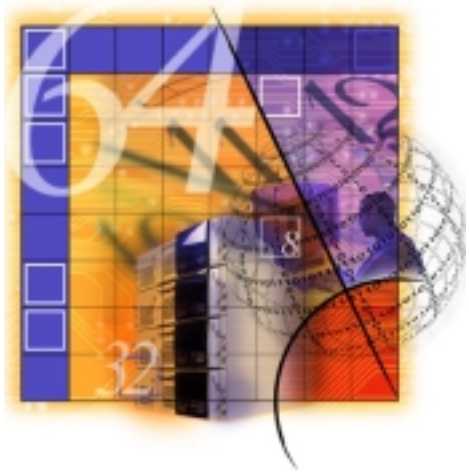

UNIX* Developer's Interface Guide for Intel®-Based Servers (UDIG)

Version 1.0
December 1999

UDIG Promoters

Adaptec
Compaq
Hewlett Packard
IBM
Intel
Interphase Corporation
LSI Logic
Mylex
Phoenix
Qlogic
SCO
Sun Microsystems

Contents

TABLES

Chapter 1: Introduction

This chapter introduces the audience, scope, and purpose of the *UNIX* Developer's Interface Guide for Intel-Based Servers* (UDIG). This chapter also summarizes the content of the rest of the document.

REFERENCES

UDIG uses acronyms to reference other documents. For example, the acronym ACPI1 refers to the *Advanced Configuration and Power Interface (ACPI) Revision 1.0b*. Acronyms are unique and enclosed in square brackets [ACPI1]. For detailed information about the document referenced by an acronym, see the References section.

1.1 Audience for This Guide

This guide is for use by operating system, peripheral device, and platform vendors. It assumes that readers are familiar with referenced technologies.

1.2 Scope of This Guide

This guide describes common system building blocks and interfaces that can be used across all UNIX operating systems on Intel® architecture server platforms. The guide focuses on software interfaces for hardware systems and includes implementation requirements, guidelines, and usage models.

When a system adheres to all required guidelines, the system is referred to as a UDIG-compliant system.

This guide provides guidelines beyond the *Developers Interface Guide for IA-64 servers* [DIG64] to support UNIX operating systems. Exceptions are noted when the guide refers to IA-32 architectures.

1.3 Purpose of This Guide

The purpose of this guide is to provide a set of common compatible interfaces for the UNIX operating system on Intel architectures.

This guide attempts to

- Reduce design and development costs for Operating System Vendors (OSVs) and device manufacturers.

- Broaden opportunities for companies producing products for servers running UNIX operating systems.
- Reduce qualification time and effort for IT customers.

1.4 Chapters of This Guide

Chapter 1 provides overview information. Chapters 2 through Appendix A address guidelines for specific architectural areas for UNIX operating systems running on Intel servers.

Table 1-1 – Contents

CHAPTER	CONTENT
Chapter 1: Introduction	This chapter introduces the audience, scope, and purpose of the guide. It summarizes the content of other chapters.
Chapter 2: Boot and Configuration Guidelines	This chapter describes the guidelines for booting and configuring the UNIX operating system.
Chapter 3: Device Drivers and Services	This chapter addresses guidelines for device drivers and services for the UNIX operating system.
Appendix A	Information about the ACPI Component Architecture (CA) and the interfaces for both the OS-independent and OS-dependent components.
References	This section lists documents that are referenced in the UDIG.

1.5 Guideline Compliance

Compliance is self-administered. It is up to each adopter of this guide to check for compliance.

Adherence to these guidelines contributes to server software availability, scalability, compatibility, and manageability. It is in the implementer's best interest to clearly identify that software/hardware interfaces are compliant, and to announce this compliance through product literature and labeling.

When a system adheres to all required guidelines, the system is referred to as a UDIG-compliant system.

The sections below describe levels of guideline compliance.

1.5.1 What Is Required Compliance?

If a feature is *required*, the feature must be implemented to comply with these guidelines. Other architectural layers assume the presence of a required feature.

NOTE

Some *required* features may be nested under parent features marked as *recommended* or *optional*. In this situation, the nested *required* feature must be implemented for compliance only if implementers include the *recommended* or *optional* parent feature.

1.5.2 What Is Recommended Compliance?

If a feature is *recommended*, the feature need not be implemented. Implementation is, however, encouraged.

The presence of a *recommended* feature is not to be assumed nor precluded by other architectural levels.

An implementation which does not include a recommended feature must be prepared to interoperate with another implementation which does include the feature, though perhaps with reduced functionality. Similarly, an implementation which does include a recommended feature must be prepared to interoperate with another implementation which does not include the feature (except, of course, for the functionality the feature provides.)

Recommended features may become requirements in the future.

1.5.3 What Is Optional Compliance?

If a feature is *optional*, the feature need not be implemented. There should be no dependencies on *optional* features by other architectural levels. Omitting an *optional* feature should have no impact on other required features or the proper functioning of the system.

Operating systems are not required to support optional features but must operate in the presence or absence of those features.

1.6 Tools

This guide references tools that help you adopt these guidelines. These elements are referenced in the individual chapters and include SDKs and reference implementations where available.

1.7 Guideline Summary

The following table is a summary of all the guidelines in this document.

Section	Guideline	Required, Recommended, Optional
2.1.2.1	Support EFI Boot Environment by Providing an EFI OS Loader	Required
2.1.2.2	Tools for Installing Initial OS Boot Loader	Required
2.1.2.2.1	Tools for creating an EFI partition	Required
2.1.2.2.2	Tools for setting non-volatile variables	Required

Section	Guideline	Required, Recommended, Optional
2.1.2.3	Provide Tools to Access and Maintain the System Partition at OS Run-time	Recommended
2.1.2.3.1	Provide tools to copy EFI applications, boot loaders, and drivers	Required if Provide Tools to access and maintain the system partition at OS run-time is implemented
2.1.2.3.2	Provide tools to verify the integrity of the system partition and its contents	Recommended
2.1.2.3.3	Provide tools to create an EFI partition	Recommended
2.1.2.4	Manipulating EFI Non-volatile Variables	Required
2.1.2.4.1	Manipulating EFI non-volatile variables with administrative tools	Recommended
2.1.2.4.2	Path to operating system boot partition	Recommended
2.2.1	IA-32 Option ROMs	Optional
2.2.2	EFI Option ROMs	Recommended
2.2.3	EFI Portable Option ROMs	Optional
2.3.1.1	Follow DIG-64 Hardware/Firmware Guidelines for ACPI	Required
2.3.2.1	Support for Memory Reporting Interfaces	Required
2.3.2.2	Support for ACPI Devices Defined in the ACPI Specification	Recommended
2.3.2.3	Provide an ACPI AML Interpreter	Required
2.3.2.4	Provide Configuration Support	Required
2.3.2.5	Provide Support for System Events via SCI and GPE	Recommended
2.3.2.6	Recognize Devices through the ACPI Namespace	Required
2.3.2.6.1	Devices that require ACPI enumeration	Required
2.3.2.7	Use Resources Presented in the Namespace	Required
2.3.2.8	Respond to Device Insertion and Removal	Recommended
2.3.2.9	Provide Device Power Management Support for D0 and D3	Recommended
2.3.3	PCI Hot-plug	Recommended
2.3.4.1	Implement an ACPI Subsystem	Required
2.3.4.2	Implement ACPI subsystem External Interfaces	Recommended
2.3.4.3	Implement ACPI Subsystem Operating System Services	Recommended
3.2.1.1.1	Supported processor types	Required
3.2.1.1.2	Endianness	Required
3.2.1.1.3	Driver packaging subdirectories	Required
3.2.1.2	Runtime Architecture	Required
3.2.1.3.1	Sizes of UDI-Specific data types	Required
3.2.1.3.2	Implementation-Dependent macros	Required
3.2.1.3.3	UDI functions implemented as macros	Required
3.2.1.4.1	Object file format	Required
3.2.1.4.2	Static driver properties encapsulation	Required

