The power button programming model can also use the generic programming model. This allows the power button to reside in any of the generic address spaces (for example, the embedded controller) instead of fixed space. If the power button programming model uses the generic programming model, 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 the ACPI driver. The AML event handler then generates a Notify command to notify the OS 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 the ACPI driver 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 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 up the system.
The power button device needs to be declared as a device within the ACPI name space for the platform and only requires an _HID. An example definition follows.
This example ASL code does 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.
    - Unaccessed fields are written as **Ones**. 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 status bit 0. This bit is written HIGH to be cleared and is the responsibility of the ASL-code to clear (the ACPI driver 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 wakeup 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 status register 0. The event handler does the following:
    - Clears the power button status bit in hardware (writes a one to it)
    - Notifies the OS 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"))
    }

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(PWRB, 0x80) // Notify OS of event
            }
        IF(PBW){
            Store(One, PBW)
            Notify(PWRB, 0x2)
            }
        }   // end of _L00 handler
    }       // end of \_GPE scope
```

4.7.2.2.1.3 Power Button Over-ride

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 over-ride. In reaction to the power button over-ride 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 the OS 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:

Indicated Support	SLEEP_BUTTON Flag	Sleep Button Device Object
No sleep button	Set HIGH	Absent
Fixed feature sleep button	Set LOW	Absent
Control method sleep button	Set HIGH	Present

4.7.2.2.2.1 Fixed Sleeping Button

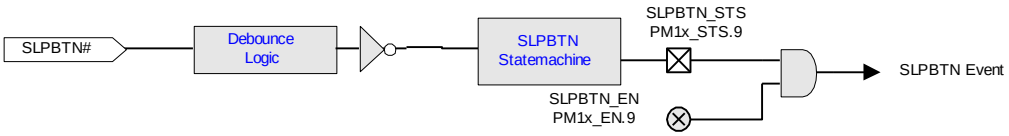


Figure 4-9 Fixed Sleep Button Logic

The fixed 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. The ACPI driver 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 awakens the if the SLP_EN bit is set. The ACPI driver responds by clearing the sleep button status bit and awakening the system.

4.7.2.2.2.2 Control Method Sleeping Button

The sleep button programming model can also use the generic programming model. This allows the sleep button to reside in any of the generic address spaces (for example, the embedded controller) instead of fixed space. If the sleep button programming model resides in generic address space, 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 the ACPI driver. The AML event handler then generates a Notify command to notify the OS 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 the ACPI driver to pass control to the sleep button driver (PNP0C0E) with the knowledge that a transition out of the G0 state is being requested by the user. 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 up the system.
The sleep button device needs to be declared as a device within the ACPI name space 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.
    - Unaccessed fields are written as Ones (these status bits clear upon writing a one 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. This bit is written HIGH to be cleared and is the responsibility of the AML code to clear (the ACPI driver 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 wakeup 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 one to it)
    - Notifies the OS 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"))
    OperationRegion(\Boo, SystemIO, 0x201, 0x1)
    Field(\Boo, ByteAcc, NoLock, WriteAsZeros){
        SBP, 1,     //  sleep request
        SBW, 1      //  wakeup request
        }           // end of sleep button device object
    }
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(SLPB, 0x80)      // Notify OS of event
            }
        IF(SBW){
            Store(One, SBW)
            Notify(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 awaken the system back to the working state upon a wake event. Note that the S4BIOS state is entered in a different manner (for more information, see section 9.1.4.2).

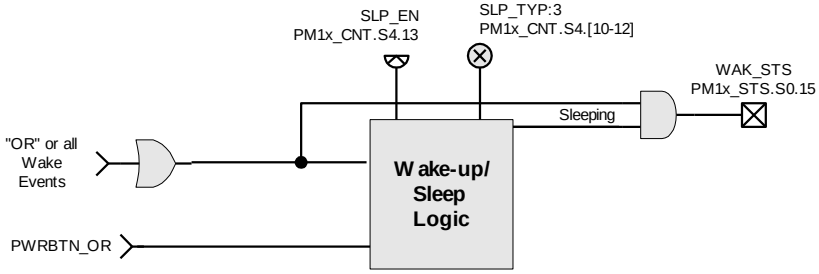


Figure 4-10 Sleeping/Wake Logic

The logic is controlled by 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. The ACPI driver gets values for the SLP_TYPx field from the _Sx objects defined in the static definition block. If the object is missing the ACPI driver assumes the hardware does not support that sleeping state. Prior to entering the desired sleeping state, the ACPI driver will read the designated _Sx object and place this value in the SLP_TYP field.

Additionally ACPI defines a fail-safe Off protocol called the "power switch 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 the ACPI driver 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 the ACPI driver 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 the ACPI driver'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 wakeup 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.

The sleeping/wake logic is required for ACPI compatibility, however only a single sleeping state is required to be supported (S1-S4). If more than a single sleeping state is supported, then the sleeping/wake logic is required

to be able to dynamically sequenced between the different sleeping states by waking the system, programming the new sleep state into the SLP_TYP field, and then by setting the SLP_EN bit.

4.7.2.4 Real Time Clock Alarm

The ACPI specification requires that the Real Time Clock (RTC)alarm generate a hardware wake-up event from 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. The ACPI provides for additional hardware to support the ACPI driver in determining that the RTC was the source of the wakeup 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, the OS will attempt to identify the RTC as a possible wakeup source; however, it might miss certain wakeup events. The RTC wake-up feature is required to work in the following sleeping states: S1-S3. S4 wakeup is optional and supported through the RTC_S4 flag within the FACP table (if set HIGH, then the platform supports RTC wakeup 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).

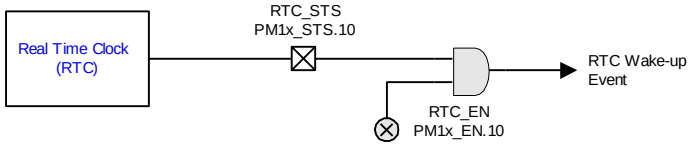


Figure 4-11 RTC Alarm

The RTC wakeup event status and enable bits within the fixed feature space is optional, and a flag within the FACP table (FIXED_RTC) indicates if the register bits are to be used by the ACPI driver or not. Having the RTC wakeup event in fixed feature space allows the ACPI driver to determine if the RTC was the source of the wakeup event without loading the entire OS. If the fixed feature event bits are not supported, then the OS will attempt to determine this by reading the RTC's status field.
The ACPI driver 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).

The RTC_STS bit is set through the RTC interrupt (IRQ8 in PC architecture systems). The OS 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 4-8 Alarm Field Decodings within the FACP Table

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	The DAY_ALRM field in the FACP table 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

[3] Note 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.

Field	Value	Address (Location) in RTC CMOS RAM (Must be Bank 0)
	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.	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 FACP table 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 FACP table 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 OS's. 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):

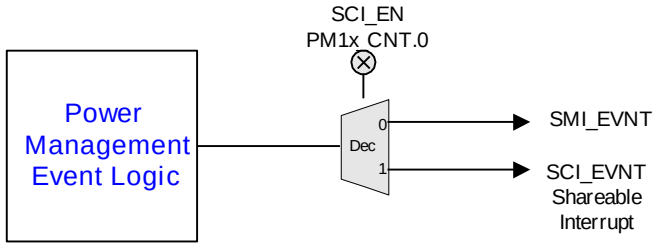


Figure 4-12 Power Management Events to SMI/SCI Control Logic

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 return HIGH for ACPI-compatible hardware that does not support a legacy power management mode (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 FACP ACPI table has an entry that indicates what interrupt the SCI interrupt is mapped to (see section 5.2.5).
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). The ACPI driver uses this register to request the hardware to switch in and out of ACPI mode. Within the FACP tables are three values that signify the system I/O 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:
1. ACPI driver checks that the SCI_EN bit is zero, and that it is in the Legacy mode.
2. The ACPI driver does an OUT to the SMI_CMD port with the data in the ACPI_ENABLE field of the FACP table.
3. The ACPI driver 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:
1. ACPI driver checks that the SCI_EN bit is one, and that it is in the ACPI mode.
2. The ACPI driver does an OUT to the SMI_CMD port with the data in the ACPI_DISABLE field of the FACP table.
3. The ACPI driver 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.

4.7.2.6 Processor Power State Control

ACPI supports placing system processors into one of four power states in the G0 working state. In the C0 state the designated processor is executing code; in the C1-C3 states it is not. While in the C0 state, ACPI allows the performance of the processor to be altered through a defined “throttling” process (the C0 Throttling state in the diagram below). Throttling hardware lets the processor execute at a designated performance level relative to its maximum performance. The hardware to enter throttling is also described in this section.

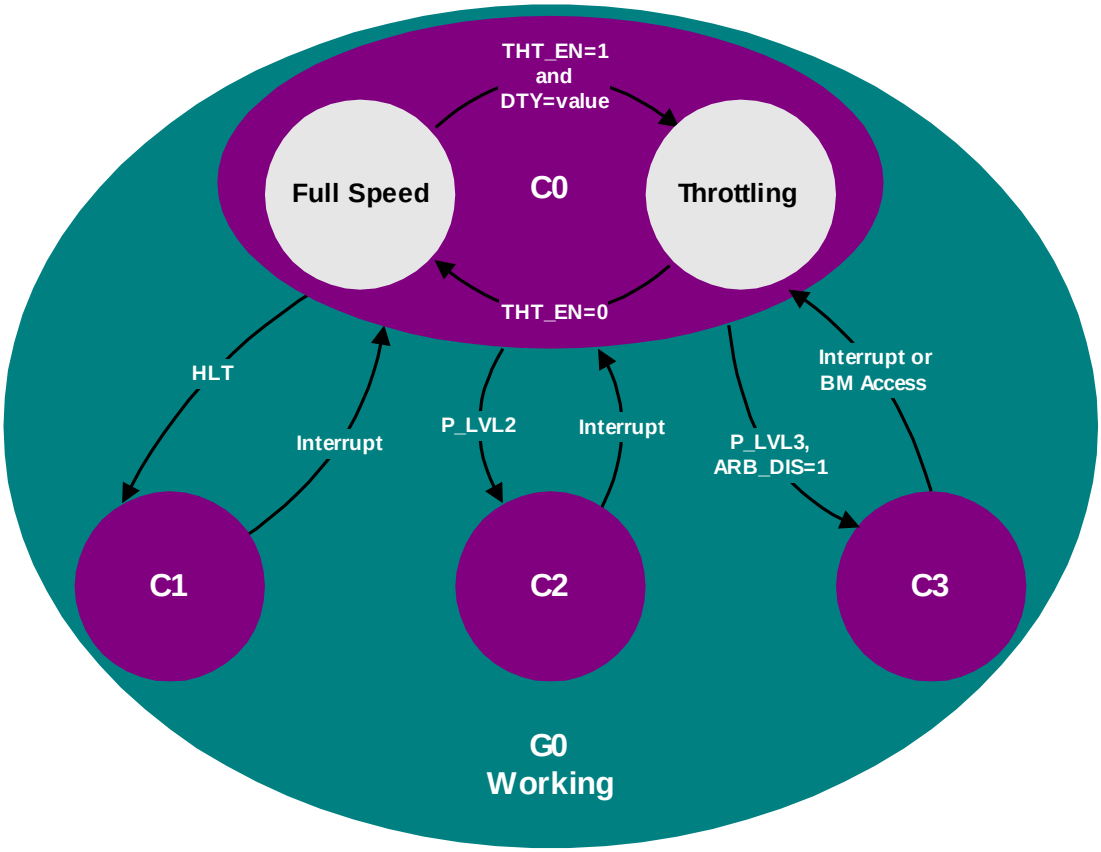


Figure 4-13 Processor Power States

In a working system (global G0 working state) the OS will dynamically transition idle CPUs into the appropriate power state. ACPI defines logic on a per-CPU basis that the OS uses to transition between the different processor power states. This logic is optional, and is described through the FACP table and processor objects (contained in the hierarchical name space). The fields and flags within the FACP 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). The ACPI specification defines four CPU power states for the G0 working state[4]: C0, C1, C2 and C3.

- In the C0 power state, the processor executes.
- In the C1 power state, the processor is in a low power state where it is able to maintain the context of the system caches. This state is supported through a native instruction of the processor (HLT for IA-PC processors), and assumes no hardware support is needed from the chipset.
- In the C2 power state, the processor is in a low power state where it is able to maintain the context of system caches. This state is supported through chipset hardware described in this section. The C2 power state is lower power and has a higher exit latency than the C1 power state.
- In the C3 power state, the processor is in a low power state where it is not necessarily able to maintain coherency of the processor caches with respect to other system activity (for example, snooping is not enabled at the CPU complex). This state is supported through chipset hardware described in this section. The C3 power state is lower power and has a higher exit latency than the C2 power state.

The P_BLK 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

[4] Note that these CPU states map into the G0 (working) state. The state of the CPU is undefined in the sleeping state (G3), the Cx states only apply to the G0 state.

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. Hardware is required to 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, all processors in a system are assumed by system software to support the same clock 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 FACP table. For example, if the P0 processor supports all power states up to and including the C3 state, but the P1 processor only supports the C1 power state, then the ACPI driver will only place idle processors into the C1 power state (P0 will never be put into the C2 or C3 power states). Note that either the C1 or C2 power state must be supported (see the PROC_C1 flag in the FACP table description in section 5.2.5).

4.7.2.6.1 C2 Power State

The C2 state puts the processor into a low power state optimized around multiprocessor (MP) and bus master systems. The system software will automatically cause an idle processor complex to enter a C2 state if there are bus masters or MP processors active (which will prevent the OS from placing the processor complex into the C3 state). The processor complex is able to snoop bus master or MP CPU accesses to memory while in the C2 state. Once the processor complex has been placed into the C2 power state, any interrupt (IRQ or reset) will bring the processor complex out of the C2 power state.

4.7.2.6.2 C3 Power State

The C3 state puts the designated processor and system into a power state where the processor's cache context is maintained, but it is not required to snoop bus master or MP CPU accesses to memory. There are two mechanisms for supporting the C3 power state:

- Having the OS flush and invalidate the caches prior to entering the C3 state.
- Providing hardware mechanisms to prevent masters from writing to memory (UP only support).

In the first case the OS will flush the system caches prior to entering the C3 state. As there is normally much latency associated with flushing processor caches, the ACPI driver is likely to only support this in MP platforms for idle processors. Flushing of the cache is through one of the defined ACPI mechanisms (described below, flushing caches).

In UP only platforms that provide the needed hardware functionality (defined in this section), the ACPI driver will attempt to place the platform into a mode that will prevent system bus masters from writing into memory while any processor is in the C3 state. This is done 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.

The ACPI driver uses the BM_STS bit to determine which Cx power state to enter. The BM_STS is an optional bit that indicates when bus masters are active. The ACPI driver uses this bit to determine the policy between the C2 and C3 power states: lots 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. The ACPI driver 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 Cx power state is exited based on bus master requests. If set, then the Cx power state is exited upon a request from a bus master; if reset, the power state is 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. The ACPI driver can optionally set this bit when using a C3 power state, and clear it when using a C1-C2 power state.

4.7.2.6.2.1 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 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:

1. Processor instruction to write-back and invalidate system caches (WBINVD instruction for IA processors).

2. 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).
3. Manual flush of caches supported by the ACPI driver.

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 FACP fields and flags indicated in this section.

ACPI specifies parameters in the FACP table 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 the ACPI driver by setting the WBINVD flag in the FACP table.

If the platform supports the write back and invalidate instruction; however, the cache is only flushed but not invalidated after its execution, then this support is indicated to the ACPI driver by setting the WBINVD_FLUSH flag in the FACP table (WBINVD flag would be cleared).

If the platform supports neither of the first two flushing options, then the ACPI driver 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 Mbytes will flush the platform caches.

There are two additional FACP 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.

4.7.2.6.3 Clock Throttling (C0 Power State)

While in the C0 power state, the ACPI driver can generate a policy to run the processor at less than maximum performance. The clock throttling hardware provides the driver with the functionality to perform this task. The logic allows the driver to program a value into a register that represents the % of maximum performance it desires the processor to execute at. When enabled, the hardware attempts to keep the processor at this minimum performance level.

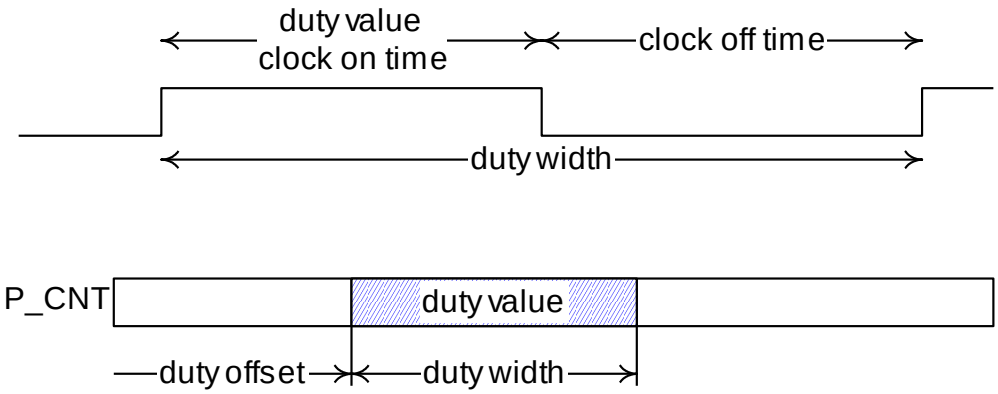


Figure 4-14 Throttling Example

The FACP table 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 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:

$$\%\,Performance = \frac{duty setting}{2^{duty width}} * 100\%$$

Equation 1 Duty Cycle Equation

Nominal performance is defined as "close as possible, but not below the indicated performance level." The ACPI driver will use the duty offset and duty width to determine how to access the duty setting field. The ACPI driver will then program the duty setting based on the thermal condition and desired power of the processor object. The ACPI driver calculates the nominal performance of the processor using the equation expressed in Equation 1. Note 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):

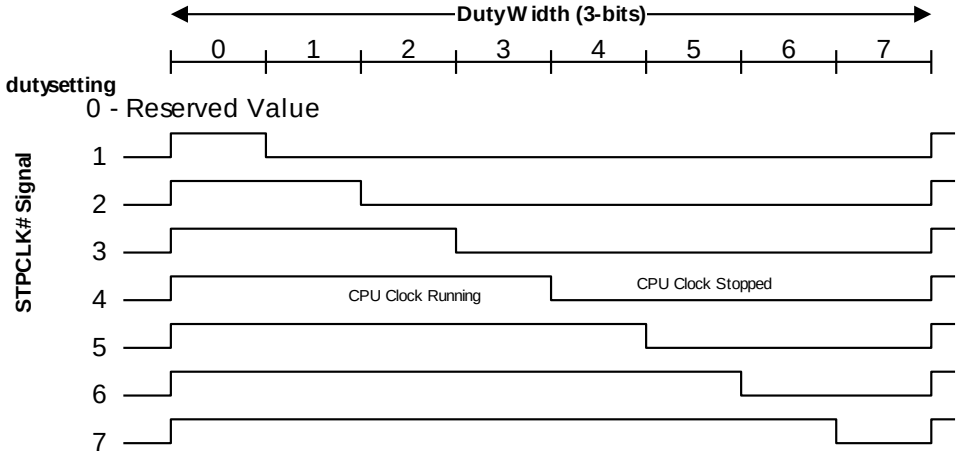


Figure 4-15 Example Control for the STPCLK#

To start the throttling logic the ACPI driver sets the desired duty setting and then set the THT_EN bit HIGH. To change the duty setting the OS will first reset the THT_EN bit LOW, 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:

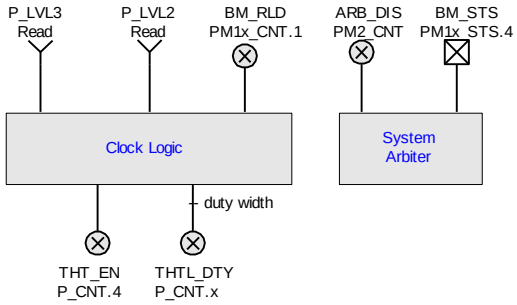


Figure 4-16 ACPI Clock Logic (One per Processor)

An ACPI platform is required to support a single CPU state (besides C0). All of the CPU states occur in the G0 system state; they have no meaning when the system transitions into the sleeping state. ACPI defines the attributes 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 processor register block ACPI requires that there be a processor register block for each CPU in the system. Additionally, ACPI requires that the clock logic for MP systems be symmetrical; if the P0 processor supports the C1, C2, and C3 states, but P1 only supports the C1 state, then the ACPI driver will limit all processors to enter the C1 state when idle.

The following sections define the different ACPI CPU states.

4.7.2.6.4 C0 Power State

This is the executing state for the CPU, in all other CPU power states the CPU is not executing instructions. The CPU's clock is running at full frequency or is running at a reduced performance (for more information, see section 4.7.2.6.3).

4.7.2.6.5 C1 Power State

The C1 CPU low power state is supported through the execution of a CPU instruction that places it into a low power state (for IA processors this would be the HLT instruction).

4.7.2.6.6 C2 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 MP activity can take place without corrupting cache context. The C2 power state is assumed by the ACPI driver to have lower power and higher exit latency than the C1 power state.

4.7.2.6.7 C3 Power State

The C3 power state is an optional ACPI feature that needs chipset hardware support. This logic consists of a P_LVL3 register which, when read, will cause the system to precisely transition into a C3 power state. When the system is in a C3 power state, the system CPU is assumed to be unable to maintain cache coherency; it is the responsibility of the OS to place the system into a condition where the caches will not become incoherent with memory. The ACPI specification provides a standard way for the ACPI driver to disable bus masters that will guarantee coherency in a uniprocessor (UP) system. In multiprocessor systems, the OS will flush and invalidate caches prior to entering the C3 state.

4.7.3 Fixed Feature Space Registers

The fixed feature space registers are manipulated directly by the ACPI driver. The following sections describes fixed features under the programming model. The ACPI driver owns all the fixed resource registers, these registers are not manipulated by ASL/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 FACP table), the bit positions is 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 Power Management 1 Status Registers

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

The PM1 status registers contains the fixed 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 FACP table. 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 the OS). 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 FACP table. If the FACP table indicates that the feature is not supported as a fixed feature, then software treats these bits as ignored.

Table 4-9 PM1 Status Registers Fixed Feature Status Bits

Bit	Name	Description
0	TMR_STS	This is the timer carry status bit. This bit gets set anytime the 23rd/31st bit of a 24/32-bit counter changes (whenever the MSB changes from low to high or high to low. 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 one to this bit position. Note 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).
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 states a wakeup 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 one to this bit position. ACPI defines an optional mechanism for unconditional transitioning a crashed platform from the G0 working state into the G2 soft-off state called the power button over-ride. 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 either the PWR_BUTTON flag in the FACP table being reset zero. If the PWR_BUTTON flag is set HIGH or a power button device object is present in ACPI name space, than this bit field is treated as ignored by software. If the power button was the cause of the wakeup (from an S1-S4 state), then this bit is set prior to returning control to the OS.
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 wakeup 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 one to this bit position. Support for the sleep button is indicated by either the SLP_BUTTON flag in the FACP table being reset zero. If the SLP_BUTTON flag is set HIGH or a sleep button device object is present in ACPI name space, than this bit field is treated as

Bit	Name	Description
		ignored by software. If the sleep button was the cause of the wakeup (from an S1-S4 state), then this bit is set prior to returning control to the OS.
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 one to this bit position. If the RTC was the cause of the wakeup (from an S1-S3 state), then this bit is set prior to returning control to the OS. If the RTC_S4 flag within the FACP table is set, and the RTC was the cause of the wakeup from the S4 state), then this bit is set prior to returning control to the OS.
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 wakeup 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 one to this bit position.

4.7.3.1.2 Power Management 1 Enable Registers

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

The PM1 enable registers contains the fixed 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 FACP table. 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 the OS). 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 FACP table. If the FACP table indicates that the feature is not supported as a fixed feature, then software treats the enable bits as write as zero.

Table 4-10 PM1 Enable Registers Fixed 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 wakeup). The PWRBTN_STS bit is set anytime the power button is asserted. The enable bit does not have to be set to

Bit	Name	Description
		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 either the PWR_BUTTON flag in the FACP table being reset zero. If the PWR_BUTTON flag is set HIGH or a power button device object is present in ACPI name space, than this bit field is treated as ignored by software.
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 wakeup). 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 either the SLP_BUTTON flag in the FACP table being reset zero. If the SLP_BUTTON flag is set HIGH or a sleep button device object is present in ACPI name space, than this bit field is treated as ignored by software.
10	RTC_EN	This optional bit is used to enable the setting of the RTC_STS bit to generate a wakeup event. The RTC_STS bit is set anytime the RTC generates an alarm. If the RTC was the cause of the wakeup (from an S1-S3 state), then this bit is set prior to returning control to the OS. If the RTC_S4 flag within the FACP table is set, and the RTC was the cause of the wakeup from the S4 state), then this bit is set prior to returning control to the OS.
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 FACP table), the bit positions specified here is 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 Power Management 1 Control Registers

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

The PM1 control registers contains the fixed 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 FACP table. Accesses to PM1 control registers are accessed through byte and word accesses.

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

Table 4-11 PM1 Control Registers Fixed 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

Bit	Name	Description
		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. The ACPI driver 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 effect 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 the ACPI driver 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 the ACPI driver.
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 _Sx object contains 3-bit binary values associated with the respective sleeping state (as described by the object). The ACPI driver takes the two values from the _Sx 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 _Sx 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 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 FACP table 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 FACP table. If the FACP table indicates that the feature is not supported as a fixed feature, then software treats these bits as ignored.

Table 4-12 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

Bit	Name	Description
		clock is stopped upon enter its Sx 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 Power Management 2 Control (PM2_CNT)

```
Register Location: <PM2_BLK> System I/O
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 FACP table. If the FACP table indicates that the feature is not supported as a fixed feature, then software treats these bits as ignored.

Table 4-13 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 LOW the system arbiter is enabled and the arbiter can grant the bus to other bus masters. When this bit is HIGH the system arbiter is disabled and the default CPU has ownership of the system. The ACPI driver 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 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. 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: <P_BLK>   System I/O 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 FACP table. Software treats all other CLK_VAL bits as ignored (those not used by the duty setting value).

Table 4-14 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: <P_BLK>+4 System   I/O Space
Default Value:     00h
Attribute:         Read-Only
Size:              8-bits
```

This register is accessed as a byte.

Table 4-15 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 a "enter a C2 power state" to the clock control logic.

4.7.3.5.3 Processor LVL3 Register (P_LVL3): 8

```
Register Location: <P_BLK>+5h   System I/O Space
Default Value:     00h
Attribute:         Read-Only
Size:              8-bits
```

This register is accessed as a byte.

Table 4-16 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 a "enter a C3 power state" to the clock control logic.

4.7.4 Generic Address Space

ACPI provides a mechanism that allows a unique piece of "value added" hardware to be described to the ACPI driver in ACPI name space. There are a number of rules to be followed when designing ACPI-compatible hardware.

Programming bits can reside in any of the defined generic 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 registers. The general-purpose registers are pointed to by the GP_REG block, and the generic register space can be any of the defined ACPI address spaces. A device's generic address space programming model is described through an associated object in the ACPI name space, 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 the ACPI driver 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 awaken 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).

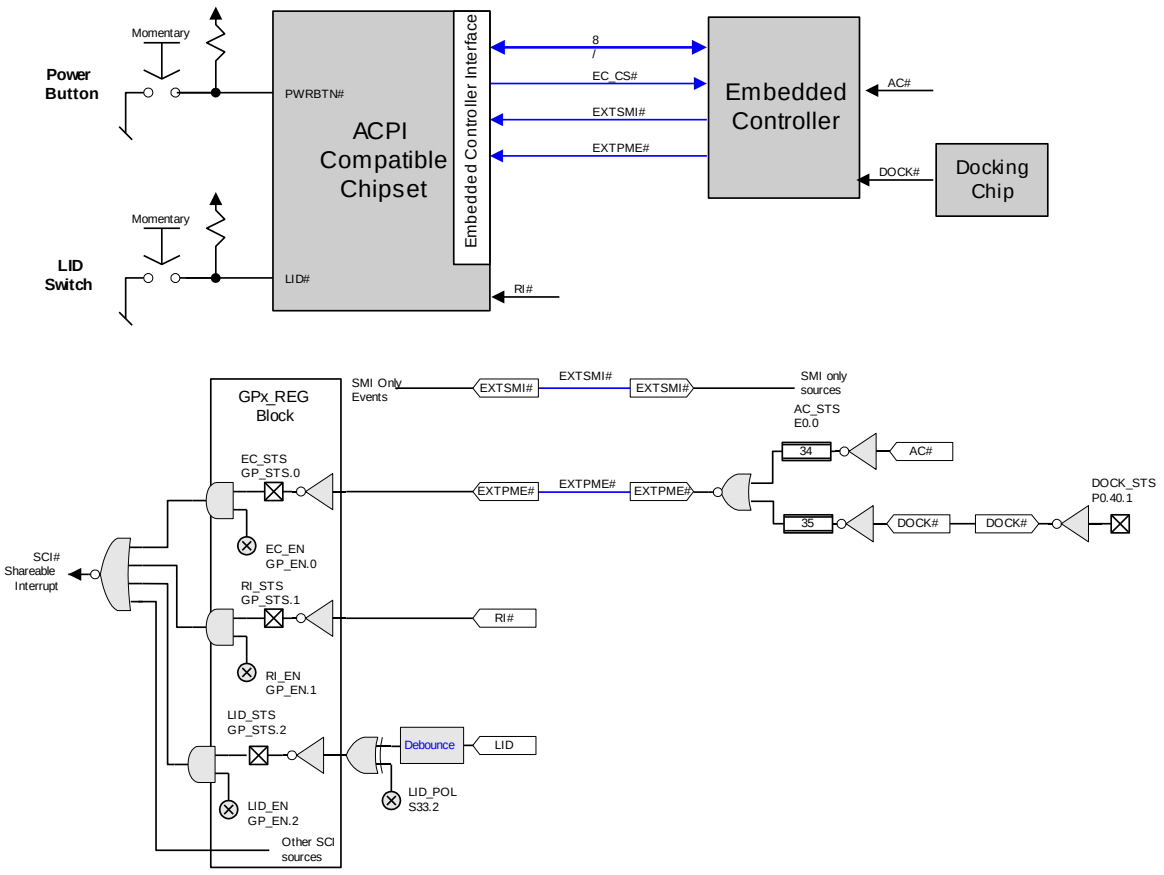


Figure 4-17 Example of General-Purpose vs Generic Address Space Events

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 awakening the system).
- Lid status.

The embedded controller event status bit (EC_STS) is used to indicate that one of two query events are 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; the ACPI driver 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. The ACPI driver 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. Note 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 HIGH and LID is HIGH) or closed (LID_POL is LOW and LID is LOW). 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 events, the ACPI driver will clear the status bits in the GPE*x* register blocks. However, AML code clears all sibling status bits in generic space.

Generic 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 the OS of certain power management and Plug and Play events. Review section 5 to understand the types of hardware functionality that supports the different types of subsystems. The following is a list of features supported by APCI; 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 wakeup events
- Plug and Play configuration

4.7.4.1 General-Purpose Register Blocks

ACPI supports up to two general-purpose register blocks. 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 operating system upon awakening.

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 FACP table's GPE0_BLK and GPE0_BLK_LEN operators. The ACPI driver owns the general-purpose event resources and these bits are only manipulated by the ACPI driver; ASL/AML code can not 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/ASL code would describe to the OS 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 ASL/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.

4.7.4.1.1.1 General-Purpose Event 0 Status Register

```
Register Location: <GPE0_STS> System I/O 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 one to its respective bit position. For the general-purpose event registers, unimplemented bits are ignored by the OS.

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

[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.

4.7.4.1.1.2 General-Purpose Event 0 Enable Register

```
Register Location: <GPE0_EN> System I/O 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. The ACPI driver 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 FACP table'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 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 one to its respective bit position. For the general-purpose event registers, unimplemented bits are ignored by the operating system.

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

The ACPI driver 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 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.

The ACPI driver 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 operating system as policy input for sleeping the system, or for waking up 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 the ACPI driver. The Lid device needs to contain a control method that returns its status. The Lid event handler AML code re-configures the lid hardware (if it needs to) to generate an event in the other direction, clear the status, and then notify the OS of the event.

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

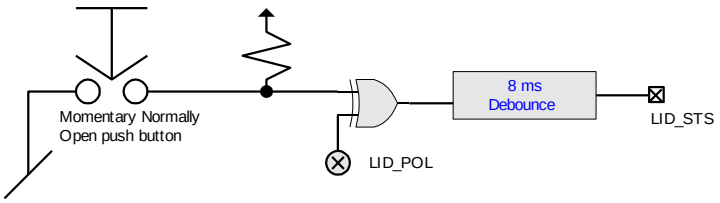


Figure 4-18 Example Generic Address Space Lid Switch Logic

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 control and status bits reside in address space.
  - System address space in registers 0x201 (control) and 0x202 (status).
- A field operator to allow AML code to access these bits:
  - Polarity control bit (LID_POL) is called LPOL and is accessed at 0x200.0.
  - Status bit (LID_STS) is called LSTS and is accessed at 0x200.1.
- Creates a device called "\LID" with the following:
  - A Plug and Play identifier "PNP0C0D" that associates the ACPI driver with this object.
  - The status control method that returns the state of the Lid's status bit.
- The lid switch event handler that does the following:
  - Defines the lid's status bit (LidS) 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:
    - Clears the lid status (write 1 to LidS bit).
    - Flips the polarity of the LPOL bit (to cause the event to be generated on the opposite condition).
    - Generates a notify to the operating system that does the following:
      - Passes the \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
        \_S4})      // 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 awakening the system).
- Lid status.

The embedded controller event status bit (EC_STS) is used to indicate that one of two query events are 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; the ACPI driver will then schedule for execution the control method associated with query value 34.

- Another query event is for the docking chip which 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. The ACPI driver 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. Note 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 HIGH and LID is HIGH) or closed (LID_POL is LOW and LID is LOW). 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 events, the ACPI driver 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 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 the OS of certain power management and Plug and Play events. Review section 5 to understand what types of hardware hooks are required to support the different types of subsystems. The following is a list of features supported by APCI, however the list is not intended to be complete or comprehensive:

- Device insertion/ejection (e.g. docking, device bay, A/C adapter)
- Batteries[6]
- Platform thermal subsystem
- Turning on/off power resources
- Mobile lid interface
- Embedded controller
- System indicators
- OEM-specific wake-up events
- Plug and Play configuration

4.7.4.3 General-Purpose Register Blocks

ACPI supports up to two general purpose register blocks. 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 according to its length (block length divided by two). It is up to the specific design to determine if these bits retain their context across sleeping or soft off states. If bits lose their context across a sleeping or soft off state, then BIOS should reset the respective enable bit prior to passing control to the operating system upon awakening.

4.7.4.3.1 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 FACP table's GPE0_BLK and GPE0_BLK_LEN operators. The ACPI driver owns the General purpose event resources and these bits are only manipulated by the ACPI driver; ASL/AML code can not 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/ASL code would describe to the OS 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 chipset, however not utilized by the platform and have no associated ASL/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.

[6] 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.

4.7.4.3.2 General Purpose Event 0 Status Register

```
Register Location: <GPE0_STS>   System I/O 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 0 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 should be set when the event is active, and can only be cleared by software writing a one 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 up the system if asserted when the system is in a sleeping state with its respective enable bit set.

4.7.4.3.2.1 General Purpose Event 0 Enable Register

```
Register Location: <GPE0_EN>   System I/O 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 in bank 0 of the general purpose registers. 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.

4.7.4.3.3 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 FACP table's GPE1_BLK and GPE1_BLK_LEN operators.

4.7.4.3.3.1 General Purpose Event 1 Status Register

```
Register Location: <GPE1_STS>   System I/O 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 in bank 0 of the general purpose registers. 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 should be set when the event is active, and can only be cleared by software writing a one 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-up the system if asserted when the system is in a sleeping state with its respective enable bit set.
The ACPI driver accesses GPE registers through byte accesses (regardless of their length).

4.7.4.3.3.2 General Purpose Event 1 Enable Register

```
Register Location: <GPE1_EN> System I/O 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 bits in bank 0 of the general purpose registers. 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, then a set status bit in the corresponding status bit will generate an SCI bit.
The ACPI driver accesses GPE registers through byte accesses (regardless of their length).

4.7.4.4 Specific Generic Devices

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

4.7.4.4.1 Lid Switch

The Lid switch is an optional feature present in most "clam shell" style mobile computers. It can be used by the operating system as policy input for sleeping the system, or for waking up 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 the ACPI driver. The Lid device needs to contain a control method that returns its status. The Lid event handler AML code re-configures the lid hardware (if it needs to) to generate an event in the other direction, clear the status, and then notify the OS of the event.
Example hardware and ASL code is shown below for such a design.

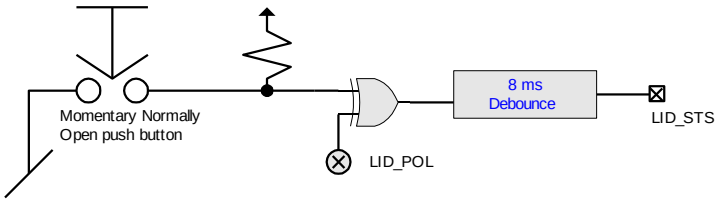


Figure 4-19 Example Generic Address Space Lid Switch Logic

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 control and status bits reside in address space.
  - System address space in registers 0x201 (control) and 0x202 (status).
- A field operator to allow AML code to access these bits:
  - Polarity control bit (LID_POL) is called LPOL and is accessed at 0x200.0
  - Status bit (LID_STS) is called LSTS and is accessed at 0x200.1
- Creates a device called "\LID" with
  - A plug and play identifier "PNP0C0D" which associates the ACPI driver with this object
  - The status control method which returns the state of the Lid's status bit
- The lid switch event handler which
  - Defines the lid's status bit (LidS) 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) which:
    - Clears the lid status (write 1 to LidS bit)
    - Flips the polarity of the LPOL bit (to cause the event to be generated on the opposite condition)
    - Generates a notify to the operating system which
      - Passes the \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(\LID) {
    Name(_HID, EISAID("PNP0C0D"))
    Method(_LID) {Return(LPOL)}
    Name(_PRW, Package(2) {
        One,                    // bit 1 of GPE to enable Lid wake-up
        \_S4})                  // can wake-up from S4 state
    }
Scope(\GPE)                     // Root level event handlers
    Method(L001)   {            // uses bit 1 of GP0_STS register
        Not(LPOL, LPOL)         // Flip the lid polarity bit
        Notify(\LID, 0x80)      // Notify OS of event
    }
```

4.7.4.4.2 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 PNP0A09 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:

```
Device(\_EC0) {
   // PnP ID
   Name(_HID, EISAID(PNP0C09))
   // Returns the "Current Resources" of EC
   Name(_CRS, Buffer(){ 0x4B, 0x62, 0, 1, 0x4B,
                        0x66, 0, 1, 0x79, 0  })
   // 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, AnyAcc, Lock, Preserve) {
      // Field definitions
      }
   Method(Q00){..}
   Method(QFF){..}
}
```

For more information on the embedded controller see section 13.

4.7.4.4.3 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.

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 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. 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 the ACPI driver 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 ACPI name space is discussed.

5.1 Overview of the System Description Table Architecture

The Root System Description Pointer 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, which references other Description Tables that provide data to the OS, supplying it with knowledge of the base system's implementation and configuration (see Figure 5-1).