Section	Guideline	Required, Recommended, Optional
3.2.2.1.1	Supported processor types	Required
3.2.2.1.2	Endianness	Required (Recommended that little endian is supported)
3.2.2.1.3	Driver packaging subdirectories	Required
3.2.2.2	Runtime Architecture	Required
3.2.2.3.1	Sizes of UDI-Specific data types	Required
3.2.2.3.2	Implementation-Dependent macros	Required
3.2.2.3.3	UDI functions implemented as macros	Required
3.2.2.4.1	Object file format	Required
3.2.2.4.2	Static driver properties encapsulation	Required
3.3.1.1	UDI Core Specification	Required
3.3.1.2	UDI Physical I/O Specification	Required
3.3.2	Metalanguages	Required
3.3.3	Bus Bindings	Required

Chapter 2: Boot and Configuration Guidelines

This chapter provides the guidelines necessary to configure and boot a UNIX operating system. The content assists UNIX operating system developers, firmware developers, and system hardware developers in building fully functional server systems based on IA-32, IA-64, and compatible architectures. This draws from a very large existing body of firmware/boot documentation to define firmware and hardware interfaces. These interfaces are required to take a system from the end of POST to the end of the operating system boot process.

Booting a system requires powering up the system, configuring the hardware for boot, and executing an OS bootstrap loader. There are two distinctly different methods for booting a system: legacy IA-32 boot and Extensible Firmware Interface (EFI). Both methods are OS neutral, so very little documentation regarding the booting of a UNIX operating system is needed.

The PC industry has been using the legacy IA-32 method for years. The legacy IA-32 method uses the system BIOS to control the boot sequence. The system BIOS provides abstracted services, enumerates hardware devices, loads Option ROMs, and performs resource management. This chapter assumes that the reader is familiar with the legacy boot process.

EFI is a comprehensive interface specification for booting an operating system. Since the *EFI Specification* [EFI1] is OS neutral, it does not provide guidelines for booting a UNIX operating system. This chapter assumes that the reader is familiar with the specification and adds guidelines for UNIX operating systems in an EFI environment. EFI allows systems to remove legacy hardware and associated legacy firmware interfaces.

The ACPI Specification provides a methodology for configuring systems. By using ACPI the operating system does not have to rely on proprietary methods for configuring the system hardware. ACPI interfaces provide a configuration layer that allows the operating system and hardware to exist and evolve independently of each other.

This chapter includes

- EFI boot model requirements and how industry uses these requirements to boot IA-64 systems.
- Requirements for EFI Option ROMs.
- ACPI requirements for UNIX in an IA-64 environment.

REFERENCES

The UDIG uses acronyms to reference other documents. For example, the acronym ACPI1 refers to the *Advanced Configuration and Power Interface (ACPI) Revision 1.0b*. Acronyms are unique and enclosed in square brackets [ACPI1]. For detailed information about the document referenced by an acronym, see the References section.

2.1 EFI Boot Model

This section describes the Extensible Firmware Interface (EFI) boot initiative and its interaction with UDIG-compliant UNIX operating systems.

2.1.1 EFI Overview

Currently, some OS loaders have a very intimate knowledge of the underlying hardware platform. Relying on specific types of hardware devices at dedicated I/O or memory addresses prevents innovation and evolution of the platform.

EFI technology is OS-independent and applies equally to both IA-64 and IA-32 platforms. EFI isolates the OS loader from the underlying hardware platform by specifying interfaces for both the OS loader and Platform hardware/firmware. EFI defines a system partition, boot-time services, run-time services, and an application environment where OS-independent applications can execute.

2.1.1.1 System Partition

EFI specifies an OS-independent area, called the system partition, for the storage of utilities, diagnostics, EFI drivers and OS loaders.

EFI supports booting from media that contains an EFI OS Loader or an EFI-defined system partition. Examples of media supported by EFI include diskette, hard drive, CD-ROM, DVD-ROM, and networks.

2.1.1.2 Boot Time Services

EFI defines boot-time services. For example, it provides:

- Memory allocation.
- Watchdog timer support.
- Console input/output.
- Block I/O, Disk I/O and file system access to devices.
- Ability to add additional EFI interfaces and drivers.
- Functions that create, signal, and destroy events and adjust a task's priority level.
- Functions that terminate the EFI application or transition from the boot environment to the OS runtime environment.

2.1.1.3 Runtime Services

EFI defines runtime services that are available even after the OS boot loader has issued the EFI disconnect command. The following are examples of EFI runtime services.

- Get and set time.
- Reboot the system.
- Get or set a system wakeup time.
- Get or set non-volatile environment variables.

2.1.2 UNIX and EFI

2.1.2.1 Support EFI Boot Environment by Providing an EFI OS Loader

Required

An EFI OS loader must be the software component that bootstraps the operating system using EFI services.

2.1.2.2 Tools for Installing Initial OS Boot Loader

Required

The operating system install process must be capable of storing the OS boot loader in an EFI partition.

2.1.2.2.1 Tools for creating an EFI partition

Required

The operating system install process must be capable of creating an EFI partition.

2.1.2.2.2 Tools for setting non-volatile variables

Required

The operating system install process must be capable of setting the environment variables required for boot.

2.1.2.3 Provide Tools to Access and Maintain the System Partition at OS Runtime

Recommended

2.1.2.3.1 Provide tools to copy EFI applications, boot loaders, and drivers

Required if Provide tools to access and maintain the system partition at OS run-time section is implemented

The EFI system partition is used to store EFI applications such as OS Boot loaders, drivers and other applications. The operating system must provide tools to copy EFI applications, boot loaders and drivers to the system partition.

2.1.2.3.2 Provide tools to verify the integrity of the system partition and its contents

Recommended

The operating system should provide tools to verify the integrity of the system partition and its contents. These tools should verify that all software components in the system partition exist to enable the system to boot.

2.1.2.3.3 Provide tools to create an EFI partition.

Recommended

The operating system should provide tools to create additional EFI partitions.

2.1.2.4 Manipulating EFI Non-volatile Variables

Required

The operating system must be capable of manipulating the EFI non-volatile variables at run-time. For example, this interface could be used to manipulate the boot order.

2.1.2.4.1 Manipulating EFI non-volatile variables with administrative tools

Recommended

The operating system should provide administrative tools to manipulate the EFI non-volatile variables at run-time.

2.1.2.4.2 Path to operating system boot partition

Recommended

The path to operating system boot partition should be specified. This data resides in the OptionalData element of the Boot load option variable (Boot####,) as defined in the *EFI Specification* [EFI1].

If implemented, the first item in the OptionalData element must be an EFI_DEVICE_PATH, including Hard Drive Media Device Path that represents the path to the OS partition. The OS loader will use this element as its primary means to locate and perform I/O to the OS partition using EFI. Future revisions of the EFI specification may redefine OptionalData as an element of type EFI_DEVICE_PATH.

2.2 Option ROMs

This section provides the implementation guidelines to assist in Option ROM development for IA-64 and compatible architectures. While the focus is on IA-64-hosted UNIX servers, this section defines guidelines that are independent of CPU architecture and operating systems for the future development of Option ROMs. Where applicable, this section refers to existing industry specifications and is intended for developers who implement EFI Option ROMs.

This section provides guidelines regarding:

- IA-32 Option ROMs.
- EFI Option ROMs.
- EFI Portable Option ROMs.

2.2.1 IA-32 Option ROMs

Optional

IA-32 Option ROMs have many legacy requirements that may not be supported in future EFI systems. A complete description of existing IA-32 Option ROM technology will be included in future documentation sponsored by Intel.

2.2.2 EFI Option ROMs

Recommended

EFI defines a mechanism for EFI drivers to reside in an Option ROM. The [EFI1] specification includes processor architecture-dependent drivers for both IA-32 and IA-64.

2.2.3 EFI Portable Option ROMs

Optional

One of the goals of a future EFI Specification is to provide a comprehensive and common Portable Option ROM implementation. Specific key goals include:

- Abstract and extensible design.
- CPU independence.
- OS independence.
- Allow multiple formats in a single Option ROM.
- Facilitate the removal of legacy infrastructure.

A portable Option ROM uses the EFI Option ROM format and includes a portable EFI boot service driver or application. Portable Option ROMs are operating system independent and achieve CPU architecture independence through the following means.

- Exclusive use of EFI Services.
- Portable byte stream format.

2.3 UNIX ACPI Usage Model

The Advanced Configuration and Power Interface (ACPI) specification provides a methodology for configuring systems.

By using ACPI the operating system does not have to rely on proprietary methods for configuring the system hardware. ACPI interfaces provide a configuration layer that allows the operating system and hardware to exist and evolve independently of each other. ACPI replaces legacy platform interfaces, such as PnP BIOS calls and the Multi-Processor Specification, and provides a means for an orderly transition away from these legacy interfaces. ACPI provides a flexible and extensible method to support advanced architectures.

2.3.1 Hardware/Firmware Requirements

The hardware/firmware requirements necessary for an ACPI implementation are described in detail in the *Developers Interface Guide for IA-64 Servers* [DIG64] and the *ACPI Specification* [ACPI1]. The UDIG focuses on the requirements necessary to implement ACPI on server systems, and primarily on system configuration using ACPI in a UNIX environment.

2.3.1.1 Follow DIG-64 Hardware/Firmware Guidelines for ACPI

Required

2.3.2 UNIX Operating System Requirements

The UNIX operating system requirements necessary for an ACPI implementation are described below. The UDIG focuses on the requirements necessary to implement ACPI on server systems running UNIX, and primarily on system configuration using ACPI. The power management mechanisms of ACPI may be implemented based on an OEM's particular market requirements, e.g., to support dynamic device configuration.

2.3.2.1 Support for Memory Reporting Interfaces

Required

The UNIX operating system must discover all memory using the EFI memory reporting interface GetMemoryMap(). This applies equally to all types of ACPI memory as well as normal physical memory, device memory, and other types defined by [EFI1].

2.3.2.2 Support for ACPI Devices Defined in the ACPI Specification

Recommended

Each operating system should support its own subset of the devices defined in [ACPI1] specification. These devices are hardware components outside of the core chipset. Examples of devices that are supported via ACPI include LCD panels, video adapters, IDE CD-ROM, hard disk controllers, COM ports, etc.

2.3.2.3 Provide an ACPI AML Interpreter

Required

Each operating system must provide an AML interpreter to create the ACPI namespace, and execute control methods for device configuration.

2.3.2.4 Provide Configuration Support

Required

ACPI requires that the OS use existing resource configuration techniques, such as PCI configuration, and combine the resources enumerated by ACPI into the OS conflict detection and resolution algorithms supporting system events.

2.3.2.5 Provide Support for System Events via SCI and GPE

Recommended

The operating system should support event handling using the System Control Interrupt (SCI) and the General Purpose Event (GPE) mechanisms defined by the ACPI specification [ACPI1].

2.3.2.6 Recognize Devices through the ACPI Namespace

Required

The operating system must have the capability to make the association between the _ADR and the bus enumeration mechanism, even if a device is not present. Devices that exist on a bus that has enumeration capability (for example the PCI bus) do not require ACPI enumeration. Devices

that do not require ACPI enumeration are still required to have a _ADR object as defined by the [ACPI1]. This object is used to map the device to the unique address of the device as it appears to the enumeration mechanisms of the bus.

2.3.2.6.1 Devices that require ACPI enumeration

Required

For devices enumerated using ACPI the operating system must identify devices in the ACPI namespace using the _HID object embedded in the device definition. The _HID Object evaluates to a device's Plug and Play Hardware ID. A device with a _HID may also have an optional _CID that contains PnP IDs for compatible devices [ACPI1].

2.3.2.7 Use Resources Presented in the Namespace

Required

ACPI requires that devices use the _CRS named object to define the resources used by the device. A device may also have a _PRS method defined that informs the operating system of other possible resource definitions for this device. The _CRS and _PRS objects, along with others, are defined in the ACPI specification [ACPI1].

2.3.2.8 Respond to Device Insertion and Removal

Recommended

The operating system should support device insertion and removal. This implies the capability to load and unload (start/stop) drivers as devices appear and disappear.

2.3.2.9 Provide Device Power Management Support for D0 and D3

Recommended

The operating system should have an understanding or at least tolerance of power management. This means that the operating system should allow devices to change power states and tolerate the latencies resulting from the power state transitions. This is particularly useful for device insertion and removal.

2.3.3 PCI Hot-plug

Recommended

The operating system should support PCI Hot-plug using ACPI mechanisms [ACPI2, and PCI Hot-Plug Specification, Revision 1.0].

2.3.4 ACPI Subsystem

This section describes the guidelines for the implementation of an ACPI driver, hereinafter, referred to as an ACPI subsystem [ACPI CA].

2.3.4.1 Implement an ACPI Subsystem

Required

Operating systems must provide an ACPI subsystem.

2.3.4.2 Implement ACPI subsystem External Interfaces

Recommended

To allow common applications to use the ACPI subsystem, the subsystem's external interfaces should comply with the reference implementation [ACPI CA].

2.3.4.3 Implement ACPI Subsystem Operating System Services

Recommended

If the operating system uses the reference implementation, it must provide services for memory management, scheduling, synchronization and mutual exclusion, interrupt handling, and stream I/O [ACPI CA].

Chapter 3: Device Drivers and Services

This chapter specifies requirements and guidelines on a device driver in UDIG-compliant systems, and the driver services and infrastructure provided in UDIG-compliant systems. These requirements and guidelines facilitate the timely development, support, validation, and testability of third-party I/O devices and drivers in Intel®-based Servers running UNIX or operating systems like UNIX. Use of these requirements and guidelines results in a high degree of customer confidence in the support and usability of I/O devices and corresponding drivers on Intel-based UNIX servers.

This chapter should be of general interest to people developing drivers, driver services, and driver infrastructure. However, the UDI Environment Implementer's Guide section is specifically aimed at services/infrastructure developers.

The basis for this chapter is the Uniform Driver Interface (UDI). More information on UDI is available at the Project UDI Web site: http://www.project-udi.org. UDI solves the problem of creating different drivers. Without UDI, developers must create different drivers for different CPU architectures and operating systems, all of which must talk to the same hardware devices. With UDI, developers write one driver. This driver is compiled and executed across all CPU architectures and operating systems that support UDI. Some of the core features of UDI that enable this capability are:

1. Implicit synchronization via regions.
2. Inter-module communication via channels.
3. Explicit definition of all driver interfaces.

This chapter is organized into the following main sections.