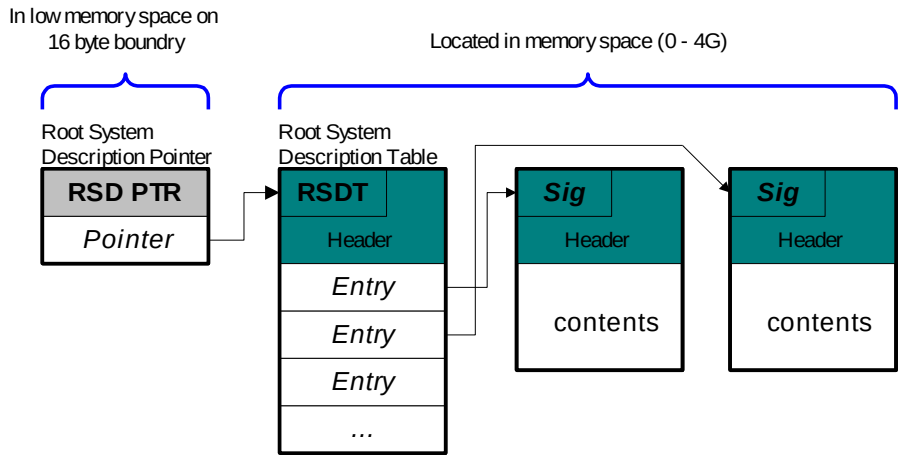


Figure 5-1 Root System Description Pointer and Table

All description tables start with identical headers. The primary purpose of the description tables is to define for the OS various industry-standard implementation details. Such definitions enable various portions of these implementations to be flexible in hardware requirements and design, yet still provide the OS 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 ("FACP"). The data within this table includes various fixed-length entries that describe the fixed ACPI features of the hardware. The FACP table always refers to the Differentiated System Description Table ("DSDT"), which contains information and descriptions for various system features. The relationships between these tables is shown in Figure 5-2.

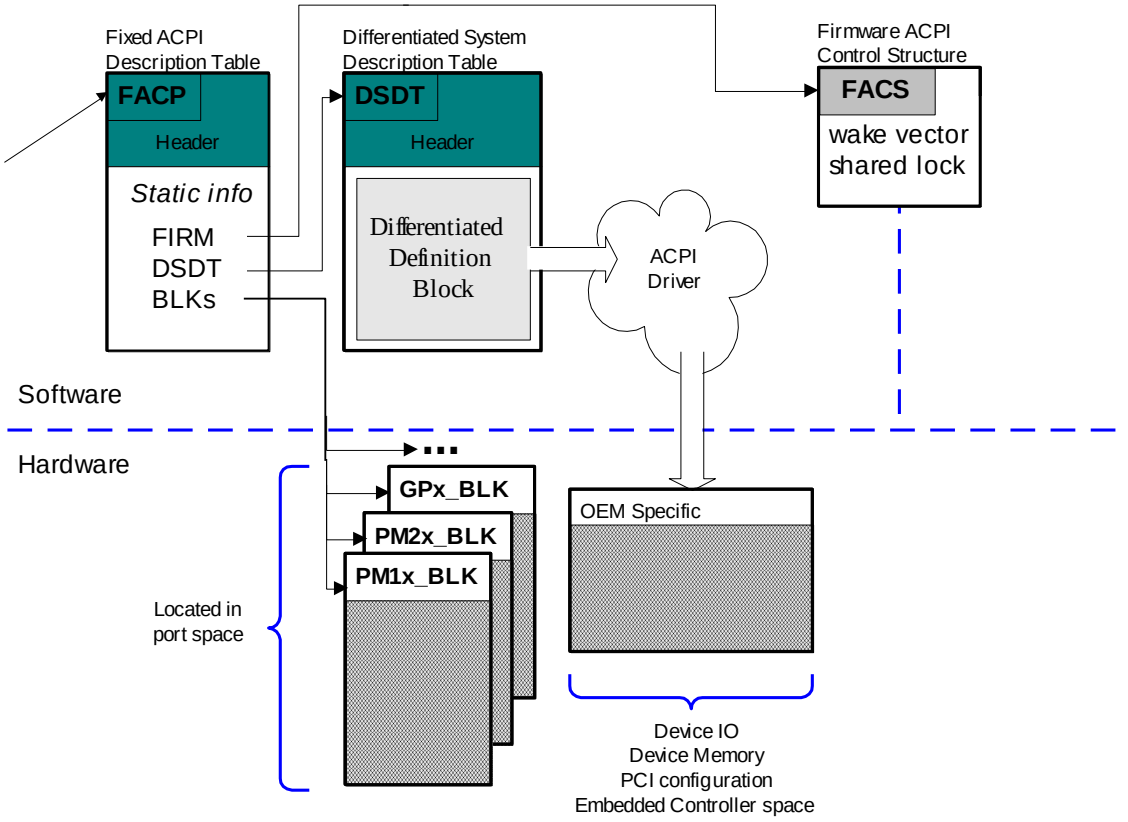


Figure 5-2 Description Table Structures

The OS searches the following physical ranges on 16-byte boundaries for a Root System Description Pointer structure. This structure is located by searching the areas listed below for a valid signature and checksum match:

- The first 1K 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.
- In the BIOS read-only memory space between 0E0000h and 0FFFFFh.

When the OS locates the structure, it looks at the physical system address for the Root 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 on other standards defined on the current 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 (FACP).

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

The purpose of the FACP is to define various static system information regarding power management. The Fixed ACPI Description Table starts with the "FACP" signature. The FACP 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. 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 GP0_BLK and GP1_BLK blocks provide the foundation for an interrupt processing model for Control Methods. The P_BLKs blocks are for controlling processor features.

Besides ACPI Hardware Register implementation information, the FACP 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 the OS 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 name space, data, and control methods encoded in AML. The OS "loads" or "unloads" an entire definition block as a logical unit. The Differentiated Definition Block is always loaded by the OS 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 IO 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 the OS. The operators present are intended to allow many useful hardware designs to be ACPI-expressed, not to allow all hardware design to be expressed.

5.2 Description Table Specifications

This section specifies the structure of the system description tables:

- Root System Description Pointer
- System Description Table Header
- Root System Description Table
- Fixed ACPI Description Table
- Firmware ACPI Control Structure
- Differentiated System Description Table
- Secondary System Description Table
- Persistent System Description Table
- Multiple APIC Description Table
- Smart Battery Table

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. Note 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.
- ACPI driver implementations, for all reserved bits in ACPI tables and in other software values:
- Ignore all reserved bits that are read.

- Preserve reserved bit values of read/write data items (for example, the driver 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.
- ACPI driver 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 Root System Description Pointer

The OS searches the following physical ranges on 16-byte boundaries for a Root System Description Pointer. This table is located by searching the areas listed below for a valid Root System Description Pointer structure signature and checksum match. When the operating system locates the Root System Description Pointer structure, it looks at the supplied physical system address for the Root System Description Table:

- The first 1K 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.
- In the BIOS read-only memory space between 0E0000h and 0FFFFFh.

Table 5-1 Root System Description Pointer Structure

Field	Byte Length	Byte Offset	Description
Signature	8	0	"RSD PTR "
Checksum	1	8	The entire Root System Description Pointer structure, including the checksum field, must add to zero to be considered valid.
OEMID	6	9	An OEM-supplied string that identifies the OEM.
Reserved	1	15	Must be zero.
RsdtAddress	4	16	Physical address of the Root System Description Table.

5.2.3 System Description Table Header

All description tables begin with the structure shown in Table 5-2. The content of the system description table is determined by the *Signature* field. System Description Table signatures defined by this specification are listed in Table 5-3.

Table 5-2 DESCRIPTION_HEADER Fields

Field	Byte Length	Byte Offset	Description
Signature	4	0	The ASCII string representation of the table identifier.

Field	Byte Length	Byte Offset	Description
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 backwards 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 the DSDT, RSDT, SSDT, and PSDT tables, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For the DSDT, RSDT, SSDT, PSDT tables, 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.

Table 5-3 contains the Description Table signatures defined by this specification.

Table 5-3 DESCRIPTION_HEADER Signatures

Signature	Description
“APIC”	Multiple APIC Description Table. See section 5.2.8.
“DSDT”	Differentiated System Description Table. See section 5.2.7.1.
”FACP”	Fixed ACPI Description Table. See section 5.2.5.
“FACS”	Firmware ACPI Control Structure. See section 5.2.6.
“PSDT”	Persistent System Description Table. See section 5.2.7.3.
“RSDT”	Root System Description Table. See section 5.2.4.
“SSDT”	Secondary System Description Table. See section 5.2.7.2.
“SBST”	Smart Battery Specification Table. See section 5.2.9

5.2.4 Root System Description Table

The OS locates that Root System Description Table by following the pointer in the Root System Description Pointer structure. The Root System Description Table, shown in Table 5-4, 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. The OS examines each table for a known signature. Based on the signature, the OS can then interpret the implementation-specific data within the table.

Table 5-4 Root System Description Table Fields

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 Root System Description Table. The length implies the number of Entry fields 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 Root System Description Table, the table ID is the manufacture model ID.
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 the DSDT, RSDT, SSDT, PSDT tables, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For the DSDT, RSDT, SSDT, and PSDT tables, this is the revision for the ASL Compiler.
Entry	4*n	36	An array of physical addresses that point to other DESCRIPTION_HEADERs. The OS assumes at least the DESCRIPTION_HEADER is addressable, and then can further address the table based upon its Length field.

5.2.5 Fixed ACPI Description Table

The Fixed ACPI Description Table defines various fixed 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_TMP_BLK, GPE0_BLK, and GPE1_BLK.

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

Table 5-5 Fixed ACPI Description Table 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 Fixed ACPI Description 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 Fixed ACPI Description Table, the table ID is the manufacture model ID.
OEM Revision	4	24	OEM revision of FACP table for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For the DSDT, RSDT, SSDT, and PSDT tables, this is the ID for the ASL Compiler.

Field	Byte Length	Byte Offset	Description
Creator Revision	4	32	Revision of utility that created the table. For the DSDT, RSDT, SSDT, PSDT tables, this is the revision for the ASL Compiler.
FIRMWARE_CTRL	4	36	Physical memory address (0-4 GB) of the Firmware ACPI Control Structure, where the OS and Firmware exchange control information. See section 5.2.6 for a description of the Firmware ACPI Control Structure.
DSDT	4	40	Physical memory address (0-4 GB) of the Differentiated System Description Table.
INT_MODEL	1	44	The interrupt mode of the ACPI description. The SCI vector and Plug and Play interrupt information assume some interrupt controller implementation model for which the OS must also provide support. This value represents the interrupt model being assumed in the ACPI description of the OS. This value therefore represents the interrupt model. This value is not allowed to change for a given machine, even across reboots. 0 Dual PIC, industry standard PC-AT type implementation with 0-15 IRQs with EISA edge-level-control register. 1 Multiple APIC. Local processor APICs with one or more IO APICs as defined by the Multiple APIC Description Table. >1 Reserved.
Reserved	1	45	
SCI_INT	2	46	System pin the SCI interrupt is wired to. The OS 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, the OS 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 SMI_DISABLE command to the SMI_CMD port. The SCI_EN bit effectively tracks the ownership of the ACPI hardware registers. The OS issues commands to the SMI_CMD port synchronously from the boot processor.
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. The OS initialization process will synchronously wait for the ownership transfer to complete, so the ACPI system releases SMI ownership as timely as possible.
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 the OS 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.

Field	Byte Length	Byte Offset	Description
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 5-8.)
Reserved	1	55	
PM1a_EVT_BLK	4	56	System port address of the Power Management 1a Event Register Block. See section 4.7.3.1 for a hardware description layout of this register block. This is a required field.
PM1b_EVT_BLK	4	60	System port address of the Power Management 1b Event Register Block. See section 4.7.3.1 for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero.
PM1a_CNT_BLK	4	64	System port address of the Power Management 1a Control Register Block. See section 4.7.3.2 for a hardware description layout of this register block. This is a required field.
PM1b_CNT_BLK	4	68	System port address of the Power Management 1b Control Register Block. See section 4.7.3.2 for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero.
PM2_CNT_BLK	4	72	System port address of the Power Management 2 Control Register Block. See section 4.7.3.4 for a hardware description layout of this register block. This field is optional; if this register block is not supported, this field contains zero.
PM_TMR_BLK	4	76	System power address of the Power Management Timer Control Register Block. See section 4.7.3.3 for a hardware description layout of this register block. This is a required field.
GPE0_BLK	4	80	System port address of Generic Purpose Event 0 Register Block. See section 4.7.4.3 for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero.
GPE1_BLK	4	84	System port address of Generic Purpose Event 1 Register Block. See section 4.7.4.3 for a hardware description of this register block. This is an optional field; if this register block is not supported, this field contains zero.
PM1_EVT_LEN	1	88	Number of bytes in port address space decoded by PM1a_EVT_BLK and, if supported, PM1b_CNT_BLK. This value is ≥ 4.
PM1_CNT_LEN	1	89	Number of bytes in port address space decoded by PM1a_CNT_BLK and, if supported, PM1b_CNT_BLK. This value is ≥ 1.
PM2_CNT_LEN	1	90	Number of bytes in port address space decoded by PM2_CNT_BLK. This value is ≥ 1.
PM_TM_LEN	1	91	Number of bytes in port address space decoded by PM_TM_BLK. This value is ≥ 4.

Field	Byte Length	Byte Offset	Description
GPE0_BLK_LEN	1	92	Number of bytes in port address space decoded by GPE0_BLK. The value is a non-negative multiple of 2.
GPE1_BLK_LEN	1	93	Number of bytes in port address space 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.
Reserved	1	95	
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.
FLUSH_SIZE	2	100	If WBINVD=0, the value of this field is the contiguous memory size that needs to be read(using cacheable addresses) to flush dirty lines from any processor's memory caches. If the system does not support a method for flushing the processor's caches, then FLUSH_SIZE and WBINVD are set to zero. Note that this method of flushing the processor caches has limitations, and WBINVD=1 is the preferred way to flush the processors caches. In particular, it is known that at least Intel Pentium Pro Processor, MP C3 support, 3rd level victim caches require WBINVD=1 support. This value is typically at least 2 times the cache size. The maximum allowed value for this setting is 2 MB for a typical maximum supported cache size of 1 MB through this mechanism. Larger cache sizes are supported using WBINVD=1. This value is ignored if WBINVD=1.
FLUSH_STRIDE	2	102	If WBINVD=0, the value of this field is the memory stride width, in bytes, to perform reads to flush the processor's memory caches. These cacheable memory reads are done for a length of FLUSH_SIZE from cacheable addresses to flush dirty lines from the processor's caches. This value is typically the smallest cache line width on any of the processor's caches. This value is ignored if WBINVD=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.
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.

Field	Byte Length	Byte Offset	Description
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 the OS can use to program the day of the month alarm. See section 4.7.2.4 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 the OS 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 the OS can use to program the centenary field.
Reserved	3	109	
Flags	4	112	Fixed feature flags. See Table 5-6 for a description of this field.

Table 5-6 Fixed ACPI Description Table Fixed Feature Flags

FACP - Flag	Bit length	Bit offset	Description
WBINVD	1	0	WBINVD is correctly supported. 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 the OS 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.
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. Note that on Intel Pentium Pro Processor machines, the WBINVD instruction must flush and invalidate the caches. If neither of the WBINVD flags are 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.
PROC_C1	1	2	A one indicates that the C1 power state is supported on all processors. A system can support more Cx states, but is required to at least support either the C1 or C2 power state.

FACP - Flag	Bit length	Bit offset	Description
P_LVL2_UP	1	3	A zero indicates that the C2 power state is configured to only work on a UP system. A one indicates that the C2 power state is configured to work on a UP or 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 power 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-up status is supported in fixed register space; a one indicates the RTC wake-up 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.
Reserved	23		

5.2.6 Firmware ACPI Control Structure

The Firmware ACPI Control Structure (FACS) is a structure in read/write memory that the BIOS has set aside for ACPI usage. This structure is passed to an ACPI-compatible OS using the Fixed ACPI Description Table. For more information about the Fixed ACPI Description Table FIRMWARE_CTRL field, see section 5.2.5.

The BIOS aligns the FACS on a 64-byte boundary anywhere within the 0-4G memory address space. The memory where the FACS structure resides must not be reported as system memory in the system's memory map. For example, the E820 memory reporting interface would report the region as AddressRangeReserved. For more information about the E820 memory reporting interface, see section 14.1.

Table 5-7 Firmware ACPI Control Structure

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.

Field	Byte Length	Byte Offset	Description
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. The OS 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, the OS assumes that the saved non-volatile image is from a different hardware configuration and can not be restored.
Firmware Waking Vector	4	12	Location into which the ACPI OS puts its waking vector. Before transitioning the system into a global sleeping state, the OS fills in this vector with the physical memory address of an OS-specific wake function. During POST, the BIOS checks this value and if it is non-zero, transfers control to the specified address. On PCs, the wake function address is in memory below 1MB and the control is transferred while in real mode. The OS wake function restores the processors' context. For PC-IA 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 & 0x000F Note that on PC-IA 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	The Global Lock is used to synchronize access to shared hardware resources between the OS environment and the SMI environment. This lock is owned exclusively by either the OS 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 the OS or the firmware will access the embedded controller interface at any one time. See section 5.2.6.1 for more information on acquiring and releasing the Global Lock.
Flags	4	20	Firmware control structure flags. See Table 5-8 for a description of this field.
Reserved	40	24	This value is zero

Table 5-8 Firmware Control Structure Feature Flags

FACS - Flag	Bit Length	Bit Offset	Description

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

5.2.6.1 Global Lock

The Global Lock is a DWORD in read/write memory in the Firmware ACPI Control Structure, accessed and updated by both the operating system environment and SMI environment in a defined manner to provide an exclusive lock. 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 it is owned by the other environment, 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 using an inter-environment 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 is 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 5-9 shows the encoding of the Global Lock DWORD in memory:

Table 5-9 Embedded Controller Arbitration Structure

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 the OS 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.

```
AcquireGlobalLock:
           mov     ecx, GlobalLock    ; ecx = address of Global Lock
acq10:     mov     eax, [ecx]         ; Value to compare against

           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

           ; Attempt to set new value
           lock cmpxchg dword ptr[ecx], edx
           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 the OS 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 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
rel10:      mov     eax, [ecx]         ; Value to compare against

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

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

            and     eax, 1             ; Was pending set?
            ret
```

Although using the Global Lock allows various hardware resources to be shared, it is important to note 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 when a logical register in the hardware is shared. For example, if bit 0 is used by ACPI (the OS) 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.

5.2.7 Definition Blocks

A Definition Block contains information about hardware implementation details in the form of objects that contain data, AML code, or other objects. The top-level organization of this information after a definition block is loaded is name-tagged in a hierarchical name space.

The OS "loads" or "unloads" an entire definition block as a logical unit. As part of the Fixed ACPI Description Table, the system provides the operating system with the Differentiated System Description Table that contains the Differentiated Definition Block to be loaded at operating system initialization time and cannot be unloaded.

It is possible for this 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 IO space.

In theory, it might be possible to define something like PCI configuration space in a Definition Block by building it from IO space, but that is not the goal of the definition block. 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 the OS.

The operators present are intended to allow many useful hardware designs to be easily expressed, not to allow all hardware design to be expressed.

5.2.7.1 Differentiated System Description Table

The Differentiated System Description Table is part of the system fixed description in Definition Block format. This Definition Block is like all other Definition Blocks, with the exception that it cannot be unloaded. See section 5.2.7 for a description of Definition Blocks.

5.2.7.2 Secondary System Description Table

Secondary System Description Tables are a continuation of the Differentiated System Description Table. There can be multiple Secondary System Description Tables present. After the Differentiated System Description Table is loaded, each secondary description table with a unique OEM Table ID is loaded. 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 Differentiated System Description Table. A Secondary System Description Table can only rely on the Differentiated System Description Table being loaded prior to itself.

5.2.7.3 Persistent System Description Table

Persistent System Description Tables are similar to Secondary System Description Tables, except a Persistent System Description Table can be saved by the OS and automatically loaded at every boot. This can be used in the case where a Definition Block is loaded dynamically, for example based on the presence of some device, and the Definition Block has the ability to be loaded regardless of the presence of its device(s). In this case, by marking the Definition Block as persistent, the operating system can load the definition prior to the device appearing thus improving the load and enumeration time for the device when it does finally appear in the system. In particular, dynamic docking station devices might want to design their Definition Blocks as persistent.

5.2.8 Multiple APIC Description Table

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 APIC interrupt controller. The choice of the interrupt model to support is up to the platform designer, but it cannot be dynamically changed by the system firmware; the OS will choose which model to use and install support for that model at the time of installation. If a platform supports both models, the OS will only use one; it will not mix models. Therefore, the ACPI interrupt model must remain constant for all time on any given system. This section provides the APIC Description Table information necessary to use an APIC implementation on ACPI.

ACPI represents all interrupt vectors as "flat" values where each system vector has a different value. The primary information needed to support APICs on such a model is to map each IO APIC's interrupt INTI to the flat system vector value used by ACPI. Additional APIC support is required to handle various multi-processor functions that APIC implementations might support (specifically, identifying each processor's local APIC ID).

Table 5-10 Multiple APIC Description Table 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 Multiple APIC Description 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 Multiple APIC Description Table, the table ID is the manufacturer model ID.
OEM Revision	4	24	OEM revision of Multiple APIC Description Table for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For the DSDT, RSDT, SSDT, and PSDT tables, this is the ID for the ASL Compiler.

Field	Byte Length	Byte Offset	Description
Creator Revision	4	32	Revision of utility that created the table. For the DSDT, RSDT, SSDT, and PSDT tables, this is the revision for the ASL Compiler.
Local APIC Address	4	36	The physical address at which each processor can access its local APIC.
Flags	4	40	Multiple APIC flags. See Table 5-11 for a description of this field.

Table 5-11 Multiple APIC Description Table 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.
Reserved	31	1	This value is zero.

Following the Multiple APIC Description Table is a list of APIC structures that declare the APIC features of the machine. The first byte of the structure declares the structure type, and the second byte declares the length of the structure.

Table 5-12 APIC Structure Types

Value	Description
0	Processor Local APIC
1	IO APIC
>1	Reserved. The OS skips structures of the reserved type.

5.2.8.1 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. Processor information cannot change during the life of an operating system boot. For example, 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 5-13 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 15.2.3.4.1.10.
APIC ID	1	3	The processor's local APIC ID.
Flags	4	4	Local APIC flags. See Table 5-14 for a description of this field.

Table 5-14 Local APIC Flags

Local APIC - Flags	Bit Length	Bit Offset	Description

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.

5.2.8.2 IO APIC

In an APIC implementation, there is one or more IO APICs. Each IO APIC has a series of interrupt inputs, called INTI*x*, where the value of *x* is from 0 to last INTI line on the specific IO APIC. The IO APIC structure declares where in the system vector space the IO APICs INTIs appear. Each IO APIC INTI has an exclusive system vector mapping. There is one IO APIC structure per IO APIC in the system.

Table 5-15 IO APIC Structure

Field	Byte Length	Byte Offset	Description
Type	1	0	1 - IO APIC structure
Length	1	1	12
IO APIC ID	1	2	The IO APIC's ID.
Reserved	1	3	0
IO APIC Address	4	4	The physical address to access this IO APIC. Each IO APIC resides at a unique address.
System Vector Base	4	8	The system interrupt vector index where this IO APIC's INTI lines start. The number of INTI lines is determined by the IO APIC's *Max Redir Entry* register.

5.2.8.3 Platforms with APIC and Dual 8259 Support

Systems that support both APIC and dual 8259 interrupt models must map interrupt vectors 0-15 to 8259 IRQs 0-15. This allows such a platform to support ACPI OSes that use the APIC model and as well as those ACPI OSes those that use the 8259 model (the OS will only use one model; it will not mix models). When an ACPI OS supports the 8259 model, it will assume that any interrupt descriptors reporting vectors 0-15 correspond to 8259 IRQs and any vectors from any interrupt descriptor greater than 15 are ignored. When an ACPI OS loads APIC support, it will enable the APIC as described by the APIC specification, and use all reported interrupt vectors. (For more information on hardware resource configuration see section 6)

5.2.9 Smart Battery Table

If the platform supports batteries as defined by the Smart Battery Specification 1.0, then a Smart Battery Table 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. The OS uses these tables with the capabilities of the batteries to determine the different trip points. For more information, see the section 11, which describes the control method battery.

Table 5-16 Smart Battery Description Table Format

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	'SBST'. Signature for the Smart Battery Description Table.
Length	4	4	Length, in bytes, of the entire Smart Battery Description Table.
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero.

Field	Byte Length	Byte Offset	Description
OEMID	6	10	OEM ID.
OEM Table ID	8	16	For the Smart Battery Description Table, the table ID is the manufacturer model ID.
OEM Revision	4	24	OEM revision of Smart Battery Description Table for supplied OEM Table ID.
Creator ID	4	28	Vendor ID of utility that created the table. For the DSDT, RSDT, SSDT, and PSDT tables, this is the ID for the ASL Compiler.
Creator Revision	4	32	Revision of utility that created the table. For the DSDT, RSDT, SSDT, and PSDT tables, this is the revision for the ASL Compiler.
Warning Energy Level	4	36	OEM suggested energy level in milliWatt-hours (mWh) at which the platform warns the user.
Low Energy Level	4	40	OEM suggested platform energy level in mWh at which the platform is placed in a sleeping state.
Critical Energy Level	4	44	OEM suggested platform energy level in mWh at which the platform performs an emergency shutdown.

5.3 ACPI Name Space

For all Definition Blocks, the system maintains a single hierarchical name space that it uses to refer to objects. All Definition Blocks load into the same name space. 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[7]. Only an unload operation of a Definition Block can remove names from the name space, so a name collision in an attempt to load a Definition Block is considered fatal. Contents of the name space only changes on a load or unload operation.

The name space 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 are 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 If a name is padded, it is done so with trailing underscores ('_').
- Names beginning with '_' are reserved by this specification. Definition Blocks can only use names beginning with '_' as defined by this specification.
- A name preceded with '\' causes the name to refer to the root of the name space ('\' is not part of the 32-bit fixed-length name).

Except for names preceded with a '\', the *current name space* determines where in the name space 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 name space, and then in the parent name space. If the name space does not have a parent (the root of the name space), the name is not found[8].

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 name space.

[7] 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. In the root of the name space, and certain locations where interaction is being designed in, will be the areas which extra care must be taken.

[8] Unless the operation being performed is explicitly prepared for failure in name resolution, this is considered an error and results in a systems crash.

Figure 5-3 shows a sample of the ACPI name space after a Differentiated Definition Block has been loaded.

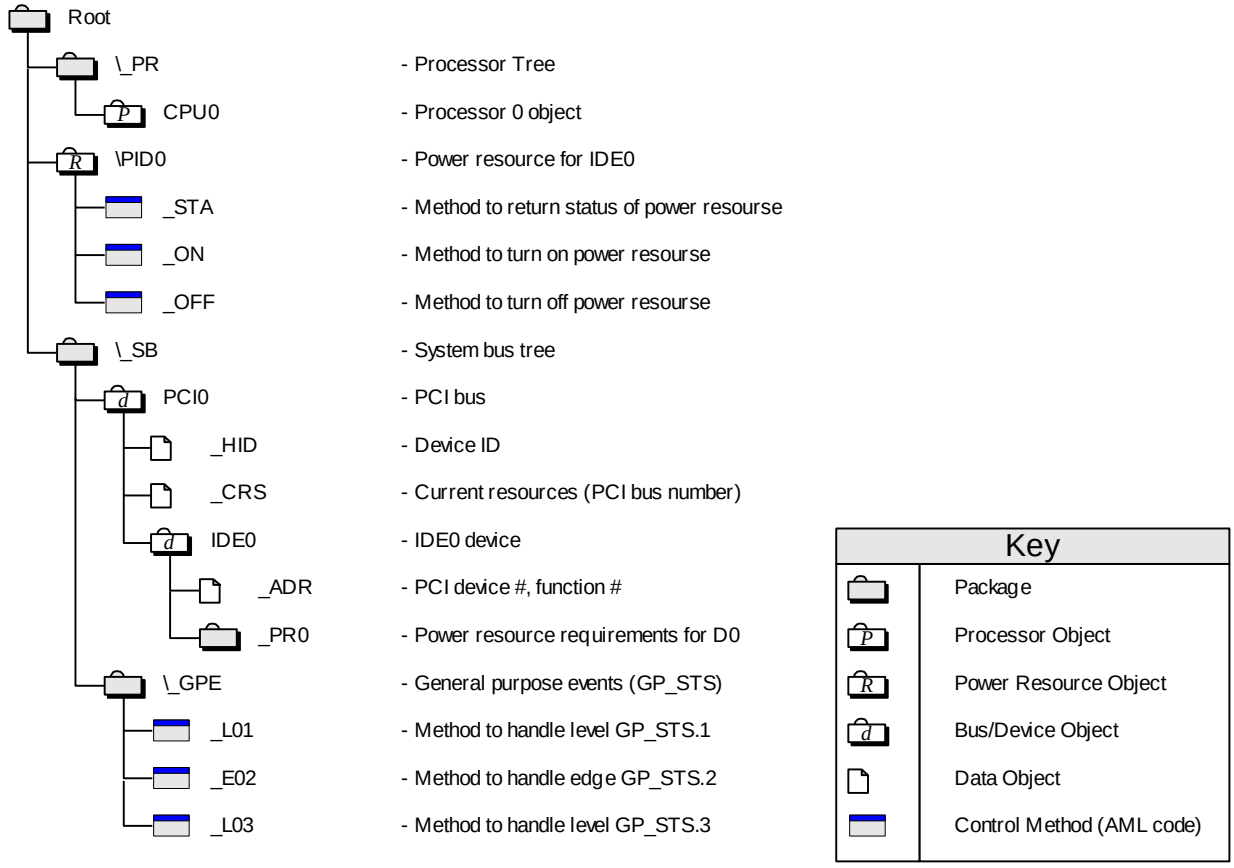


Figure 5-3 Example ACPI Name Space

5.3.1 Defined Root Names Spaces

The following name spaces are defined under the name space root.

Table 5-17 Name Spaces Defined Under the Name Space Root

Name	Description
_GPE	General events in GPE register block.
_PR	All Processor objects are defined under this name space. For more information about defining Processor objects, see section 8
_SB	All Device / Bus Objects are defined under this name space.
_SI	System indicator objects are defined under this name space. For more information about defining system indicators, see section 10.1.
_TZ	All Thermal Zone objects are defined under this name space. For more information about defining Thermal Zone objects, see section 12.

5.3.2 Objects

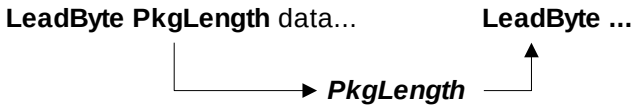
All objects, except locals, have a global scope. Local data objects have a per-invocation scope and lifetime and are used to process the current invocation from beginning to end.

The contents of objects varies 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 begin to end. The lead byte in the stream comes from the AML encoding tables shown in section 16 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 encodings, see section 16.
Within the stream there are two levels of datum 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.



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 0x0FFFFFFFF.

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.

Figure 5-4 shows a sample ACPI Machine Language (AML) byte stream encoding, and illustrates the use of *PkgLength* values in the byte stream.

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 name space and initializes objects accordingly. The name space for which population occurs is either from the *current name space location*, as defined by all nested packages or from the root if the name is preceded with '\'. For example, the byte stream shown in the left column of Figure 5-4 (indented by logical packaging level) produces the objects in the name space shown in the right column.

Byte Stream	Description

Byte Stream	Description
5B 80	Define Operation Region operator.
5C 47 49 4F 5F	\GIO_ (Name of operational region located at root level of ACPI Name Space at definition block load time.)
01	Operational region is in System IO address space.
0B 25 01	Operational region starts at 0x0125 (leading 0x0B means Word constant)
0A 01	Length of operational region is one byte (leading 0x0A means byte constant).
5B 81	Define Fields operator.
0C	Length of Define Fields definition in byte stream is 12 bytes, starting from the beginning of this byte.
5C 47 49 4F 5F	\GIO_ (Name of operational region within which to define field.)
00	Flag byte set to AnyAcc, NoLock, and Preserve (for more information, see section 16).
43 54 30 31	CT01 (Name of field.)
01	Length of field named 'CT01' is one bit.
10	Name Scope operator.
42 04	Scope package length is 66 bytes (from the beginning of this byte). Bits 6 and 7 of leading byte are set when package lenght is greater than 0x3F bytes (for more information, see section 16).
5C 5F 53 42 5F	_SB_ (Name of scope package)
5B 82	Define Bus/Device Package operator.
39	Length of Bus/Device package is 57 bytes from the beginning of this byte.
50 43 49 30	PCI0 (Name of device)
5B 84	Power Resource operator; begins a package that declares a named Power Resource object.
32	Length of Power Resource package is 50 bytes from the beginning of this byte.
46 45 54 30	FET0 (Name of Power Resource object)
00	Indicates lowest-power system sleep state OS must maintain to keep this power resource on; value of zero indicates S0.
00 00	Apply control methods contained in this power resource package at first step of power operation sequencing.
14	Define Method operator.
10	Length of this control method definition is 16 bytes from the beginning of this byte.
5F 4F 4E 5F 00	_ON_ (System-defined control method name.)
00	Number of parameters for this control method.
70	Store operator.
FF	Source of Store operation is OnesByte
43 54 30 31	CT01 (Name of target for Store operation.)
5B 22	Sleep operator.
1E 00	Duration of sleep is 30 milliseconds.
14	Define Method operator.
0C	Length of this control method definition is 12 bytes from the beginning of this byte.
5F 4F 46 46 00	_OFF (System-defined control method name.)
00	Number of parameters for this control method.
70	Store operator.
00	Source of Store operation is ZeroByte.
43 54 30 31	CT01 (Name of target for Store operation).
14	Define Method operator.
0B	Length of this control method definition is 11 bytes from the beginning of this byte.
5F 5B 54 41	_STA (System-defined control method name.)
00	Number of parameters for this control method.
A4	Return operator
43 54 30 31	CT01 (Name of object to return)

Figure 5-4 Sample Data Format Byte Stream Encoding with Corresponding Name Space

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.