Source Specifications	Defines, through reference to the *UDI Specifications* [UDI1], the source-level interface requirements for UDIG-compliant device drivers and services.
Binary Specifications	Defines the binary interface requirements for a device driver on IA-32 and IA-64 platforms. UDIG-compliant systems support these binary interfaces, providing binary portability to a compliant device driver.
Technology Profiles	Provides the required and optional I/O technologies that are provided in UDIG-compliant systems.
UDI Environment Implementer's Guide	Provides guidelines for system developers to facilitate the development of driver services and infrastructure in UDIG-compliant systems.
Reference Implementation	Describes tools that are available to implementers.

Driver Development Kits	UDI Driver Development Kits (DDKs) are expected to be available from various OSVs. The Kit provides UDI build and runtime environments, including driver packaging and build tools, and other driver development materials, such as test suites and driver writer's guides.

3.1 Source Specifications

The I/O device driver support defined in this document provides for the broadest possible range of I/O devices on Intel-based servers. To accomplish this goal and accommodate future growth, the *Uniform Driver Interface (UDI) Specifications* [UDI1] serve as the foundation for UDIG-compliant device driver support. The UDIG is based on the [UDI1].

3.1.1 UDI Specifications [UDI1]

The *UDI Specifications* define the environment for an I/O device driver by defining the interface between a driver and the surrounding operating system. These specifications insulate a device driver from the implementation details of the operating system and server platform the operating system is installed on. Yet, the specifications are flexible enough to allow the driver full control of its device within that operating system and platform environment.

The *UDI Specifications* are divided into a number of separate specifications based on technology areas. The Technology Profiles section in this chapter lists the specific UDI specifications that apply to UDIG-compliant systems.

The *UDI Specifications* define source-level interface requirements for a device driver. The specifications also support the ability to easily define binary-level interface definitions based on the source-level definitions. Binary interface definitions for IA-32 and IA-64 are defined in the next section.

3.2 Binary Specifications

Binary-level specifications, referred to as Architected Binary Interface (ABI) bindings in the [UDI1], are defined in this section for binary portability of UDI drivers for IA-32 and IA-64 systems. UDIG-compliant IA-32 systems must support the IA-32 binding; UDIG-compliant IA-64 systems must support the IA-64 ABI binding.

As described in the *UDI Core Specification* [UDI2], a UDI ABI binding has the following components.

- A processor architecture, supported instruction sets, endianness modes, and corresponding subdirectory names for the UDI packaging format.
- An associated runtime architecture that defines the procedure calling conventions, register usage, stack conventions, data layout, and so on.

- Binary bindings to the source-level UDI specifications: specifies the sizes of UDI fundamental data types, and the binary-portability requirements of implementation-dependent UDI macros.
- Building the driver object: specifies the object file format and the encapsulation of the static driver properties in the object file.

3.2.1 IA-32 ABI Binding

The IA-32 ABI binding defines binary bindings for UDI driver modules that run on IA-32 processors. The driver is compiled once for IA-32 and distributed to any conforming IA-32 platform.

3.2.1.1 Processor Architecture

3.2.1.1.1 Supported processor types

Required

The IA-32 binding works across any processor that conforms to the 32-bit Intel architecture. This includes the Intel processor types listed in Table 3-1. For maximum binary portability across IA-32 processors, you must compile the driver so that it does not depend on any specific IA-32 target processor type. Driver packages use the subdirectory named specified in Table 3-1 for its IA-32 driver binaries. All binary driver packages for IA-32 must include at least a generic **IA32** binary, but may also include one or more specific binaries.

3.2.1.1.2 Endianness

Required

The IA-32 ABI binding supports only little endian IA-32 binary UDI modules.

3.2.1.1.3 Driver packaging subdirectories

Required

The UDI packaging format specification defines a directory hierarchy. This hierarchy must contain the various components of a UDI driver "package". The package includes the files that must be provided with a driver to make it installable and useable. In a driver package, one or more **bin** directories contain the driver binaries. Beneath the bin directory, **<abi>** subdirectories contain the driver binaries for a particular ABI. The **<abi>** subdirectory names defined for the IA-32 ABI, with their corresponding processor types, are defined in Table 3-1 below. For more information, see [UDI6] for additional details.

Note that the Pentium II Xeon™ processor and Celeron™ Processor share the same instruction set with the Pentium II processor and thus correspond to **IA32_P2**. Likewise, the Pentium III Xeon processor corresponds to **IA32_P3**.

Table 3-1 – <abi> subdirectories

Directory name	Processor type
IA32	Any IA-32 Processor
IA32_P3	Pentium III Processor
IA32_P2	Pentium II Processor
IA32_PMMX	Pentium Processor with MMX™ Technology
IA32_PPRO	Pentium Pro Processor
IA32_P	Pentium Processor
IA32_486	Intel486™ Processor
IA32_386	Intel386™ Processor

3.2.1.2 Runtime Architecture

Required

The runtime architecture for this IA-32 ABI binding shall be the IA-32 runtime architecture as defined in the *System V Application Binary Interface Specification* [SAB1] and the *System V Application Binary Interface Intel 386 Architecture Processor Supplement Specification* [SAB2].

In particular, a compliant environment must provide an execution environment, implement object file support, and provide linking and loading.

3.2.1.3 Binary Bindings to the Source Specifications

This section defines the binary bindings to the source-level UDI Specifications. Only a few aspects of the source-level UDI Specifications are implementation-dependent, requiring additional specification for binary portability. These fall into three categories: the sizes of fundamental UDI data types, the binary-portability requirements of implementation-dependent UDI macros, and the specification of UDI functional interfaces, other than the UDI utility functions, that are allowed to be implemented as macros. Utility functions in UDI can always be implemented as macros without breaking binary portability. For more information, see the [UDI2].

3.2.1.3.1 Sizes of UDI-Specific data types

Required

As indicated in the introduction of [UDI2], an ABI specification must specify the size of each of the following data types, defined as follows for the IA-32 ABI binding.

Table 3-2– UDI specific data types

Data Type	Size	Description
void *	32 bits	Pointer type
udi_size_t	32 bits	Abstract type – Size type
udi_index_t	8 bits	Abstract type – Index type
udi_channel_t	32 bits	Opaque type – Channel handle
udi_constraints_t	32 bits	Opaque type – Constraints handle
udi_timestamp_t	32 bits	Opaque type – Timestamp type
udi_pio_handle_t	32 bits	Opaque type – PIO handle
udi_dma_handle_t	32 bits	Opaque type – DMA handle

3.2.1.3.2 Implementation-Dependent macros

Required

An implementation-dependent macro is an interface specified to be a macro, other than a utility macro, in a UDI Specification [UDI1]. The macro expansion definition is implementation-specific. Such macros may contain environment and platform dependencies that provide binary portability. Such dependencies must be hidden behind an external function call. The ABI must specify any such external symbol references.

Only the *UDI Core Specification* [UDI2] and the *UDI Physical I/O Specification* [UDI6] may specify implementation-dependent macros. There are only two such macros in those specifications: the **UDI_HANDLE_ID** macro and the **UDI_HANDLE_IS_NULL** macro.