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 15.

```
// ASL Example
DefinitionBlock (
    "forbook.aml",      // Output Filename
    "DSDT",             // Signature
    0x10,               // DSDT 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 which 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 15. For a detailed specification of the ACPI Control Method Machine Language (AML), upon which the output of the ASL translator is based, see section 16.

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 5-18 ASL Built-in Macros

ASL Statement	Description
Offset(*a*)	Used in a FieldList 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.

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 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.

5.5.3.1 Control Methods, Objects, and Operation Regions

Control Methods can reference any objects anywhere in the Name Space as well as objects that have shorthand encodings shown in section 15.1.3.1. Shorthand encodings are provided for common operators. The operators can access the contents of a object. An object's contents are either in dynamic storage (RAM) or, in some cases, in hardware registers. Access to hardware registers from within a control method is eventually accomplished through an Operation Region. Operation Regions are required to have exclusive access to the hardware registers[9]. 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 GP_BLK is not directly accessed by a control method but is used to provide an extensible interrupt handling model for control method invocation.

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 have shorthand encodings. For the definition of the Arg*x* shorthand encoding, see section 15.2.3.3.4.

The number of arguments passed to any control method is fixed and is defined when the control method package is created. For the definition of the Method operator, see section 15.2.3.4.1.6.

[9] This means the registers are not used by non-ACPI OS device drivers or SMI handling code.

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. For the definition of the Local*x* shorthand encoding, see section 15.2.3.3.4.2).

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. For the definition of the Return operator, see section 15.2.3.5.1.14.

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:

- ACPI driver
- Fixed ACPI Description Table (FACP)
- PM1a_STS, PM1b_STS and PM1a_EN, PM1b_EN fixed register blocks
- GPE0_BLK and GPE1_BLK register blocks
- 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 5-19 ACPI Event Programming Model Components

Component	Description
ACPI driver	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 the ACPI driver are fixed ACPI events; interrupts handled by control methods are general-purpose events.
Fixed ACPI Description Table (FACP)	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).

Component	Description
ACPI device-specific model events	Devices in the ACPI name space 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 block 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 the ACPI driver 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.

Table 5-20 Fixed ACPI Events

Event	Comment
Power management timer carry bit set.	A power management timer is required for ACPI-compatible hardware. 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 as well as the TMR_VAL register in the PM_TMR_BLK in section 4.7.3.3.
Power button signal	A power button is required for ACPI compatible platforms, but 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. Note that during the S0 state, both the power and sleep buttons merely notify the OS that they were pressed. If the system does not have a sleep button, it is recommended that the OS use the power button to initiate sleep operations as requested by the user.
Sleep button signal	A sleep button is an optional ACPI event. If supported, it 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-compatible hardware is required to have an RTC wake alarm function with a minimum of one-month granularity; however, 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.
Wake status	At least one system sleep state is required for an ACPI-compatible platform. The wake status bit is used to determine when the sleeping state has been completed.

Event	Comment
	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.
System bus master request	Optional. 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.
Global release status	This status is raised as a result of the global lock protocol, and is handled by the ACPI driver 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. For more information on global lock, see section 5.2.6.1.

5.6.2.2 General-Purpose Event Handling

When the ACPI driver 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 between zero and 255 general-purpose event inputs in hardware, each as either a level or edge event. An example of a general-purpose event is specified in section 4, where EC_STS and EC_EN bits are defined to enable the ACPI driver 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. Note 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 4.7.4.3.

The ACPI driver 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 the ACPI driver receives a general-purpose event (the event is from a GPEx_BLK STS bit), the ACPI driver 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 _PWR 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, the ACPI driver 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 - 1. The event values for status bits in GPE1_BLK start at GPE1_BASE and end at GPE1_BASE + GPE1_BLK_LEN - 1. GPE0_BLK_LEN, GPE1_BASE, and GPE1_BLK_LEN are all defined in the Fixed ACPI description table.

For the ACPI driver 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:

1. When a device signals its wake signal, the general-purpose status event bit used to track that device is set.
2. While the corresponding general-purpose enable bit is enabled, the SCI interrupt is asserted.
3. If the system is sleeping, this will cause the hardware, if possible, to transition the system into the S0 state.
4. Once the system is running, ACPI will dispatch the correspond GPE handler.
5. The handler needs to determine which device object has signaled wake and performs a wake Notify operation on the corresponding device object(s) that have asserted wake.
6. In turn the OS will notify the OS native driver(s) for each device that will wake its device to service it.

It is recommended that events that wake are not intermixed with events that do not wake 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, the 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 reported by the embedded controller by using the Query command. When an embedded controller event occurs, the ACPI-aware driver queues control methods to handle each event. Another way the OEM AML code can perform OEM-specific functions custom to each event on the particular platform is to queue a control method to handle these events. For an embedded controller event, the ACPI drive will queue the control method of the name _Q*XX*, where *XX* is the hex format of the query code. Note that each embedded controller device can have query event control methods.

5.6.2.2.3 Queuing the Matching Control Method for Execution

When a general-purpose event is raised, the ACPI driver uses a naming convention to determine which control method to queue for execution and how the GPE EIO 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 EIO 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, 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 - 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 (\_TZ.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.

The OS 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, the OS 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 ACPI OS of various object-related events. All such notification are done with the Notify operator that 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 15.2.3.5.1.11.

Table 5-21 Device Object Notification Types

Value	Description
0	**Device Check**. This notification is performed on a device object to indicate to the OS that it needs to perform the Plug and Play re-enumeration operation on the device tree starting from the point where has been notified. The OS will only perform this operation at boot, and when notified. It is the responsibility of the ACPI AML code to notify the OS 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 Device Check notification on wake to inform the OS that it needs to check the configuration for a device change.
1	**Ejection Request.** Used to notify the OS that the device's ejection button has been pressed, and that the OS needs to perform the Plug and Play ejection operation. This event can only be notified in response to a physical user action to remove the device.
2	**Device Wake.** Used to notify the OS that the device has signaled its wake event, and that the OS needs to notify the OS native device driver for the device. This is only used for devices that support _PRW.
3-7F	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 5-22 Control Method Battery Device Notification Values

Hex value	Description
80	**Battery Status Changed**. Used to notify that the control method battery device status has changed.
81	**Battery Information Changed.** Used to notify that the control method battery device information has changed. This only occurs when a battery is replaced.
>81	Reserved.

Table 5-23 Power Source Object Notification Values

Hex value	Description
80	**Power Source Status Changed**. Used to notify that the power source status has changed.
>80	Reserved.

Table 5-24 Thermal Zone Object Notification Values

Hex value	Description
80	**Thermal Zone Status Changed**. Used to notify that the thermal zone temperature has changed.
81	**Thermal Zone Trip points Changed.** Used to notify that the thermal zone trip points have changed.
>81	Reserved.

Table 5-25 Control Method Power Button Notification Values

Hex value	Description
80	**S0 Power Button Pressed.** Used to notify that the power button has been pressed while the system is in the S0 state. Note 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 5-26 Control Method Sleep Button Notification Values

Hex value	Description
80	**S0 Sleep Button Pressed.** Used to notify that the sleep button has been pressed while the system is in the S0 state. Note 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 5-27 Control Method Lid Notification Values

Hex value	Description
80	**Lid Status Changed.** Used to notify that the control method lid device status has changed.
>80	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 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 5-28 ACPI Device IDs

Plug and Play ID	Description
PNP0C08	**ACPI.** Not declared in ACPI as a device. This ID is used by the operating system the ACPI driver 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 ACPI Bus.** A device that is only a bus whose bus settings are totally controlled by its ACPI resource information, and otherwise needs no bus-specific driver support.
PNP0A06	**Extended IO 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).
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 for more details.
ACPI0001	**SMBus Host Controller.** SMBus host controller using the embedded controller interface (as specified in section 13.9).
ACPI0002	**Smart Battery Subsystem.** The Smart battery Subsystem specified in section 11.

5.6.5 Defined Generic Object and Control Methods

The following table lists all the generic object and control methods defined in this specification and gives a reference to the defining section of the specification.

Table 5-29 Defined Generic Object and Control Methods

Object	Description
_ADR	Device identification object that evaluates to a device’s address on its parent bus. See section 6.1.
_AC*x*	Thermal zone object that returns Active trip point in Kelvin (to 0.1 degrees) See section 12.2.

Object	Description
_AL*x*	Thermal zone object containing list of pointers to active cooling device objects. See section 12.2.
_CID	Device identification object that evaluates to a device's Plug and Play Compatible ID list. See section 6.1.
_CRS	Device configuration object that specifies a device's *current* resource settings, or a control method that generates such an object. See section 6.2.
_CRT	Thermal zone object that returns critical trip point in Kelvin (to 0.1 degrees). See section 12.2.
_DCL	Thermal zone object that returns list of pointers to Bay device objects within the thermal zone. See section 12.2.
_DIS	Device configuration control method that disables a device. See section 6.2.
_EC	Control Method used to define the offset address and Query value of an SMBus host controller defined within an embedded controller device. See section 13.12.
_EJD	Device insertion/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. See section 6.3.
_EJ*x*	Device insertion/removal control method that ejects a device. See section 6.3.
_HID	Device identification object that evaluates to a device's Plug and Play Hardware ID. See section 6.1.
_IRC	Power management object that signifies the device has a significant inrush current draw. See section 7.3.1.
_LCK	Device insertion/removal control method that locks or unlocks a device. See section 6.3.
_MSG	System indicator control that indicates messages are waiting. See section 10.1.
_OFF	Power resource object that sets the resource off. See section 7.4.
_ON	Power resource object sets the resource on. See section 7.4.
_PCL	Power source object that contains a list of pointers to devices powered by a power source. See section 11.3.2.
_PRS	Device configuration object that specifies a device's *possible* resource settings, or a control method that generates such an object. See section 6.2.
_PRW	Power management object that evaluates to the device's power requirements in order to wake the system from a system sleeping state. See section 7.2.1
_PR0	Power management object that evaluates to the device's power requirements in the D0 device state (device fully on). See section 7.2.2.
_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. See section 7.2.3
_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. See section 7.2.4.
_PSC	Power management object that evaluates to the device's current power state. See section 7.3.3.
_PSL	Thermal zone object that returns list of pointers to passive cooling device objects. See section 12.2.
_PSR	Power source object that returns present power source device. See section 11.3.1.
_PSV	Thermal zone object that returns Passive trip point in Kelvin (to 0.1 degrees). See section 12.2.
_PSW	Power management control method that enables or disables the device's WAKE function. See section 7.2.
_PS0	Power management control method that puts the device in the D0 device state. (device fully on). See section 7.2.
_PS1	Power management control method that puts the device in the D1 device state. See section 7.2.
_PS2	Power management control method that puts the device in the D2 device state. See section 7.2.

Object	Description
_PS3	Power management control method that puts the device in the D3 device state (device off). See section 7.2.
_RMV	Device insertion/removal object that indicates that the given device is removable. See section 6.3.
_SBC	System indicator control method that indicates the system battery charge level. See section 10.1.
_SBS	System indicator control method that indicates the system battery state. See section 10.1.
_SCP	Thermal zone object that sets user cooling policy (Active or Passive). See section 12.2.
_SLN	Device identification object that evaluates to the slot number for a slot. See section 6.1.4.
_STA	Device insertion/removal control method that returns a device's status. See section 6.3.
_STA	Power resource object that evaluates to the current on or off state of the Power Resource. See section 7.4.
_SRS	Device configuration control method that sets a device's settings. See section 6.2.
_SST	System indicator control method that indicates the system status. See section 10.1.
_TC1	Thermal zone object that contains thermal constant for Passive cooling. See section 12.2.
_TC2	Thermal zone object that contains thermal constant for Passive cooling. See section 12.2.
_TMP	Thermal zone object that returns current temperature in Kelvin (to 0.1 degrees). See section 12.2.
_TSP	Thermal zone object that contains thermal sampling period for Passive cooling. See section 12.2.
_UID	Device identification object that specifies a device's unique persistent ID, or a control method that generates it. See section 6.1.
_PTS	Power management control method used to prepare to sleep. See section 7.4.1.
_S0	Power management package that defines system **_S0** state mode. See section 7.4.1.
_S1	Power management package that defines system **_S1** state mode. See section 7.4.1.
_S2	Power management package that defines system **_S2** state mode. See section 7.4.1.
_S3	Power management package that defines system **_S3** state mode. See section 7.4.1.
_S4	Power management package that defines system **_S4** state mode. See section 7.4.1.
_S5	Power management package that defines system **_S5** state mode. See section 7.4.1.
_WAK	Power management control method run once system is awakened. See section 7.4.1.

5.7 OS-Defined Object Names

A list of OS-supplied object names are shown in the following table.

Table 5-30 Predefined Global Events

Name	Description
_GL	Global Lock
_OS	Name of the operating system.
_REV	Revision of the AML interpreter for the specified OS.

5.7.1 _GL Global Lock Mutex

This object is a Mutex object that behaves like a Mutex as defined in section 15.2.3.4.1.7 with the added behavior that acquiring this Mutex also acquires the shared environment Global Lock defined in section 5.2.6.1. This allows Control Methods to explicitly synchronize with the Global Lock if necessary.

5.7.2 _OS Name object

This object is contains a string that identifies the operating system. This value does not change with different revisions of the AML interpreter.

5.7.3 _REV data object

This object is contains the revision of the AML interpreter for the specified _OS as a Dword. Larger values are newer revisions of the interpreter.

6. Configuration

This section specifies the objects the OS 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 configure hardware resources 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 in the following table:

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.
_HID	Object which evaluates to a device's Plug and Play Hardware ID.
_SUN	Object that evaluates to the slot UI number for a slot.
_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), the ACPI driver enumerates the devices' Plug and Play ID(s) and the ACPI BIOS must supply a _HID object (plus an optional _CID object) for each device to enable the ACPI driver to do that. For devices on an enumerable type of bus, such as a PCI bus, the ACPI system needs to 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.

6.1.1 _ADR

This object is used to supply the OS 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..

The OS infers the parent bus from the location of the _ADR object's Device Package in the ACPI name space. For more information about the positioning of Device Packages in the ACPI name space, see "Named Object Creation Encodings."

_ADR object information must be static, and can be defined for the following bus types listed in the following table.

Table 6-2 _ADR Object Bus Types

BUS	Address encoding
EISA	EISA slot number 0 - F
IDE Controller	0=Primary Channel, 1=Secondary Channel
IDE Channel	0=Master drive, 1=Slave drive
PCI	High word = Device #, Low word = Function #. (e.g., 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

BUS	Address encoding
SMB	Lowest Slave Address

6.1.2 _CID

This optional object is used to supply the OS 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 a compatible device ID, or a package of compatible device IDs, for the device in the order of preference. A compatible ID must be either a numeric 32-bit compressed EISA type ID or a PCI ID. The format of PCI IDs 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.

6.1.3 _HID

This object is used to supply the OS with the device's Plug and Play Hardware ID. 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 the ACPI driver. The ACPI driver only enumerates a device when no bus enumerator can detect the device ID. For example, devices on an ISA bus are enumerated by the ACPI driver. Use the _ADR object to describe devices enumerated by bus enumerators other than the ACPI driver.
A _HID object evaluates to either a numeric 32-bit compressed EISA type ID or a string.

6.1.4 _SUN

_SUN is used by the OS user interface 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 which is the number to be used in the user interface. This number must match any slot number printed on the physical slot.

6.1.5 _UID

This object provides the OS 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 only needs to be unique amongst all devices with the same device ID. The OS 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 which returns the unique docking station ID.

A _UID object evaluates to either a numeric value or a string.

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, etc.

When the ACPI driver enumerates a device, it will call _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 will determine what resources the device should consume and set 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. These descriptors point to a single device object (resource producer) that claims the shared resource in it's _PRS. This allows the OS to clearly understand the resource dependencies in the system and move all related devices together if it needs to change resources. Further, it allows the OS to only allocate resources to resource producers when devices that consume that resource appear.

The device configuration objects are listed in the following table.

Table 6-3 Device Configuration Objects

Object	Description
_CRS	An object that specifies a device's *current* resource settings, or a control method that generates such an object.
_DIS	A control method that disables a device.
_PRS	An object that specifies a device's *possible* resource settings, or a control method that generates such an object.
_PRT	An object that specifies the PCI interrupt Routing Table.
_SRS	A control method that sets a device's settings.

6.2.1 _CRS

This required object evaluates to a byte stream that describes the system resources currently allocated to a device. 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 will be 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, a compatible extension of the formats specified in the PNPBIOS Specification. 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

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, the OS 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 _PRT

PCI interrupts are inherently non-heirarchical. PCI interrupt pins are typically wired together to four interrupt vectors in the interrupt controller. PRT provides a mapping table from PCI interrupt pins to the interrupt vectors the pins are connected to. PRT is a package that contains a list of packages, each of which describes the mapping of an interrupt pin. These mapping packages have the following fields:

Table 6-4 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	Name	Name of a the device that allocates the interrupt the above pin is connected to. If this field is null, then the interrupt is allocated from the global interrupt vector pool.
Source Index	BYTE	An index that indicates which resource descriptor in the resource template of the device pointed to in Source this interrupt is allocated from. If Source is null, this field is the interrupt vector number the pin is connected to.

There are two ways that _PRT can be used. Typically, the vector 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 device object. These objects have _PRS, _CRS, _SRS, and _DIS control methods to allocate the interrupt vectors. Then, the PCI driver handles the interrupts not as interrupt vectors 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 different vectors on the interrupt controller, the OS will use _PRS, _CRS, _SRS, and _DIS control methods for the interrupt's device object.
In the second model, the PCI interrupts are hard-wired to specific interrupt vectors on the interrupt controller and are not configurable. In this case, the Source field in _PRT does point to a device, but is null, and the Source Index field contains the global interrupt vector that the PCI interrupt is hard wired to.

6.2.3.1 Example: Using _PRT to describe PCI IRQ routing

The following example describes two PCI slots and a PCI video chip. Note that the interrupts on the two PCI slots are wired up differently (barber polled).

```
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, LNKA, 0},  // Slot 1, INTA
            Package{0x0004ffff, 1, LNKB, 0},  // Slot 1, INTB
            Package{0x0004ffff, 2, LNKC, 0},  // Slot 1, INTC
            Package{0x0004ffff, 3, LNKD, 0},  // Slot 1, INTD
            Package{0x0005ffff, 0, LNKB, 0},  // Slot 2, INTA
            Package{0x0005ffff, 1, LNKC, 0},  // Slot 2, INTB
            Package{0x0005ffff, 2, LNKD, 0},  // Slot 2, INTC
            Package{0x0005ffff, 3, LNKA, 0},  // Slot 2, INTD
            Package{0x0006ffff, 0, LNKC, 0}   // Video, INTA
        })
    }
}
```

6.2.4 _PRS

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 the OS to select a conflict-free resource allocation without user intervention.
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.

6.2.5 _SRS

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.

If the device is disabled, _SRS will enable 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. The OS 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. Note 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 (i.e., hot insertion), the hardware generates an SCI general purpose event.
2. The _ control method for the event uses the Notify(*device*,0) command to inform the OS of which bus 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. Performance can be optimized by having Notify point as closely as possible in the 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 7.5.3) 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.
3. The OS 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, the OS 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. The OS will then load and configure all devices in 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. (i.e. 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 an SCI 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*, 1) command to inform the OS 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) will automatically be removed.
3. The operating system will shut down and unload devices that will be removed.
4. If the device has a _LCK control method, the OS will run this control method to unlock the device.
5. The operating system 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 (i.e. an undock) and the battery is low, dead, or not present, the OS will use the lowest supported sleep state _EJ*x* listed; otherwise it will use the highest state _EJ*x*. Having made this decision, the OS will run the appropriate _EJ*x* control method to prepare the hardware for eject.
6. If the removal will be a warm removal, the OS puts the system in the appropriate S*x* state. If the removal will be a hot removal, the OS skips to step 8, below.
7. When the hardware is put into the sleep state, it can use any motors, etc to eject the device. Immediately after ejection, the hardware will wake the computer to an S0 state. If the system was sleeping when the eject notification came in, the operating system will return the computer to a sleeping state consistent with the user's wakeup settings.
8. The OS will call _STA to determine if the eject successfully occurred. (In this case, control methods do not need to call Notify() to tell the OS of the change in _STA) If there were any mechanical failures, _STA will return 3: device present and not functioning, and the OS will inform the user of the problem.

Note: this mechanism is the same for removing a single device as well as 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 since system and data integrity cannot be guaranteed when a surprise-style removal occurs. Because the operating system 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 the following table.

Table 6-5 Device Insertion and Removal Objects

Object	Description
_EJD	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.
_EJ*x*	A control method that ejects a device.
_LCK	A control method that locks or unlocks a device.
_RMV	Object that indicates that the given device is removable.
_STA	A control method that returns a device's status

6.3.1 _EJD

This object is used to name the device object of another device upon which a device is dependent and is primarily used to support docking stations. Whenever the named device is ejected, the dependent device will also receive an ejection notification.

An _EJD object evaluates to the name of another device object. This object's EJx methods will be used to eject all the dependent devices. Devices that have an _EJD object cannot have any _EJx control methods.

A device's dependents will be ejected when the device itself is ejected.

When describing a platform that includes a docking station, usually more than one _EJD object will be required. For example, if a dock attaches both a PCI device and an ACPI-configured device to a portable, 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 submits an eject notify (_EJN) request, the OS would first attempt to disable and unload the drivers for both the PCI and ACPI configured devices.

6.3.2 _EJ*x*

These control method are optional and are only supplied for a device which supports a software-controlled VCR-style ejection mechanism. To support warm and hot removal, an _EJ*x* control method is listed for each sleep state the device supports removal from, 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 the OS calls the _EJ0 control method. The _EJ0 control method does not return until ejection is complete. After calling _EJ0, the OS will call _STA to determine whether or not the eject succeeded.

For warm removal, the _EJ1-_EJ4 control methods do not cause the device to be immediately ejected. Instead, they only 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 the OS has put the system into a sleep state by writing to the SLP_EN register. After the system resumes, the OS will call _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 enable warm eject.
0 = Disable (cancel) warm eject (EJ0 will never be called with this value).

A device object may have at most 2 _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 or EJ5 control method to be used when the system will not have power (e.g. no battery) after the eject. For example, a hot-docking notebook might list _EJ0 and _EJ5.

6.3.3 _LCK

This control method is optional and is only required for a device which supports a software-controlled locking mechanism. When the operating software 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.

6.3.4 _RMV

The _RMV object indicates to the OS that the device can be removed while the system is in the working state (i.e., 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 the OS to indicate to the user that the device can be removed and for the OS to provide a way for shutting down this device before removing it.

6.3.5 _STA

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 user interface
bit 3: Set if the device is functioning properly (cleared if the device failed its diagnostics)
bits 4:31: Reserved (must be cleared)

If bit 0 is cleared, then bit 1 must also be cleared (i.e., 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 the OS user interface, bit 2 is set. For example, a notebook could have joystick hardware in the notebook (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.

If a device object does not have an _STA object, then the OS will assume that all of the above bits are set (i.e. the device is Present, Enabled, Shown in the UI, and Functioning).

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 ResourceTemplate macro creates Buffer for 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:

```
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.
For example, given the above resource template, the following code changes the minimum and maximum addresses for the IO descriptor named IO2:

```
Store(0xA000, IO2._MIN)
Store(0xB000, IO2._MAX)
```

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.

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.

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 6-6 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 6-7 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 6-8 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.

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. Bit[7:5] *Reserved and must be 0* Bit[4] Interrupt is sharable, _SHR Bit[3] Low true level sensitive, _LL Bit[2:1] *Ignored* Bit[0] High true edge sensitive, _HE

NOTE: *Low true, level sensitive interrupts may be electrically shared, 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.

6.4.2.1.1 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
    )
    {
    ByteConst [, ByteConst ...]   // 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
    )
    {
    ByteConst [, ByteConst ...]   // list of IRQ numbers (valid values: 0-15)
    }
```

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 6-9 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.

Offset	Field Name
Byte 2	Bit[7] *Reserved and 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 Bit[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*

6.4.2.2.1 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
    )
    {
    ByteConst [, ByteConst ...]                   // List of channel numbers
                                                  //(valid values: 0-17)
    }
```

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 6-10 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 it's 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 6-11 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

Note 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 which appears earliest (nearest the beginning) in the structure has the highest priority, and so on.

6.4.2.3.1 ASL Macro for Start Dependent Function Descriptor

The following macro generates a Start Dependent Function descriptor with the optional priority byte:

```
StartDependentFn(
    ByteConst,      // Compatibility priority (valid values: 0-2)
    ByteConst       // 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
    }
```

6.4.2.4 End Dependent Functions (Type 0, Small Item Name 0x7, Length=0)

Table 6-12 End Dependent Functions

Offset	Field Name
Byte 0	Value = 0_0111_000B (Type = 0, small item name = 0x7 length =0)

Note that only one End Dependent Function item is allowed per logical device. This enforces the fact that Dependent Functions cannot be nested.

6.4.2.4.1 ASL Macro for End Dependent Functions descriptor

The following macro generates an End Dependent Functions descriptor:

```
EndDependentFn(
)
```

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 6-13 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.

6.4.2.5.1 ASL Macros for IO Port Descriptor

The following macro generates a short IO descriptor:

```
IO(
    Decode16 | Decode10,      // _DEC
    WordConst,                // _MIN, Address minimum
    WordConst,                // _MAX, Address max
    ByteConst,                // _ALN, Base alignment
    ByteConst                 // _LEN, Range length
    NameString | Nothing      // A name to refer back to this resource
)
```

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 6-14 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.

Offset	Field Name	Definition
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.

6.4.2.6.1 ASL Macro for Fixed IO Port Descriptor

The following macro generates a short Fixed IO descriptor:

```
FixedIO(
    WordConst,            // _BAS, Address base
    ByteConst             // _LEN, Range length
    NameString | Nothing  // A name to refer back to this resource
    )
```

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 6-15 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

6.4.2.7.1 ASL Macro for Vendor Defined Descriptor

The following macro generates a short vendor specific descriptor:

```
VendorShort(
    NameString | Nothing          // A name to refer back to this resource
    )
    {
    ByteConst [, ByteConst ...]   // List of bytes, up to 7 bytes
    }
```

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 6-16 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.

6.4.2.8.1 ASL Macro for End Tag

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 K of data.

Table 6-17 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

There following large information items are currently defined for Plug and Play ISA devices:

Table 6-18 Large Resource Items

Large Item Name	Value
24-bit memory range descriptor	0x1
Reserved	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
Reserved	0xA - 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 6-19 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)
Byte 3	Information	This field provides extra information about this memory. Bit[7:1] *Ignored* Bit[0] Write status, _RW Status 1 writeable 0 non-writeable (ROM)
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

Offset	Field Name, ASL Field Name	Definition
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 KByte)
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 KByte)
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.

NOTE: Address bits [7:0] of memory base addresses are assumed to be 0.

NOTE: 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.

NOTE: 24-bit Memory Range descriptors are used for legacy devices.

NOTE: Mixing of 24-bit and 32-bit memory descriptors on the same device is not allowed.

6.4.3.1.1 ASL Macro for 24-bit Memory Descriptor

The following macro generates a long 24 bit memory descriptor:

```
Memory24(
    ReadWrite | ReadOnly,      // _RW
    WordConst,                 // _MIN, Minimum base memory address [23:8]
    WordConst,                 // _MAX, Maximum base memory address [23:8]
    WordConst,                 // _ALN, Base alignment
    WordConst                  // _LEN, Range length
    NameString | Nothing       // A name to refer back to this resource
    )
```

6.4.3.2 Vendor Defined (Type 1, Large Item Name 0x4)

The vendor defined resource data type is for vendor use.

Table 6-20 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

6.4.3.2.1 ASL Macro for Vendor Defined Descriptor

The following macro generates a long vendor specific descriptor:

```
VendorLong(
   NameString | Nothing          // A name to refer back to this resource
   )
   {
   ByteConst [, ByteConst ...]   // List of bytes
   }
```

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 6-21 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 Status 1 writeable 0 non-writeable (ROM)
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
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.

Offset	Field Name	Definition
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.

6.4.3.3.1 ASL Macro for 32-Bit Memory Descriptor

The following macro generates a long 32-bit memory descriptor:

```
Memory32(
    ReadWrite | ReadOnly,      // _RW
    DWordConst,                // _MIN, Minimum base memory address
    DWordConst,                // _MAX, Maximum base memory address
    DWordConst,                // _ALN, Base alignment
    DWordConst                 // _LEN, Range length
    NameString | Nothing       // A name to refer back to this resource
    )
```

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 6-22 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 Status 1 writeable 0 non-writeable (ROM)
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

Offset	Field Name	Definition
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.

6.4.3.4.1 ASL Macros for 32-bit Fixed Memory Descriptor

The following macro generates a long 32 bit fixed memory descriptor:

```
Memory32Fixed(
    ReadWrite | ReadOnly,          // _RW
    DWordConst,                    // _BAS, Range base
    DWordConst                     // _LEN, Range length
    NameString | Nothing           // A name to refer back to this resource
    )
```

6.4.3.5 Address Space Descriptors

The 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 busses), and resource types found on some RISC processors.

6.4.3.5.1 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 6-23 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 = 22 (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

Offset	Field Name	Definition
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)
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 all be set. (i.e. 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 primary 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 primary side of the bridge.
Byte 15	Address range maximum, _MAX bits [15:8]	

Offset	Field Name	Definition
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 primary side to obtain the address on the secondary side. Non-bridge devices must list 0 for all Address Translation offset bits.
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	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.

6.4.3.5.2 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
    DWordConst,                                      // _GRA, Address granularity
    DWordConst,                                      // _MIN, Address range minimum
    DWordConst,                                      // _MAX, Address range max
    DWordConst,                                      // _TRA, Translation
    ByteConst | Nothing,                             // Resource Source Index;
                                                     // if Nothing, not generated
    NameString | Nothing                             // Resource Source;
                                                     // if Nothing, not generated
    NameString | Nothing                             // A name to refer back
                                                     // to this resource
    )
```

The following generates a DWORD Address descriptor with ResourceType = IO:

```
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
    DWordConst,                                      // _GRA: Address granularity
    DWordConst,                                      // _MIN: Address range minimum
    DWordConst,                                      // _MAX: Address range max
    DWordConst,                                      // _TRA: Translation
    ByteConst | Nothing,                             // Resource Source Index;
                                                     // if Nothing, not generated
    NameString | Nothing                             // Resource Source;
                                                     // if Nothing, not generated
    NameString | Nothing                             // A name to refer back to this resource
    )
```

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 7-19; the only difference is that the address fields are 16 bits wide rather than 32.

Table 6-24 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)

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 all be set. (i.e. 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 primary 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 primary 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 primary side to obtain the address on the secondary side. Non-bridge devices must list 0 for all Address Translation offset bits.
Byte 13	Address Translation offset, _TRA bits [15:8]	
Byte 14	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.

6.4.3.5.4 ASL Macros for WORD Address Descriptor

The following macro generates a WORD Address descriptor with ResourceType = IO

```
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
    WordConst,                                          // _GRA: Address granularity
    WordConst,                                          // _MIN: Address range minimum
    WordConst,                                          // _MAX: Address range max
    WordConst,                                          // _TRA: Translation
    ByteConst | Nothing,                                // Resource Source Index;
                                                        // if Nothing, not generated
    NameString | Nothing                                // Resource Source;
                                                        // if Nothing, not generated
    NameString | Nothing                                // A name to refer back
                                                        // to this resource
    )
```

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
    WordConst,                                          // _GRA, Address granularity
    WordConst,                                          // _MIN, Address range minimum
    WordConst,                                          // _MAX, Address range max
    WordConst,                                          // _TRA: Translation
    ByteConst | Nothing,                                // Resource Source Index;
                                                        // if Nothing, not generated
    NameString | Nothing                                // Resource Source;
                                                        // if Nothing, not generated
    NameString | Nothing                                // A name to refer back
                                                        // to this resource
    )
```

6.4.3.5.5 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 6-25 Memory Resource Flag (Resource Type = 0) Definitions

Bits	Meaning
Bits[7:4]	Reserved; must be 0
Bits[4:1]	Memory attributes, _MEM Value Meaning 0 The memory is noncacheable 1 The memory is cacheable 2 The memory is cacheable and supports write combining 3 The memory is cacheable and prefetchable >3 Reserved
Bit[0]	Write status, _RW 1: This memory range is read-write 0: This memory range is read-only

Table 6-26 I/O Resource Flag (Resource Type = 1) Definitions

Bits	Meaning
Bit[7:2]	Reserved; must be 0

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 name space.
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 names pace.

Table 6-27 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 6-28 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 = 12 (minimum)
Byte 2	Length, bits[15:8]	Variable: Value = 0 (minimum)
Byte 3	Interrupt Vector Flags	Interrupt Vector Information. Bit[7:4] Reserved, must be 0. Bit[3] Interrupt is shareable, _SHR Bit[2] Low true level sensitive, _LL Bit[1] High true level sensitive, _HE 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 the OS passes this descriptor to _SRS, this field must be set to 1.
Byte 4n+5	Interrupt Number, _INT bits [7:0]	Interrupt number.
Byte 4n+6	Interrupt Number, _INT bits [15:8]	
Byte 4n+7	Interrupt Number, _INT bits [23:16]	
Byte 4n+8	Interrupt Number, _INT bits [31:24]	
…	…	Additional interrupt numbers

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 operating system 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.

6.4.3.6.1 ASL Macro for Extended Interrupt Descriptor

The following macro generates an extended interrupt descriptor:

```
Interrupt(
    ResourceConsumer | ResourceProducer | Nothing,   // Nothing=>ResourceConsumer
    Edge | Level,                                    // _LL, _HE
    ActiveHigh | ActiveLow ,                         // __LL, _HE
    Shared | Exclusive | Nothing,                    // _SHR: Nothing=>Exclusive
    ByteConst | Nothing,                             // Resource Source Index;
                                                     // if Nothing, not generated
    NameString | Nothing                             // Resource Source;
                                                     // if Nothing, not generated
    NameString | Nothing                             // A name to refer back
                                                     // to this resource
    )
    {
    DWordConst [, DWordConst ...]                    // _INT, list of interrupt numbers
    }
```

7. Power Management

This section specifies the device power management objects and system power management objects the OS can use to perform power management on a platform. The system state indicator objects are also specified in this section.

7.1 Declaring a PowerResource 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 name space.
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 15.

Systemlevel is the lowest power system sleep level the OS 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. The OS 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 Name Space location. For a description of the ACPI-defined named objects for a Power Resource, see section 7.2.

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
    }
}
```

7.2 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.

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.

Table 7-1 Device Power Management Child Objects

Object	Description
_IRC	Object that signifies the device has a significant inrush current draw.
_PRW	Object that evaluates to the device's power requirements in order to wake the system from a system sleeping 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 which 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 which can achieve the defined D2 device state according to the related device class.
_PSC	Object that evaluates to the device's current power state.
_PSW	Control method that enables or disables the device's WAKE function.
_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).

7.2.1 _PRW

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 7-2 Wake Power Requirements Package

	Object	Description
0	numeric	The bit index in GPEx_EN of the enable bit that is enabled for the wake event.
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 the OS 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.

7.2.2 _PR0

This object evaluates to a package of the following definition:

Table 7-3 Power Resource Requirements Package

	Object	Description
0	numeric	The lowest power system sleeping state in which the device can still maintain this power state.
1	object reference	Reference to required Power Resource #0.
…		
N	object reference	Reference to required Power Resource #N.

For the OS to put the device in the D0 device state, the following must occur:
1. The OS determines the lowest power system state the system must maintain to power the device in the D0 state.
2. All Power Resources referenced by elements 1 through N must be in the ON state.
3. All Power Resources no longer referenced by any device in the system must be in the OFF state.
4. If present, the _PS0 control method is executed to set the device into the D0 device state.

7.2.3 _PR1

This object evaluates to a package as defined in Table 7-3. For the OS to put the device in the D1 device state, the following must occur:
1. The OS determines the lowest power system state the system must maintain to power the device in the D1 state.
2. All Power Resources referenced by elements 1 through N must be in the ON state.
3. All Power Resources no longer referenced by any device in the system must be in the OFF state.
4. If present, the _PS1 control method is executed to set the device into the D1 device state.

7.2.4 _PR2

This object evaluates to a package as defined in Table 7-3. For the OS to put the device in the D2 device state, the following must occur:
1. The OS determines the lowest power system state the system must maintain to power the device in the D2 state.
2. All Power Resources referenced by elements 1 through N must be in the ON state.
3. All Power Resources no longer referenced by any device in the system must be in the OFF state.
4. If present, the _PS2 control method is executed to set the device into the D2 device state.

7.3 Power Resources for OFF

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 the OS to put the device in the D3 state, the following must occur:
1. All Power Resources no longer referenced by any device in the system must be in the OFF state.
2. 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.

7.3.1 _IRC

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 the ACPI driver to the OS such that the OS can serialize all devices in the system that have in-rush current serialization requirements. The OS can only transition one device flagged with _IRC to the D0 state at a time.

7.3.2 _PSW

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

7.3.3 _PSC

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 7-4.

Table 7-4 _PSC Control Method Result Codes

Result	Device State
0	D0
1	D1
2	D2
3	D3

7.3.4 _PS0

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

7.3.5 _PS1

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

7.3.6 _PS2

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

7.3.7 _PS3

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 IO ports are available to other devices).

Arguments:
None
Result code:
None

7.4 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 the OS changes the state of device objects in the system, the power resources which are needed will change which will cause the ACPI driver to turn power resources on and off. To determine the initial power resource settings the _STA method can be used.

Table 7-5 Power Resource Child Objects

Object	Description
_STA	Object that evaluates to the current on or off state of the Power Resource. 0 = OFF, 1 = ON
_ON	Set the resource on.
_OFF	Set the resource off.

7.4.1 _STA

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

7.4.2 _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. The ACPI driver 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

7.4.3 _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. The ACPI driver 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

7.5 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 the OS 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
_PTS	Control method used to prepare to sleep
_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.

7.5.1 _PTS Prepare To Sleep

The _PTS control method is executed by the operating system 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 the OS 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, the OS can abort the sleeping operation without notification to the ACPI driver, in which case another _PTS would occur some time before the next attempt by the OS 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).

7.5.2 System _Sx states

All system states supported by the system must provide an object containing the Dword value of the following format in the static Definition Block. The system states, known as S0 - S5, are referenced in the name space as _S0 - _S5 and for clarity the short Sx names are used unless specifically referring to the named _Sx object. For each Sx 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, the OS 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, the OS 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 runtime 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;
        }
```

7.5.2.1 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 the OS assumes the system to be in the S0 state.

7.5.2.2 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 which solely reference Power Resources which 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 which 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 operating software does not have to flush the processor's cache.

7.5.2.3 System _S2 State

The S2 sleeping state is logically lower then 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 which solely reference Power Resources which 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 which 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 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 DRAM.

7.5.2.4 System _S3 State

The S3 state is logically lower then 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.

[10] Or 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.

- Devices states are compatible with the current Power Resource states. only devices which solely reference Power Resources which 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 which 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 required to exit an S3 state and passes control to the firmware resume vector. See section 9.3.2 for more details on BIOS initialization.

From the software view point, 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 can be required to be in 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.

7.5.2.5 System _S4 State

While the system is in this state, it is in the system S4 sleeping state. The state is logically lower then the S3 state and is assumed to conserved 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 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 D4 device state can initiate a hardware event which transitions the system state to S0. This transition causes the processor to begin execution at its boot location.

After the OS has executed the _PTS control method and put the entire system state into main memory, there are two ways which the OS may handle the next phase of the S4 state for saving and restoring main memory. The first way is where the operating system uses its drivers to access the disks and file system structures to save a copy of memory to disk, and then initiates the hardware S4 sequence by setting the SLP_EN register bit. When the system wakes, the firmware performs a normal boot process and loads the OSes loader. The loader then restores the systems memory and wakes the OS.

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 the OS by transferring control to the FACS waking vector.

The S4BIOS transition is optional, but any system which supports this mechanism is required to 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 OSes which do not have support for the direct method.

7.5.2.6 System _S5 State (Soft Off)

The S5 state is similar to the S4 state except that the OS has not saved any context nor set any devices to wake the system. 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. The OS must have all wake events disabled before initiating SLP_EN for the S5 state.

7.5.3 _WAK (System Wake)

After the system has awakened 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 up. The AML code for this control method issues device, thermal, and other notifications to ensure that the OS 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). Note that a device check notification from the _SB node will cause the OS to re-enumerate the entire tree[11].

Hardware is not obligated to track the state needed to supply the resulting status; however, this method can return status concerning the last sleep operation initiated by the OS. The result codes can be used to provide additional information to the OS 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.

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 to be 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 the OS runtime aspects of managing the processor's 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
- Processor clock throttling
- Cooling control

These controls are used in combination by the operating software to achieve the desired balance of the following, sometimes paradoxical, 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 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 P_BLK. For more information, see section 15.

8.2 Processor Power States

By putting a processor into a power state (C1, C2, or C3), 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 sleeping state has a latency associated with entering and exiting that corresponds to the power savings. To conserve power, the operating software puts the processor into one of its supported sleeping states when idle.

8.2.1 Processor Power State C0

While the processor is in this state, it executes instructions. No specific power or thermal savings are realized.

8.2.2 Processor Power State C1

All processors support this power state. This processor power state has the lowest latency, and on IA-PC processors is entered by the "STI-HLT" instruction sequence[14]. The hardware latency on this state is required to 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 power state, this state has no other software-visible effects.

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.

[12] In any system sleeping state, the processors are not executing instructions (that is, not "runtime"), 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] The C1 sleeping state specifically defines interrupts to be enabled while halted.

8.2.3 Processor Power State C2

This processor power state is optionally supported by the system. If present, the state offers improved power savings of 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 FACP Table 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 power state, this state has no other software-visible effects.

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.

8.2.4 Processor Power State C3

This processor power state is optionally supported by the system. If present, the state offers improved power savings of 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 FACP Table, 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. 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.

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 or when BM_RLD is set and a bus master is attempting to gain access to memory.

8.3 Processor State Policy

The operating software can implement control policies based on what is best suited for it. Below is an example policy for IA-PC processors.

```
; ProcessorIdleHandlers is initialized at system initialization time.
; It contains the handler to use for each of the C1, C2, C3 processor
; states.  If the given processor state is not supported, the next
; best handler is installed.
ProcessorIdleHanders      dd     4 dup (?)

ProcessorIdle:
; System determines that processor is idle, and has interrupts
; disabled as that idleness can only be maintained until the next
; interrupt

    call    [IdleHandler]      ; Invoke currently selected idle handler

; IdleHandler enabled interrupts
    jmp     TopOfIdleCode      ; Go check to see if we are still idle
```

Example idle handers are shown below. The strategy shown is for each idle handler to quickly determine that the installed IdleHandler should be demoted to the next lower level. Not shown is an operating environment-specific task of very low priority that waits for the processor's "idleness" to get sufficiently high for a long amount of time, at which point it promotes the IdleHandler to its next higher level.

```
    IdleC1:
        sti
        hlt
        ret

    IdleC2:
    mov eax,LastIdleStart]        ; (eax) = last idle start time
    sub eax,[LastIdleEnd]         ; (eax) = length of last idle
    and eax,0ffffffh              ; mask off sign
    cmp eax,RequiredC2IdleTime    ; was last idle long enough?
    jc short IdleC2Short          ; no, go check for demotion

    mov edx, PM_TMR
    in eax, dx                    ; Get current time
    mov [LastIdleStart], eax      ; This is new LastIdleStart

    mov edx, P_LVL2
    in al, dx                     ; Enter C2

    mov edx, PM_TMR
    in eax, dx                    ;  Ensure C2 entered
    in eax, dx                    ; Get current time
    mov [LastIdleEnd], eax        ; This is new LastIdleEnd

    sti
    ret
```

A demotion policy from **C2** could be to demote to **C1** after two short **C2** idles in a row.

```
IdleC3Uniprocessor:
    mov edx, PM1a_STS
    in  al, dx
    mov edx, PM1b_STS
    mov ah, al
    in  al, dx
    or  ah, al
    test ah, BM_STS               ; Any bus master activity?
    jnz short SetIdleHandlerC2    ; Yes, switch to C2 idle

    mov eax, [LastIdleStart]      ; (eax) = last idle start time
    sub eax, [LastIdleEnd]        ; (eax) = length of last idle
    and eax, 0ffffffh             ; mask off sign
    cmp eax, RequiredC3IdleTime   ; was last idle long enough?
    Jc short IdleC3Short          ; no, go check for demotion

    mov edx, PM_TMR
    in  eax, dx                   ; Get current time
    mov [LastIdleStart], eax      ; This is new LastIdleStart

    mov edx, PM2_CNT              ; disable bus master arbitration
    in  al, dx
    mov ah, al
    or  al, ARB_DIS
    out dx, al

    mov edx, P_LVL3
    in  al, dx                    ; Enter C3.

    Mov edx, PM_TMR
    in  eax, dx                   ; Ensure C3 entered
    in  eax, dx                   ; Get current time
    mov [LastIdleEnd], eax        ; This is new LastIdleEnd

    mov edx, PM2_CNT              ; enable bus master arbitration
    mov al, ah
    out dx, al

    sti
    ret
```

A demotion policy from the C3 handler could be to demote to C2 after two short C2 idles in a row or on one short C3 idle time if the RequiredC3IdleTime and last execution time (difference from current time to LastIdleEnd time) are sufficiently high.

The IdleC3Multiprocessor handler can be used only on systems that identify themselves as having working WBINDV instructions. The handler can take a long time to enter the C3 state, so both the promotion and demotion from this handler would likely be conservative.

```
IdleC3Multiprocessor:
    mov eax, [LastIdleStart]          ; (eax) = last idle start time
    sub eax, [LastIdleEnd]            ; (eax) = length of last idle
    and eax, 0ffffffh                 ; mask off sign
    cmp eax, RequiredMPC3IdleTime     ; was last idle long enough?
    Jc  short IdleC3Short             ; no, go check for demotion

    wbinvd                            ; requires wbinvd support

    mov edx, PM_TMR
    in  eax, dx                       ; Get current time
    mov esi, eax                      ; Remember it

    mov edx, P_LVL3
    in  al, dx                        ; Enter C3.

    mov edx, PM_TMR
    in  eax, dx                       ; Ensure C3 entered
    in  eax, dx                       ; Get current time
    mov [LastIdleStart], esi          ; New LastIdleStart
    mov [LastIdleEnd], eax            ; New LastIdleEnd

    sti
    ret
```

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 state, 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 trade-off cost, power, and wake-up 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-up latency sleeping state (transitioning through the G0 state) [15].

ACPI defines a programming model that provides a mechanism for the ACPI driver 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. The hardware implements different low-power sleeping states and then associates these states with the defined ACPI sleeping states (through the SLP_TYPx fields). The ACPI hardware 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 the ACPI driver 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 the OS 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), the ACPI driver will execute OEM-specific AML/ASL code contained in the Prepare To Sleep, _PTS, control method. One use of the _PTS control method indicates 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.

Upon waking up, the OS will execute the Wake (_WAK) control method. This control method again contains OEM-specific AML/ASL code. One use of the _WAK control method requests the OS 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.

This section discusses the initialization sequence required by an ACPI platform. This includes the boot sequence, different wake-up scenarios, and an example to illustrate how to sue the new E820 calls.

9.1 Sleeping States

The illustration below shows the transitions between the working state, the sleeping states, and the Soft Off state.

[15] The OS uses the RTC wakeup feature to program in the time transition delay. Prior to sleeping, the OS 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] Note 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.

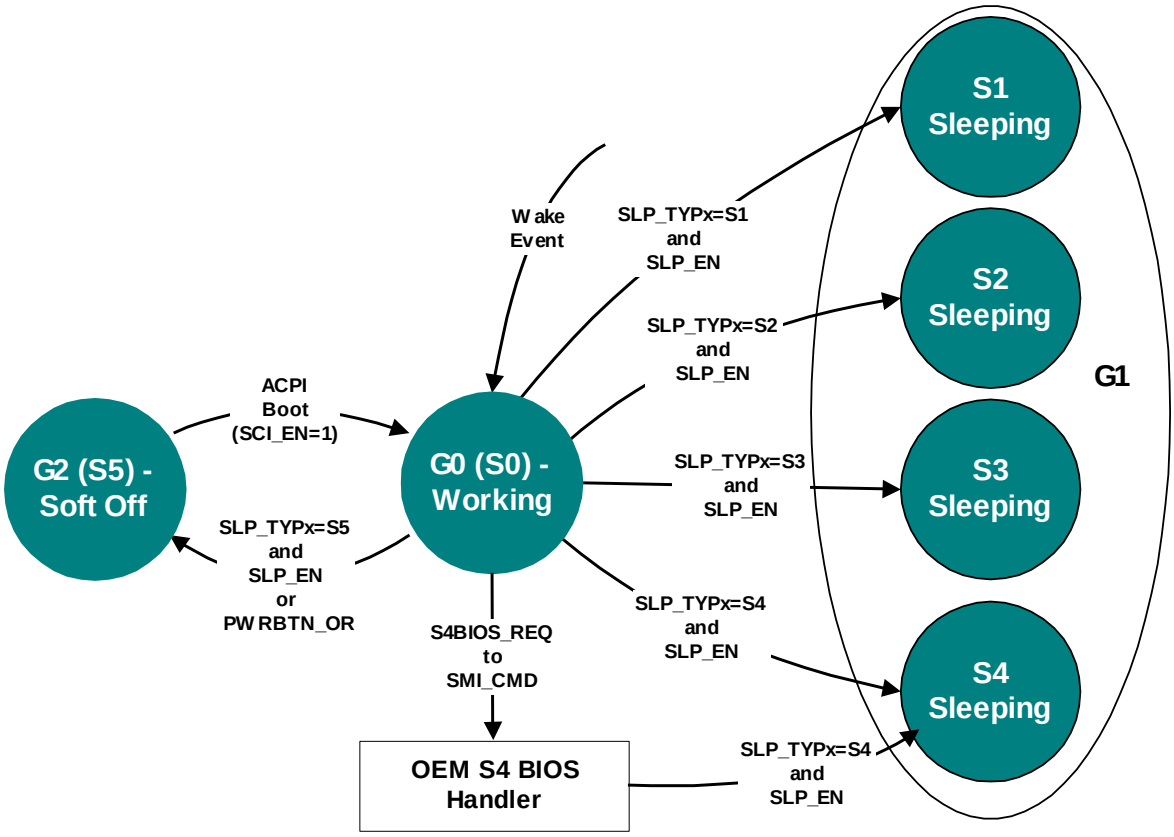


Figure 9-1 Example Sleeping States

ACPI defines distinct differences between the G0 and G1 system states.

- In the G0 state, work is being performed by the OS and 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, the OS will place devices in the D3 state; if a device is enabled to "wake up the system," then the OS will place these devices into the lowest Dx state for which the device still supports wakeup. This is defined in the power resource description of that object; for information, see section 7. This definition of the G1 state implies:
    - The CPU executes no OS code while in the G1 state.
    - To the OS, hardware devices are not operating (except possibly to generate a wakeup event).
    - ACPI registers are affected as follows:
        - Wakeup event bits are enabled in the corresponding fixed or general-purpose registers according to enabled wakeup 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 have different attributes. The attributes were chosen to allow differentiation of sleeping states that vary in power, wakeup latency, and implementation cost tradeoffs. Running the processor at a divided clock rate is not an ACPI sleeping state (G1); this is a working (G0) state. The CPU cannot be executing any instructions when in the sleeping state; the ACPI driver 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. (Note that S4 and S5 are very similar from a hardware standpoint.) At least one sleeping state must be implemented by ACPI-compatible hardware. Many platforms will support multiple sleeping states. ACPI specifies that a 3-bit binary number be associated with the sleeping state (these numbers are given objects within ACPI's root name space: _S0, _S1, _S2, _S3, _S4 and _S5). The ACPI driver will do the following:

1. Pick the closest sleeping state supported by the platform and enabled waking devices.
2. Execute the Prepare To Sleep (_PTS) control method (which passes the type of intended sleep state to OEM AML code) if it is an S1-S4 sleeping state. The _PTS control method is not executed for the S5 soft off 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, the ACPI driver 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, the ACPI driver 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. 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.)

The Prepare To Sleep (_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, so no hardware power-plane sequencing takes place by execution of the _PTS control method.

When the ACPI driver gets control again (after waking up) it will call the wakeup control method (_WAK). This control method executes OEM-specific ASL/AML code to have the OS search for any devices that might 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 wakeup latency sleeping state. In this state no system context is lost (CPU or chip set), and the hardware is responsible for maintaining all system context, which includes the context of the CPU, caches, memory, and all chipset I/O. Examples of S1 sleeping state implementation alternatives follow.

9.1.1.1 S1 Sleeping State Implementation (Example 1)

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 wakeup event should cause the hardware to de-assert the STPCLK# signal to the processor.

9.1.1.2 S1 Sleeping State Implementation (Example 2)

When SLP_TYPx is programmed to the S1 value and the SLP_ENx bit is subsequently set, the hardware will implement an S1 state by doing the following:

1. Place the processor into the stop grant state.
2. Stop the processor's input clock, placing the processor into the stop clock state.
3. Places 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. Stop 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's clock). Hardware must reverse the process (restarting system clocks) upon any enabled wakeup event.

9.1.2 S2 Sleeping State

The S2 state is defined as a low wakeup latency sleep 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). Additionally, control starts from the processor's reset vector after the wakeup event. Before setting the SLP_EN bit, the ACPI driver will flush the system caches. If the platform supports the WBINVD instruction (as indicated by the WBINVD and WBINVD_FLUSH flags in the FACP table), the OS will execute the WBINVD instruction. If the platform does not support the WBINVD instruction to flush the caches, then the ACPI driver will attempt to manually flush the caches using the FLUSH_SIZE and FLUSH_STRIDE fields in the FACP table. The hardware is responsible for maintaining chipset and memory context. An example of a S2 sleeping state implementation follows.

9.1.2.1 S2 Sleeping State Implementation Example

When SLP_TYPx is programmed to the S2 value (found in the _S2 object) and then the SLP_EN bit is set, the hardware will implement an S2 state by doing the following:

- Stop system clocks (the only running clock is the RTC).
- Place system memory into a self or suspend refresh state.
- Power off the CPU and cache subsystem.

In this case, the CPU is reset upon detection of the wakeup 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.
- Call the waking vector.

9.1.3 S3 Sleeping State

The S3 state is defined as a low wakeup latency sleep state, where all system context is lost except for system memory. CPU, cache, and device context are lost in this state; the OS and drivers must restore all device context. Hardware must maintain memory context and restore some CPU and L2 configuration context. Control starts from the processor's reset vector after the wakeup event. Prior to setting the SLP_EN bit, the ACPI driver will flush the system caches. If the platform supports the WBINVD instruction (as indicated by the WBINVD and WBINVD_FLUSH flags in the FACP table), the OS will execute the WBINVD instruction. If the platform does not support the WBINVD instruction then the ACPI driver will attempt to manually flush the cache using the FLUSH_SIZE and FLUSH_STRIDE fields within the FACP table. The hardware is responsible for maintaining chip set and memory context. Examples of an S3 sleeping state implementation follows.

9.1.3.1 S3 Sleeping State Implementation Example

When SLP_TYPx is programmed to the S3 value (found in the _S3 object) and then the SLP_EN bit is set, the hardware will implement an S3 state by doing the following:

- Memory is placed into a low power auto or self refresh state.
- Devices that are maintaining memory isolate themselves from other devices in the system.
- Power is removed 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 wakeup event re-powers the system and resets most devices (depending on the implementation). 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 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.

Note that the BIOS is required to reconfigure the L2 and memory controller to their pre-sleeping states. The BIOS can store the values of the L2 controller into the reserved memory space, where it can then retrieve the values after waking up. The OS will call the Prepare To Sleep method (_PTS) once a 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 this in a non-volatile memory area (as with RTC CMOS RAM) to enable it to restore the values during the waking process.
When the OS re-enumerates buses coming out of the S3 sleeping state, it will discover any devices that have come and gone, and configure devices as they are turned on.

9.1.4 S4 Sleeping State

The S4 sleeping state is the lowest power, longest wakeup latency sleeping state supported by ACPI. In order to reduce power to a minimum, its 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 OS-initiated mechanism is similar to the entry into the S1-S3 sleeping states; the OS driver writes the SLP_TYPx fields and sets the SLP_EN bit. The BIOS-initiated mechanism occurs by the OS transferring control to the BIOS by writing the S4BIOS_REQ value to the SMI_CMD port.
In the OS-initiated S4 sleeping state, the OS is responsible for saving all system context. Before entering the S4 state, the OS will save context of all memory. Upon awakening, the OS will then restore the system context. When the OS 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, the OS 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 OS Initiated S4 Transition

If the OS supports the OS-initiated S4 transition, it will not generate a BIOS-initiated S4 transition. Platforms that support the BIOS-initiated S4 transition also support the OS-initiated S4 transition.
The OS-initiated S4 transition is initiated by the OS driver by saving system context, writing the SLP_TYPx fields, 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 the OS 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 should update 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

For the BIOS-initiated S4 transition, entry into the S4 state occurs by the ACPI driver passing control to BIOS to software. Transfer of control occurs by the OS driver writing the S4BIOS_REQ value into the SMI_CMD port (these values are specified in the FACP table). After BIOS has control, it 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 the OS issuing the ACPI_ENABLE command. If the S4BIOS_F flag in the FACS table is set, this indicates that the ACPI driver can request the BIOS to transition the platform into the S4BIOS sleeping state by writing the S4BIOS_REQ value (found in the FACP table) to the SMI_CMD port (identified by the SMI_CMD value in the FACP table).
Upon waking up the BIOS, software restores memory context and calls the waking vector (similar to wakeup 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 the OS 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 the OS to turn the machine off. Note that the S5 state is not a sleeping state (it is a G2 state) and no context is saved by the OS or hardware. Also note that from a hardware perspective, the S4 and S5 states are 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 the wakeup due to a power button press. Upon waking up, the BIOS does a normal power-on reset, loading the boot sector, and executing (not the waking vector, as it does not exist yet).

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. The OS decides (through a policy scheme) to place the system into the sleeping state.
2. The OS examines all devices who are enabled to wake up the system and determines the deepest possible sleeping state the system can enter to support the enabled wakeup functions. The _PRW named object under each device is examined, as well as the power resource object it points to.
3. The OS executes the Prepare To Sleep (_PTS) control method, passing an argument that indicates the desired sleeping state (1, 2, 3, or 4 representing S1, S2, S3, and S4).
4. The OS places all device drivers into their respective Dx state. If the device is enabled for wakeup, it enters the Dx state associated with the wakeup capability. If the device is not enabled to wakeup the system, it enters the D3 state.
5. OS saves any other processor's context (other than the local processor) to memory
6. OS saves the local processor's context to memory
7. OS writes the waking vector into the FACS table in memory.
8. OS clears the WAK_STS in the PM1a_STS and PM1b_STS registers.
9. OS flushes caches (only if entering S2 or S3).
10. If entering an S4 state using the S4BIOS mechanism, the OS writes the S4BIOS_REQ value (from the FACP table) to the SMI_CMD port. This passes control to the BIOS, which then transitions the platform into the S4BIOS state.
11. If not entering an S4BIOS state, then the OS writes SLP_TYPa (from the associated sleeping object) with the SLP_ENa bit set to the PM1a_CNT register.
12. The OS writes SLP_TYPb with the SLP_EN bit set to the PM1b_CNT register.
13. The OS loops on the WAK_STS bit (in both the PM1a_CNT and PM1b_CNT registers).
14. The system enters the specified sleeping state.

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. The OS prepares its components to shut down (flushing disk caches).
2. The OS writes SLP_TYPa (from the _S5 object) with the SLP_ENa bit set to the PM1a_CNT register.
3. The OS writes SLP_TYPb (from the _S5 object) with the SLP_ENb bit set to the PM1b_CNT register.
4. The system enters the Soft Off state.

9.2 Flushing Caches

Before entering the S2 or S3 sleeping states, the OS is responsible for flushing the system caches. ACPI provides a number of mechanisms to flush system caches:

1. Use the IA instruction WBINVD to flush and invalidate platform caches.
   WBINVD_FLUSH flag set HIGH in the FACP table indicates this support.
2. Use IA instruction WBINVD to flush but NOT invalidate the platform caches.
   WBINVD flag set HIGH in the FACP table indicates this support.
3. Use FLUSH_SIZE and FLUSH_STRIDE to manually flush system caches.
   Both the WBINVD and WBINVD_FLUSH flags both reset LOW indicate this support.

The manual flush mechanism has a number of caveats:

1. Largest cache is 1 MB in size (FLUSH_SIZE is a maximum value of 2 MB).
2. 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 FACP 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. The OS 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 wakeup 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.

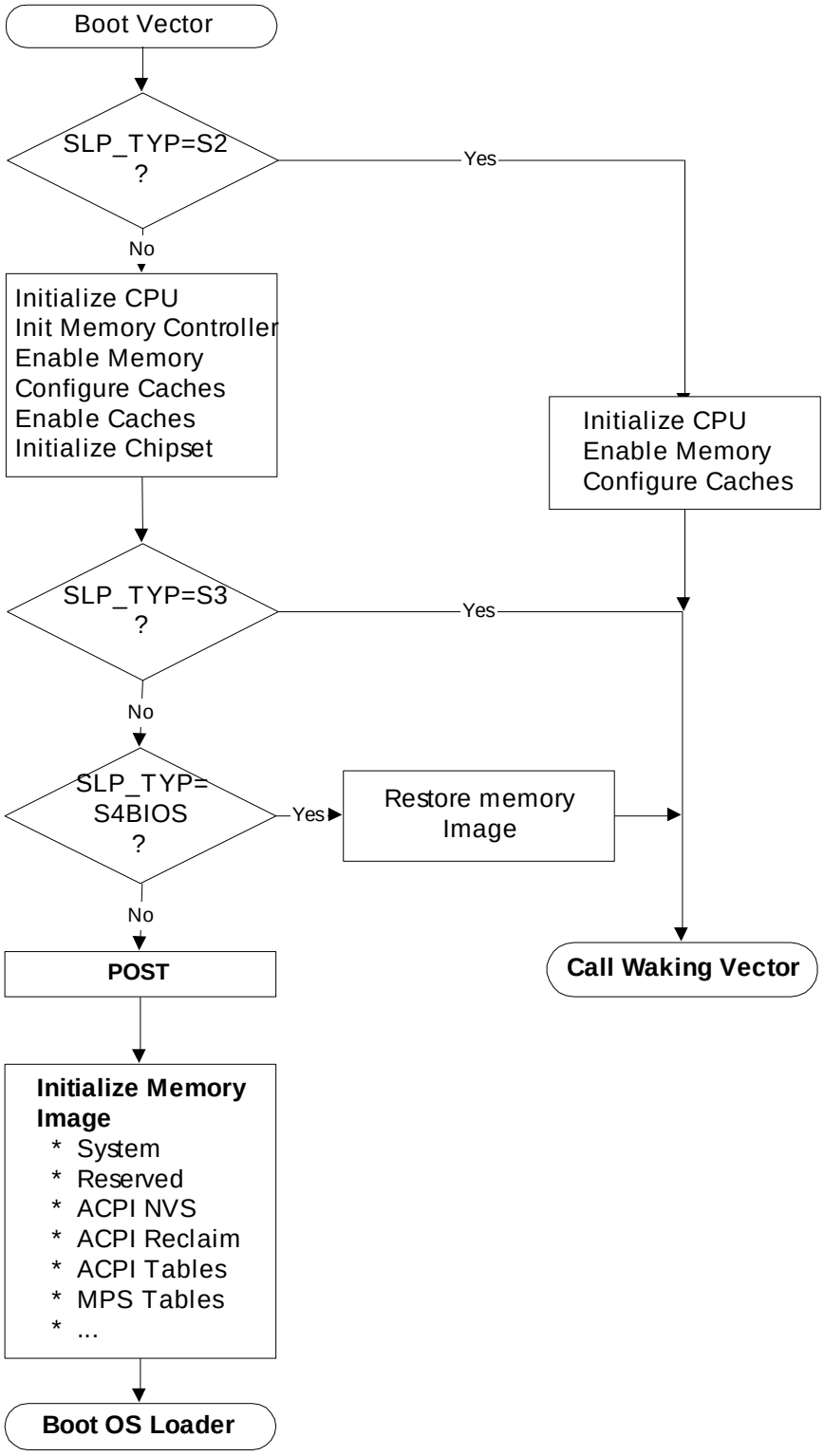


Figure 9-2 BIOS Initialization

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 during a hard or soft reset. The sleeping attributes are such that the power-on sequence (and hard and soft reset) is similar to waking up from an S4 state, the system is configured to a boot

configuration, and then the OS loader is called. Waking up in the S2, S3, or S4 states only requires a partial configuration by the hardware, followed by calling the waking vector (found in the FACP table).
First, the BIOS determines whether this is an S2 wakeup by examining the SLP_TYP register value, which should be preserved between sleeping sessions. If this is an S2 wakeup, then the BIOS handler should enable the memory controller to accept memory accesses (some programming might be required to exit the memory controller from the auto refresh state). At this point, the BIOS reconfigures the caches (cache configuration data having been saved in the ACPI NVS RAM area prior to sleeping), and then calls the waking vector (thus passing control on to the OS).
If this was not a wakeup from an S2 sleeping state (an S3, S4, or boot), then the BIOS initializes the memory controller, configures the caches, and enables access to memory and caches. For the S3 state, there are two classes of hardware: those that lose the configuration of the memory controller when maintaining memory context, and those that don't. If the memory controller's configuration is lost while in the S3 state, then this configuration information should be stored in BIOS non-volatile memory (like RTC CMOS memory) before suspending. Other information such as the cache controller's configuration and processor configuration can be stored in ACPI NVS RAM area, which is available after the memory controller has been enabled and read/write access is enabled. After this is done, the BIOS can call the waking vector.
As mentioned previously, waking up from an S4 state is treated the same as a cold boot: the BIOS runs POST and then initializes memory to contain the required system tables. After it has finished this, it can call the OS loader, and control is passed to the OS.
To wake from S4 using the S4BIOS mechanism, the BIOS runs POST, restores memory context, and calls the waking vector.

9.3.1 Turning On ACPI

When a platform initializes from a cold boot (mechanical off or from an S4 state), the hardware platform is assumed to 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, the OS will then enable the ACPI mode by first scanning memory for the ACPI tables, and then generates a write of the ACPI_ENABLE value to the SMI_CMD port (as described in the FACP table). The hardware platform will set the SCI_EN bit to indicate to the OS that the hardware platform is now configured for ACPI.
When the platform is awakening from an S1, S2 or S3 state, the OS 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:

- 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 wakeup 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, the IA-PC INT15 E820 specification has been updated with two new memory range types:

- **ACPI Reclaim Memory.** Memory identified by the BIOS that contains the ACPI tables. This memory can be any place above 1 MB and contains the ACPI tables. When the OS 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. The OS 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, the OS is not allowed to use this physical memory. The ACPI driver will call the Prepare To Sleep (_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, the OS will restore this memory area and call the wakeup control method (_WAK) to enable the BIOS to reclaim its memory image.

Note: The memory information returned from INT15 E820 should be the same before and after an S4 sleep.

These new memory range types are in addition to the previous E820 memory types of system and reserved. When the OS is first booting, it will make E820 calls to obtain a system memory map. As an example, the following memory map represents a typical IA-PC platform physical memory map.
For more information about the INT15H, E820H definition, see section 14.1.

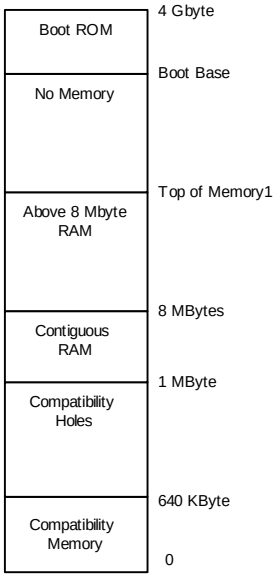


Figure 9-3 Example Physical Memory Map

The names and attributes of the different memory regions are listed below:

- **0 - 640K:** Compatibility Memory. Application executable memory for an 8086 system.
- **640K - 1MB:** 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).
- **1MB - 8MB:** Contiguous RAM. An area of contiguous physical memory addresses. The OS requires 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.)
- **8MB - Top of Memory1:** This area contains memory to the "top of memory1" boundary. In this area, memory-mapped I/O blocks are possible.
- **Top of Memory1- Boot Base:** This area contains the bootstrap ROM.

The platform 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. Note 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.

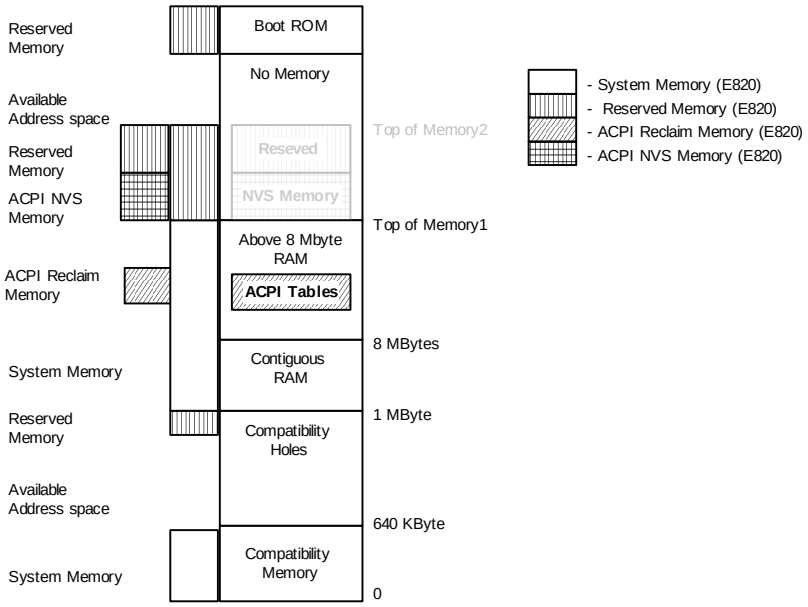


Figure 9-4 Memory as Configured after Boot

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).

The OS 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 the OS 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 up from an S4 state, the OS 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 the OS, either by using the OS boot loader (a result of awakening from an S4/S5 or boot condition) or the OS waking vector (a result of awakening from an S2 or S3 state). For the Boot OS Loader path, the OS will get the system memory map through an INT15H E820h call. If the OS 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, the OS will not restore the system context and can boot from scratch (from the S4 state). Next, for an S4 wakeup, the OS will check the NVS file to see whether it is valid. If valid, then the OS will load the NVS image into system memory. Next, the OS will ask BIOS to switch into ACPI mode and will reload the memory image from the NVS file.

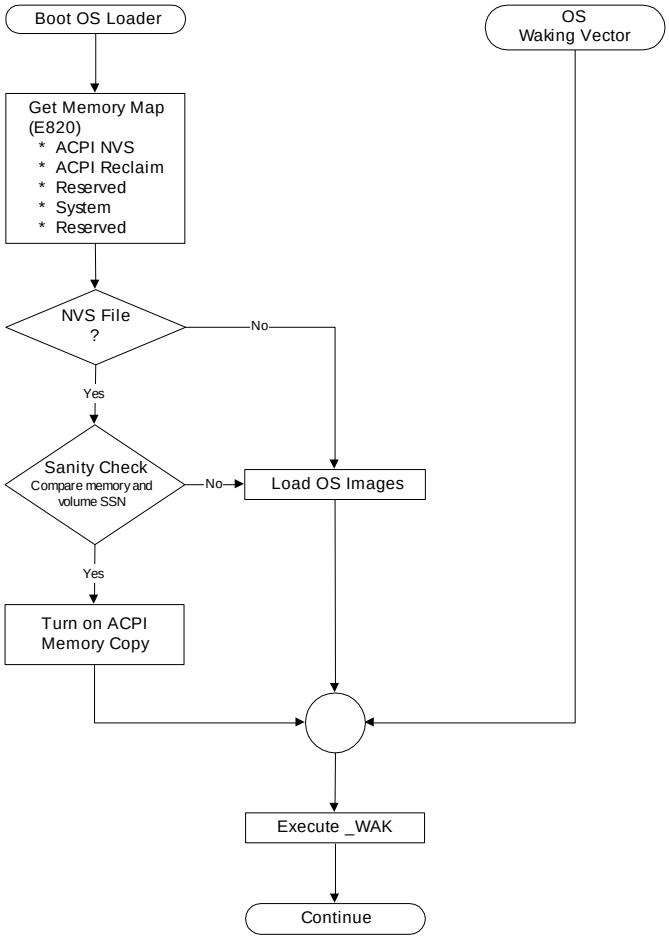


Figure 9-5 OS Initialization

If an NVS image file did not exist, then the OS loader will load the OS from scratch. At this point, the OS 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 Turning Off ACPI

ACPI provides a mechanism that enables the operating system to disable ACPI. The following occurs:

1. The OS unloads all ACPI drivers (including the APIC driver).
2. The OS disables all ACPI events.
3. The OS finishes using all ACPI registers.
4. The OS issues an I/O access to the port at the address contained in the SMI_CMD field (in the FACP table) with the value contained in the ACPI_DISABLE field (in the FACP table).
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 operating system passes control to the legacy mode.

When and if the legacy operating system returns control to the ACPI OS, if the legacy OS has wiped out the ACPI tables (in reserved memory and ACPI NVS memory), then 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, which go in the _SI region of the Name Space, 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 lid devices (for more information about control method lid devices, see section 10.3.
- Control method power and sleep button devices (for more information about control method power and sleep button devices, see section 4.7.2.2.
- Embedded controller devices (for more information about embedded controller devices, see section 13).
- System Management Bus (SMBus) host controller (for more information, see section 13.9.)
- Fan devices (for more information about fan devices, see section 12).
- 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.

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 name space. The following table lists all defined system indicators. (Note 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

Operating software invokes this control method to set the system status indicator as desired.

Arguments:

0 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

This control method sets the systems message waiting status indicator.

Arguments:

0 Number of messages waiting.

10.2 Control Method Battery Device

A battery device is required to either have a ACPI Smart Battery Table or a Control Method Battery (CMBatt) 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 SMB bus, which in turn will locate the battery selector, and charger SMB devices.

The Control Method Battery interface is defined in section 11.2.

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, 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

10.3.1.1.1.1 _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, 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 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 the OS has actually awakened, and a button pressed notification (**Notify**(*Object*, 0x80)) must not occur.

The Wake Notification indicates that the system has awakened 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.

The embedded controller device object provides the _HID (Hardware ID) 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).

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 for primarily cooling within that zone. Note 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.

10.7 Generic Bus Bridge Device

A generic bus bridge device is a bridge that does not require a special OS driver because the bridge does not provide/require any features not described within the standard ACPI device functions. The resources the bridge supports are supported through the standard ACPI resource handling. All device enumeration for child devices is supported through standard ACPI device enumeration (for example, name space), and no other features of the bus are needed by OS drivers. 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-IO devices. The bridged resources are likely to be positively decoded as either a function of the bridge or the integrated devices. In either case, for this example, a generic bus bridge device would be used to declare the bridge, then further devices would be declared below the bridge for the integrated Super-IO devices.

10.8 IDE Controller Device

A method is supported to allow the OEM to define how the IDE controller's transfer mode register is set. The operating system's IDE device driver is responsible for providing the transfer ns timings to set, and to use either a device-specific register to make the IDE controller mode setting[17] or the device's ACPI control method to effect the IDE transfer timing settings. The OS native driver is responsible for setting this whenever the IDE controller has been in the D3 state.

Table 10-3 IDE Specific Controls

Object	Description
_STM	Optional control method to use to set the IDE controller transfer timings.

10.8.1 _STM

This **Control Method** sets the IDE controller's transfer timings to the setting requested. The AML code is required to convert and set the ns timing to the appropriate transfer mode settings for the IDE controller.

Arguments:

0 The timing for drive 0 on the channel

1 The timing for drive 1 on the channel

Result code:

None

[17] This is the prefer way to set the IDE controller mode; however, not all controllers will fit into this model.

11. Power Source Devices

This section specifies the battery and AC adapter device objects the OS uses to manage power resources.
A battery device is required to either have a Smart Battery subsystem or a **Control Method Battery (CMBatt)** interface as described in this section. The OS 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 host controller. This will install an OS-specific driver for the SMBus, which in turn will locate the battery and battery selector SMB devices.

11.1 Smart Battery Subsystems

Smart Batteries are defined as using the smart battery subsystem as defined by the:

- System Management Bus Specification (SMBS),
- Smart Battery Data Specification (SBDS),
- Smart Battery Selector Specification (SBSS), and the
- Smart Battery Charger Specification (SBCS)

An ACPI compatible smart battery subsystem consists of:

- An SMBus host controller (CPU to SMB host controller) interface
- At least one smart battery
- A smart battery charger
- 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 (SBDS) and smart charger (SBCS) is through the SMBus (SMBS) physical protocols. The smart battery selector provides event notification (battery insertion/removal, …) and charger SMBus routing capability for any smart battery subsystem. A typical smart battery subsystem is illustrated below:

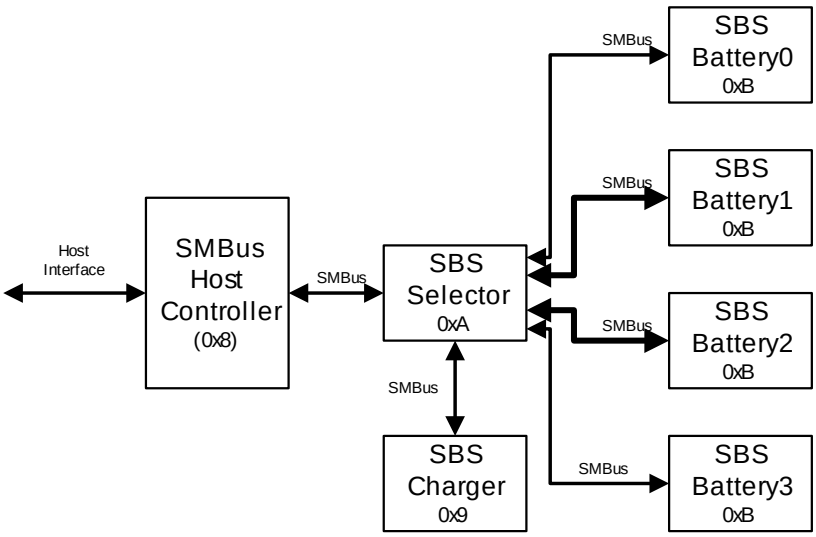


Figure 11-1 Typical Smart Battery Subsystem

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
SBS Charger/Charger Selector	0x9

SMBus Device Description	SMBus Slave Address (A0-A6)
SBS Selector	0xA
SBS 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 1 can have a byte, and command register 2 can have a word.

The SMBus host slave interface provides a standard mechanism for the host CPU to generate SMBus protocol commands which are required to communicate with SMBus devices (i.e., the smart battery components). ACPI defines such an SMBus host controller that resides in embedded controller address space, however an OS can support any SMBus host controller which has a native SMBus host controller device driver.

The SBS selector provides a standard programming model to control multiple smart batteries in a smart battery subsystem. A smart battery selector provides the following types of battery management functions:

- Event notification for battery insertion removal
- Event notification for AC power connected or disconnected
- Status/Control of which battery is communicating with the SMBus host controller
- Status/Control of which battery is powering the system
- Status/Control of which battery is connected to the charger
- Status of which batteries are present in the system
- Event notification when the selector switches from one power source to another
- Hardware switching to a secondary battery upon the primary battery running low
- Hardware switching to AC

A smart battery selector function can reside in a standalone SMBus slave device (SBS Selector which responds to the 0xA slave address), or may be present within a smart charger device (SBS Charger which responds to the 0x9 slave address). If both smart charger and stand alone selectors are present in the same smart battery subsystem, then the driver assumes that the stand alone selector is wired to the batteries.

The SBS 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 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 battery is capable of informing the smart charger 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 Charger Requirements

The smart battery charger specification 1.0 defines an optional mechanism for notifying the system that the battery or AC status has changed. ACPI requires that this interrupt mechanism be through the SMBus Alarm Notify mechanism.

For a charger only device this requires the smart charger, upon a battery or AC status change, to generate an SMBus Alarm Notify. This generates an event from the SMBus host controller after the contents of the ChargerStatus() command register (0x13) are placed in the SMBus host slave data port and the slave address of the messaging device (in this case, the charger [18]) is placed in the SMBus host slave command port (at slave address 0x8).

If a smart battery charger contains the optional selector function (as indicated by ChargerSpecInfo() command register, 0x11, bit 4), this requires the smart charger, upon a battery or AC status change, to generate an SMBus Alarm Notify. This generates an event from the SMBus host controller after the contents of the SelectorState() command register (0x21) are placed in the SMBus host slave data port and the slave address of the messaging device (in this case, the charger[18]) is placed in the SMBus host slave command port (at slave address 0x8).

[18] Note 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.

When the selector function is present in the smart charger, Battery and AC status changes should be reported through the SelectorState() notify and not the ChargerStatus() notify.

11.1.2 ACPI Smart Battery Selector Requirements

The smart battery selector specification 1.0 defines an optional mechanism for notifying the system that the battery or AC status has changed. ACPI requires that this interrupt mechanism be through the SMBus Alarm Notify mechanism.

For a smart battery selector device this requires the smart battery selector, upon a battery or AC status change, to generate an SMBus Alarm Notify. This generates an event from the SMBus host controller after the contents of the SelectorState() command register (0x1) are placed in the SMBus host slave data port and the slave address of the messaging device (in this case, the selector[18]) is placed in the SMBus host slave command port (at slave address 0x8).

11.1.3 Smart Battery Objects

The smart battery subsystems 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 which 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 which 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 selector. 1: Maximum of one smart battery and a selector. 2: Maximum of two smart batteries and a selector. 3: Maximum of three smart batteries and a selector. 4: Maximum of four smart batteries and a 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. Note that a value of 0 indicates a maximum support of one battery and there is no selector present in the system.

11.1.4 Smart Battery Subsystem Control Methods

As the SMBus is not an enumerable bus, all devices on the bus are required to be declared in ACPI name space. As the smart battery driver understands the SBS battery, charger, and 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 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 an SBS selector is present. If 0, then the smart battery driver assumes a single smart battery and no SBS selector.