For the IA-32 ABI, the **UDI_HANDLE_ID** macro must return the exact, non-transformed contents of the specified handle, without producing any external symbol references. The actual C definition of the macro may be provided in an implementation-dependent manner in udi.h. Depending on the actual type definition of the handle type, acceptable definitions include

```
#define UDI_HANDLE_ID(handle, handle_type) \
                ((void *)(handle))
```

or

```
#define UDI_HANDLE_ID(handle, handle_type) \
                (*(void **)&(handle))
```

For the IA-32 ABI, all handle types are 32 bits for the IA-32 ABI. Therefore, the **UDI_HANDLE_IS_NULL** macro must return TRUE if the handle value is zero. This can be done using appropriate **void *** casts and comparing against NULL. The actual C definition of the macro may be provided in an implementation-dependent manner in udi.h. Depending on the actual type definition of the handle type, acceptable definitions include

```
#define UDI_HANDLE_IS_NULL(handle, handle_type) \
                ((void *)(handle) == NULL)
```

or

```
#define UDI_HANDLE_IS_NULL(handle, handle_type) \
                (*(void **)&(handle) == NULL)
```

3.2.1.3.3 UDI functions implemented as macros

Required

As previously noted, utility functions in UDI can always be implemented as macros without breaking binary portability. However, ABI bindings must require each other UDI functional interface to be implemented in one of two ways. One way is to implement the UDI functional interface as an external function call. Another way is to optionally implement the interface as a macro with any environment or platform dependencies hidden behind an external function call. This external function call is specified by the ABI as part of the macro expansion.

For the IA-32 ABI, the only non-utility function that may be implemented as a macro is **udi_assert**, which must call the following function when the assertion condition test fails:

```
void __udi_assert(const char *expr,
                  const char *filename,
                  int lineno);
```

3.2.1.4 Building the Driver Object

The [UDI2] defines the generic aspects of the packaging and distribution of UDI drivers, whether distributed in source or binary format. The ABI binding defines the object file format and the binding of the static driver properties to the driver's object files.

3.2.1.4.1 Object file format

Required

The object file format for this IA-32 binding shall be the ELF-32 object file format as defined in the [SAB1] and the [SAB2].

Any driver package that includes IA-32 binary modules must include IA-32 binary modules for all modules in the driver package.

3.2.1.4.2 Static driver properties encapsulation

Required

A driver's static driver properties shall be encapsulated in the ELF-32 object file of the driver's primary module, and in a special no-load section of that object file called **.udiprops**. The content of this section is ASCII-encoded, null-terminated strings from the **udiprops.txt** file. The only difference between the **udiprops.txt** file and the **.udiprops** section is that:

- Comments are removed.
- Extraneous white space is removed.
- Extraneous line terminations for a property value is removed so that each property value is a single, null terminated string.

3.2.2 IA-64 ABI Binding

The IA-64 ABI binding defines binary bindings for UDI driver modules that run on IA-64 processors. This definition allows the driver to be compiled once for a target endianness for IA-64 systems and be distributed to any conforming IA-64 platform supporting that endianness.

3.2.2.1 Processor Architecture

3.2.2.1.1 Supported processor types

Required

The IA-64 binding is designed to work across any processor that conforms to the 64-bit Intel architecture. This includes the Intel processor types listed in Table 3-3. For maximum binary portability across IA-64 processors, you must compile the driver in a way that does not depend on any specific IA-64 target processor type. In this case, the driver uses the subdirectory named **IA64_LE** or **IA64_BE** for its IA-64 driver binaries. All binary driver packages for IA-64 must include at least a generic **IA64_LE/BE** binary and may also include one or more specific binaries.

3.2.2.1.2 Endianness

Required

The IA-64 ABI binding supports both big and little endian IA-64 binary UDI modules, but UDIG-compliant IA-64 systems are only required to support one endianness or the other.

Recommended

It is recommended that little endian be supported.

3.2.2.1.3 Driver packaging subdirectories

Required

The [UDI1] must define a directory hierarchy that contains the various components of a UDI driver "package", such as the files provided with a driver to make it installable and useable. Each driver package has one or more **bin** directories containing the driver binaries. The bin directory has **<abi>** subdirectories containing the driver binaries for a particular ABI. The **<abi>** subdirectory names defined for the IA-64 ABI, with their corresponding processor types, are defined in Table 3-3. See [UDI1] for additional details.

Table 3-3 – <abi> directories for various processors

Directory Name	Processor Type
IA64_LE	Any IA-64 Processor in little-endian mode
IA64_BE	Any IA-64 Processor in big-endian mode
IA64_LE_ITANIUM	Intel® Itanium™ processor in little endian mode
IA64_BE_ITANIUM	Itanium processor in big endian mode

3.2.2.2 Runtime Architecture

Required

The runtime architecture for this IA-64 ABI binding shall be the IA-64 runtime architecture as defined in the *IA-64 Software Conventions and Runtime Architecture Guide* [SRC1] from Intel and Hewlett-Packard. This is the definition of the runtime architecture that is common to all operating environments on the IA-64 architecture.

3.2.2.3 Binary Bindings to the Source Specifications

This section defines the binary bindings to the source-level UDI Specifications. Only a few aspects of the source-level UDI Specifications are implementation-dependent, requiring additional specification for binary portability. These fall into three categories.

- The sizes of fundamental UDI-specific data types.
- The binary-portability requirements of implementation-dependent UDI macros.
- The specification of UDI functional interfaces, other than the UDI utility functions, that are allowed to be implemented as macros. Utility functions in UDI can always be implemented as macros without breaking binary portability. See the [UDI2] for details.

3.2.2.3.1 Sizes of UDI-Specific data types

Required

As indicated in the *UDI Core Specification* [UDI2], an ABI specification must specify the size of each of the following data types for the IA-64 ABI binding.

Table 3-4 – Data types for the IA-64 ABI binding

Data Type	Size	Description
void *	64 bits	Pointer type
udi_size_t	64 bits	Abstract type – Size type
udi_index_t	8 bits	Abstract type – Index type
udi_channel_t	64 bits	Opaque type – Channel handle
udi_constraints_t	64 bits	Opaque type – Constraints handle
udi_timestamp_t	64 bits	Opaque type – Timestamp type
udi_pio_handle_t	64 bits	Opaque type – PIO handle
udi_dma_handle_t	64 bits	Opaque type – DMA handle

3.2.2.3.2 Implementation-Dependent macros

Required

An implementation-dependent macro is an interface specified to be a macro, other than a utility macro, in a UDI Specification [UDI1]. The Macro expansion definition is implementation-specific. Such macros may contain environment and platform dependencies that, to provide binary portability, must be hidden behind an external function call; any such external symbol references must be specified in the ABI.

Only the *UDI Core Specification* [UDI2] and the *UDI Physical I/O Specification* [UDI6] may specify implementation-dependent macros. There are only two such macros in those specifications: the **UDI_HANDLE_ID** macro and the **UDI_HANDLE_IS_NULL** macro.

For the IA-64 ABI, the **UDI_HANDLE_ID** macro must return the exact, non-transformed contents of the specified handle, without producing any external symbol references. The actual C definition of the macro may be provided in an implementation-dependent manner in udi.h. Depending on the actual type definition of the handle type and the target programming model (e.g., LP64 vs. P64), acceptable definitions include

```
#define UDI_HANDLE_ID(handle, handle_type) \
                ((void *)(handle))
```

or

```
#define UDI_HANDLE_ID(handle, handle_type) \
                (*(void **)&(handle))
```

For the IA-64 ABI, all handle types are 64 bits for the IA-64 ABI. Therefore, the **UDI_HANDLE_IS_NULL** macro must return TRUE if the handle value is zero. This can be done using appropriate **void *** casts and comparing against NULL. The actual C definition of the macro may be provided in an implementation-dependent manner in udi.h. Depending on the actual type definition of the handle type, acceptable definitions include

```
#define UDI_HANDLE_IS_NULL(handle, handle_type) \
                ((void *)(handle) == NULL)
```

or

```
#define UDI_HANDLE_IS_NULL(handle, handle_type) \
                (*(void **)&(handle) == NULL)
```

3.2.2.3.3 UDI functions implemented as macros

Required

As previously noted, you can implement utility functions in UDI as macros without breaking binary portability. However, ABI bindings must require each other UDI functional interface to be implemented in one of two ways. One way is to implement the interface as an external function call. Another way is to optimally implement the interface as a macro with any environment or platform dependencies hidden behind an external function call. This external function call is specified by the ABI as part of the macro expansion.