11.1.4.1 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:

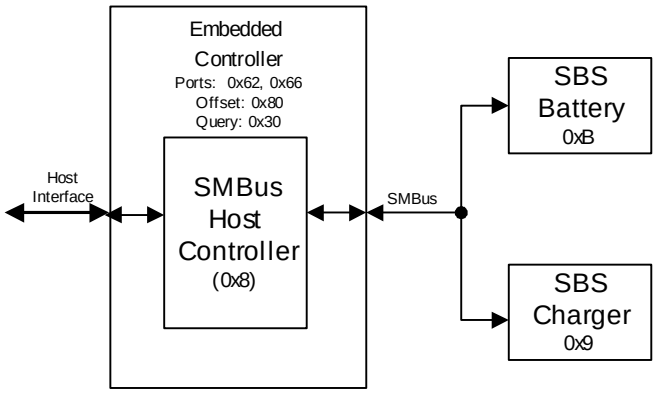


Figure 11-2 Single Smart Battery Subsystem

In this example the platform is using an SMBus host controller that resides within the embedded controller and meets the ACPI standard for an embedded controller interface and SMBus host controller interface. The embedded controller interface sits at system I/O port addresses 0x62 and 0x66. The SMBus host controller is 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 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
```

11.1.4.2 Example: Multiple Smart Battery Subsystem

This section illustrates how to define a smart battery subsystem that contains three smart batteries, a SBS selector and a charger. The platform implementation is illustrated below:

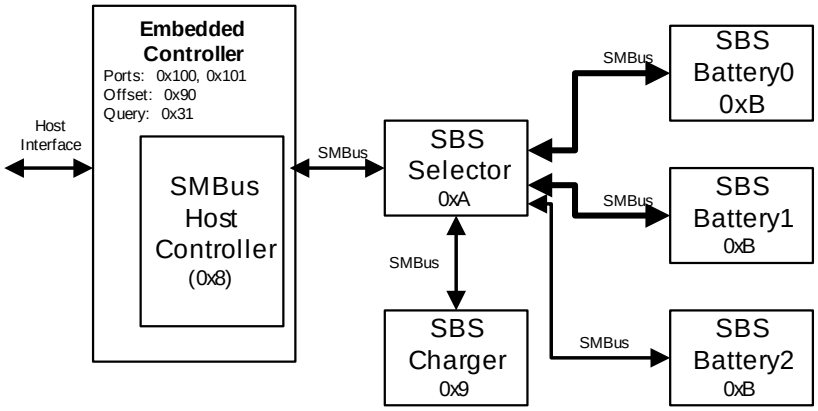


Figure 11-3 Smart Battery Subsystem

In this example, the platform is using an SMBus host controller that resides within the embedded controller and meets the ACPI standard for an embedded controller interface and SMBus host controller interface. The embedded controller interface sits at system I/O port addresses 0x100 and 0x101. The SMBus host controller resides 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, an SBS charger and an SBS selector. 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 which battery's the 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). 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).
As with other devices, when a battery device is inserted to the system or removed from the system, the hardware asserts a GP event. The AML code handler for this event will issue a Notify(*battery_device* , 0x00) or Notify(*battery_device* ,0x01) on the battery device to initiate the standard device Plug and Play actions.
When the present state of the battery has changed or when the trip point set by the _BTP control method is crossed, the hardware will assert a GP event. The AML code handler for this event issues a Notify(*battery_device*,0x80) on the battery device.
In the unlikely case that the battery becomes critical, AML code interface can issue Notify(*battery_device*, 0x80) and reports the battery critical flag in the _BST object. The OS performs critical shutdown.

11.2.2 Battery Control Methods

The Control Method Battery (CMBatt) 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.
A Control Method Battery is described as a device object. Each device object supporting the CMBatt 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 (i.e., model number, serial number, design voltage, etc.)
_BST	Returns the current battery status (i.e., dynamic information about the battery such as whether the battery is currently charging or discharging, an estimate of the remaining battery capacity, etc.).
_BTP	Sets the Battery Trip point which generates an SCI when the battery(s) capacity 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 6.3.5.

11.2.2.1 _BIF

This object returns the static portion of the Control Method Battery information. This information remains constant until the battery is changed.

Arguments:

None

Results 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

Field	Format	Description
Battery Technology	DWORD	**0x00000000** = Primary (ex., non-rechargeable) **0x00000001** = Secondary (ex., 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. **0x000000000 - 0x7FFFFFFF** in [mWh] or [mAh]
Design capacity of Low	DWORD	OEM-designed low battery capacity. **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 it to display the OEM information about the Battery. If the OEM does not support this information, this should be reserved as 0x00.

Note: A secondary-type battery should report the corresponding capacity (except for Unknown).
Note: On a multiple battery system, all batteries in the system should return the same granularity.
Note: OSes prefer these control methods to report data in terms of power (watts).

11.2.2.2 _BST

This object that returns the present battery status. Whenever the *Battery State* value changes, the system will generate an SCI to notify the OS.

Arguments:

None

Results 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. Note: The *Charging* bit and the *Discharging* bit are mutually exclusive and must not both be set at the same time. 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 Even in critical state, hardware should report the corresponding charging/discharging state. When the battery reports critical energy state and also reports the battery is discharging (bits 0 and 2 are both set) the OS will perform a critical system shutdown.

Field	Format	Description
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
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	Battery Present Voltage 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 is a Primary battery can report Unknown voltage).

11.2.2.3 _BTP

This object is used to set a trip-point to generate an SCI when the *Battery Remaining Capacity* reaches the value specified in the _BTP object. This information 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
Results 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

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
Results code:

0x00000000 = Off-line
0x00000001 = On-line

11.3.2 _PCL

This object evaluates to a list of pointers, each pointing to a device or a bus powered by the power source device. Pointing a bus means that all devices under the bus is powered by it power source device.

11.4 Power Source Name Space Example

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 the illustration (Figure 11.4) below.

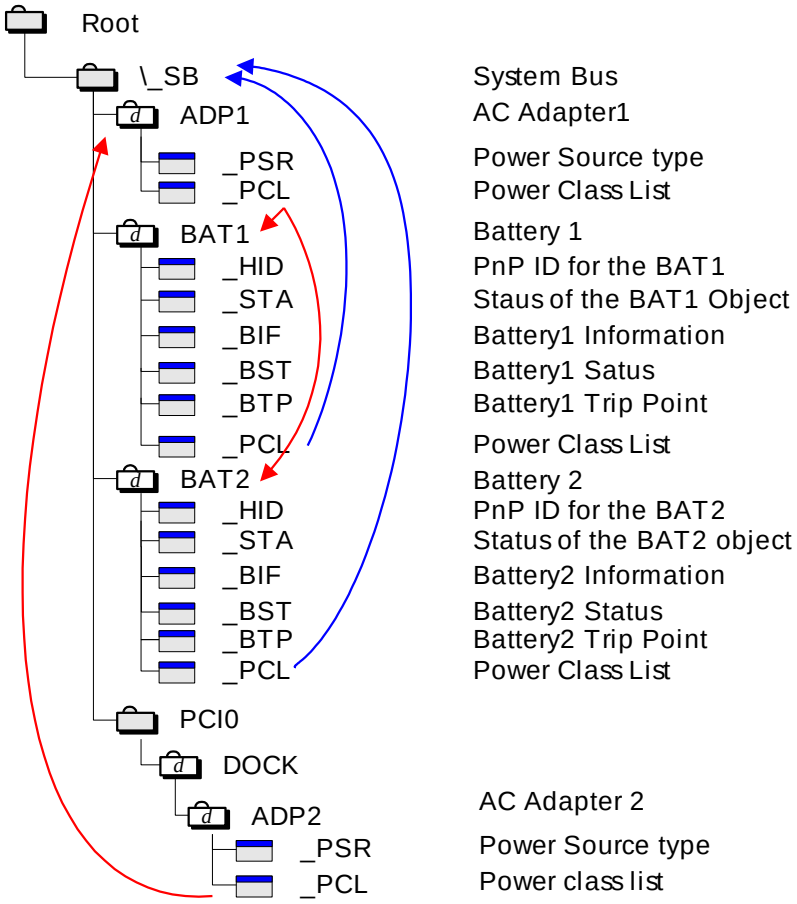


Figure 11-4 Power Source Name Space Example that Includes a Docking Station

12. Thermal Management

This section specifies the objects the OS uses for thermal management of a platform.

12.1 Thermal Control

ACPI allows the OS to be proactive in its system cooling policies. With the OS in control of the operating environment, cooling decisions can be made based on 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 the OS to control platform temperature. ACPI expects all temperatures to be given in tenths of 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 primary cooling policies that the OS uses to control the thermal state of the hardware. The policies are *Active, Passive* and *Critical:*

- **Passive cooling:** The OS reduces the power consumption of the system to reduce the thermal output of the machine by slowing the processor clock. The _PSV control method is used to declare the temperature to start passive cooling.
- **Active cooling:** The OS takes a direct action such as turning on a fan. The _AC*x* control methods declare the temperatures to start different active cooling levels.
- **Critical trip point:** This is the threshold temperature at which the OS performs an orderly, but critical, shut down of the system. The _CRT object declares the critical temperature at which the OS must perform a critical shutdown.

When a thermal zone appears, the OS runs control methods to retrieve the three temperature points at which it executes the cooling policy. When the OS receives a thermal SCI it will run the _TMP control method, which returns the current temperature of the thermal zone. The OS checks the current temperature against the thermal event temperatures. If _TMP is greater than or equal to _AC*x* then the OS will turn on the associated active cooling device(s). If _TMP is greater than or equal to _PSV then the OS will perform CPU throttling. Finally if _TMP is greater than or equal to _CRT then the OS will shutdown the system.

An optimally designed system that uses several SCI events can notify the OS of thermal increase or decrease by raising an interrupt every several degrees. This enables the OS to anticipate _AC*x*, PSV, or _CRT events and incorporate heuristics to better manage the systems temperature.

The operating system can request that the hardware change the priority of active cooling vs passive cooling.

12.1.2 Dynamically Changing Cooling Temperatures

An OEM can reset _ACx and _PSV and notify the OS to reevaluate the control methods to retrieve the new temperature settings. The following three causes are the primary uses for this thermal notification:

- When a user changes 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 the _PSV policy settings

In each situation, the OEM-provided AML code must execute a **Notify**(*thermal_zone*, 0x80) statement to request the OS to re-evaluate each policy temperature by running the _PSV and _AC*x* control methods.

12.1.2.1 Resetting Cooling Temperatures from the User Interface

When the user employs the UI to change from one cooling mode to the other, the following occurs:

1. The OS notifies the hardware of the new cooling mode by running the Set Cooling Policy (_SCP) control method.
2. When the hardware receives the notification, it can set a new temperature for both cooling policies and notify the OS that the thermal zone policy temperatures have changed.

3. The OS re-evaluates _PSV and _ACx.

12.1.2.2 Resetting Cooling Temperatures to Adjust to Bay Device Insertion or Removal

The hardware 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 and resets the _PSV and/or _AC*x* and then notifies the OS of the thermal and device insertion events.
2. The OS reenumerates the devices and reevaluates _PSV and _ACx.

12.1.2.3 Resetting Cooling Temperatures to Implement Hysteresis

An OEM can build hysteresis into platform thermal design by dynamically resetting cooling temperatures. For example,

1. When the heat increases to the temperature designated by _AC*x*, the OS will turn on the associated active cooling device and the hardware will reset the AC*x* value to a lower temperature.
2. The hardware will then run the Notify command and the OS will reevaluate the new temperatures. Because of the lower _AC*x* value now, the fan will be turned off at a lower temperature than when turned on.
3. When the temperature hits the lower _AC*x* value, the OS will turn off the fan and reevaluate the control methods when notified.

12.1.3 Hardware Thermal Events

An ACPI-compatible OS expects the hardware to generate a thermal event notification through the use of the SCI. When the OS receives the SCI event, it will run the _TMP control method to evaluate the current temperature. Then the OS will compare the value to the cooling policy temperatures. If the temperature has crossed over one of the three policy thresholds, then the OS will actively or passively cool (or stop cooling) the system, or shutdown the system entirely.

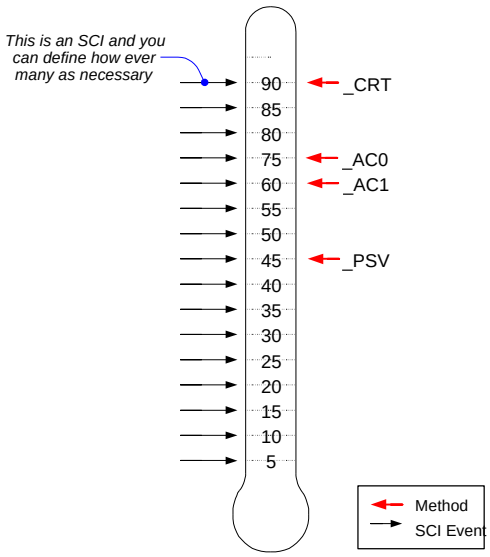


Figure 12-1 SCI Events

Both the number of SCI events to be implemented and the granularity of the temperature separation between each SCI event is OEM-specific. However, it is important to note that since the OS can use heuristic knowledge to help cool the system, the more events the OS receives the better understanding it will have of the system's thermal characteristic.

12.1.4 Active Cooling Strength

The Active cooling methods (_Ac*x*) in conjunction with active cooling lists (_AL*x*), allows an OEM to use a device that offers varying degrees of cooling capability or multiple cooling devices. The _AC*x* method

designates the temperature at which the Active cooling is enabled or disabled (depending upon the direction in which the temperature is changing). The _AL*x* method 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 the policy is represented by the temperature to which _AC0 evaluates, 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 section 12.4.

12.1.5 Passive Cooling Equation

Unlike the case for _AC*x*, during passive cooling the OS 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.

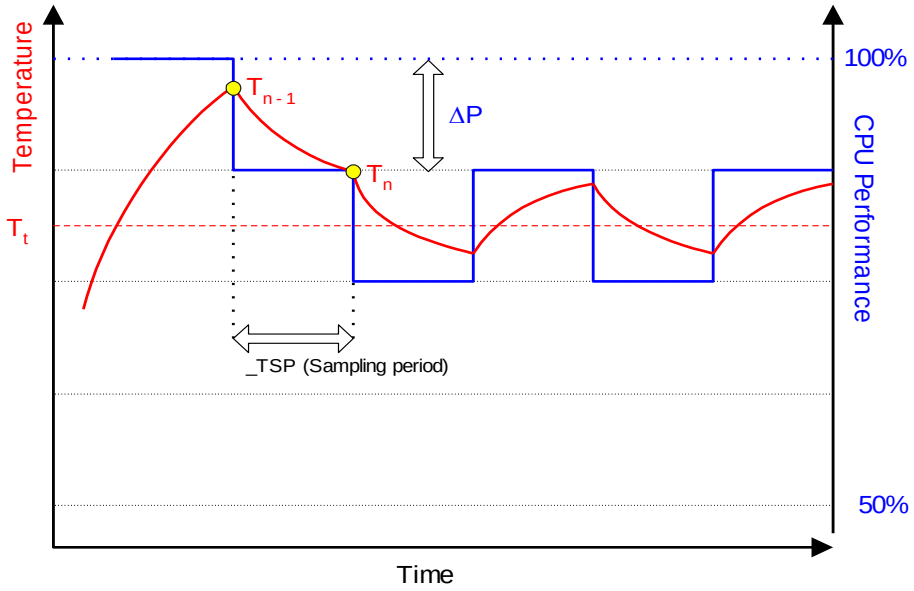


Figure 12-2 Temperature and CPU Performance Versus Time

For the OS to assess the optimum CPU performance change required to bring the temperature down, the following equation must be incorporated into the OS.

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 the OS (for more information, see section 12.3). The object _TSP contains a time interval that the OS uses to poll the hardware to sample the temperature. Whenever _TSP time has elapsed, the OS will run _TMP to sample the current temperature (shown as Tn in the above equation). Then the OS 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. (For more information about CPU throttling, see section 4.7.2.6). A detailed explanation of this thermal feedback equation is beyond the scope of this specification.

12.1.6 Critical Shutdown

When the heat reaches the temperature indicated by _CRT, the OS must immediately shutdown the system. 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 the OS 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

- Heat might rise rapidly in some systems and slower 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 Other Implementation Of Thermal Controllable Devices

The ACPI thermal event model is flexible enough to accommodate control of almost any system device capable of controlling heat. For example, if a mobile PC requires the battery charger to reduce the charging rate in order to reduce heat it can be seamlessly implemented as an ACPI cooling device. This is done by associating the charger as an Active cooling device and reporting to the OS target temperatures that will enable or disable the power resource to the device. Figure 12-3 illustrates the implementation. Because the example does not create noise, this will be an implementation of *silence* mode.

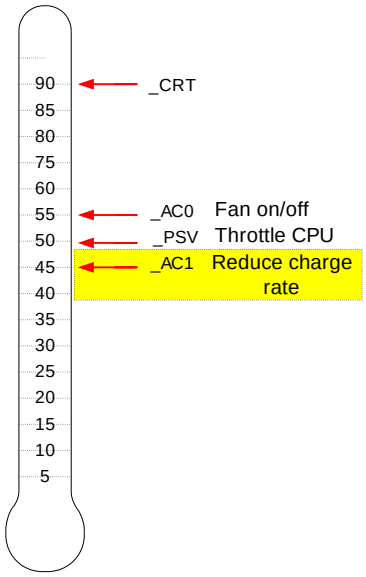


Figure 12-3 Other Thermal Control

12.3 Thermal Control Methods

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 trip point in tenths Kelvin
_AL*x*	List of pointers to active cooling device objects
_CRT	Returns critical trip point in tenths Kelvin
_PSL	List of pointers to passive cooling device objects

Object	Description
_PSV	Returns Passive trip point in tenths Kelvin
_SCP	Sets user cooling policy (Active or Passive)
_TC1	Thermal constant for Passive cooling
_TC2	Thermal constant for Passive cooling
_TMP	Returns current temperature in tenths Kelvin
_TSP	Thermal sampling period for Passive cooling in tenths of seconds

12.3.1 _AC*x*

This control method returns the temperature at which the OS 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* method.

Arguments:

None.

Result Code:

Temperature in tenths Kelvin.

The result code is an integer value which describes up to 0.1 precision in Kelvin. For example, 300.0K is represented by the integer 3000.

12.3.2 _AL*x*

This object evaluates to a list of Active cooling devices to be turned on when the associated _ACx trip point is exceeded. For example, these devices could be fans.

12.3.3 _CRT

This control method returns the critical temperature at which the OS must shutdown the system.

Arguments:

None.

Result Code:

Temperature in tenths Kelvin.

The result is an integer value that describes up to 0.1 precision in Kelvin. For example, 300.0K is represented by the integer 3000.

12.3.4 _PSL

This object evaluates to a list of processor objects to be used for Passive cooling.

12.3.5 _PSV

This control method returns the temperature at which the OS must activate CPU throttling.

Arguments:

None.

Result Code:

Temperature in tenths Kelvin.

The result code is an integer value that describes up to 0.1 precision in Kelvin. For example, 300.0 Kelvin is represented by 3000.

12.3.6 _SCP

This control method notifies the hardware of the current user cooling mode setting. The hardware can use this as a trigger to reassign _AC*x* and _PSV temperatures. The operating system will automatically evaluate _ACx and _PSV objects after executing _SCP.

Arguments:

0 - Active

1 - Passive

Result Code:

None.

12.3.7 _TC1

This is a thermal object that evaluates to the constant _ TC1 for use in the Passive cooling formula:

ΔPerformance [%]= _TC2 * (Tn - Tn-1) + _TC1 * (Tn. - Tt)

12.3.8 _TC2

This is a thermal object that evaluates to the constant _TC2 for use in the Passive cooling formula:

ΔPerformance [%]= _TC2 * (Tn - Tn-1) + _TC1 *.(Tn. - Tt)

12.3.9 _TMP

This control method returns the thermal zone's current operating temperature in Kelvin.

Argument:

None.

Result Code:

Temperature in tenths Kelvin.

The result is an integer value that describes up to 0.1 precision in Kelvin. For example, 300.0K is represented by the integer 3000.

12.3.10 _TSP

This is an object that evaluates to a thermal sampling period used by the OS to implement the Passive cooling equation. This value, along with _TC1 and _TC2, will enable the OS to provide the proper hysteresis required by the system to accomplish an effective passive cooling policy. The granularity of the sampling period is 0.1second. 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. The OS can normalize the sampling over a longer period if necessary.

12.4 Thermal Block and Name Space Example for One Thermal Zone

Following is an example ASL encoding of a thermal zone. This is an example only.

```
Scope(\_PR) {
        Processor(
                CPU0,
                1,      //unique number for this processor
                0x110,  //System IO address of Pblk Registers
                0x06    //length in bytes of PBlk
                ) {}
} //end of \_PR scope

Scope(\_SB) {
    Device(EC0) {
        Name(_HID, EISAID("PNP0C09"))     // ID for this EC
        // current resource description for this EC
        Name(_CRS, Buffer (){ 0x4B, 0x62, 0x00, 0x01, 0x4B,
                              0x66, 0x00, 0x01, 0x79, 0x00})
        Name(_GPE, 0)  // GPE index for this EC

        // create EC's region and field for thermal  support
        OperationRegion(EC0, EmbeddedControl, 0, 0xFF)
        Field(EC0, AnyAcc, Lock, Preserve) {
            MODE,   1,      // thermal policy (quiet/perform)
            FAN,    1,      // fan power (on/off)
            ,       5,
            AC0,    8,      // active cooling temp (fan high)
            PSV,    8,      // passive cooling temp
            CRT,    8,      // critical temp
            }

        // following is a method that the OS will schedule after
        // it receives an SCI and queries the EC to receive value 7
        Method(_Q07) {
            Notify (\_TZ.THRM, 0x80)
        }   // end of Notify method
    }   // end of ECO device
}   // end of scope

Scope(\_TZ) {
    PowerResource(PFAN, 0, 0) {
        Method(_STA) { Return (EC0.FAN) }        // check power state
        Method(_ON) { Store (One, EC0.FAN) }     // turn on fan
        Method(_OFF) { Store ( Zero, 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 (THRM) {
        Method(_TMP) { Return (EC0.TMP )}            // get current temp
        Method(_AC0) { Return ( EC0.AC0) }           // fan high temp
        Name(_AL0, Package(){FAN})                   // fan is act cool dev
        Method(_PSV) { Return ( EC0.PSV) }           // passive cooling temp
        Name(_PSL, Package (){\_PR.CPU0})            // cpu is pass cool dev
        Method(_CRT) { Return ( EC0.CRT) }           // get critical temp
        Method(_SCP, 1) { Store (Arg1, EC0.MODE) }   // set cooling mode
        Name(_TC1, 4)  // bogus example constant
        Name(_TC2, 3)  // bogus example constant
        Name(_TSP, 60)                               // sample every 60 sec
    }
}
```

12.5 Controlling Multiple Fans in a Thermal Zone

The following is an example encoding of a thermal block with a thermal zone and a single fan that has two cooling speeds. This is an example only.

```
Scope(\_PR) {
    Processor(
                CPU0,
                1,          //unique number for this processor
                0x110,      //System IO address of Pblk Registers
                0x06        //length in bytes of PBlk
                ) {}
} //end of \_PR scope

Scope(\_SB) {
    Device(EC0) {
        Name(_HID, EISAID("PNP0C09"))     // ID for this EC
        // current resource description for this EC
        Name(_CRS, Buffer (){  0x4B, 0x62, 0x00, 0x01, 0x4B,
                               0x66, 0x00, 0x01, 0x79, 0x00})
        Name(_GPE, 0)                     // GPE index for this EC

        // create EC's region and field for thermal support
        OperationRegion(EC0, EmbeddedControl, 0, 0xFF)

        // following is a method that the OS will schedule after it
        // receives an SCI and queries the EC to receive value 7
        Method(_Q07) {
            Notify (\_TZ.THM1, 0x80)
        }
    }
}

Scope(\_TZ) {
    // fan cooling mode high/off - engaged at AC0 temp
    PowerResource(FN10, 0, 0) {
        Method(_STA) { Return (THM1.FAN0) }      // check power state
        Method(_ON) { Store (One, THM1.FAN0) }   // turn on fan at high
        Method(_OFF) { Store (Zero, THM1.FAN0) } // turn off fan
    }

    // fan cooling mode low/off   - engaged at AC1 temp
    PowerResource(FN11, 0, 0) {
        Method(_STA) { Return (THM1.FAN1) }      // check power state
        Method(_ON) { Store (One, THM1.FAN1) }   // turn on fan at low
        Method(_OFF) { Store (Zero, THM1.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 on high.
    //
    // Create FAN device object FN1
    Device (FN1) {
        // Device ID for the FAN
        Name(_HID, EISAID("PNP0C0B"))
        Name(_PR0, Package(){FN10, FN11})
    }

    // Create FAN device object FN2
    Device (FN2) {
        // Device ID for the FAN
        Name(_HID, EISAID("PNP0C0B"))
        Name(_PR0, Package(){FN10})
    }

    // create a thermal zone
    ThermalZone (THM1) {
        // field used by this thermal zone
        Field(\EC0, AnyAcc, 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
            CRT,    8,      // critical temp
        }
```

```
Method(_TMP) { Return (TMP )}                    // get current temp
       Method(_AC0) { Return (AC0) }             // fan high temp
       Method(_AC1) { Return (AC1) }             // fan low temp
       Name(_AL0, Package() {FN1, FN23})         // active cooling (high)
       Name(_AL1, Package() {FN2})               // active cooling (low)
       Method(_PSV) { Return (PSV) }             // passive cooling temp
       Name(_PSL, Package() {\_PR.CPU0})         // cpu is pass cool dev
       Method(_CRT) { Return (CRT) }             // get crit. temp
       Method(_SCP, 1) { Store (Arg1, MODE) }    // set cooling mode
       Name(_TC1, 1)  // bogus example constant
       Name(_TC2, 2)  // bogus example constant
       Name(_TSP, 150)                           // sample every 15 seconds
    // END: declare objects for thermal zone
    }
} // end of TZ
```

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 and applications can take advantage of.

ACPI also defines a standard hardware and software communications interface between an OS driver and an SMBus Host Controller via an Embedded Controller.

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.

The specification supports optional extensions to the interface that allow the driver to communicate to an SMBus controller within the embedded controller (actual or emulated). This will allow standard drivers to be created for SMBus devices that appear on the SMBus whether they are actual or emulated.

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 5.2.6.1 of the ACPI specification. 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.

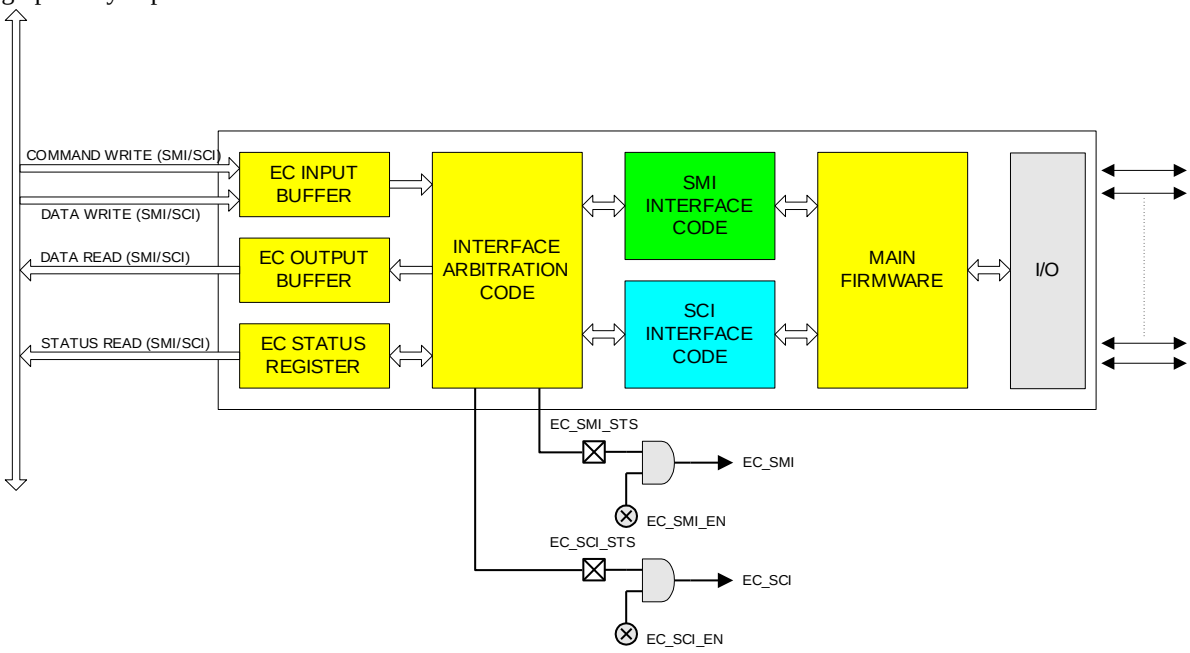


Figure 13-1 Shared Interface

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 the OS 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 the OS driver. The following diagram illustrates the embedded controller architecture that includes a dedicated ACPI interface.

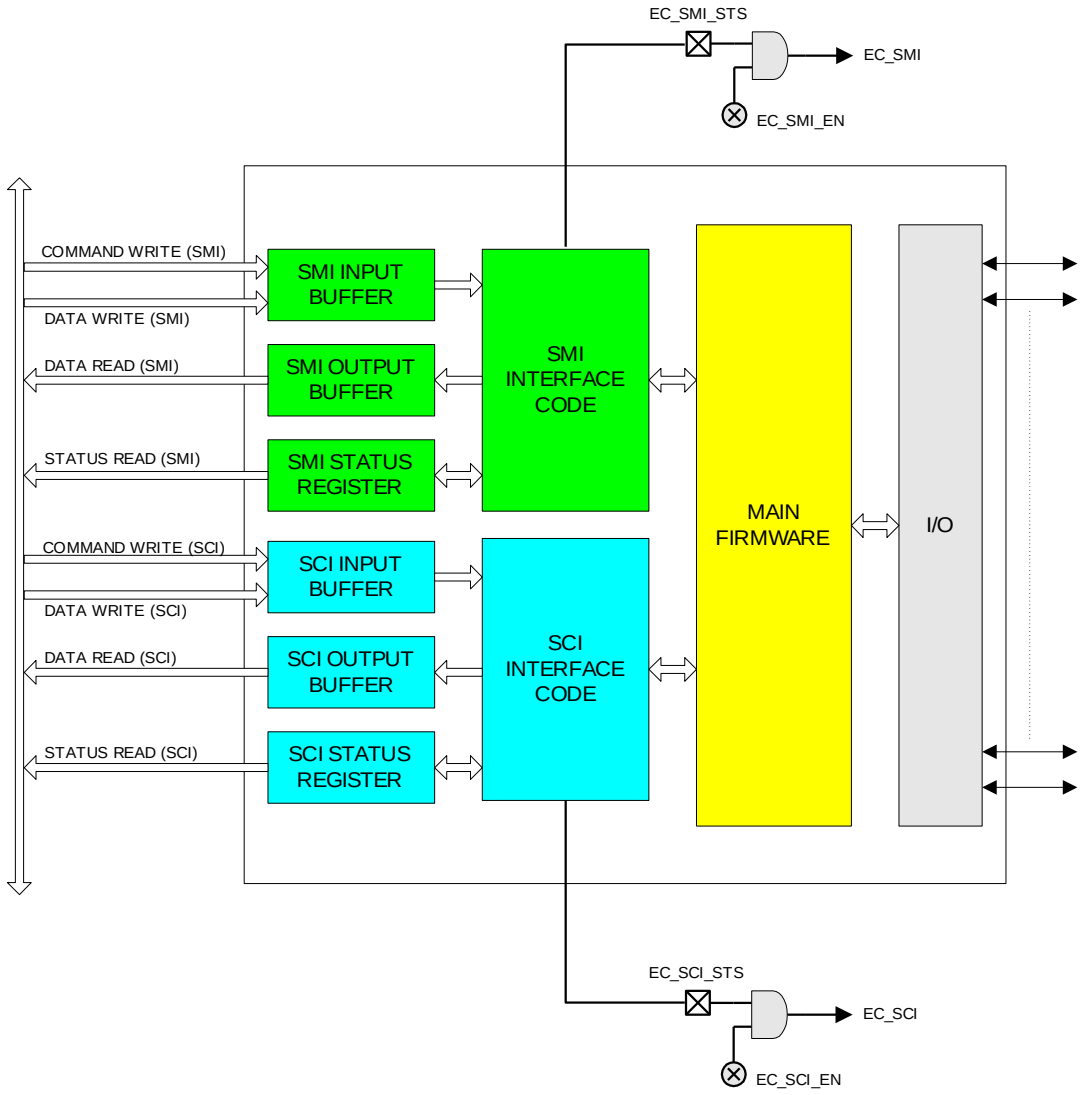


Figure 13-2 Private Interface

The private interface allows the OS 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. The OS 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.
- Two 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 5.2.6.1.

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 data could be received by the incorrect recipient. 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 the OS 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 the OS. The OS needs this bit to differentiate command-complete SCIs from notification SCIs. The OS uses the query command to request the cause of the SCI_EVT and take action. For more information, see section 13.3)

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 the OS 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 the OS 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 the OS 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

Embedded Controller Command	Command Byte Encoding
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 the OS to read a byte in the address space of the embedded controller. This command byte is reserved for exclusive use by the OS, 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 the OS to write a byte in the address space of the embedded controller. This command byte is reserved for exclusive use by the OS, 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 the OS 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 the OS 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 the OS). 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 the OS to indicate the change.

While in burst mode, the embedded controller follows these guidelines for the OS 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:

1. The OS 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 the OS interface for ~ 1 ms.
2. 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 the OS that it is in Burst mode.

Burst mode is exited the following manner:

1. The OS 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).

2. 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 the OS 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)

The OS 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 the OS, it first sets the SCI_EVT flag in the EC_SC register, generates an SCI, and then waits for the OS to send the query (QR_EC) command. The OS 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 the OS 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 SMBus host controller device object in the ACPI name space.

13.5 Embedded Controller Firmware

The embedded controller firmware must obey the following rules in order to be ACPI-compatible:

1. **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 an 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.
2. **SCI/SMI Task Queuing:** If the system design is sharing the interface between both a system management interrupt handler and the OS, 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 operating system 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 operating system or system management handler issues a query command to the embedded controller to request the cause of the notification event.
3. **Notification Management:** The use of the embedded controller means using the query (QR_EC) command to notify the OS 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 the OS 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 the OS driver treats this as an edge event (the EC SCI GPE cannot be shared).

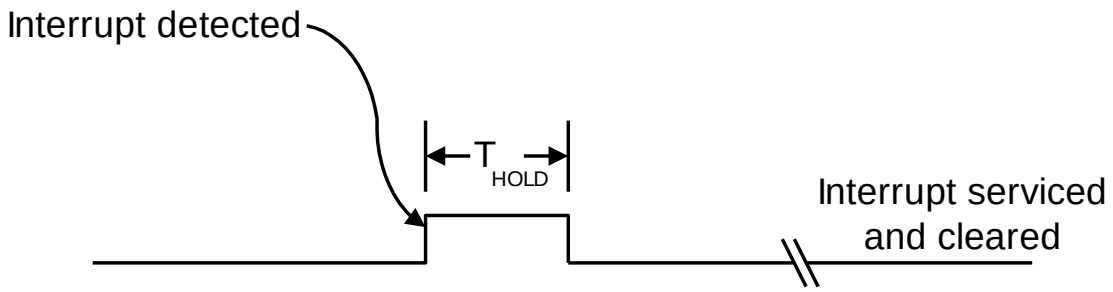


Figure 13-3 EC Interrupt Waveform

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. The OS 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 (Three 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 (Three 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 (Two Bytes)

Byte #1	(Command byte Header)	No Interrupt
Byte #2	(Query value to host)	Interrupt on OBF=1

- BURST ENABLE COMMAND (Two Bytes)

Byte #1	(Command byte Header)	No Interrupt

Byte #2 (Burst acknowledge byte) Interrupt on OBF=1

- BURST DISABLE COMMAND (One Byte)
  Byte #1 (Command byte Header) Interrupt on IBF=0

13.7 Embedded Controller Interfacing Algorithms

To initiate communications with the embedded controller, the OS or system management handler acquires ownership of the interface. This ownership is acquired through the use of the Global Lock (described in section 5.2.6.1), or is owned by default by the OS 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 SMBus controller.

For implementation details of the above listed information, see sections 13.11 and 13.12.

13.9 SMBus Host Controller Interface via Embedded Controller

This section describes the System Management Bus (referred to as SMBus) Host Interface, which is a mechanism to allow the OS to address components on the SMBus. SMBus address space is one of the generic address spaces defined in the ACPI specification, and this section specifies how to implement a host controller interface in order to have the OS communicate directly with SMBus devices.

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 Intel Corporation.

The SMBus host interface provides a method of communicating on the SMBus through a block of registers that reside in embedded controller space. Some SMBus host controller interfaces have special requirements that certain SMBus commands are filtered by the host controller. For example, to prevent an errant application or virus from potentially damaging the battery subsystem. This is most easily accomplished by providing the host interface controller through an embedded controller, because the embedded controller can easily filter out the potentially problematic commands.

The SMBus host controller interface allows the host processor (under control of the OS) to manage devices on the SMBus. Among typical devices that reside on the SMBus are smart batteries, smart chargers, contrast/backlight control, and temperature sensors.

This section specifies a standard set of registers an ACPI-compatible OS can use to communicate with SMBus devices. Any SMBus host interface that does not comply with this standard can be communicated with using control methods (as described in section 5).

13.9.1 Register Description

The SMBus host interface is a flat array of registers that are arranged sequentially in address space.

13.9.1.1 Status Register, SMB_STS

This register indicates general status on the SMBus. This includes SMBus host controller 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 SMBus host controller query event (that is, an SMBus host controller interrupt) is raised after the clearing of the protocol register.

NOTE: The OS driver 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.

Status Code	Name	Description
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.

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 SMBus host controller, a write to this register initiates the transaction on the SMBus.

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
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

When the OS 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.

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 (i.e. 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 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
RES			**BCNT**				

Where:

RES: Reserved

BCNT: Block Count for Block Read and Block Write Protocols

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 SMBus host controller 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 the OS can take immediate corrective actions. Once an alarm message has been received, the SMBus host controller will 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 the OS driver until the alarm status bit is set. The OS driver then reads the three 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 the section 13.9.1. 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 quick write 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 quick read 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 send byte protocol.

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 receive byte protocol.

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 write byte protocol.

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 read byte protocol.

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 write word protocol.

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 read word protocol.

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 write block protocol.

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 read block protocol.

Data Returned:

SMB_BCNT: Number of data bytes (1-32) received.

SMB_DATA[0:3]: 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 process call protocol.

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.

13.9.3 SMBus Register Set

The register set for the SMBus host controller has the following format. All registers are eight 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.
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.

LOCATION	REGISTER NAME	DESCRIPTION
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.

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

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 a good example 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 the 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 Name Space

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 Controller's 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.
_HID	Named object that provides the Embedded Controller's Plug and Play identifier. This value is be set to PNP0A09. _HID is a standard device configuration control method defined in section 6.1.3.
_GPE	Named object that returns what SCI interrupt within the GPx_STS register (bit assignment). 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 Name Space

An embedded controller device is created using the named device object. The embedded controller's device object requires the following elements:

Table 13-6 EC SMBus Host Controller Device Objects

Object	Description
_HID	Named object that provides the Embedded Controller'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.
_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.

13.12.1 Example EC SMBus Host Controller ASL-Code

Example ASL-code that defines an SMBus Host Controller from within an embedded controller device is shown below:

```
Device(EC0) {
    Name(_HID, EISAID("PNP0C09"))
         Name(_CRS,
                 ResourceTemplate(){              // port 0x62 and 0x66
                         IO(Decode16, 0x62, 0x62, 0, 1),  // Status port
                         IO(Decode16, 0x66, 0x66, 0, 1)   // command port
                         }
              )
    Name(_GPE, 0)

    Device (SMB1) {
        Name (_HID, "ACPI0001")
                Name(_EC, 0x8030)                    // EC offset, Query
                OperationRegion(PHO1, SMBus, 0x51, 0x1)
        Device(DEVA){
            Name(_ADR, 0x51)
            Field(PHO1, ByteAcc, NoLock, Preserve) {
                TST0,  1,
                TST1,  1,
                NULL,  5,
                TST7,  1,
                }
            }       // end of DEVA
        }           // end of SMB1

    Device (SMB2) {
        Name (_HID, "ACPI0001")
                Name(_EC, 0x9031)               // EC offset, Query
                OperationRegion(PHO1, SMBus, 0x62, 0x1)
                OperationRegion(PHO2, SMBus, 0x50, 0x2)
        Device(DEVB){
            Name(_ADR, 0x62)
                        Field(PHO1, SMBQuickAcc, NoLock, Preserve) {
                              TSTC,   8
                              }              // end of DEVB
        Device(EPRM){
            Name(_ADR, 0x50)
                        Field(PHO2, AnyAcc, NoLock, Preserve){
                                FLD1, 256,
                                FLD2, 8,
                                FLD3, 16,
                                FLD4, 8,
                                FLD5, 224
                                }
            }       // end of EPRM
        }           // end of SMB2
    }               // end of EC
```

14. Query System Address Map

This section explains the special INT 15 call that Intel and Microsoft developed for use in IA-PC based systems. The call supplies the operating system with a clean memory map indicating address ranges that are reserved and ranges that are available in the motherboard.

14.1 INT 15H, E820H - Query System Address Map

This call can be used in real mode only.

This call returns a memory map of all the installed RAM, and of physical memory ranges reserved by the BIOS. The address map is returned by making successive calls to this API, each returning one run of physical address information. Each run has a type that dictates how this run of physical address range is to be treated by the operating system.

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 16Mb, the INT-15 88 and INT-15 E801 functions must return the top of memory below the AddressRangeACPI and AddressRangeNVS memory ranges.

Table 14-1 Input

EAX	Function Code	E820h
EBX	Continuation	Contains the continuation value to get the next run 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 14-2 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 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 a carry. The caller will ignore any other information returned by the BIOS when the carry flag is set.

Table 14-3 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 the following table.

Table 14-4 Address Ranges in the Type Field

Value	Mnemonic	Description
1	AddressRangeMemory	This run is available RAM usable by the operating system.
2	AddressRangeReserved	This run 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 run is available RAM usable by the operating system after it reads the ACPI tables.
4	AddressRangeNVS	ACPI NVS Memory. This run 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. Any range of this type must be treated by the OS 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.

14.2 Assumptions and Limitations

- The BIOS returns address ranges describing base board memory and ISA or PCI memory that is contiguous with that base board 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 operating system 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 base board 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. Example video memory at `A0000` to `BFFFF` physical are not described by this function. The range from `E0000` to `EFFFF` is specific to the base board and is reported as it applies to that base board.
- All of lower memory is reported as normal memory. The operating system 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).

14.3 Example Address Map

This sample address map (for an Intel processor-based system) describes a machine which has 128 MB of RAM, 640K of base memory and 127 MB of extended memory. The base memory has 639K available for the user and 1K 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-64K.

The 639K 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 14-5 Sample Memory Map

Base (Hex)	Length	Type	Description
0000 0000	639K	AddressRangeMemory	Available Base memory - typically the same value as is returned using the INT 12 function.
0009 FC00	1K	AddressRangeReserved	Memory reserved for use by the BIOS(s). This area typically includes the Extended BIOS data area.
000F 0000	64K	AddressRangeReserved	System BIOS
0010 0000	7MB	AddressRangeMemory	Extended memory, which is not limited to the 64-MB address range.
0080 0000	4MB	AddressRangeReserved	Chip set memory hole required to support the LFB mapping at 12 MB.
0100 0000	120MB	AddressRangeMemory	Base board RAM relocated above a chip set memory hole.
FEC0 0000	4K	AddressRangeReserved	I/O APIC memory mapped I/O at FEC00000.
FEE0 0000	4K	AddressRangeReserved	Local APIC memory mapped I/O at FEE00000.
FFFF 0000	64K	AddressRangeReserved	Remapped System BIOS at end of address space.

14.4 Sample 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
        .
        .
        .
    }
```

15. ACPI Source Language (ASL) Reference

This section formally defines the ACPI Control Method Source Language (ASL). ASL is a source language for writing ACPI control methods. OEMs and BIOS developers 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 Control Method Machine Language (AML) Specification, section 16.

AML and ASL are ***different languages*** though they are closely related.

Every ACPI-compatible OSes 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.

15.1 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 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. 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 two ways: ordinary decimal, or hexadecimal, using the notation 0x*dd*
- **nothing** indicates an empty item. For example **{ nothing }** is equivalent to **{}**

15.1.1 ASL Grammar Notation

The notation used to express the ASL grammar is specified in the following table.

Table 15-1 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.
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.	See the examples for ellipses and square brackets.
Square brackets ([ ])	Used to indicate optional items.	Term [Term . . .] means an ordered list of one or more terms.
Ellipses (. . .)	Indicates continuation of a list of terms of the same type.	,Arg … means one or more Arg terms, separated by commas.
Word in bold.	Denotes the name of a term in the ASL grammar, representing any instance of such a term.	In the following ASL term definition: **ThermalZone** (*ZoneName*) {NamedObjectList} 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*) {NamedObjectList} 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 2 hexadecimal digits.	0x21 means a value of hexadecimal 21, or decimal 37. Note 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.

15.1.2 ASL Names

```
leadnamechar :=          'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|
                         'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'|'_'
namechar :=              'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|
                         'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'|'_'|'0'|
                         '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
nameseg :=               <leadnamechar namechar namechar namechar> |
                         <leadnamechar namechar namechar> |
                         <leadnamechar namechar > |
                         <leadnamechar >
namepath :=              nameseg | <namepath '.' nameseg>
rootpath :=              '\'
prefixpath :=            '^' ['^'...]
NameString :=            <rootpath namepath> | <prefixpath namepath> | namepath
```

15.1.3 ASL Language and Terms

```
ASL :=                   DefinitionBlockTerm
DefinitionBlockTerm :=   DefinitionBlock(
                                OutPutFileName,        //String
                                Signature,             //String
                                DSDT Revision,         //ByteConst
                                OEMID,                 //String
                                TableID,               //String
                                OEMRevision            //DWordConst
                                )
                                {TermList}
TermList :=              Nothing | Term [Term ...]
Term :=                  CompilerDirectiveTerm | DeclarationTerm | OperatorCodeTerm
CompilerDirectiveTerm := IncludeTerm
IncludeTerm :=           Include(
                                Pathname               //String (file system pathname)
                                )
DataObjectTerm :=        BufferTerm | LiteralDataTerm | PackageTerm
BufferTerm :=            Buffer(
                                ByteCount | Nothing    //OpCode=>Integer
                                )
                                {Initializer}          //String | ByteList
LiteralDataTerm :=       Integer | String
Integer :=               ByteConst | WordConst | DwordConst
ByteConst :=             0x00 through 0xFF, inclusive
WordConst :=             0x0000 through 0xFFFF, inclusive
DwordConst :=            0x00000000 to 0xFFFFFFFF,inclusive
ByteList :=              Nothing | ByteConst [,ByteConst ...]
String :=                '"' AsciiCharList '"'
AsciiCharList :=         Nothing | AsciiChar [AsciiChar ...]
AsciiChar :=             0x01 through 0x7F, inclusive
NullChar :=              0x00
PackageTerm :=           Package(
                                ElementCount | Nothing    //ElementCount is ByteConst
                                )
                                {PackageList}
PackageList :=           Nothing | PackageElement [,PackageElement ...]
PackageElement :=        DataObjectTerm | ConstantTerm | SuperName
ConstantTerm :=          OneTerm | OnesTerm | ZeroTerm
OneTerm :=               One
OnesTerm :=              Ones                          //ByteConst | WordConst | DwordConst
ZeroTerm :=              Zero
SuperName :=             Namestring | MethodObjectTerm | DebugTerm
ResultName :=            Nothing | SuperName
MethodObjectTerm :=      ArgTerm | LocalTerm
ArgTerm :=               Arg0 | Arg1 | Arg2 | Arg3 | Arg4 | Arg5 | Arg6
LocalTerm :=             Local0 | Local1 | Local2 | Local3 | Local4 | Local5 | Local6 |
                         Local7
DebugTerm :=             Debug

DeclarationTerm :=       NamedObjectTerm | NameSpaceModifierTerm
KeywordTerm :=           AccessTypeKeyword | LockRuleKeyWord | MatchOpKeyword |
                         RegionSpaceKeyword | SerializeRuleKeyword |
                         UpdateRuleKeyword
AccessTypeKeyword :=     AnyAcc | ByteAcc | WordAcc | DWordAcc | BlockAcc |
                         SMBSendRecvAcc | SMBQuickAcc
```

```
LockRuleKeyword :=          Lock | NoLock
MatchOpKeyword :=           MEQ | MGE | MGT | MLE | MLT | MTR
RegionSpaceKeyword :=       EmbeddedControl | PCI_Config | SystemIO | SystemMemory | SMBus
SerializeRuleKeyword :=     Serialized | NotSerialized | Nothing
UpdateRuleKeyword :=        Preserve | WriteAsOnes | WriteAsZeros

NamedObjectTerm :=          BankFieldTerm | DeviceTerm | EventTerm | FieldTerm |
                            IndexFieldTerm | MethodTerm | MutexTerm | OperationRegionTerm |
                            PowerResourceTerm | ProcessorTerm | ThermalZoneTerm
BankFieldTerm :=            BankField(
                                  RegionName,               //NameString
                                  BankName,                 //NameString
                                  BankValue,                //OpCode=>DWord
                                  AccessType,               //AccessTypeKeyword
                                  LockRule,                 //LockRuleKeyword
                                  UpdateRule                //UpdateRuleKeyword
                                  )
                                  {FieldList}
FieldList :=                Nothing | Field [,Field ...]
Field :=                    OffsetTerm | FieldEntry | AccessTerm
FieldEntry :=               <Nothing | NameSeg> ',' Integer
OffsetTerm :=               Offset(ByteOffset)              //integer
AccessTerm :=               AccessAs(
                                  AccessType,               //AccessTypeKeyword
                                  AccessAttribute           //ByteConst
                                  )
DeviceTerm :=               Device(
                                  BusDeviceName             //NameString
                                  )
                                  {NamedObjectList}
NamedObjectList :=          Nothing | NamedObjectTerm [NamedObjectTerm ...]
EventTerm :=                Event(
                                  EventName                 //NameString
                                  )
FieldTerm :=                Field(
                                  RegionName,               //NameString
                                  AccessType,               //AccessTypeKeyword
                                  LockRule,                 //LockRuleKeyword
                                  UpdateRule                //UpdateRuleKeyword
                                  )
                                  {FieldList}
IndexFieldTerm :=           IndexField(
                                  IndexName,                //NameString
                                  DataName,                 //NameString
                                  AccessType,               //AccessTypeKeyword
                                  LockRule,                 //LockRuleKeyword
                                  UpdateRule                //UpdateRuleKeyword
                                  )
                                  {FieldList}
MethodTerm :=               Method(
                                  MethodName,               //NameString
                                  ArgCount,                 //ByteConst
                                  SerializeRule             //SerializeRuleKeyword
                                  )
                                  {TermList}
MutexTerm :=                Mutex(
                                  MutexName,                //NameString
                                  SyncLevel                 //ByteConst
                                  )
OperationRegionTerm :=      OperationRegion(
                                  RegionName,               //NameString
                                  RegionSpace,              //RegionSpaceKeywd
                                  Offset,                   //OpCode=>DWord
                                  Length                    //OpCode=>DWord
                                  )
PowerResourceTerm :=        PowerResource(
                                  ResourceName,             //NameString
                                  SystemLevel,              //ByteConst
                                  ResourceOrder             //WordConst
                                  )
                                  {NamedObjectList}
ProcessorTerm :=            Processor(
```

```
                                ProcessorName,          //NameString
                                ProcessorID,            //ByteConst
                                PBlockAddress,          //ByteConst
                                PBlockLength            //DWordConst
                                )
                                {NamedObjectList}
ThermalZoneTerm :=          ThermalZone(
                                ZoneName                //NameString
                                )
                                {NamedObjectList}

NameSpaceModifierTerm :=    AliasTerm | NameTerm | ScopeTerm
AliasTerm :=                Alias   (
                                Source,                 //NameString
                                Destination             //NameString
                                )
NameTerm :=                 Name    (
                                ObjectName,             //NameString
                                Target                  //DataObjectTerm
                                )
ScopeTerm :=                Scope   (
                                Location   //NameString
                                )
                                {TermList}

OperatorCodeTerm :=         Type1OpCode | Type2OpCode | Type2Macro

UserMethodTerm :=           SuperName(ArgList)
ArgList :=                  Nothing | Arg [,Arg ...]
Arg :=                      OpCode
OpCode :=                   Type2OpCode | SuperName | ConstantTerm | LiteralDataTerm |
                            Type2Macro

// A Type1OpCode 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 Type2OpCode term can be used in an expression

Type1OpCode := BreakTerm | BreakPointTerm| CreateBitFieldTerm |
               CreateByteFieldTerm | CreateDWordFieldTerm |
               CreateFieldTerm | CreateWordFieldTerm |
               ElseTerm | FatalTerm | IfTerm | NoOpTerm | NotifyTerm |
               ReleaseTerm | ResetTerm | ReturnTerm | SignalTerm |
               SleepTerm | StallTerm | UnloadTerm | WhileTerm

BreakTerm :=                Break
BreakPointTerm :=           BreakPoint
CreateBitFieldTerm :=       CreateBitField(
                                Source,             //OpCode=>Buffer
                                BitIndex,           //OpCode=>Integer
                                Destination         //SuperName
                                )
CreateByteFieldTerm :=      CreateByteField(
                                Source,             //OpCode=>Buffer
                                ByteIndex,          //OpCode=>Integer
                                Destination         //SuperName
                                )
CreateDwordFieldTerm :=     CreateDwordField(
                                Source,             //OpCode=>buffer
                                ByteIndex,          //OpCode=>Integer
                                Destination         //SuperName
                                )
CreateFieldTerm :=          CreateField(
                                Source,             //OpCode=>Buffer
                                Offset,             //OpCode=>Integer
                                NumBits,            //OpCode=>Integer
                                Destination         //SuperName
                                )
CreateWordFieldTerm :=      CreateWordField(
                                Source,             //OpCode
                                ByteIndex,          //OpCode
                                Destination         //SuperName
```

```
                                )
ElseTerm :=                 Else
                                {False}             //TermList
FatalTerm :=                Fatal(
                                Type,               //ByteConst
                                Code,               //DWordConst
                                Arg                 //OpCode=>Integer
                                )
IfTerm :=                   If(
                                Predicate           //OpCode=>Integer
                                )
                                {True}              //TermList
NoopTerm :=                 Noop
NotifyTerm :=               Notify(
                                NotifyObject,       //SuperName
                                NotificationValue   //OpCode=>Byte
                                )
ReleaseTerm :=              Release(
                                SynchObject         //SuperName
                                )
ResetTerm :=
                            Reset(
                                SynchObject         //SuperName
                                )
ReturnTerm :=               Return(
                                Arg                 //OpCode=>Object
                                )
SignalTerm :=               Signal(
                                SynchObject         //SuperName
                                )
SleepTerm :=                Sleep(
                                Millisecond         //OpCode=>Integer
                                )
StallTerm :=                Stall(
                                Microseconds        //OpCode=>Byte
                                )
UnloadTerm :=               Unload(
                                NameSpaceObjectRef  //NameString
                            )
WhileTerm :=                While(
                                Predicate)          //OpCode
                                {True}              //TermList

Type2OpCode :=              AcquireTerm | AddTerm | AndTerm | ConcatenateTerm |
                            DecrementTerm | DivideTerm |
                            ElseTerm | FindSetLeftBitTerm |
                            FindSetRightBitTerm | FromBCDTerm | IncrementTerm |
                            IndexTerm | LAndTerm | LEqualTerm | LGreaterTerm |
                            LGreaterEqualTerm | LLessTerm | LLessEqualTerm |
                            LNotTerm | LNotEqualTerm | LoadTerm | MatchTerm |
                            MultiplyTerm | NAndTerm | NorTerm | NotTerm |
                            ObjectTypeTerm | OrTerm | ShiftLeftTerm |
                            ShiftRightTerm | SizeOfTerm | StoreTerm |
                            SubtractTerm | ToBCDTerm | WaitTerm | XOrTerm

AcquireTerm :=          Zero | Ones <= //Ones means timed-out
                            Acquire(
                                SynchObject,        //SuperName
                                TimeOut             //WordConst
                                )
AddTerm :=              Integer <=
                            Add(
                                Addend1,            //OpCode=>Integer
                                Addend2,            //OpCode=>Integer
                                Result              //ResultName
                                )
AndTerm :=              Integer <=
                            And(
                                Source1,            //OpCode=>Integer
                                Source2,            //OpCode=>Integer
                                Result              //ResultName
                                )
ConcatenateTerm :=      Integer | String | Buffer <=
```

```
                        Concatenate(
                                Source1,          //OpCode
                                Source2,          //OpCode
                                Destination       //SuperName
                                )
CondRefOfTerm :=        Ones | Zero <=
                        CondRefOf(
                                Source            //SuperName
                                Destination       //SuperName
                                )
DecrementTerm :=        Integer <=
                        Decrement(
                                Addend            //SuperName
                                )
DivideTerm :=           Integer <=      //returns Result
                        Divide(
                                Dividend,         //OpCode=>Integer
                                Divisor,          //OpCode=>Integer
                                Remainder,        //ResultName
                                Result            //ResultName
                                )
FindSetLeftBitTerm :=   ByteConst <=
                        FindSetLeftBit(
                                Source,           //OpCode=>Integer
                                BitNo             //SuperName
                                )
FindSetRightBitTerm :=  ByteConst <=
                        FindSetRightBit(
                                Source,           //OpCode=>Integer
                                BitNo             //SuperName
                                )
FromBCDTerm :=          Integer <=
                        FromBCD(
                                BcdValue,         //OpCode=>Integer
                                Destination       //ResultName
                                )
IncrementTerm :=        Integer <=
                        Increment(
                                Addend            //SuperName
                                )
IndexTerm :=            PackageElement <=
                        Index(
                                Source,           //OpCode=>PackageObject
                                Index,            //OpCode=>Integer
                                Destination       //NameString
                                )
LAndTerm :=             Ones | Zero <=
                        LAnd(
                                Source1,          //OpCode=>Integer
                                Source2           //OpCode=>Integer
                                )
LEqualTerm :=           Ones | Zero <=     //Ones means equal
                        LEqual(
                                Source1,          //OpCode=>Integer
                                Source2           //OpCode=>Integer
                                )
LGreaterTerm :=         Ones | Zero <=     //Ones means S1 > S2
                        LGreater(
                                Source1,          //OpCode=>Integer
                                Source2           //OpCode=>Integer
                                )
LGreaterEqualTerm:=     Ones | Zero  <=    //Ones means S1 >= S2
                        LGreaterEqual(
                                Source1,          //OpCode=>Integer
                                Source2           //OpCode=>Integer
                                )
LLessTerm :=            Ones | Zero <=     //Ones means S1 < S2
                        LLess(
                                Source1,          //OpCode=>Integer
                                Source2           //OpCode=>Integer
                                )
LLessEqualTerm :=       Ones | Zero <=     //Ones means S1 <= S2
```

```
                        LLessEqual(
                                Source1,          //OpCode=>Integer
                                Source2           //OpCode=>Integer
                                )
LNotTerm :=         Ones | Zero <=
                        LNot(
                                Source            //OpCode=>Integer
                                )
LNotEqualTerm :=    Ones | Zero <=     //Ones means S1 <> S2
                        LNotEqual(
                                Source1,          //OpCode=>Integer
                                Source2           //OpCode=>Integer
                                )
LoadTerm :=         DwordConst <=  // Name space object reference
                        Load(
                                RegionName        //NameString
                                )
LOrTerm :=          Integer <=
                        LOr(
                                Source1,          //OpCode=>Integer
                                Source2           //OpCode=>Integer
                                )
MatchTerm :=        Integer | Ones <=
                        Match(
                                SearchPackage,    //OpCode=>Package
                                Op1,              //MatchOpKeyword
                                V1,               //OpCode=>Object
                                Op2,              //MatchOpKeyword
                                V2,               //OpCode=>Object
                                Start             //OpCode=>Object
                                )
MultiplyTerm :=     Integer <=
                        Multiply(
                                Multiplcand,      //OpCode=>Integer
                                Multiplier,       //OpCode=>Integer
                                Result            //ResultName
                                )
NAndTerm :=         Integer <=
                        NAnd(
                                Source1,          //OpCode=>Integer
                                Source2,          //OpCode=>Integer
                                Result            //ResultName
                                )
NOrTerm :=          Integer <=
                        NOr(
                                Source1,          //OpCode=>Integer
                                Source2,          //OpCode->Integer
                                Result            //ResultName
                                )
NotTerm :=          Integer <=
                        Not(
                                Source1,          //OpCode=>Integer
                                Result            //ResultName
                                )
ObjectTypeTerm :=   ByteConst <=
                        ObjectType(
                                Object            //SuperName
                                )
OrTerm :=           Integer <=
                        Or(
                                Source1,          //OpCode=>Integer
                                Source2,          //OpCode=>Integer
                                Result            //ResultName
                                )
RefOfTerm :=        ObjectReference <=
                        RefOf(
                                Object            //SuperName
                                )
ShiftLeftTerm :=    Integer <=
                        ShiftLeft(
                                Source,           //OpCode=>Integer
                                Count             //OpCode=>Integer
```

```
                                Result              //ResultName
                                )
ShiftRightTerm :=       Integer <=
                            ShiftRight(
                                Source,             //OpCode=>Integer
                                Count               //OpCode=>Integer
                                Result              //ResultName
                                )
SizeOfTerm :=           Integer <=
                            SizeOf(
                                DataObject          //SuperName=>DataObject
                                )
StoreTerm :=            Buffer | Integer | String <=
                            Store(
                                Source,             //OpCode
                                Destination         //SuperName
                                )
SubtractTerm :=         Integer <=
                            Subtract(
                                Addend1,            //OpCode=>Integer
                                Addend2,            //OpCode=>Integer
                                Result              //ResultName
                                )
ToBCDTerm :=            Integer <=
                            ToBCD(
                                Value               //OpCode=>Integer
                                Destination         //ResultName
                                )
WaitTerm :=             Zero | Ones <=     //Ones means timed-out
                            Wait(
                                SynchObject,        //SuperName
                                TimeOut             //OpCode
                            )
XOrTerm :=              Integer <=
                            XOr(
                                Source1,            //OpCode=>Integer
                                Source2,            //OpCode=>Integer
                                Result              //ResultName
                                )

Type2Macro :=           EISADTerm | ResourceTemplateTerm
EISADTerm :=            DwordConst <=
                            EISAID(
                                ID                  //String
                                )
ResourceTemplateTerm := Buffer <=
                            ResourceTemplate()
                                {ResourceMacroList}
ResourceMacroList :=    ResourceMacroTerm [ResourceMacroTerm ...]
ResourceMacroTerm :=    DMATerm | DWORDIOTerm | DWORDMemoryTerm |
                        EndDependentFnTerm | FixedIOTerm |
                        InterruptTerm | IOTerm | IRQNoFlagsTerm | IRQTerm |
                        Memory24Term | Memory32FixedTerm | Memory32Term |
                        StartDependentFnTerm | StartDependentFnNoPriTerm |
                        VendorLongTerm | VendorShortTerm |
                        WORDBusNumberTerm | WORDIOTerm

DMATerm := Buffer <=
    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
        )
        {
        ByteConst [, ByteConst ...]       // List of channel numbers(valid values: 0-17)
        }
DWORDIOTerm := Buffer <=
    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
        DWordConst,                                     // _GRA, Address granularity
        DWordConst,                                     // _MIN, Address range minimum
        DWordConst,                                     // _MAX, Address range max
        DWordConst,                                     // _TRA, Translation
        ByteConst | Nothing,                            // Resource Source Index;
                                                        // if Nothing, not generated
        NameString | Nothing                            // Resource Source;
                                                        // if Nothing, not generated
        NameString | Nothing                            // A name to refer back to
                                                        // this resource
        )
DWORDMemoryTerm := Buffer <=
    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
        DWordConst,                                     // _GRA, Address granularity
        DWordConst,                                     // _MIN, Address range minimum
        DWordConst,                                     // _MAX, Address range max
        DWordConst,                                     // _TRA, Translation
        ByteConst | Nothing,                            // Resource Source Index;
                                                        // if Nothing, not generated
        NameString | Nothing                            // Resource Source;
                                                        // if Nothing, not generated
        NameString | Nothing                            // A name to refer back
                                                        // to this resource
        )
EndDependentFnTerm :=  Buffer <=
    EndDependentFn(
                  )
FixedIOTerm := Buffer <=
    FixedIO(
        WordConst,                                      // _BAS, Address base
        ByteConst,                                      // _LEN, Range length
        NameString | Nothing                            // A name to refer back
                                                        // to this resource
        )
InterruptTerm := Buffer <=
    Interrupt(
        ResourceConsumer | ResourceProducer | Nothing,  // Nothing=>ResourceConsumer
        Edge | Level,                                   // _LL, _HE
        ActiveHigh | ActiveLow,                         // _LL, _HE
        Shared | Exclusive | Nothing,                   // _SHR, Nothing=>Exclusive
        ByteConst | Nothing,                            // Resource Source Index;
                                                        // if Nothing, not generated
        NameString | Nothing,                           // Resource Source;
                                                        // if Nothing, not generated
        NameString | Nothing                            // A name to refer back
                                                        // to this resource
        )
        {
        DWordConst [, DWordConst ...]                   // _INT, list of interrupt
                                                        // numbers
        }

IOTerm := Buffer <=
    IO(
        Decode16 | Decode10,                            // _DEC
        WordConst,                                      // _MIN, Address minimum
        WordConst,                                      // _MAX, Address max
        ByteConst,                                      // _ALN, Base alignment
        ByteConst                                       // _LEN, Range length
        NameString | Nothing                            // A name to refer back
                                                        // to this resource
        )
IRQNoFlagsTerm := Buffer <=
    IRQNoFlags(
        NameString | Nothing                            // A name to refer back
```

```
                                                        // to this resource
        )
        {
        ByteConst [, ByteConst ...]                     // list of IRQ numbers
                                                        // (valid values: 0-15)
        }
IRQTerm := Buffer <=
    IRQ(
        Edge | Level,                                   // _LL, _HE
        ActiveHigh | ActiveLow,                         // _LL, _HE
        Shared | Exclusive | Nothing,                   // _SHR, Nothing=>Exclusive
        NameString | Nothing                            // A name to refer back to
                                                        // this resource
        )
        {
        ByteConst [, ByteConst ...]                     // List of IRQ numbers
                                                        // (valid values: 0-15)
        }
Memory24Term := Buffer <=
    Memory24(
        ReadWrite | ReadOnly,                   // _RW
        WordConst,                              // _MIN, Minimum base memory address [23:8]
        WordConst,                              // _MAX, Maximum base memory address [23:8]
        WordConst,                              // _ALN, Base alignment
        WordConst                               // _LEN, Range length
        NameString | Nothing                    // A name to refer back to this resource
        )
Memory32Fixedterm := Buffer <=
    Memory32Fixed(
        ReadWrite | ReadOnly,                   // _RW
        DWordConst,                             // _BAS, Range base
        DWordConst                              // _LEN, Range length
        NameString | Nothing                    // A name to refer back to this resource
        )
Memory32Term := Buffer <=
    Memory32(
        ReadWrite | ReadOnly,                   // _RW
        DWordConst,                             // _MIN, Minimum base memory address
        DWordConst,                             // _MAX, Maximum base memory address
        DWordConst,                             // _ALN, Base alignment
        DWordConst                              // _LEN, Range length
        NameString | Nothing                    // A name to refer back to this resource
        )
StartDependentFnTerm := Buffer <=
    StartDependentFn(
        ByteConst,                              // Compatibility priority (valid values: 0-2)
        ByteConst                               // Performance/Robustness priority
                                                // (valid values: 0-2)
        )
        {
        // List of descriptors for this dependent function
        }
StartDependentFnNoPriTerm := Buffer <=
    StartDependentFnNoPri(
        )
        {
        // List of descriptors for this dependent function
        }
VendorLongTerm := Buffer <=
    VendorLong(
        NameString | Nothing                    // A name to refer back to this resource
        )
        {
        ByteConst [, ByteConst ...]             // List of bytes
        }
VendorShortTerm := Buffer <=
    VendorShort(
        NameString | Nothing                    // A name to refer back to this resource
        )
        {
        ByteConst [, ByteConst ...]             // List of bytes, up to 7 bytes
        }
WORDBusNumberTerm := Buffer <=
    WORDBusNumber(
        ResourceConsumer | ResourceProducer | Nothing,   // Nothing=>ResourceConsumer
```

```
        MinFixed | MinNotFixed | Nothing,              // _MIF, Nothing=>MinNotFixed
        MaxFixed | MaxNotFixed | Nothing,              // _MAF, Nothing=>MaxNotFixed
        SubDecode | PosDecode | Nothing,               // _DEC, Nothing=>PosDecode
        WordConst,                                     // _GRA, Address granularity
        WordConst,                                     // _MIN, Address range minimum
        WordConst,                                     // _MAX, Address range max
        WordConst,                                     // _TRA: Translation
        ByteConst | Nothing,                           // Resource Source Index;
                                                       // if Nothing, not generated
        NameString | Nothing                           // Resource Source;
                                                       // if Nothing, not generated
        NameString | Nothing                           // A name to refer back
                                                       // to this resource
        )
WORDIOTerm := Buffer <=
    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
        WordConst,                                     // _GRA, Address granularity
        WordConst,                                     // _MIN, Address range minimum
        WordConst,                                     // _MAX, Address range max
        WordConst,                                     // _TRA, Translation
        ByteConst | Nothing,                           // Resource Source Index;
                                                       // if Nothing, not generated
        NameString | Nothing                           // Resource Source;
                                                       // if Nothing, not generated
        NameString | Nothing                           // A name to refer back
                                                       // to this resource
        )
```

15.2 Full ASL Reference

This reference section is for developers who are writing ASL code while developing definition blocks for platforms.

15.2.1 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.

Table 15-2 Control Method Named Object Reference Encodings

Value	Description	
41-5A, 5F	Lead character of name (‘A’ - ‘Z’, ‘_’)	**LeadNameChar**
30-39, 41-5A, 5F	Non-lead (trailing) character of name (‘A’ - ‘Z’, ‘_’, ’0 - 9’)	**NameChar**

The following table lists the name modifiers.

Table 15-3 Definition Block Name Modifier Encodings

	Description	NamePrefix :=	**Followed by …**
5C	Name space root (‘\’)	RootPrefix	Name
5E	Parent name space (‘^’)	ParentPrefix	Name
2E	Name extender: 1	DualNamePrefix	Name Name
2F	Name extender: N	MultiNamePrefix	*count* Name^{count}

15.2.2 ASL Data Types

The contents of an object, or the data it references, may be abstract entities (for example, “Device Object”) or can be one of three computational data types. The computational data type can be used as arguments to many of the ASL Operator terms.

Table 15-4 Data Types

Data Type	Description
Integer	32-bit little endian unsigned value.
Buffer	Arbitrary fixed length array of bits.
String	ASCIIZ string 1 to 200 characters in length (including NullChar).

These data types are automatically converted as needed during computation as shown in the following table:

Table 15-5 Data Type Conversion

Data Type	Comment
Integer	Converts to string of its hex representation Converts to a 32-bit buffer.
Buffer	Buffers of 32 bits or less convert to their integer representation. Buffers which are larger then 32 bits cannot be converted to integer. Converts to string by its hex-dump representation with a space every 8 bits.
String	Does not convert.

15.2.3 ASL Terms

This section describes all the ASL terms and provides sample ASL code that uses the terms. For other sample ASL code, see the three sections of the specification that describe ACPI concept machines.

The ASL terms are grouped into the following categories:

- Definition block term
- Compiler directive terms
- Data object terms
- Declaration terms
- Operator terms

15.2.3.1 Definition Block Term

```
DefinitionBlockTerm :=    DefinitionBlock(
                              OutPutFileName,       //String
                              Signature,            //String
                              DSDT Revision,        //ByteConst
                              OEMID,                //String
                              TableID,              //String
                              OEMRevision           //DWordConst
                              )
                              {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 addition 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.

15.2.3.2 Compiler Directive Terms

```
IncludeTerm :=            Include(
                              Pathname              //String (file system pathname)
                              )
```

Pathname is the full OS file system path to another file that contains ASL terms to be included in the current file of ASL terms.

15.2.3.3 Data Object Terms

There are four types of ASL declaration terms: constant terms, data object declaration terms, the debug object term, and method object terms.

15.2.3.3.1 Constant Terms

The constant declaration terms are **One**, **Ones**, and **Zero**.

15.2.3.3.1.1 One - Constant One Object

```
OneTerm :=              One
```

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 have no effect and are ignored.

15.2.3.3.1.2 Ones - Constant Ones Object

```
OnesTerm :=             Ones                        //ByteConst | WordConst | DwordConst
```

The constant ones object is an object of type Integer that will always read as all bits set. Writes to this object have no effect and are ignored.

15.2.3.3.1.3 Zero - Constant Zero Object

```
ZeroTerm :=             Zero
```

The constant zero object is an object of type Integer that will always read as all bits clear. Writes to this object have no effect and are ignored.

15.2.3.3.2 Data Object Declaration Terms

The data object declaration terms are:

- Buffer declarations (used to declare and initialize long strings).
- Literal data declarations (used to declare and initialize integers and short strings).
- Package data declarations (used to declare arrays and data structures).

15.2.3.3.2.1 Buffer - Declare Buffer

```
BufferTerm :=           Buffer(
                               ByteCount | Nothing    //OpCode=>Integer
                               )
                               {Initializer}          //String | ByteList
```

Declares a Buffer, of size *ByteCount* and initial value of *Initializer*.

The optional *ByteCount* parameter specifies the size of the buffer and the initial value is specified in *Initializer*. If *ByteCount* 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 datum in name space, 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}
```

15.2.3.3.2.2 Literal Data Declarations

```
LiteralDataTerm :=      Integer | String
```

15.2.3.3.2.2.1 Integers

```
Integer :=              ByteConst | WordConst | DwordConst
ByteConst :=            0x00 through 0xFF, inclusive
WordConst :=            0x0000 through 0xFFFF, inclusive
DwordConst :=           0x00000000 to 0xFFFFFFFF,inclusive
ByteList :=             Nothing | ByteConst [,ByteConst ...]
```

Using the above grammar to define an object containing the value of integer causes the ASL compiler to automatically pick the proper width of the defined integer (byte, word, or Dword).

15.2.3.3.2.2.2 Strings

```
String :=               '"' AsciiCharList '"'
AsciiCharList :=        Nothing | AsciiChar [AsciiChar ...]
AsciiChar :=            0x01 through 0x7F, inclusive
NullChar :=             0x00
```

The above grammar can be used to define an object containing a read-only string value. The default string value is the null string, which has 0 bytes available for storage of other values.

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)
```

15.2.3.3.2.3 Package - Declare Package Object

```
PackageTerm :=          Package(
                            ElementCount | Nothing     //ElementCount is ByteConst
                            )
                            {PackageList}
```

Declares an unnamed aggregation of data items, constants, and/or references to control methods. The size of the package is *ElementCount*. *PackageList* contains the list data items, constants, and/or control method references used to initialize the package. If *ElementCount* is absent, it is set to match the number of elements in the PackageList. If *ElementCount* is present and greater than the number of elements in the PackageList, the default entry Undefined is used to initialize the package elements beyond those initialized from the PackageList. Evaluating an undefined element will yield an error, but they can be assigned values to make them defined. It is an error for *ElementCount* to be less than the number of elements in the PackageList

There are two types of package elements in the PackageList: references to data objects and references to control methods.

Note: Version 1.0 of the Microsoft-provided ASL compiler does not allow non-method code package objects (code packages). However, there is nothing in the ACPI specification that precludes this. If implemented in an ASL compiler, evaluation on non-method code package objects are performed in the scope of the invoking method. The targets of all stores, loads, and references to the locals, arguments, or constant terms are the same invoking method's objects.

Example 1:

```
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 **IndexField** term definitions).

```
Package (10) {}
```

15.2.3.3.3 Debug Data Object

```
DebugTerm :=            Debug
```

The debug data object is an object of type Operation Region, which has "virtual content." 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 15-6 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.

15.2.3.3.4 Method Objects

The method objects can be used in control methods to pass and receive arguments and for local storage variables.

15.2.3.3.4.1 Arg0 | Arg1 | Arg2 … - 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.

15.2.3.3.4.2 Local0 | Local1 | Local2… - Local Data Object

```
LocalTerm :=            Local0 | Local1 | Local2 | Local3 | Local4 | Local5 | Local6 |
                        Local7
```

Up to 8 locals can be referenced as one byte encodings. 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.

15.2.3.4 Declaration Terms

There are two types of ASL declaration terms: named object terms and name space modifier terms.

15.2.3.4.1 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 15-7 Named Object Creators

ASL Statement	Description
BankField	Declares fields in a banked configuration object.
Device	Declares a bus/device object.

ASL Statement	Description
Event	Declares an event synchronization object.
Field	Declares fields.
IndexField	Declares fields in an index/data configuration object.
Method	Declares a control method.
Mutex	Declares a synchronization method.
OperationRegion	Declares an operational region.
PowerResource	Declares a power resource object.
Processor	Declares a processor package.
ThermalZone	Declares a thermal zone package.

15.2.3.4.1.1 BankField - Declare Bank/Data Field

```
BankFieldTerm :=        BankField(
                            RegionName,             //NameString
                            BankName,               //NameString
                            BankValue,              //OpCode=>DWord
                            AccessType,             //AccessTypeKeyword
                            LockRule,               //LockRuleKeyword
                            UpdateRule              //UpdateRuleKeyword
                            )
                            {FieldList}
```

This statement creates a data object at load time. The contents of the created object are obtained by a reference to a bank selection register.
This encoding is used to define named data 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 FieldList are the same format as the Field operator.

The following is a block of ASL sample code using *BankField:*

- Creates a 4-bit bank select register in system I/O space.
- Creates overlapping fields in the same system I/O space which are selected via the bank register.

```
// define 256-byte operational region in SystemIO space
// and name it GIO0
OperationRegion (GIO0, 1, 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
}
```

15.2.3.4.1.2 Device - Declare Bus/Device Package

```
DeviceTerm :=           Device(
                            BusDeviceName           //NameString
                            )
                            {NamedObjectList}
```

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 name space corresponding to that device's location in the system. Within the name space 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 Name Space correct.

The following block of ASL sample code shows a nested use of Device objects to describe an IDE controller.

```
Device (IDE0) {        // primary controller
    Name(_ADR, 0)      // put device/function id here

    // define region for IDE mode register
    OperationRegion (PCIC, PCI_Config, 0x50, 0x10) { }
    Field (PCIC, AnyAcc, NoLock, Preserve) {
        …
    }

    Device(MSTR) {     // master channel
        Name(_ADR, 0)
        Name(_PR0, Package(){0, PIDE})
        Method (_STM, 2) {
            …
        }
    }

    Device(SLAV) {
        Name(_ADR, 1)
        Name(_PR0, Package(){0, PIDE})
        Method (_STM, 2) {
            …
        }
    }
}
```

15.2.3.4.1.3 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 in section 15.2.3.5.1.

15.2.3.4.1.4 Field - Declare Field Objects

```
FieldTerm :=            Field(
                                RegionName,                 //NameString
                                AccessType,                 //AccessTypeKeyword
                                LockRule,                   //LockRuleKeyword
                                UpdateRule                  //UpdateRuleKeyword
                                )
                                {FieldList}
```

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 scope of the **Field** term. **Field** opens a name scope.

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.

All accesses within the parent object are performed naturally aligned. If desired, *AccessType* can be used to force minimum access width. Note 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. 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 15-8 OperationRegion Region Types and Access Types

Region Types	Access Type	Description
SystemMemory **SystemIO** **PCI_Config**	**ByteAcc** **WordAcc** **DWordAcc** **AnyAcc**	Read/Write byte, word, Dword access
EmbeddedControl	**ByteAcc**	
SMBus	**ByteAcc**	Read/Write SMBus byte protocol
	WordAcc	Read/Write SMBus word protocol
	BlockAcc	Read/Write SMBus block protocol
	AnyAcc	Read/Write linear SMBus byte, word, block protocol
	SMBSendRecvAcc	Send/Receive SMBus protocol
	SMBQuickAcc	QuickRead/QuickWrite SMBus protocol

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 an **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**(byte_offset) macro within a *FieldList* to skip to the bit position of the supplied byte offset.

For support of non-linear address devices, such as SMBus devices, a protocol is required to be associated with each command value. The ASL compiler supports the AccessAs(*AccessType, AccessAttribute*) macro within a *FieldList*. The *AccessAttribute* portion of the macro is interpreted differently depending on the address space. For System Memory, SystemIO, PCI_Config or Embedded Controller space the *AccessAttribute* is reserved. For SMBus devices the *AccessAttribute* indicates the command value of the SMBus device to use for the field being defined. The *AccessAttribute* allows a specific protocol to be associated with the fields following the macro and can contain any of the Access Type listed in the table.

15.2.3.4.1.4.1 SMBus Slave Address

SMBus device Addressing supports both a linear and non-linear addressing mechanism. This section clarifies how ACPI treats these types of devices and how they should be defined and accessed. SMBus devices are defined to have a fixed 7-bit slave address. This can be illustrated by the smart battery subsystem devices:

Table 15-9 Examples of SMBus Devices and Slave Addresses

SMBus Device Description	Slave Address (A0-A6)
SMBus Host Slave Interface	0x8
SBS Charger	0x9
SBS Selector	0xA
SBS Battery	0xB

The SMBus driver expects a 7-bit slave address for the device to be passed to it. The 1.0 System Management Bus specification defines the address protocols (how data is passed on the wiggling pins) as:

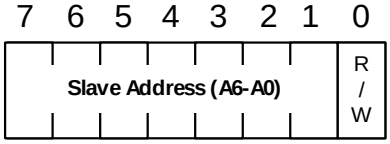


Figure 15-1 SMBus Slave Address Protocol

This indicates that bit 0 of the protocol represents whether this access is a read or write cycle, and the next six bits represent the slave address. Note that the driver expects a zero-based address, not a one-based address. For example, the SBS battery has a slave address of 0xB, or 0001011b (bits 0, 1 and 4 being set). This value is represented by 0x16 for writes or 0x17 for reads to the smart battery in the SMBus protocol format. The protocol format of the slave address and the actual slave address should not be confused as the SMBus driver expects the actual slave address, not the protocol format with the read/write value; the driver will shift the slave address left by 1 bit and mask in the read/write protocol.