For the IA-64 ABI, the only non-utility function that may be implemented as a macro is **udi_assert**, which must call the following function when the assertion condition test fails:

```
void __udi_assert(const char *expr,
                  const char *filename,
                  int lineno);
```

3.2.2.4 Building the Driver Object

The [UDI2] defines the generic aspects of the packaging and distribution of UDI drivers, whether you are distributing it in source or binary format. The ABI binding defines the object file format, and the binding of the static driver properties to the driver's object files.

3.2.2.4.1 Object file format

Required

The object file format for this IA-64 binding shall be the ELF-64 object file format as defined in the *ELF-64 Object File Format Specification* [ELF1], and in the *Processor-Specific ELF Supplement for IA-64* [ELF2].

The appropriate endian bit indicating the target endianness of the driver module must be set in the ELF-64 file header, as defined in the [ELF1].

Any driver package that includes little endian IA-64 binary modules must include full IA-64 binary modules for all modules in the driver package. It is recommended that IA-32 little endian binary modules be supplied as well.

System V ABI [SAB3] updates are available at http://www.sco.com/developer/devspecs/

3.2.2.4.2 Static driver properties encapsulation

Required

A driver's static driver properties shall be encapsulated in the ELF-64 object file of the driver's primary module, in a special no-load section of the object file called **.udiprops**. The content is ASCII-encoded, null-terminated strings from the **udiprops.txt** file. The only difference between the **udiprops.txt** file and the **.udiprops** section is:

- Comments are removed.
- Extraneous white space is removed.
- Extraneous line terminations for a property value are removed so that each property value is a single, null terminated string.

3.3 Technology Profiles

While IA-64 UDIG-compliant platforms are required to include the I/O hardware technologies specified in [DIG64], this section lists those technologies that are supported through portable driver interfaces. Corresponding software technologies are designated as either unconditionally "required" or "required when present". Unconditionally required technologies must be supported on all UDIG-compliant systems by providing the corresponding portable driver interfaces specified in this section. Technologies that are required when present are only required on systems that support the corresponding hardware technology. When provided, they must use the corresponding portable driver interfaces specified in this section.

To support the various types of I/O technologies that exist, UDI has defined three general classes of interfaces: driver services (interfaces that a driver can call to obtain various services from the embedding environment), metalanguages (interfaces for communication between driver modules, typically grouped into technology areas such as SCSI, networking, or block storage), and bus bindings (bindings of how the driver services are used with a particular type of I/O bus such as PCI, EISA, or VME). The required technologies must be profiled relative to these three classes of UDI interfaces.

To learn more about what metalanguages are available, see http://www.project-udi.org/metalanguages.html.

3.3.1 Driver Services

There are two UDI Specifications that specify general services available to the driver by the UDI environment: the [UDI2] and the [UDI6]. These services must be provided in UDIG-compliant systems as described below.

3.3.1.1 UDI Core Specification

Required

The [UDI2] defines the core set of UDI interfaces that are available to all UDI drivers and that are required to be provided by all UDI environment implementations. Thus, the implementation of [UDI2] is required in UDIG-compliant systems.

3.3.1.2 UDI Physical I/O Specification

Required

The [UDI6] defines UDI interfaces for PIO, DMA, and interrupts. Drivers that use such interfaces are called "physical I/O drivers". While the UDI Specifications do not require all environments to support physical I/O drivers, those that do must conform to the generic portions of the [UDI6]. UDIG-compliant systems are required to support physical I/O drivers and must therefore implement the interfaces in the [UDI6].

3.3.2 Metalanguages

On UDI systems, metalanguage libraries and mappers must be provided to support external bus and device class technologies such as SCSI, networking, USB, and block storage. External bus and device class technologies listed in the [DIG64] are supported by the following metalanguages.

Table 3-5 – Metalanguages

Metalanguage	Compliance	Comments
SCSI	Required	Support for SCSI adapters, both those that interface to a parallel SCSI bus and those that provide (together with the driver) SCSI encapsulated over a serial link such as Fibre Channel, is required in UDIG-compliant systems. Therefore, UDIG-compliant systems are required to support the *UDI SCSI Driver Specification* [UDI4].
Networking	Required	Support for Network Interface Controllers (NICs) is required on UDIG-compliant systems. Therefore, UDIG-compliant systems are required to support the *UDI Network Driver Specification* [UDI5].
USB	Required when present	UDIG-compliant systems that support USB devices must support the *Open USBDI Specification* [USB1].

Currently, specification for Block Storage, Bus Bridge, and Intelligent I/O (I2O) are under development.

3.3.3 Bus Bindings

The [UDI1] defines a number of bus bindings that establish UDI interfaces specific to particular I/O bus types. UDIG-compliant systems that support any of the I/O bus types below are required to support the corresponding *UDI Bus Binding Specification*.

Table 3-6 – UDI Bus Binding Specification

Bus	Compliance	Comments
PCI	Required	Support for the *UDI PCI Bus Binding Specification* [UDI3] is required.

Currently, Bus Bindings for System, ACPI, ISA, EISA, and VME are under development under the guidance of Project UDI.

3.4 UDI Environment Implementer's Guide

The UDI Environment Implementer's Guide is expected to be available from UDI. The Uniform Driver Interface (UDI) specifications define a complete environment for the development of a device driver. This includes a driver architectural model and the complete set of services and other interfaces needed by a device driver to control its device or pseudo-device, and to interact properly with the rest of the system in which it operates. This environment in which a driver operates, called the UDI environment, must be provided in each operating system or operating environment in which UDI drivers operate. See the Project UDI web page at http://www.project-udi.org for additional information.

3.5 Reference Implementation

An implementation of the UDI environment services and sample UDI drivers are provided by Project UDI. This open source code implementation is released into the public domain to enable quick spread of the UDI technology to a wide range of platforms and drivers. It is expected to initially provide support for most of the major UNIX vendors as well as Linux*.

This reference implementation has been co-developed by a number of OSVs and IHVs with this goal – easing the spread of the UDI technology – in mind, and the result is an implementation of the UDI environment in which much of the environment code is common across the range of supported operating systems. For example, UDI's buffer management services are coded as a common piece of source code with OS dependencies abstracted via a handful of OS-dependent macros. Thus, to get these buffer management services implemented for a new operating system all that the OS implementor has to do is to fill in the handful of macros to map to the needs of that operating system. Some operating systems may require additional work, depending on how closely the operating system's kernel services map to the reference implementation assumptions, but in any case this is a significant aid to new development.

The reference implementation is a tool that is used by the industry at large in the development of UDI environments and drivers. See the Project UDI Web page at http://www.project-udi.org for additional information.

3.6 Driver Development Kits

UDI Driver Development Kits (DDKs) are expected to be available from various OSVs. The Kit provides UDI build and runtime environments, including driver packaging and build tools, and other driver development materials, such as test suites and driver writer's guides. See the Project UDI Web page at http://www.project-udi.org for additional information.

This Web page also includes introductory and tutorial materials useful for UDI device driver writers.