15.2.3.4.1.4.2 SMBus Addressing

Associated with each SMBus device is an 8-bit command register that represents an additional address space within the device, allowing up to 256 registers within an SMBus device. For some devices this is treated as a linear address space; for other devices such as the Smart Battery, this is treated as a non-linear address space. The SMBus driver differentiates these types of devices so that it can understand how to use the different SMBus protocols on the device.

A linear address device treats the command and slave address fields as a byte-linear 15-bit address space where the address is formed as follows:

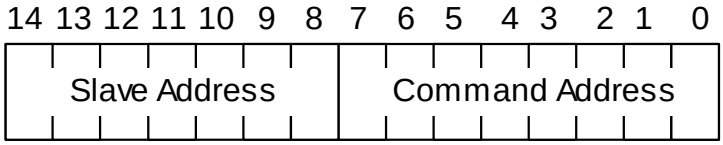


Figure 15-2 SMBus Linear Address Decode

For example an SMBus memory device that consumes slave address 0x40 would be accessing a linear address range of 0x4000-0x40FF (256 bytes of address space). A byte access to 0x4000 (slave 0x40, command 0) would access byte location 0x4000 (slave 0x40, command 0), and a word access to 0x4000 (slave 0x40, command 0) would access byte locations 0x4000-0x4001 (slave 0x40, commands 0-1). For a device that behaves in this manner, ASL should indicate an AnyAcc in the field operator defining the SMBus device. This indicates to the SMBus driver that it can use the read/write block, read/write word, or read/write byte protocols to access this device.

A non-linear address device (such as the smart battery) defines each command value within the device to be a potentially different size. The ACPI driver treats such a device differently from a linear address device by only accessing command values with the specified protocol only. For example the smart battery device has a slave address of 0xB and a definition for the first two command values as follows:

Table 15-10 Example Command Codes from the Smart Battery

Command Address	Data Type	Protocol to Access
0x0	Manufacture Access	Word Read/Write
0x1	Remaining Capacity Alarm	Word Read/Write
0x2	Remaining Time Alarm	Word Read/Write
…		
0x20	Manufacture Name	Block Read/Write
0x21	Device Name	Block Read/Write

The Smart Battery uses a non-linear programming model. Each command register can be a different size and has a specific SMBus protocol associated with it. For example command register 0x0 contains a word of data (which in a linear device would take up two command registers 0 and 1) that represents the “Manufacture Access” and command register 0x1 contains the next word of data (which in a linear device would take up two command registers 0 and 1) that represents the “Remaining Capacity.” In a linear address model these registers would overlap; however, this is legitimate SMBus device definition. As a further example command register 0x20 can

represent up to 32 bytes of data (block read/write) and command register 0x21 also represents up to 32 bytes of data.

15.2.3.4.1.4.3 SMBus Protocols

This section describes the different SMBus protocols and how the SMBus driver treats them. It also gives examples of how to define and then access such devices in ASL.

15.2.3.4.1.4.3.1 Quick Protocol (QuickAcc)

The SMBus Quick protocol does not transfer any data. This protocol is used to control simple devices and consists of the slave address with the R/W bit set high or low. Therefore, two types of Quick commands can be generated: QuickRead with the R/W protocol bit reset LOW or QuickWrite with the R/W protocol bit set HIGH. A device defined to use the quick protocol has no command registers, and consumes the entire 7-bit slave address.

To define a quick device an operation region is generated using the SMBus address type. Next a field is generated in the operation region using the "QuickAcc" access type. To generate a QuickWrite protocol to this device, ASL would generate a write to this field. To generate a QuickRead protocol to this device, ASL would generate a read to this field. Note that even though the ASL read the field and a QuickRead protocol was sent to the device, the device does not return any data and the numeric result returned by the SMB driver to the ASL will be 0. For example,

```
Device(\_SB.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, Zero)
    Device (SMB1) {
        Name(_ADR, "ACPI0001")
        Name(_EC, 0x8030)          // EC offset(0x80), Query (0x30)
        OperationRegion(PHO1, SMBus, 0x61, 0x1)
        Device(DEVA){
            Name(_ADR, 0x61)       // Slave Address 0x61
            Field(PHO1, QuickAcc, NoLock, Preserve) {
                QCKA,  1
            }
        }       // end of DEVA
    }           // end of SMB1
}               // end of EC0
```

This example creates a quick SMBus device residing at slave address 0x61 called "QCKA". Examples of generating the Quick0 and Quick1 commands from ASL is illustrated below:

```
Method(Test){
    Store(1, QCKA)         // Generates a QuickRead command to slave address 0x61
    Store(QCKA, Local0)    // Generates a QuickWrite command to slave address 0x61
}
```

15.2.3.4.1.4.3.2 Send/Receive Command Protocol (SMBSendRecvAcc)

The SMBus Send/Receive protocol transfers a byte of data between the selected SMBus slave address and the ASL code performing a read/write to the field. The SMBus protocol for send-command is defined that the byte being written is presented in the "command" field, while the data returned from a read-command is defined to be the byte in the data field. The SMBus driver will read and write the data to a SMBSendRecvAcc field accordingly.

To define a send/receive command to a device an operation region is generated using the SMBus address type. Next a field is generated in the operation region using the "SMBSendRecvAcc" access type. To generate a send byte protocol to this device, ASL would generate a write to this field. To generate a receive byte protocol to this device, ASL would generate a read to this field. For example,

```
Device(\_SB.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, Zero)
    Device (SMB1) {
        Name(_ADR, "ACPI0001")
        Name(_EC, 0x8030)          // EC offset(0x80), Query (0x30)
        OperationRegion(PHO1, SMBus, 0x62, 0x1)
        Device(DEVB){
            Name(_ADR, 0x62)       // Slave Address 0x62
            Field(PHO1, SMBSendRecvAcc, NoLock, Preserve) {
                TSTA,   1,
                TSTB,   1,
                TSTC,   5
            }
        }       // end of DEVB
    }           // end of SMB1
}           // end of EC0
```

This example creates a send/receive byte SMBus device residing at slave address 0x62. There are three fields that reference this single byte called “TSTA”, “TSTB” and “TSTC”. Examples of generating the send/receive byte protocols from ASL are illustrated below:

```
Method(Test){
    Store(1, TSTA)         // Sets TSTA, preserved TSTB and TSTC, sendbyte
    Store(0, TSTB)         // Clears TSTB, preserved TSTA and TSTC, sendbyte
    Store(0x7, TSTC)       // Sets TSTC to 0111b, preserved TSTA and TSTB, sendbyte
    Store(TSTA, Local0)    // returns 1, receive byte
    Store(TSTB, Local0)    // returns 0, receive byte
    Store(TSTC, Local0)    // returns 7, receive byte
}
```

Read/Write Byte Protocol (*ByteAcc*)

The SMBus Read/Write Byte protocol transfers a byte of data between the selected SMBus slave address and command value. The command address is defined through the use of the AccessAs(*AccessType*, *AccessAttribute*) macro. In this case the *AccessAtrribute* represents the byte aligned command value, and *AccessType* would be set to *ByteAcc*.

To define a ByteAcc device an operation region is generated using the SMBus address type. Next a field is generated in the operation region using the “ByteAcc” access type. In the field list an AccessAs(ByteAcc, *command_value*) macro is used to define what command address is associated with this field. The absence of the macro assume a starting command value of 0. The SMBus driver assumes that after the AccessAs(ByteAcc, *command_value*) macro is declared, the next 8-bits represent this command register. If a field is defined that crosses over this 8-bit boundary, then the SMBus driver assumes this field resides in multiple byte-wide command registers with a command address value of *command_value*+1 (for each new register) using the ByteAcc protocol.

To generate a write byte protocol to this device, ASL would generate a write to this field. To generate a read byte protocol to this device, ASL would generate a read to this field. For example,

```
Device(\_SB.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, Zero)
    Device (SMB1) {
        Name(_ADR, "ACPI0001")
        Name(_EC, 0x8030)          // EC offset(0x80), Query (0x30)
        OperationRegion(PHO1, SMBus, 0x63, 0x1)
        Device(DEVB){
            Name(_ADR, 0x63)       // Slave Address 0x63
            Field(PHO1, ByteAcc, NoLock, Preserve) {
                AccessAs(ByteAcc, 0),
                TSTA,   1,
                TSTB,   1,
                TSTC,   5,
                TSTD,   4       // this field spans command address 0 and 1
            }
        }       // end of DEVB
    }           // end of SMB1
}               // end of EC0
```

This example creates a read/write byte SMBus device residing at slave address 0x63. There are four fields that use two command registers (0 and 1), called “TSTA”, “TSTB”, “TSTC”, and “TSTD”. TSTA, TSTB and TSTC reference command register 0. TSTD references both command registers 0 and 1: bit0 of TSTD represents bit 7 of command register 0, while bits 1-3 of field TSTD represent bits 0-2 of command register 1. Examples of generating the read/write byte protocols from ASL is illustrated below:

```
Method(Test){
    Store(1, TSTA)          // Sets TSTA, preserved TSTB and TSTC, write byte
    Store(0, TSTB)          // Clears TSTB, preserved TSTA and TSTC, write byte
    Store(0x7, TSTC)        // Sets TSTC to 0111b, preserved TSTA and TSTB, write byte
    Store(0xF, TSTD)        // Sets TSTD to 0xF, command registers 0 and 1
    Store(TSTA, Local0)     // returns 1, read byte
    Store(TSTB, Local0)     // returns 0, read byte
    Store(TSTC, Local0)     // returns 7, read byte
    Store(TSTD, Local0)     // returns 0xF from command registers 0 and 1
}
```

15.2.3.4.1.4.3.3 Read/Write Word Protocol (WordAcc)

The SMBus Read/Write Word protocol transfers a word of data between the selected SMBus slave address and command value. The command address is defined through the use of the AccessAs(*AccessType*, *AccessAttribute*) macro. In this case the *AccessAttribute* represents the byte aligned command value, and *AccessType* should be set to *WordAcc*.

To define a WordAcc device an operation region is generated using the SMBus address type. Next a field is generated in the operation region using the “WordAcc” access type. In the field list an AccessAs(WordAcc, *command_value*) macro is used to define what command address is associated with this field. The absence of the macro assume a starting command value of 0. The SMBus driver assumes that after the AccessAs(WordAcc, *command_value*) macro is declared, the next 16-bits represent this command register. If a field is defined that crosses over this 16-bit boundary, then the SMBus driver assumes this field resides in multiple word wide command registers with a command address value of *command_value*+2 (for each new register) using the WordAcc protocol.

To generate a write word protocol to this device, ASL would generate a write to this field. To generate a read word protocol to this device, ASL would generate a read to this field.

15.2.3.4.1.4.3.4 Read/Write Block Protocol (*BlockAcc*)

The SMBus Read/Write Block protocol transfers up to a 32 byte buffer of data between the selected SMBus slave address and command value. The command address is defined through the use of the

AccessAs(*AccessType, AccessAttribute*) macro. In this case the *AccessAttribute* represents the byte aligned command value, and *AccessType* would be set to *BlockAcc*.

To define a BlockAcc device an operation region is generated using the SMBus address type. Next a field is generated in the operation region using the "BlockAcc" access type. In the field list an AccessAs(BlockAcc, *command_value*) macro is used to define what command address is associated with this field. The absence of the macro assume a starting command value of 0. The SMBus driver assumes that after the AccessAs(BlockAcc, *command_value*) macro is declared the command register is 32 bytes or less. Each block field must start on the a *command_value* boundary.

The SMBus driver passes block data to and from ASL through the buffer data type. The buffer is structured such that the byte count of the data to write is in record 0 followed by the buffer data. For example a 5 byte buffer with the contents of 1, 2, 3, 4 would be generated as:

```
Buffer(5){4, 1, 2, 3, 4}
```

Where the length of the buffer is its byte data width plus 1, and the first entry is the length of data (buffer length minus 1). On reads, ASL will return a buffer with the first entry set to the number of data bytes returned. For example,

```
Device(\_SB.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, Zero)

    Device (SMB1) {
        Name(_ADR, "ACPI0001")
        Name(_EC, 0x8030)          // EC offset(0x80), Query (0x30)
        OperationRegion(PHO1, SMBus, 0x65, 0x1)
        Device(DEVB){
            Name(_ADR, 0x65)       // Slave Address 0x65
            Field(PHO1, BlockAcc, NoLock, Preserve) {
                AccessAs(BlockAcc, 0),
                FLD1,   128,
                AccessAs(BlockAcc, 0x10),
                FLD2,   32
            }
        }       // end of DEVB
    }           // end of SMB1
}               // end of EC0
```

This example creates a read/write block SMBus device residing at slave address 0x65. There are two fields that use two command registers (0 and 0x10), called "FLD1", and "FLD2". Examples of generating the read/write block protocols from ASL is illustrated below:

```
Method(Test){
    Name(BUF1, Buffer(){8, 1, 2, 3, 4, 5, 6, 7, 8}   // 8 is the number of bytes
    Name(BUF2, Buffer(){4, 9, 10, 11, 12}            // 4 is the number of bytes
    Store(BUF1, FLD1)      // Sets FLD1 SMBus device block register
    Store(BUF2, FLD2)      // Sets FLD2 SMBus device block register
    Store(FLD1, Local0)    // local0 contains buf: 8,1,2,3,4,5,6,7,8
    Store(FLD2, Local0)    // local0 contains buf: 4,9,10,11,12
}
```

15.2.3.4.1.4.3.5 SMBus Memory Devices (AnyAcc)

The AnyAcc access type allows any of the Read/Write byte, word or Block protocol transfers to be made to the selected SMBus slave address and command value. The combined slave and command value generates a single byte granular address space. The command address (A0-A7 of the 15-bit address) is defined through the use of the AccessAs(*AccessType, AccessAtrribute*) macro. In this case the *AccessAttribute* represents the byte aligned command value, and *AccessType* would be set to *AnyAcc*.

To define a AnyAcc device an operation region is generated using the SMBus address type. Next a field is generated in the operation region using the "AnyAcc" access type. In the field list an AccessAs(AnyAcc, *command_value*) macro is used to define what command address is associated with this field. The absence of the macro assume a starting command value of 0. The SMBus driver assumes that after the AccessAs(*AnyAcc, command_value)* macro is declared then command registers are byte-granular and linear. If a field is defined that crosses over a byte boundary, then the SMBus driver assumes this field resides in multiple command registers with a command address value of *command_valuet*+1 (for each new register). The SMBus driver will use the most appropriate protocol for accessing the registers associated with the fields. For example, if a field spans more than three bytes a read/write block protocol access can be made, while if only spanning a byte then the read/write byte protocol can be used.

For example, a 5-byte buffer with the contents of "ACPI" would be generated as:

```
Buffer(){"ACPI"}
```

On reads, ASL will return a buffer with the first entry set to the number of data bytes returned. For example,

```
Device(\_SB.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, Zero)
    Device (SMB1) {
        Name(_ADR, "ACPI0001")
        Name(_EC, 0x8030)          // EC offset(0x80), Query (0x30)
        OperationRegion(PHO1, SMBus, 0x66, 0x1)
        Device(DEVB){
            Name(_ADR, 0x66)       // Slave Address 0x66
            Field(PHO1, AnyAcc, NoLock, Preserve) {
                FLD1,   512,
                FLD2,   256,
                FLD3,   32,
                FLD4,   16,
                FLD5,   8
            }
        }       // end of DEVB
    }           // end of SMB1
}               // end of EC0
```

This definition creates a linear SMBus device residing at slave address 0x66. There are six fields that use 102 command registers (0-101), called "FLD1", "FLD2" , "FLD3", "FLD4" and "FLD5". FLD1 references command registers 0-63 (first 64 bytes) and will be accessed by the block protocol (data is over 3 bytes). FLD2 represents command registers 64-95 (next 32 bytes) and will be accessed by the block command protocol (data is over 3 bytes). FLD3 represents command registers 96-99 (next four bytes) and will be accessed by the block command protocol (data is over 3 bytes). FLD4 represents command registers 100-101 (next two bytes) and will be accessed by the word command protocol. FLD5 represents command register 102 (next byte) and will be accessed by the byte command protocol. Examples of generating the accesses from ASL is illustrated below:

```
Method(Test){
    Name(BUF1, Buffer(){"Hannibal"}
    Name(BUF2, Buffer(){"Scipio Africanus"}
    Name(BUF3, Buffer(){"Zama"}
    Store(BUF1, FLD1)      // writes "Hannibal" to linear addresses for FLD1
    Store(BUF2, FLD2)      // writes "Scipio Africanus" to linear addresses for FLD2
    Store(BUF3, FLD3)      // writes "Zama" to linear addresses for FLD3
    Store(0xFF12, FLD4)    // sets FLD4 to 0xFF12
    Store(0xEF, FLD5)      // sets FLD5 to 0xEF
    Store(FLD1, Local0)    // local0 contains 64 byte buffer with: "Hannibal",0,…
    Store(FLD2, Local0)    // local0 contains 32 byte buffer with: "Scipio Africanus",0,…
    Store(FLD3, Local0)    // local0 contains 4 bytes: "Zama"
    Store(FLD4, Local0)    // local0 contains 2 bytes: 0xFF12
    Store(FLD5, Local0)    // local0 contains 1 byte: 0xEF
}
```

15.2.3.4.1.4.3.6 Mixed Example (AnyAcc)

Some devices can be accessed through multiple protocols. This section gives an example of such a device.

```
Device(\SB._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, Zero)
    Device (SMB1) {
        Name(_ADR, "ACPI0001")
        Name(_EC, 0x8030)          // EC offset(0x80), Query (0x30)
        OperationRegion(PHO1, SMBus, 0x67, 0x1)
        Device(DEVB){
            Name(_ADR, 0x67)       // Slave Address 0x67
            Field(PHO1, ByteAcc, NoLock, Preserve) {
                AccessAs(AnyAcc, 0),
                FLD1,   512,
                FLD2,   256,
                FLD3,   32,
                AccessAs(WordAcc, 0x70),
                FLD4,   16,
                AccessAs(ByteAcc, 0x80),
                FLD5,   8
            }
        }       // end of DEVB
    }           // end of SMB1
}               // end of EC0
```

This definition creates an SMBus device using various protocols residing at slave address 0x67. There are three fields that use four command registers (0, 1, 2 and 3), called "FLD1", "FLD2" and "FLD3". FLD1 references command registers 0-1 (32 bytes per command register) and will be accessed by the byte, word and block linear protocols. FLD2 represents command register 064 and will be accessed by the byte, word and block linear protocols. FLD3 represents command register 96 and will be accessed by the byte, word and block linear protocols. FLD4 represents command register 0x70 and will be accessed by the word command protocol. FLD5 represents command register 0x80 and will be accessed by the byte command protocol.

15.2.3.4.1.5 IndexField - Declare Index/Data Fields

```
IndexFieldTerm :=       IndexField(
                            IndexName,              //NameString
                            DataName,               //NameString
                            AccessType,             //AccessTypeKeyword
                            LockRule,               //LockRuleKeyword
                            UpdateRule              //UpdateRuleKeyword
                            )
                            {FieldList}
```

Creates a data object at load time. The contents of the created object are accessed by an index/data-style reference to *BaseName* and *DataName*. **IndexField** opens a name scope.

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 *data* object by using an *index* aligned on an *AccessType* boundary, with synchronization occurring on the operation region which 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)
}
```

15.2.3.4.1.6 Method - Declare Control Method

```
MethodTerm :=          Method(
                           MethodName,              //NameString
                           ArgCount,                //ByteConst
                           SerializeRule            //SerializeRuleKeyword
                           )
                           {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.

The current name space location used during name creation is adjusted to be the current location on the name space tree. Any names created within this scope are “below” the name of this package. The current name space location is assigned to the method package, and all name space references that occur during control method execution for this package are relative to that location.

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
    }
```

15.2.3.4.1.7 Mutex - Declare Synchronization / Mutex Object

```
MutexTerm :=            Mutex(
                                MutexName,              //NameString
                                SyncLevel               //ByteConst
                                )
```

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* check is not performed on a Mutex when the ownership count is nesting.

15.2.3.4.1.8 OperationRegion - Declare Operation Region

```
OperationRegionTerm :=  OperationRegion(
                                RegionName,             //NameString
                                RegionSpace,            //RegionSpaceKeywd
                                Offset,                 //OpCode=>DWord
                                Length                  //OpCode=>DWord
                                )
```

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 IO space. Any reads or writes to the named object will results in accesses to the IO 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.

In general, Operation Regions have “virtual content” and are only accessible via **Field** objects. If the length of the Operation Region is equal to or less than 4 bytes long, control methods may directly access the entire region as a data value. Operation Region objects may be defined down to actual bit controls using **Field** data object definitions. The actual bit content of a **Field** are bits from within a larger **Buffer** that are normalized for that field (i.e., shifted down and masked to the proper length), and as such the data type of a **Field** is **Buffer**. Therefore **Fields** which 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). Note 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.

The following example ASL code shows the use of **OperationRegion** combined with Field to describe IDE 0 and 1 controlled through general IO 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#
}
```

15.2.3.4.1.9 PowerResource - Declare Power Resource

```
PowerResourceTerm :=    PowerResource(
                            ResourceName,           //NameString
                            SystemLevel,            //ByteConst
                            ResourceOrder           //WordConst
                            )
                            {NamedObjectList}
```

Declares a power resource. **PowerResource** opens a name scope.
For a definition of the **PowerResource** term, see section 7.1.

15.2.3.4.1.10 Processor - Declare Processor

```
ProcessorTerm :=        Processor(
                            ProcessorName,          //NameString
                            ProcessorID,            //ByteConst
                            PBlockAddress,          //ByteConst
                            PBlockLength            //DWordConst
                            )
                            {NamedObjectList}
```

Declares a named processor object. **Processor** opens a name scope. Each processor is required to have a unique *ProcessorID* value from any other *ProcessorID* value.

The ACPI BIOS declares one processor object per processor in the system under the _PR name space. *PBlockAddress* provides the system IO 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 which is either 0 (for no P_BLK) or 0x20. 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,      // name space name
    1,
    0x120,          // PBlk system IO address
    0x20            // PBlkLen
    )
    { }
```

15.2.3.4.1.11 ThermalZone - Declare Thermal Zone

```
ThermalZoneTerm :=      ThermalZone(
                            ZoneName                //NameString
                            )
                            {NamedObjectList}
```

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 *ZoneName*.

For sample ASL code that uses a ThermalZone statement, see section 12.4.

15.2.3.4.2 Name Space Modifier Terms

The name space modifiers are as follows:

Table 15-11 Name Space 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 name space when the definition block that contains the Scope statement is loaded.

15.2.3.4.2.1 Alias - Declare Name Alias

```
AliasTerm :=            Alias   (
                                Source,                 //NameString
                                Destination             //NameString
                                )
```

Creates a new name, *Destination*, which refers to and acts exactly the same as *Source.*
At load time, *Destination* is created as an alias of *Source* in the name space. The *Source* name must already exist in the name space. If the alias is to a name within the same definition block the *Source* 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)
```

15.2.3.4.2.2 Name - Declare Named Object

```
NameTerm :=             Name    (
                                ObjectName,             //NameString
                                Target                  //DataObjectTerm
                                )
```

Attaches *Target* to *ObjectName* in the Global ACPI name space.

This encoding is a load-time encoding that creates *ObjectName* in the name space, which references the *Target* object.

The following example creates the name PTTX in the root of the name space 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 name space that references an integer data object with the value 5.

```
Name(\CNT, 5)
```

15.2.3.4.2.3 Scope - Declare Name Scope

```
ScopeTerm :=            Scope   (
                                Location    //NameString
                                )
                                {TermList}
```

Gives a base scope to a collection of objects at load time. All object names defined within the scope act relative to *Location*. Note that *Location* does not have to be below the surrounding scope. Note also that the **Scope** term does not create objects, but only locates objects in the name space at load time; the located objects are created by other ASL terms.

The **Scope** term alters the current name space location to *Location.* This causes the defined objects within *TermList* to occur relative to the new location in the name space.

The following example ASL code

```
Scope(\PCI0) {
    Name(X, 3)
    Scope(\) {
        Method(_RQ) { Return(0) }
    }
    Name(^Y, 4)
}
```

places the defined objects in ACPI name space as shown in the following:

```
\PCI0.X
\_RQ
\Y
```

15.2.3.5 Operator Terms

There are two types of ASL operator terms: Type 1 operators and Type 2 operators.

- A Type1 operator 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 operator term can be used in an expression; when used in an expression the argument that names the object in which to store the result can be optional.

Note that in the operator 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** operator.

15.2.3.5.1 Type 1 Operators

```
Type1OpCode := BreakTerm | BreakPointTerm | CreateBitFieldTerm |
               CreateByteFieldTerm | CreateDWordFieldTerm |
               CreateFieldTerm | CreateWordFieldTerm |
               ElseTerm | IfTerm | FatalTerm | NoOpTerm | NotifyTerm |
               ReleaseTerm | ResetTerm | ReturnTerm | SignalTerm |
               SleepTerm | StallTerm | UnloadTerm |
               WhileTerm
```

The Type 1 operators are listed in the following table.

Table 15-12 Type 1 Operators

ASL Statement	Description
Break	Stop executing the current code package at this point
BreakPoint	Used for debugging. Stops execution in the debugger
CreateBitField	Create a bit field
CreateByteField	Create a byte field
CreateDwordField	Create a Dword field
CreateField	Create a field
CreateWordField	Create a word field
Else	Else
Fatal	Fatal check
If	If
Noop	No operation
Notify	Notify the OS that a specified notification value for a NotifyObject has occurred
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

ASL Statement	Description
Sleep	Sleep n milliseconds (yields the processor)
Stall	Delay n microseconds (does not yield the processor)
Unload	Unload differentiating definition block
While	While

15.2.3.5.1.1 Break - Break

```
BreakTerm :=            Break
```

The break operation causes the current package execution to complete.

15.2.3.5.1.2 BreakPoint - BreakPoint

```
BreakPointTerm :=       BreakPoint
```

Used for debugging. Stops execution in the debugger. In the retail version of the interpreter, **BreakPoint** is equivalent to **Noop**.

15.2.3.5.1.3 CreateBitField

```
CreateBitFieldTerm :=   CreateBitField(
                            Source,         //OpCode=>Buffer
                            BitIndex,       //OpCode=>Integer
                            Destination     //SuperName
                            )
```

Source is evaluated as a buffer. *Index* is evaluated as an integer. A new field object is created for the bit of *Source* at the bit index of *Index*, and a reference is Stored into *Destination*. The bit-defined field within *Source* must exist.

15.2.3.5.1.4 CreateByteField

```
CreateByteFieldTerm :=  CreateByteField(
                            Source,         //OpCode=>Buffer
                            ByteIndex,      //OpCode=>Integer
                            Destination     //SuperName
                            )
```

Source is evaluated as a buffer. *Index* is evaluated as an integer. A new field object is created for the byte of *Source* at the byte index of *Index*, and a reference is Stored into *Destination*. The byte-defined field within *Source* must exist.

15.2.3.5.1.5 CreateDWordField

```
CreateDwordFieldTerm :=  CreateDwordField(
                            Source,         //OpCode=>buffer
                            ByteIndex,      //OpCode=>Integer
                            Destination     //SuperName
                            )
```

Source is evaluated as a buffer. *Index* is evaluated as an integer. A new field object is created for the Dword of *Source* at the Dword index of *Index*, and a reference is Stored into *Destination*. The Dword-defined field within *Source* must exist.

15.2.3.5.1.6 CreateField - Field

```
CreateFieldTerm :=      CreateField(
                            Source,         //OpCode=>Buffer
                            Offset,         //OpCode=>Integer
                            NumBits,        //OpCode=>Integer
                            Destination     //SuperName
                            )
```

Source is evaluated as a buffer and *Offset* and *NoBits* are evaluated as integers. A new field object is created for the bits of *Source* at *Offset* for *NoBits*, and a reference is Stored into *Destination*. The entire bit range of the defined field within *Source* must exist.

15.2.3.5.1.7 CreateWordField

```
CreateWordFieldTerm :=    CreateWordField(
                                Source,             //OpCode
                                ByteIndex,          //OpCode
                                Destination         //SuperName
                                )
```

Source is evaluated as a buffer. *Index* is evaluated as an integer. A new field object is created for the word of *Source* at the word index of *Index*, and a reference is Stored into *Destination*. The word-defined field within *Source* must exist.

15.2.3.5.1.8 Else - Else Operator

```
ElseTerm :=               Else
                                {False}             //TermList
```

In an **If** term, if *Predicate* evaluates to 0, it is false, and the term list in *False* of the **Else** term is executed. If *Predicate* evaluates to Not 0 if the **If** term, then it is considered true, and the term list in *False* of the **Else** term is not executed. The execution result from an **Else** operation in undefined.

The following example checks Local0 to be zero or non-zero. On non-zero, CNT is incremented; otherwise, CNT is decremented.

```
If (Local0) {
    Increment (CNT)
} Else {
    Decrement (CNT)
}
```

15.2.3.5.1.9 Fatal - Fatal Check

```
FatalTerm :=              Fatal(
                                Type,               //ByteConst
                                Code,               //DWordConst
                                Arg                 //OpCode=>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.

15.2.3.5.1.10 If - If Operator

```
IfTerm :=                 If(
                                Predicate           //OpCode=>Integer
                                )
                                {True}              //TermList
```

Predicate is evaluated as an integer. If the integer is non-zero, the term list in *True* is executed. The execution result from an **If** operation is undefined.

The following examples all check for bit 3 in **Local0** being set, and clear it if set.

```
// example 1
if (And(Local0, 4)) {
    NAnd (Local0, 4, Local0)
}
// example 2
Store(4, Local2)
if (And(Local0, Local2)) {
    NAnd (Local0, Local2, Local0)
}
// example 3
CreateBitField(Local0, 3, Local2)
if (Local2) {
    Store (Zero, Local2)
}
```

15.2.3.5.1.11 Noop Code - No Operation

```
NoopTerm :=               Noop
```

This operation has no effect.

15.2.3.5.1.12 Notify - Notify

```
NotifyTerm :=           Notify(
                                NotifyObject,       //SuperName
                                NotificationValue   //OpCode=>Byte
                                )
```

Notifies the OS that the *NotificationValue* for the *NotifyObject* has occurred. *NotifyObject* must be a reference to a device or thermal zone object.
Notification values are determined by the *NotifyObject* type. For example, the notify values for a thermal zone object are different than the notify 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.

15.2.3.5.1.13 Release - Release a Mutex Synchronization Object

```
ReleaseTerm :=          Release(
                                SynchObject         //SuperName
                                )
```

SynchObject must be a reference to a synchronization object. If the synchronization object is a Mutex and it 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.

15.2.3.5.1.14 Reset - Reset an Event Synchronization Object

```
ResetTerm :=
                        Reset(
                                SynchObject         //SuperName
                                )
```

SynchObject must be a reference to 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.

15.2.3.5.1.15 Return - Return

```
ReturnTerm :=           Return(
                                Arg                 //OpCode=>Object
                                )
```

Returns control to the invoking control method, optionally returning an object named in *Arg*.

15.2.3.5.1.16 Signal - Signal a Synchronization Event

```
SignalTerm :=           Signal(
                                SynchObject         //SuperName
                                )
```

SynchObject must be a reference to an Event synchronization object. The Event object is signaled once, allowing one invocation to acquire the event.

15.2.3.5.1.17 Sleep - Sleep

```
SleepTerm :=            Sleep(
                                Millisecond         //OpCode=>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.

15.2.3.5.1.18 Stall - Stall for a Short Time

```
StallTerm :=            Stall(
                                Microseconds        //OpCode=>Byte
                                )
```

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**.

15.2.3.5.1.19 Unload - Unload Differentiated Definition Block

```
UnloadTerm :=           Unload(
                                NameSpaceObjectRef //NameString
                        )
```

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 name space objects created as a result of the corresponding Load operation will be removed from the name space).

15.2.3.5.1.20 While - While

```
WhileTerm :=            While(
                                Predicate)          //OpCode
                                {True}              //TermList
```

Predicate is evaluated as an integer. If the integer is non-zero, the list of terms in *True* is executed. The operation repeats until the *Predicate* evaluates to zero. The execution result from an **While** term is undefined.

15.2.3.5.2 Type 2 Operators

```
Type2OpCode := AcquireTerm | AddTerm | AndTerm | ConcatenateTerm |
               CondRefOfTerm | DecrementTerm | DivideTerm |
               FindSetLeftBitTerm |
               FindSetRightBitTerm | FromBCDTerm | IncrementTerm |
               IndexTerm | LAndTerm | LEqualTerm | LGreaterTerm |
               LGreaterEqualTerm | LLessTerm | LLessEqualTerm |
               LNotTerm | LNotEqualTerm | LoadTerm | MatchTerm |
               MultiplyTerm | NAndTerm | NorTerm | NotTerm |
               ObjectTypeTerm | OrTerm | RefOfTerm | ShiftLeftTerm |
               ShiftRightTerm | SizeOfTerm | StoreTerm |
               SubtractTerm | ToBCDTerm | WaitTerm | XOrTerm
```

The ASL terms for Type 2 Operators are listed in the following table.

Table 15-13 Type 2 Operators

ASL Statement	Description
Acquire	Acquire a synchronization object
Add	Add two values

ASL Statement	Description
And	Bitwise And
Concatenate	Concatenate two strings
CondRefOf	Conditional reference to an object
Decrement	Decrement a value.
Divide	Divide
FindSetLeftBit	Index of first set Lsb
FindSetRightBit	Index of first set Msb
FromBCD	Convert from BCD to numeric
Increment	Increment a value
Index	Reference the nth element of a package
LAnd	Logical And
LEqual	Logical Equal
LGreater	Logical Greater
LGreaterEqual	Logical Not less
LLess	Logical Less
LLessEqual	Logical Not greater
LNot	Logical Not
LNotEqual	Logical Not equal
Load	Load differentiating definition block
LOr	Logical Or
Match	Search for match in package array
Multiply	Multiply
NAnd	Bitwise Nand
NOr	Bitwise Nor
Not	Bitwise Not
ObjectType	Type of object
Or	Bitwise Or
RefOf	Reference to an object
SizeOf	Get the size of a buffer, string, or package
ShiftLeft	Shift value left
ShiftRight	Shift value right
Store	Store value
Subtract	Subtract values
ToBCD	Convert numeric to BCD
Wait	Wait
XOr	Bitwise Xor

15.2.3.5.2.1 Acquire - Acquire a Mutex Synchronization Object

```
AcquireTerm :=          Zero | Ones <= //Ones means timed-out
                            Acquire(
                                    SynchObject,        //SuperName
                                    TimeOut             //WordConst
                                    )
```

SynchObject refers to the mutex to be acquired. *SynchObject* must be a reference to a Mutex synchronization object.

Ownership of the referenced 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 *TimeOut* ms have elapsed. A Mutex can be acquired more than once by the same invocation.

This operation returns a non-zero value if a *TimeOut* occurred and the mutex ownership was not acquired. A *TimeOut* of 0xFFFF indicates that there is no time out and the operation will wait indefinitely.

15.2.3.5.2.2 Add - Add

```
AddTerm :=              Integer <=
                            Add(
                                    Addend1,            //OpCode=>Integer
                                    Addend2,            //OpCode=>Integer
                                    Result              //ResultName
                                    )
```

Addend1 and *Addend2* are evaluated as integer data types and are added, and the result is **Stored** into *Result.* Overflow conditions are ignored.

15.2.3.5.2.3 And - Bitwise And

```
AndTerm :=              Integer <=
                            And(
                                    Source1,            //OpCode=>Integer
                                    Source2,            //OpCode=>Integer
                                    Result              //ResultName
                                    )
```

Source1 and *Source2* are evaluated as integer data types, a bit-wise *and* is performed, and the result is **Stored** in *Result*.

15.2.3.5.2.4 Concatenate - Concatenate

```
ConcatenateTerm :=      Integer | String | Buffer <=
                            Concatenate(
                                    Source1,            //OpCode
                                    Source2,            //OpCode
                                    Destination         //SuperName
                                    )
```

Source1 and *Source2* are evaluated. *Source2* is converted to the data type of *Source1,* and then concatenated based on the following rules:

Data Type	Description
Integer	Treated as type buffer of 32 bits.
Buffer	Results in a buffer of size (Source1.buffersize + Source2.buffersize), where the bits from Source2 are concatenated to the bits from Source1.
String	Results in a string where the characters from Source2 are concatenated to the characters (less NullChar) from Source1.

Source1 and *Source2* must be of the same data type (that is, both integers, both strings, or both buffers).

15.2.3.5.2.5 CondRefOf - Conditional Reference Of

```
CondRefOfTerm :=        Ones | Zero <=
                            CondRefOf(
                                    Source              //SuperName
                                    Destination         //SuperName
                                    )
```

Attempts to set *Destination* to refer to *Source*. The source of this operation can be any object type (e.g., data package, device object, etc.). On success, the *Destination* object is set to refer to *Source* and the execution result of this operation is the constant **Ones** object. On failure the execution result of this operation is the constant **Zero** object and the *Destination* object is unchanged. This can be used to reference items in the name space which may appear dynamically (e.g., 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**.

15.2.3.5.2.6 Decrement - Decrement

```
DecrementTerm :=        Integer <=
                            Decrement(
                                    Addend              //SuperName
                                    )
```

Same as **Add**(*Addend*, -1, *Addend*)

15.2.3.5.2.7 Divide - Divide

```
DivideTerm :=          Integer <=     //returns Result
                           Divide(
                                  Dividend,          //OpCode=>Integer
                                  Divisor,           //OpCode=>Integer
                                  Remainder,         //ResultName
                                  Result             //ResultName
                                  )
```

Dividend and *Divisor* are evaluated as integer data. *Dividend* is divided by *Divisor*, then the resulting remainder is Stored into *Remainder* and the resulting quotient is Stored into *Result*. Divide by zero exceptions are fatal.

15.2.3.5.2.8 FindSetLeftBit - Find Set Left Bit

```
FindSetLeftBitTerm :=  ByteConst <=
                           FindSetLeftBit(
                                  Source,            //OpCode=>Integer
                                  BitNo              //SuperName
                                  )
```

Source is evaluated as buffer data type, and the one-based bit location of the first LSb (least significant bit) is stored in *BitNo*. 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.

15.2.3.5.2.9 FindSetRightBit - Find Set Right Bit

```
FindSetRightBitTerm := ByteConst <=
                           FindSetRightBit(
                                  Source,            //OpCode=>Integer
                                  BitNo              //SuperName
                                  )
```

Source is evaluated as buffer data type, and the one-based bit location of the most least significant set bit is **Stored** in *BitNo.* The result of 0 means no bit was set, 32 means the first bit set is the 32nd bit, 31 means the first bit set is the 31st bit, and so on.

15.2.3.5.2.10 FromBCD - Convert from BCD

```
FromBCDTerm :=         Integer <=
                           FromBCD(
                                  BcdValue,          //OpCode=>Integer
                                  Destination        //ResultName
                                  )
```

The FromBCD operation is used to convert *BcdValue* to a numeric format and store the numeric value in *Destination*.

15.2.3.5.2.11 Increment - Increment

```
IncrementTerm :=       Integer <=
                           Increment(
                                  Addend             //SuperName
                                  )
```

Equivalent to **Add**(*Addend*, 1, *Addend*)

15.2.3.5.2.12 Index - Index

```
IndexTerm :=           PackageElement <=
                           Index(
                                  Source,            //OpCode=>PackageObject
                                  Index,             //OpCode=>Integer
                                  Destination        //NameString
                                  )
```

Source is evaluated and must be a package. *Index* is evaluated as an integer. The object at *Index* within *Source* is stored as a reference into *Destination.* For example:

```
Name(P1,
    Package () {
        Package () { 1, 2, 3 },
        Package () { 4, 5, 6 }
    }
)

Example 1:
Store(Index(P1, 1),  temp)        // temp refers to the first package
Store(Index(temp, 2), temp2)      // temp2 now refers to the  "2" entry
Store(52, temp2)                  // 2 is now 52

Example 2:
Store(52, Index(Index(P1,1),2))   // 2 is now 52
```

15.2.3.5.2.13 LAnd - Logical And

```
LAndTerm :=           Ones | Zero <=
                          LAnd(
                                  Source1,          //OpCode=>Integer
                                  Source2           //OpCode=>Integer
                                  )
```

Source1 and *source2* are evaluated as integers. If both values are non-zero, the **Ones** constant object is returned, otherwise the **Zero** constant object is returned.