Appendix A: ACPI Component Architecture: ACPI Subsystem

Table of Figures

Introduction

This appendix does not contain extensions for IA-64. As such, it is an interim document describing IA-32 implementation. The IA-64 extensions will be incorporated into a future release of the document after IA-64 architecture public disclosure.

Document Structure

This appendix consists of five major sections:

- **Introduction:** A brief overview of the ACPI Component Architecture (CA) and the interfaces for both the OS-independent and OS-dependent components.
- **Using the ACPI Subsystem:** Contains the information required to use the subsystem (both the OSI and OSD components) and the related interfaces. There is a summary of the computational and architectural model that is implemented by the ACPI component architecture, as well as a summary of the major data types and data structures that are exposed via the external interfaces.
- **OS-Independent Component Interfaces:** This section includes a detailed description of the programmatic interfaces to the OS-independent component of the ACPI subsystem.
- **OS-Dependent Component Interfaces:** This section includes a detailed description of the programmatic interfaces that must be implemented in the OS-dependent component.
- **User Guide:** This section includes tips and techniques on using the OS-independent interfaces, and implementing the OS-dependent interfaces to host a new operating system.

Rationale and Justification

The complexity of the ACPI specification leads to a lengthy and difficult implementation in operating system software. The purpose of the ACPI component architecture is to simplify ACPI implementations for operating system vendors (OSVs) by providing major portions of an ACPI implementation in OS-independent ACPI modules that can be integrated into any operating system. The OS-independent software can be hosted on any operating system by creating a small and relatively simple OS-dependent translation service between the OS-independent component of the ACPI subsystem and the host operating system.

Reference Documents

- *Advanced Configuration and Power Interface Specification*, Revision 1.0b, February 8, 1999

Overview of the ACPI Component Architecture (ACPI CA)

The ACPI Component Architecture (ACPI CA) defines and a group of software components that combine to create an implementation of the ACPI specification. A major goal of the architecture is to maximize the amount of code that has no dependencies on any individual operating system (the OS-independent code) and to minimize the amount of code that is specific to a particular operating system (the OS-dependent code). The components of the architecture include:

- A user interface to the power management and configuration features.
- A power management and power policy component (OSPM).
- A configuration management component.
- ACPI-related device drivers (for example, drivers for the Embedded Controller, SMBus, Smart Battery, and Control Method Battery.
- An ACPI Subsystem component that provides the fundamental ACPI services.

This document describes the ACPI Subsystem portion of the architecture only. Other components of the Component Architecture are described in related documents.

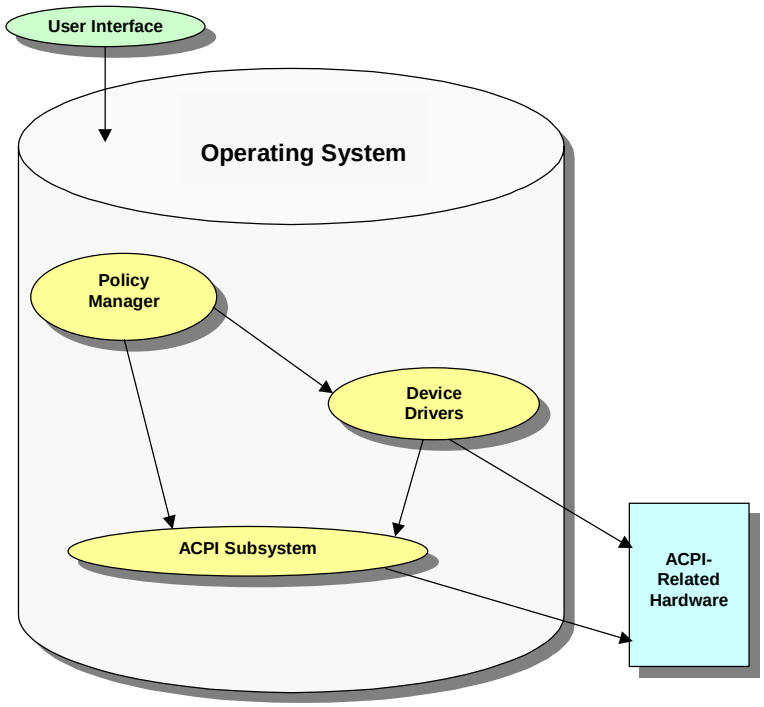

Figure 1. The ACPI Component Architecture

Overview of the ACPI Subsystem

The ACPI Subsystem implements the low level or fundamental aspects of the ACPI specification. Included are an AML interpreter, ACPI namespace, table, and device support, and event handling. Similar to the other components of the ACPI Component Architecture, the ACPI Subsystem is divided into a part that is independent of specific operating systems and a part that is implemented to run on one specific OS. Thus, the ACPI Subsystem consists of two major software components:

- An OS-independent component (OSI) provides the fundamental ACPI services that are independent of any particular operating system.
- The OS-dependent component (OSD) provides the conversion layer that interfaces the OS-independent component to a particular host operating system.

When combined into a single loadable software module such as a device driver or kernel subsystem, these two major components form the *ACPI Subsystem*. Throughout this document, the term "ACPI Subsystem" refers to the combination of the OSI and OSD components into a single module, driver, or load unit.

ACPI OS-Independent Component

The OS-independent component (OSI) supplies the major building blocks or subcomponents that are required for any ACPI implementation—including an AML interpreter, a namespace manager, ACPI event and resource management, and ACPI hardware support.

One of the goals of the OSI is to provide an abstraction level high enough such that the OSD does not need to understand or know about the very low-level ACPI details. For example, all AML code is hidden from the OSD and host operating system. Also, the details of the ACPI hardware are abstracted to higher level software interfaces.

The OSI implementation makes no assumptions about the host operating system or environment. The only way it can request operating system services is via interfaces provided by the *OS-dependent component*.

The primary user of the services provided by the OS-independent component is the OS-dependent component, since it is the OS-dependent component that provides an external interface appropriate for the host operating system (for example, the ACPI subsystem may be constructed to appear as a device driver to the host OS).

ACPI OS-Dependent Component

The OS-dependent component (OSD) operates as a bi-directional translation service for both requests from the host OS to the ACPI subsystem, and from the ACPI subsystem to the host OS. These two functions are independent of each other in many ways. In one direction, the OSD translates host OS requests from the native format into one or more calls to the ACPI OS-independent component. In the other direction, the OSD implements a generic set of OS service interfaces by using the primitives available from the host OS.

Because of its nature, the OS-dependent component must be implemented anew for each supported host operating system. There is a single OS-independent component, but there must be

an OS-dependent component for each operating system supported by the ACPI component architecture.

The primary function of the OSD in the ACPI Component Architecture is to be the small glue layer that binds the much larger OSI to the host operating system. Because of the nature of ACPI itself—such as the requirement for an AML interpreter and management of a large namespace data structure—most of the implementation of the specification is independent of any operating system services. Therefore, the OSI is the larger of the two components.

The overall ACPI Component Architecture in relation to the host operating system is diagrammed below.