15.2.3.5.2.14 LEqual - Logical Equal

```
LEqualTerm :=         Ones | Zero <=     //Ones means equal
                          LEqual(
                                  Source1,          //OpCode=>Integer
                                  Source2           //OpCode=>Integer
                                  )
```

Source1 and *Source2* are evaluated as buffers. If the buffers are equal, the **Ones** constant object is returned; otherwise, the **Zero** constant object is returned.

15.2.3.5.2.15 LGreater - Logical Greater

```
LGreaterTerm :=       Ones | Zero <=     //Ones means S1 > S2
                          LGreater(
                                  Source1,          //OpCode=>Integer
                                  Source2           //OpCode=>Integer
                                  )
```

Source1 and *Source2* are evaluated as integers. If *Source1* is greater than *Source2,* the **Ones** constant object is returned; otherwise, the **Zero** constant object is returned.

15.2.3.5.2.16 LGreaterEqual - Logical Greater Than Or Equal

```
LGreaterEqualTerm:=   Ones | Zero  <=    //Ones means S1 >= S2
                          LGreaterEqual(
                                  Source1,          //OpCode=>Integer
                                  Source2           //OpCode=>Integer
                                  )
```

Source1 and *Source2* are evaluated as integers. If *Source1* is greater than or equal to *Source2,* the **Ones** constant object is returned; otherwise, the **Zero** constant object is returned.

15.2.3.5.2.17 LLess - Logical Less

```
LLessTerm :=          Ones | Zero <=     //Ones means S1 < S2
                          LLess(
                                  Source1,          //OpCode=>Integer
                                  Source2           //OpCode=>Integer
                                  )
```

Source1 and *Source2* are evaluated as integers. If *Source1* is less than *Source2,* the **Ones** constant object is returned; otherwise, the **Zero** constant object is returned.

15.2.3.5.2.18 LLessEqual - Logical Less Than Or Equal

```
LLessEqualTerm :=       Ones | Zero <=     //Ones means S1 <= S2
                            LLessEqual(
                                    Source1,            //OpCode=>Integer
                                    Source2             //OpCode=>Integer
                                    )
```

Source1 and *Source2* are evaluated as integers. If *Source1* is less than or equal to *Source2*, then the **Ones** constant object is returned; otherwise, the **Zero** constant object is returned.

15.2.3.5.2.19 LNot - Logical Not

```
LNotTerm :=             Ones | Zero <=
                            LNot(
                                    Source              //OpCode=>Integer
                                    )
```

Source1 is evaluated as buffer. If the buffer is non-zero, the **Zero** constant object is returned; otherwise, the **Ones** constant object is returned.

15.2.3.5.2.20 LNotEqual - Logical Not Equal

```
LNotEqualTerm :=        Ones | Zero <=     //Ones means S1 <> S2
                            LNotEqual(
                                    Source1,            //OpCode=>Integer
                                    Source2             //OpCode=>Integer
                                    )
```

Source1 and *Source2* are evaluated as integers. If *Source1* is not equal to *Source2*, then the **Ones** constant object is returned; otherwise, the **Zero** constant object is returned.

15.2.3.5.2.21 Load - Load Differentiated Definition Block

```
LoadTerm :=             DwordConst <=  // Name space object reference
                            Load(
                                    RegionName          //NameString
                                    )
```

Performs a run time load of a Definition Block. The Region parameter is the buffer that contains a DESCRIPTION_HEADER of type SSDT or PSDT. This table is read into memory, the checksum is verified, and then it is loaded into the ACPI name space.

The **Load** operation is used to perform a run time load of a Definition Block. **Region** is the buffer that contains a DESCRIPTION_HEADER of type SSDT or PSDT. This table is read into memory, the checksum is verified, and then it is loaded into the ACPI name space. 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 execution result of this operator is a object reference that can be used to unload the Definition Block at a future date. The Definition Block must be totally contained within the supplied operational region.

The default name space location to load the Definition Block is relative to the current name space. The new Definition Block can override this by specifying absolute names or by adjusting the name space 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.

15.2.3.5.2.22 LOr - Logical Or

```
LOrTerm :=              Integer <=
                            LOr(
                                    Source1,            //OpCode=>Integer
                                    Source2             //OpCode=>Integer
                                    )
```

Source1 and *Source2* are evaluated as integers. If either values is non-zero, the **Ones** constant object is returned; otherwise, the **Zero** constant object is returned.

15.2.3.5.2.23 Match - Find Object Match

```
MatchTerm :=          Integer | Ones <=
                      Match(
                            SearchPackage,     //OpCode=>Package
                            Op1,               //MatchOpKeyword
                            V1,                //OpCode=>Object
                            Op2,               //MatchOpKeyword
                            V2,                //OpCode=>Object
                            Start              //OpCode=>Object
                            )
```

SearchPackage is treated as a one-dimension array. A comparison is performed for each element of the package/array, starting with the index value indicated by *Start* (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 V1) and (P[i] Op2 V2) then Match -> i is returned.

If the comparison succeeds, the index of the element that succeeded is returned; otherwise, the **Ones** object is returned.

Op1 and *Op2* have the following values and meanings listed in the following table.

Table 15-14 Match Term Operator Meanings

Operator	Encoding	Macro
TRUE - a don't care, always returns TRUE	0	**MTR**
EQ - returns TRUE if P[i] == V	1	**MEQ**
LE - returns TRUE if P[i] <= V	2	**MLE**
LT - returns TRUE if P[i] < V	3	**MLT**
GE - returns TRUE if P[i] >= V	4	**MGE**
GT - returns TRUE if P[i] > V	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

// some examples with defaults

// Find entry GT 1984, starting with element #3 (Op2 defaults to MTR, V2 defaults to 0)
Match(P1, MGT, 1984,,,3)

// Find the first entry GE some value (Op2, V2, Start default to MTR, 0, 0)
Match(P1, MGE, FindValue)
```

15.2.3.5.2.24 Multiply - Multiply

```
MultiplyTerm :=       Integer <=
                      Multiply(
                            Multiplcand,       //OpCode=>Integer
                            Multiplier,        //OpCode=>Integer
                            Result             //ResultName
                            )
```

Multiplicand and *Multiplier* are evaluated as integer data types. *Multiplicand* is multiplied by *Multiplier,* and the result is **Stored** into *Result.* Overflow conditions are ignored.

15.2.3.5.2.25 NAnd - Bit-wise NAnd

```
NAndTerm :=             Integer <=
                            NAnd(
                                    Source1,            //OpCode=>Integer
                                    Source2,            //OpCode=>Integer
                                    Result              //ResultName
                                    )
```

Source1 and *Source2* are evaluated as integer data types, a bit-wise *nand* is performed, and the result is **Stored** in *Result.*

15.2.3.5.2.26 NOr - Bitwise NOr

```
NOrTerm :=              Integer <=
                            NOr(
                                    Source1,            //OpCode=>Integer
                                    Source2,            //OpCode->Integer
                                    Result              //ResultName
                                    )
```

Source1 and *Source2* are evaluated as integer data types, a bit-wise *nor* is performed, and the result is **Stored** in *Result.*

15.2.3.5.2.27 Not - Not

```
NotTerm :=              Integer <=
                            Not(
                                    Source1,            //OpCode=>Integer
                                    Result              //ResultName
                                    )
```

Source1 is evaluated as an integer data type, a bit-wise *not* is performed ,and the result is stored in *Result*.

15.2.3.5.2.28 ObjectType - Object Type

```
ObjectTypeTerm :=       ByteConst <=
                            ObjectType(
                                    Object              //SuperName
                                    )
```

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 the following table.

Table 15-15 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	Alias
>14	Other

15.2.3.5.2.29 Or - Bit-wise Or

```
OrTerm :=           Integer <=
                        Or(
                                Source1,        //OpCode=>Integer
                                Source2,        //OpCode=>Integer
                                Result          //ResultName
                                )
```

Source1 and *Source2* are evaluated as integer data types, a bit-wide *or* is performed, and the result is **Stored** in *Result.*

15.2.3.5.2.30 RefOf - Reference Of

```
RefOfTerm :=        ObjectReference <=
                        RefOf(
                                Object          //SuperName
                                )
```

Returns a reference to *Source*. The source of this operation can be any object type (for example, a package, a device object, and so on).
If the *Source* object does not exist, the result of a **RefOf** operation is fatal. Use the **CondRefOf** term in cases where the *Source* object might not exist.

15.2.3.5.2.31 ShiftLeft - Shift Left

```
ShiftLeftTerm :=    Integer <=
                        ShiftLeft(
                                Source,         //OpCode=>Integer
                                Count           //OpCode=>Integer
                                Result          //ResultName
                                )
```

Source and *Count* are evaluated as integer data types. *Source* is shifted left with the least significant bit zeroed *Count* times. The result is Stored into *Result.*

15.2.3.5.2.32 ShiftRight - Shift Right

```
ShiftRightTerm :=   Integer <=
                        ShiftRight(
                                Source,         //OpCode=>Integer
                                Count           //OpCode=>Integer
                                Result          //ResultName
                                )
```

Source and *Count* are evaluated as integer data types. *Source* is shifted right with the most significant bit zeroed *Count* times. The result is **Stored** into *Result.*

15.2.3.5.2.33 SizeOf - SizeOf Data Object

```
SizeOfTerm :=       Integer <=
                        SizeOf(
                                DataObject      //SuperName=>DataObject
                                )
```

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, the size in bytes of the string NOT counting the trailing NULL. For a package, it returns the number of elements.

15.2.3.5.2.34 Store - Store

```
StoreTerm :=        Buffer | Integer | String <=
                        Store(
                                Source,         //OpCode
                                Destination     //SuperName
                                )
```

This operation evaluates *Source* converts to the data type of *Destination* and writes the results into *Destination.* If the *Destination* is of the type Uninitialized, then the *Destination* object is initialized as shown in the following table.

Table 15-16 Store Operator Initialization Data Types for Uninitialized Destinations

Data Type	Description
Integer	Destination initialized as integer.
Buffer	If greater than 32 bits, destination initialized as buffer; otherwise, destination initialized as integer.
String	Destination initialized as string.

The Buffer data type is a fixed length data type. The source argument must be of equal or greater length then the buffers size. Extra bits are truncated. Stores to buffer data types within an Operational Region 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 discarded. Stores to read-only objects are fatal. The execution result of the operation is *Destination.*

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)
```

15.2.3.5.2.35 Subtract - Subtract

```
SubtractTerm :=         Integer <=
                            Subtract(
                                    Addend1,            //OpCode=>Integer
                                    Addend2,            //OpCode=>Integer
                                    Result              //ResultName
                                    )
```

Addend1 and *Addend2* are evaluated as integer data types. *Addend2* is subtracted from *Addend1*, and the result is **Stored** into *Result.* Underflow conditions are ignored.

15.2.3.5.2.36 ToBCD - Convert to BCD

```
ToBCDTerm :=            Integer <=
                            ToBCD(
                                    Value               //OpCode=>Integer
                                    Destination         //ResultName
                                    )
```

The ToBCD operation is used to convert *Value* from a BCD format to a numeric format and store the numeric value in *Destination*.

15.2.3.5.2.37 Wait - Wait for a Synchronization Event

```
WaitTerm :=             Zero | Ones <=     //Ones means timed-out
                            Wait(
                                    SynchObject,        //SuperName
                                    TimeOut             //OpCode
                                )
```

SynchObject refers to an event; the calling method blocks waiting for the event to be signaled. *SynchObject* must be a reference to an Event synchronization object.

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 *TimeOut* ms have elapsed.

This operation returns a non-zero value if a time out occurred and a signal was not acquired. A *TimeOut* of 0xFFFF indicates that there is no time out and the operation will wait indefinitely.

15.2.3.5.2.38 XOr - Bitwise XOr

```
XOrTerm :=          Integer <=
                        XOr(
                                Source1,            //OpCode=>Integer
                                Source2,            //OpCode=>Integer
                                Result              //ResultName
                                )
```

Source1 and *Source2* are evaluated as integer data types, a bit-wise *xor* is performed, and the result is **Stored** in *Result.*

15.2.3.6 Type 2 Macros

```
Type2MacroCode := EISADTerm | ResourceTemplateTerm
```

For a full definition of the ResourceTemplateTerm macro, see section 6.4.1.

15.2.3.6.1 EISAID - Convert EISA ID

```
EISADTerm :=        DwordConst <=
                        EISAID(
                                ID                  //String
                                )
```

Converts *ID*, a 7-character text string argument, into its corresponding 4-byte numeric EISA ID encoding. The can be used when declaring IDs for devices that have EISA IDs.

16. 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, a machine code for a virtual machine which 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 method 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 interpreter will compile into the ACPI name space at definition block load time. For example, notice that DefByte allocates an anonymous integer variable with a byte size initial value in ACPI space, 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 OSes 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.

16.1 Notation Conventions

The notation conventions in the table below help the reader to interpret the AML formal grammar.

Table 16-1 AML Grammar Notation Conventions

Notation Convention	Description	Example
0x*dd*	Refers to a byte value expressed as 2 hexadecimal digits.	0x21
Word in bold.	Denotes the name of a term in the AML grammar, representing any instance of such a term.	
Word in bold followed by an argument in parenthesis.	Represents a particular instance of a term in the AML grammar with a particular value.	
Words in italics.	Names of arguments to objects that are replaced for a given instance.	
Term => representation	Shows the actual representation in the byte stream of the term to the left of the =>.	
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.	
Square brackets ([ ])	Indicates optional terms	
Bar symbol (|)	Separates alternatives.	aterm := bterm | [cterm dterm] means the following constructs are possible: bterm

Notation Convention	Description	Example
		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.
Superscript following a term.	The superscript is a repeat count.	Aterm^3 means aterm aterm aterm
Colon (:)	term:n indicates that term is a bit field composed of n bits. aterm:bterm refers to the bitfield bterm within the term aterm.	Term:3 means that term is a 3-bit field.
Aterm |= bterm:n cterm:m …	Bitwise OR of the terms to the right of the |=	

16.2 AML Grammar Definition

This section defines the byte values that make up an AML byte stream.

16.2.1 Names

```
LeadNameChar :=     'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' |
                    'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' |
                    'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
                    'Y' | 'Z' | '_'
NameChar :=         'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' |
                    'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' |
                    'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
                    'Y' | 'Z' | '_' | '0' | '1' | '2' | '3' | '4' |
                    '5' | '6' | '7' | '8' | '9'
    '_' =>              0x5F
    'A'-'Z' =>          0x41 - 0x5A, inclusive
    '0'-'9' =>          0x30 - 0x39, inclusive
NameSeg :=          [LeadNameChar NameChar NameChar NameChar] |
                    [LeadNameChar NameChar NameChar '_']  |
                    [LeadNameChar NameChar '_' '_']  |
                    [LeadNameChar '_' '_' '_']
                    // '\_____' refers to the ROOT, it is used
                    // mostly in Scope operators.
Name :=             NameTail | <RootPrefix NameTail> | <ParentNameHead NameTail>
    RootPrefix =>       0x5C        // Single use per name - anchors the name
                                    // to the root of the name space
NameTail :=         NameSeg  |
                    <MultiNamePrefix(segmentcount)  NameSeg^segmentcount> |
                    <DualNamePrefix  NameSeg  NameSeg>
    DualNamePrefix => 0x2E          // Two name components follow -
                                    // shorthand for MultiNamePrefix(2)
MultiNamePrefix := MultiNamePrefixOp ByteConst
    MultiNamePrefixOp => 0x2F       // N name components follow -
                                    // N can be from 1 to 255, inclusive.
                                    // For example:
                                    // MultiNamePrefix(x23) => x2F x23
                                    // and refers to a 35-character segment
                                    // name (unreasonably long.)
                                    // Note that
                                    //   DualNamePrefix  NameSeg  NameSeg
                                    // has a smaller encoding than the equivalent encoding
                                    //   MultiNamePrefix(2) NameSeg NameSeg.
ParentNameHead :=  ParentNamePrefix [ParentNamePrefix ...]
    ParentNamePrefix => 0x5E        // 1 to N uses per name - anchors the name to a
                                    // parent's location in the name space tree.
SuperName :=   Name  |  DefArg  |  DefLocal | DefDebug
```

16.2.2 Declarations

```
DefConstant :=          OneOp | OnesOp | ZeroOp
    OneOp =>                0x01
    OnesOp =>               0xff | 0xffff | 0xffffffff
    ZeroOp =>               0x00
DefDataObject :=        DefBuffer | DefNum | DefPackage | DefString
DefBuffer :=            BufferOp ByteConst(buffsize) ByteConst^buffsize
   BufferOp =>              0x11
DefNum :=               DefByte  |  DefWord  |  DefDWord
DefByte :=              ByteOp ByteConst
   ByteOp =>                0x0A
   ByteConst =>             0x00 - 0xff, inclusive
DefWord :=              WordOp WordConst
    WordOp =>               0x0B
    WordConst =>            0x0000 - 0xffff, inclusive
DefDWord :=             DWordOp DWordConst
    DWordOp =>              0x0C
    DWordConst =>           0x00000000 - 0xffffffff, inclusive
DefPackage :=           PackageOp  PkgLength(Length(Data)) ByteConst^ElementCount Data
    PackageOp =>            0x12
    Length(expression) =>   The length, in bytes, of expression.
    PkgLength :=            PkgLead  ByteConst^followcount
    PkgLead  |=             < 6-7: followcount => 1-3
                              4-5: reserved
                              0-3: 0-15
                            >
                            |
                            < 6-7: followcount => 0
                            0-5: 0-63
                            >
```

```
                                // Note: The high 2 bits of the first byte reveal how many
                                // follow bytes are in the PkgLength. If the PkgLength value
                                // is more than 1 byte long (value > 63) then only 4 bits
                                // are used from the first byte. Maximum value
                                // is 3*8 + 4 = 28 bits. The value is little endian.
                                // For example:
                                //      PkgLength(63) => 0x3F
                                //      PkgLength(64) => 0x40 0x04
                                //      PkgLength(0xF13BA4) => 0xC4 0xBA 0x13 0x0F
DefString :=                StringOp  Asciib  NullChar
    StringOp =>                 0x0D
    Asciib :=                   AsciiChar [AsciiChar ...]
    AsciiChar =>                0x01 - 0x7f         // Any standard ASCII character
                                                    // except for NullChar).
    NullChar =>                 0x00
DefDebug :=                 DebugOp
    DebugOp =>                  0x5B 0x31
DefMethodDataObject :=      DefArg | DefLocal
DefArg :=                   Arg0 | Arg1 | Arg2 | Arg3 | Arg4 | Arg5 | Arg6
    Arg0 =>                     0x68
    Arg1 =>                     0x69
    Arg2 =>                     0x6A
    Arg3 =>                     0x6B
    Arg4 =>                     0x6C
    Arg5 =>                     0x6D
    Arg6 =>                     0x6E
DefLocal :=Local0 | Local1 | Local2 | Local3 | Local4 | Local5  Local6 | Local7
    Local0 =>                   0x60
    Local1 =>                   0x61
    Local2 =>                   0x62
    Local3 =>                   0x63
    Local4 =>                   0x64
    Local5 =>                   0x65
    Local6 =>                   0x66
    Local7 =>                   0x67
DefNameSpaceModifier :=     DefAlias | Defname | DefScope
DefAlias :=                 AliasOp  Name  Name
    AliasOp =>                  0x06
DefName :=                  NameOp  Name  DataTerm
    NameOp =>                   0x08
    Data :=                     DataTerm [DataTerm ...]
    DataTerm :=                 DataItem  | DefPackage
    DataItem :=                 DefBuffer | DefNum | DefString
DefScope :=                 ScopeOp PkgLength Name  NamedObjectList
    ScopeOp =>                  0x10
NamedObjectList :=          NamedObject [NamedObject ...]
NamedObject :=              DefBankField | DefDevice | DefEvent | DefField | DefIndexField
|                           DefMethod | DefMutex | DefPowerResource | DefProcessor |
                            DefRegion | DefThermalZone
DefBankField :=             DefBankFieldOp  Name Name BankVal FieldFlag DefFieldList
    DefBankFieldOp =>           0x5B 0x87
    BankVal =>                  DwordConst
    FieldFlag |=                0-3: AccessType
                                4-4: LockRule
                                5-6: UpdateRule
                                7-7: Reserved
        AccessType:4 :=             AnyAcc | ByteAcc | WordAcc | DWordAcc |
                                    BlockAcc | SMBSendRecvAcc | SMBQuickAcc
            AnyAcc =>                   0
            ByteAcc =>                  1
            WordAcc =>                  2
            DWordAcc =>                 3
            BlockAcc =>                 4
            SMBSendRecvAcc =>           5
            SMBQuickAcc =>              6
        LockRule:1 :=               Lock | NoLock
            NoLock =>                   0
            Lock =>                     1
        UpdateRule:2 :=             Preserve | WriteAsOnes | WriteAsZeros
            Preserve =>                 0
            WriteAsOnes =>              1
            WriteAsZeros   =>           2
    DefFieldList :=         DefFieldElement | [DefFieldElement ... ]
        DefFieldElement :=      NamedField | ReservedField | AccessField
        NamedField :=           NameSeg  PkgLength
        ReservedField :=        0x00  PkgLength
```

```
       AccessField :=        0x01  Pkglength
DefDevice :=                 DeviceOp PkgLength Name NamedObjectList
    DeviceOp =>              0x5B 0x82
DefEvent :=                  EventOp Name
    EventOp =>               0x5B 0x02
DefField :=                  DefFieldOp  PkgLength  Name  FieldFlag  DefFieldList
    DefFieldOp =>            0x5B 0x81
DefIndexField :=             IndexFieldOp Name Name FieldFlag FieldList
    IndexFieldOp =>          0x5B 0x86
DefMethod :=                 MethodOp  PkgLength  Name  MethodFlags  CodeList
    MethodOp =>              0x14
    MethodFlags |=           0-2: ArgCount
                             3-3: Synchronized
                             4-7: reserved
       ArgCount:3 :=             0-7
DefMutex :=                  MutexOp  Name  SyncFlags
    MutexOp                  0x5B 0x01
    SyncFlags |=             0-3: SyncLevel => 0x0 - 0xf
                             4-7: Reserved
DefPowerResource :=          PowerResOp PkgLength Name SysLevel Level NamedObjectList
    PowerResOp =>            0x5B 0x84
    SysLevel :=              ByteConst
    Level :=                 WordConst
DefProcessor :=              ProcessorOp PkgLength Name ProcID PBlk PBlkLen NamedObjectList
    ProcessorOp =>           0x5B 0x83
    ProcID :=                ByteConst
    PBlk :=                  DWordConst
    PBlkLen :=               ByteConst
DefRegion :=                 RegionOp Name RegionSpace RegionAddress RegionLength
    RegionOp =>              0x5B 0x80
    RegionSpace :=           Byte => 0 |     // SystemMemory
                                     1 |     // SystemIO
                                     2 |     // PCIConfig
                                     3 |     // EmbeddedControl
                                     4       // SMBus
    RegionAddress :=         DWordConst
    RegionLength :=          DWordConst
DefThermalZone :=            ThermalZoneOp  PkgLength  Name NamedObjectList
    ThermalZoneOp =>         0x5B 0x85
```

16.2.3 Operators

```
OpCode :=                    DefType2OpCode | SuperName | DefConstant | DefNum | DefString

Source :=                    OpCode
Source1 :=                   OpCode
Source2 :=                   OpCode
Count :=                     OpCode
BitNum :=                    SuperName
Result :=                    SuperName
Destination :=               SuperName
Index :=                     OpCode

DefType1OpCode :=            DefBreak | DefBreakPoint | DefBitField | DefByteField |
                             DefDwordField | DefCreateField | DefWordField | DefElse |
                             DefFatal | DefIf | DefNoop | DefNotify | DefRelease |
                             DefReset | DefReturn | DefSignal | DefSleep | DefStall |
                             DefUnload | DefWhile
DefBreak :=                  BreakOp
    BreakOp =>               0xA5
DefBreakPoint :=             BreakPointOp
    BreakPointOp =>          0xCC
DefBitField :=               BitFieldOp Source Index Destination
    BitFieldOp =>            0x8D
DefByteField :=              ByteFieldOp Source Index Destination
    ByteFieldOp =>           0x8C
DefDwordField :=             DwordFieldOp Source Index Destination
    DwordFieldOp =>          0x8A
DefCreateField :=            FieldOp Source Offset NoBits Destination
    FieldOp =>               0x5B 0x13
    Offset :=                Opcode
    NoBits :=                Opcode
DefWordField :=              WordFieldOp Source Index Destination
    WordFieldOp =>           0x8B
```

```
DefElse :=              ElseOp PkgLength FalseCode
    ElseOp =>               0xA1
    FalseCode :=        OpCode | CodePkg
DefFatal :=             FatalOp FType FCode FArg
    FatalOp =>              0x5B 0x32
    FType :=                ByteConst
    FCode :=                DwordConst
    FArg :=                 OpCode
DefIf :=                IfOp PkgLength Predicate TrueCode
    IfOp =>                 0xA0
    Predicate :=            OpCode
    TrueCode :=             OpCode | CodePkg
DefNoOp :=              NoOp
    NoOp =>                 0xA3
DefNotify :=            NotifyOp SuperName ByteConst
    NotifyOp =>             0x86
DefRelease :=           ReleaseOp SuperName
    ReleaseOp =>            0x5B 0x27
DefReset :=             ResetOp SuperName
    ResetOp =>              0x5B 0x26
DefReturn :=            ReturnOp ReturnValue
    ReturnOp =>             0xA4
    ReturnValue :=          Opcode
DefSignal :=            SignalOp SuperName
    SignalOp =>             0x5B 0x24
DefSleep :=             SleepOp Milliseconds
    SleepOp =>              0x5B 0x22
    Milliseconds :=         WordConst
DefStall :=             StallOp Microseconds
    StallOp =>              0x5B 0x21
    Microseconds :=         ByteConst
DefUnLoad :=            UnLoadOp OpCode
    UnloadOp =>             0x5B 0x2A
DefWhile :=             WhileOp PkgLength Predicate TrueCode
    WhileOp =>              0xA2

Type2OpCode :=          DefAcquire | DefAdd | DefBitAnd |
                        DefBitNand | DefBitOr | DefBitNor | DefBitNot |
                        DefBitXor | DefConcat | DefCondRefOf | DefDecrement |
                        DefDivide | DefFindSetLeftBit |
                        DefFindSetRightBit | DefFromBCD | DefIncrement | DefLAnd |
                        DefLEqual | DefLGreaterEqual | DefLGreater |
                        DefLLessEqual | DefLLess | DefLNot | DefLNotEqual |
                        DefLoad | DefLOr |
                        DefMultiply | DefRefIndex | DefRefOf | DefRefMatch |
                        DefReturn | DefShiftLeftBit | DefShiftLeft
                        DefShiftRightBit | DefShiftRight | DefSizeOf | DefStore |
                        DefSubtract | DefToBCD | DefType | DefWait

DefAcquire :=           AcquireOp SuperName
    AcquireOp =>            0x5B 0x23
DefAdd :=               AddOp Addend1 Addend2 Result
    AddOp =>                0x72
    Addend1 :=              OpCode
    Addend2 :=              OpCode
DefBitAnd :=            AndOp Source1 Source2 Result
    AndOp =>                0x7B
DefBitNAnd :=           NAndOp Source1 Source2 Result
    NAndOp =>               0x7C
DefBitOr :=             OrOp Source1 Source2 Result
    OrOp =>                 0x7D
DefBitNor :=            NorOp Source1 Source2 Result
    NorOp =>                0x7E
DefBitNot :=            NotOp Source1 Result
    NotOp =>                0x80
DefBitXor :=            XOrOp Source1 Source2 Result
    XorOp =>                0x7F
DefConcat :=            ConcatOp Source1 Source2 Destination
    ConcatOp =>             0x73
DefCondRefOf :=         CondRefOfOp Source Destination
    CondRefOfOp =>      0x5B 0x12
DefDecrement :=         DecrementOp SuperName
    DecrementOp =>          0x76
DefDivide :=            DivideOp Dividend Divisor Remainder Quotient
    DivideOp =>             0x78
    Dividend :=             Opcode
```

```
    Divisor :=              Opcode
    Remainder :=            SuperName
    Quotient :=             SuperName
DefFindSetLeftBit :=    FindSetLeftBitOp Source Destination
    FindSetLeftBitOp =>     0x81
DefFindSetRightBit :=   FindSetRightBitOp Source Destination
    FindSetRightBitOp =>    0x82
DefFromBCD :=           FromBCDOp BcdValue Destination
    FromBCDOp =>            0x5B 0x28
    BcdValue :=             OpCode
DefIncrement :=         IncrementOp SuperName
    IncrementOp =>          0x75
DefLAnd :=              LAndOp Source1 Source2
    LAndOp =>               0x90
DefLEqual :=            LEQOp Source1 Source2
    LEQOp =>                0x93
DefLGreaterEqual :=     LGEQOp Source1 Source2
    LGEQOp =>               0x95 0x92
DefLGreater :=          LGOp Source1 Source2
    LGOp =>                 0x94
DefLLessEqual :=        LLEQOp Source1 Source2
    LLEQOp =>               0x94 0x92
DefLLess :=             LLOp Source1 Source2
    LLOp =>                 0x95
DefLNot :=              LNotOp Source1
    LNotOp =>               0x92
DefLNotEqual :=         LNotEQOp Source1 Source2
    LNotEQOp =>             0x93 0x92
DefLoad :=              LoadOp Name
    LoadOp =>               0x5B 0x20
DefLOr :=               LOrOp Source1 Source2
    LOrOp =>                0x91
DefMultiply :=          MultiplyOp Multiplican Multiplier Product
    MultiplyOp =>           0x77
    Multiplican :=          Opcode
    Multiplier :=           Opcode
    Product :=              SuperName
DefRefIndex :=          RefIndexOp OpCode OpCode
   RefIndexOp =>            0x88
DefRefOf :=             RefOfOp Source
    RefOfOp =>              0x71
DefRefMatch :=          RefMatchOp Source MatchOp Source1 MatchOp Source2 Start
    RefMatchOp =>           0x89
    MatchOp =>              OpCode
    Start =>                OpCode
DefShiftLeftBit : =     ShiftLeftBitOp Source BitNum
    ShiftLeftBitOp =>       0x5B 0x11
DefShiftLeft :=         ShiftLeftOp Source Count Result
    ShiftLeftOp =>          0x79
DefShiftRightBit :=     ShiftRightBitOp Source BitNum
    ShiftRightBitOp =>      0x5B 0x10
DefShiftRight : =       ShiftRightOp Source Count Result
    ShiftRightOp =>         0x7A
DefSizeOf :=            SizeOfOp SuperName
    SizeOfOp =>             0x87
DefStore :=             StoreOp Source Destination
    StoreOp =>              0x70
DefSubtract :=          SubtractOp  Minuend Subtraend Difference
    SubtractOp =>           0x74
    Minuend :=              OpCode
    Subtraend :=            OpCode
    Difference :=           SuperName
DefToBCD :=             ToBCDOp Value Destination
    ToBCDOp =>              0x5B 0x29
    Value :=                OpCode
DefType :=              TypeOp SuperName
    TypeOp =>               0x8E
DefWait :=              WaitOp SuperName WordConst
    WaitOp =>               0x5B 0x25
```

16.3 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 16-2 AML Byte Stream Byte Values

Byte Value (in Hex)	Meaning	Fixed List Arguments	Variable List Arguments?
0x00	ZeroOp	Nothing	No
0x01	OneOp	Nothing	No
0x02 - 0x05	Null	--	--
0x06	AliasOp	Name Name	No
0x07	Null	--	--
0x08	NameOp	Name Data	No
0x09	Null	--	--
0x0A	ByteOp	ByteConst	No
0x0B	WordOp	WordConst	No
0x0C	DWordOp	DWordConst	No
0x0D	StringOp	AsciiChar [AsciiChar ...] NullChar	No
0x0E - 0F	Null	--	--
0x10	ScopeOp	Name	Yes
0x11	BufferOp	ByteConst	Yes
0x12	PackageOp	ByteConst	Yes
0x13	Null	--	--
0x14	MethodOp	Name Byte	Yes
0x15 - 0x2D	Null	--	--
0x2E ('.')	DualNamePrefix	NameSeg NameSeg	--
0x2F ('/')	MultiNamePrefix	Byte NameSeg [NameSeg ...]	No
0x30 - 0x5a	Null	--	--
0x5B ('[')	ExtendedOperator Prefix	Byte	--
0x5B 0x01	MutexOp	Name Byte	No
0x5B 0x02	EventOp	Name	No
0x5B 0x10	ShiftRightBitOp	OpCode SuperName	No
0x5B 0x11	ShiftLeftBitOp	OpCode SuperName	No
0x5B 0x12	CondRefOfOp	OpCode SuperName	No
0x5B 0x13	CreateFieldOp	OpCode OpCode OpCode SuperName	No
0x5B 0x20	LoadOp	Name	No
0x5B 0x21	StallOp	OpCode	No
0x5B 0x22	SleepOp	OpCode	No
0x5B 0x23	AcquireOp	SuperName	No
0x5B 0x24	SignalOp	SuperName	No
0x5B 0x25	WaitOp	SuperName WordConst	No
0x5B 0x26	ResetOp	SuperName	No
0x5B 0x27	ReleaseOp	SuperName	No
0x5B 0x28	FromBCDOp	OpCode SuperName	No
0x5B 0x29	ToBCD	OpCode SuperName	No
0x5B 0x2A	UnloadOp	OpCode	No

Byte Value (in Hex)	Meaning	Fixed List Arguments	Variable List Arguments?
0x5B 0x31	DebugOp	Nothing	No
0x5B 0x32	FatalOp	Nothing	No
0x5B 0x80	OpRegionOp	Name Byte OpCode OpCode	No
0x5B 0x81	FieldOp	Name Byte	Yes
0x5B 0x82	DeviceOp	Name	Yes
0x5B 0x83	ProcessorOp	Name Byte Dword Byte	Yes
0x5B 0x84	PowerResOp	Name Name Word	Yes
0x5B 0x85	ThermalZoneOp	Name	Yes
0x5B 0x86	IndexFieldOp	Name Name Byte	Yes
0x5B 0x87	BankFieldOp	Name Name Dword Byte	Yes
0x5C ('\')	RootNamePrefix	Name	--
0x5D	Null	--	--
0x5E ('^')	ParentNamePrefix	Name	--
0x5F	Null	--	--
0x60 (`)	Local0	Nothing	No
0x61 ('a')	Local1	Nothing	No
0x62 ('b')	Local2	Nothing	No
0x63 ('c')	Local3	Nothing	No
0x64 ('d')	Local4	Nothing	No
0x65 ('e')	Local5	Nothing	No
0x66 ('f')	Local6	Nothing	No
0x67 ('g')	Local7	Nothing	No
0x68 ('h')	Arg0	Nothing	No
0x69 ('i')	Arg1	Nothing	No
0x6A ('j')	Arg2	Nothing	No
0x6B ('k')	Arg3	Nothing	No
0x6C ('l')	Arg4	Nothing	No
0x6D ('m')	Arg5	Nothing	No
0x6E ('n')	Arg6	Nothing	No
0x6F	Null	--	--
0x70	StoreOp	OpCode SuperName	No
0x71	RefOfOp	SuperName	No
0x72	AddOp	OpCode OpCode SuperName	No
0x73	ConcatOp	OpCode OpCode SuperName	No
0x74	SubtractOp	OpCode OpCode SuperName	No
0x75	IncrementOp	SuperName	No
0x76	DecrementOp	SuperName	No
0x77	MultiplyOp	OpCode OpCode SuperName	No
0x78	DivideOp	OpCode OpCode SuperName SuperName	No

Byte Value (in Hex)	Meaning	Fixed List Arguments	Variable List Arguments?
0x79	ShiftLeftOp	OpCode OpCode SuperName	No
0x7A	ShiftRightOp	OpCode OpCode SuperName	No
0x7B	AndOp	OpCode OpCode SuperName	No
0x7C	NAndOp	OpCode OpCode SuperName	No
0x7D	OrOp	OpCode OpCode SuperName	No
0x7E	NOrOp	OpCode OpCode SuperName	No
0x7F	XOrOp	OpCode OpCode SuperName	No
0x80	NotOp	OpCode SuperName	No
0x81	FindSetLeftBitOp	OpCode SuperName	No
0x82	FindSetRightBitOp	OpCode SuperName	No
0x83 - 0x85	Null	--	--
0x86	NotifyOp	SuperName ByteConst	No
0x87	SizeOfOp	Name	No
0x88	IndexOp	OpCode OpCode	No
0x89	MatchOp	OpCode OpCode OpCode OpCode OpCode	No
0x8A	DWordFieldOp	OpCode OpCode SuperName	No
0x8B	WordFieldOp	OpCode OpCode SuperName	No
0x8C	ByteFieldOp	OpCode OpCode SuperName	No
0x8D	BitFieldOp	OpCode OpCode SuperName	No
0x8E	ObjTypeOp	Name	No
0x8F	Null	--	--
0x90	LAndOp	OpCode OpCode	No
0x91	LOrOp	OpCode OpCode	No
0x92	LNotOp	OpCode	No
0x93	LEQOp	OpCode OpCode	No
0x93 0x92	LNotEQOp	OpCode OpCode	No
0x94	LGOp	OpCode OpCode	No
0x94 0x92	LLEQOp	OpCode OpCode	No
0x95	LLOp	OpCode OpCode	No
0x95 0x92	LGEQOp	OpCode OpCode	No
0x96 - 9F	Null	--	--
0xA0	IfOp	OpCode	Yes
0xA1	ElseOp	Nothing	Yes
0xA2	WhileOp	OpCode	Yes
0xA3	NoOp	Nothing	No
0xA4	ReturnOp	OpCode	No
0xA5	BreakOp	Nothing	No
0xA6 - 0xCB	Null	--	--
0xCC	BreakPointOp	Nothing	No
0xCD - 0xFE	Null	--	--

Byte Value (in Hex)	Meaning	Fixed List Arguments	Variable List Arguments?
0xFF	OnesOp	Nothing	No

16.4 Examples

16.4.1 Relationship Between ASL, AML, and Byte Streams

For an example that shows the relationship between ASL code and the AML byte stream it produces, see sections 5.4 and 5.5.

16.5 AML Encoding of Names in the Name Space

Assume the following name space 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.MEM.SET.STP1) {
    XYZ
    ^ABC
    ^ABC.DEF
}
```

This will be encoded in AML as:

```
'STP1'
ParentNamePrefix  'GET_'
ParentNamePrefix  ParentNamePrefix  'PCI0'
ParentNamePrefix  ParentNamePrefix  DualNamePrefix  'PCI0' 'SBS_'
RootPrefix  'S2__'
RootPrefix  MultiNamePrefix  3  'S2__'  'ISA_'  'COM1'
ParentNamePrefix  ParentNamePrefix  ParentNamePrefix  'S3__'
ParentNamePrefix  ParentNamePrefix  ParentNamePrefix  DualNamePrefix  'S2__'  'MEM_'
ParentNamePrefix  ParentNamePrefix  ParentNamePrefix  MultiNamePrefix3  'S2__'  'MEM_'
'SET_'
```

After the block is loaded, the name space will look like this (names added to the name space by the loading operation are shown in italics).

```
\
    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
```