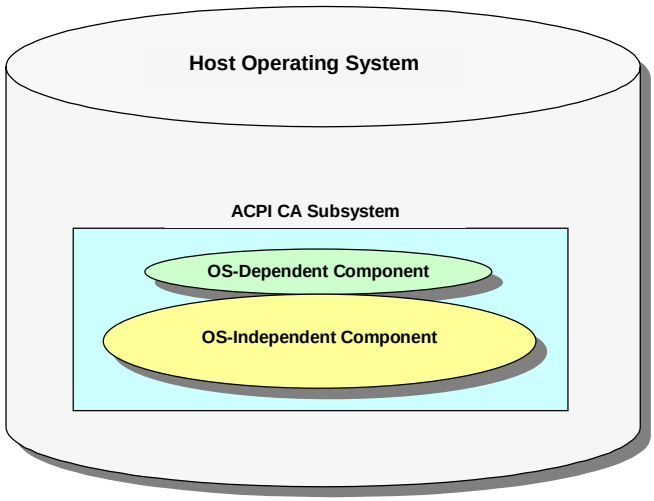

Figure 2. ACPI Subsystem Architecture

Relationships Between the Host Operating System, OSD, and OSI

Host Operating System Interaction

The Host Operating System makes requests to the ACPI subsystem using the OS-dependent interface that is defined between the OSD component and the Host OS. The host typically does *not* make calls directly to the OSI component because the **Acpi*** interfaces are typically too low-level for the host. Also, the direct call interface to the OSI is probably not appropriate for the host-to-OSD interface—a device driver interface is far more likely to be used instead. In this sense, the OSD component acts as a "wrapper" for the OSI component.

The OSD component "calls up" to the host operating system whenever operating system services are required, either for the OSD itself, or on behalf of the OSI component. All native calls directly to the host are confined to the OS-dependent component, for obvious reasons.

OS-Dependent Component Interaction

The OS-dependent component implements two types of interfaces, one for each of two distinct callers:

- The Host OS interface is the only external (public) interface from the host OS into the ACPI subsystem. The mechanism used to implement this interface can be whatever is appropriate for the host OS—such as a device driver or internal subsystem interface. The OSD-host OS interface receives ACPI requests from the operating system and translates them into one or more requests to the OSI component. Therefore, the OSD calls the OSI to implement the host OS interface.

- The **Osd*** interfaces provide common operating system services to the OSI such as memory allocation ,mutual exclusion, hardware access, and I/O. The OSI component uses these interface to gain access to OS services in an OS-independent manner. Therefore, the OSD component makes calls to the host operating system to implement the **Osd*** interface.

OS-Independent Component Interaction

The OS-independent component implements a single type of interface:

- The **Acpi*** interfaces provide the actual ACPI services. When operating system services are required during the servicing of an ACPI request, the OSI makes requests to the host OS indirectly via the fixed **Osd*** interfaces.

The diagram below illustrates the relationships and interaction between the various architectural elements by showing the flow of control between them. Note that the host never calls the OSI directly—it accesses services that are provided by the OSD. Also, the OSI never calls the host directly—instead it makes calls to the **Osd*** interfaces in the OSD. It is this level of indirection in both directions that allows the OSI to be truly operating system independent.

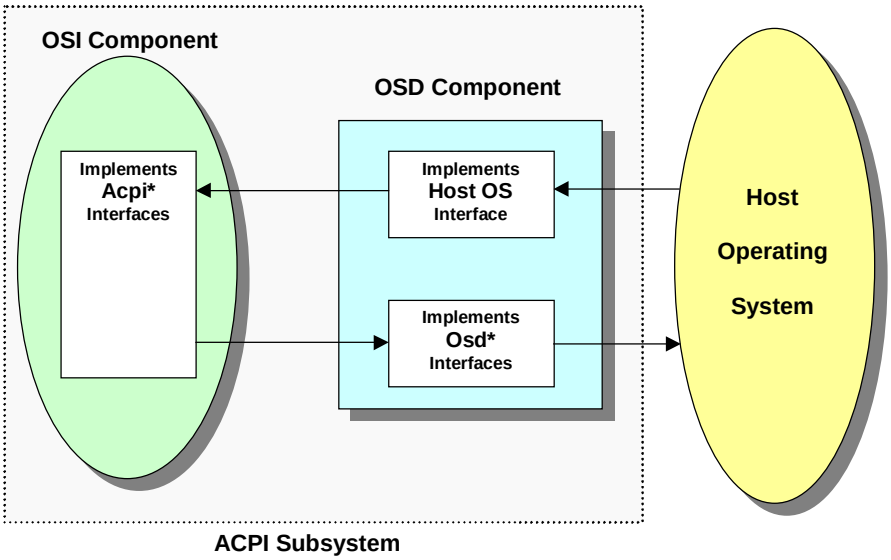

Figure 3. Interaction Between the Architectural Components

Architecture of the OS-Independent Component (OSI)

The OS-independent component is divided into several logical modules or sub-components. Each module implements a service or group of related services. This section describes each sub-component and identifies the classes of external interfaces to the components, the mapping of these classes to the individual components, and the interface names.

These ACPI modules are the OS-independent parts of an ACPI implementation that can share common code across all operating systems. These modules are delivered in source code form (the language used is ANSI C), and can be compiled and integrated into an OS-specific ACPI driver or subsystem (or whatever packaging is appropriate for the host OS.)

The diagram below shows the various internal modules of the OS-independent component and their relationship to each other. The AML interpreter forms the foundation of the component, with additional services built upon this foundation.

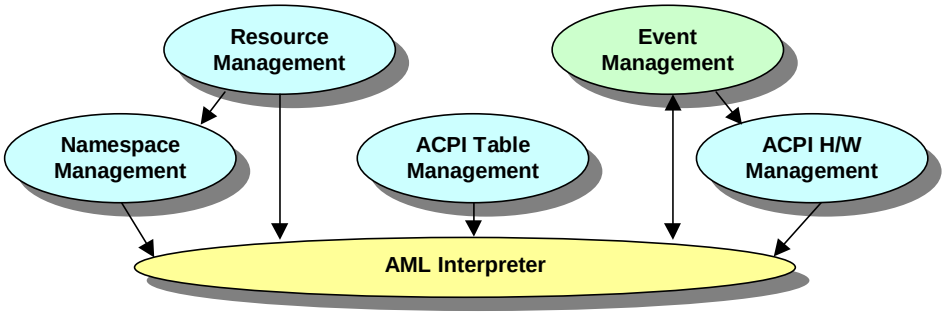

Figure 4. Internal Modules of the OS-Independent Component

AML Interpreter

The AML interpreter is responsible for execution of the AML code that is provided by the computer system vendor. Most of the other services are built upon the AML interpreter. Therefore, there are no direct external interfaces to the interpreter. The services that the interpreter provides to the other services include:

- AML Control Method Execution
- Evaluation of Namespace Objects

ACPI Table Management

This component manages the ACPI tables such as the RSDT, FACP, FACS, DSDT, etc. The tables may be loaded from the firmware or directly from a buffer provided by the host operating system. Services include:

- ACPI Table Parsing
- ACPI Table Verification
- ACPI Table installation and removal

Namespace Management

The Namespace component provides ACPI namespace services on top of the AML interpreter. It builds and manages the internal ACPI namespace. Services include:

- Namespace Initialization from either the BIOS or a file
- Device Enumeration
- Namespace Access
- Access to ACPI data and tables

Resource Management

The Resource component provides resource query and configuration services on top of the Namespace manager and AML interpreter. Services include:

- Getting and Setting Current Resources
- Getting Possible Resources
- Getting IRQ Routing Tables
- Getting Power Dependencies

ACPI Hardware Management

The hardware manager controls access to the ACPI registers, timers, and other ACPI-related hardware. Services include:

- ACPI Status register and Enable register access
- ACPI Register access (generic read and write)
- Power Management Timer access
- Legacy Mode support
- Global Lock support
- Sleep Transitions support (S-states)
- Processor Power State support (C-states)
- Other hardware integration: Throttling, Processor Performance, etc.

Event Handling

The Event Handling component manages the ACPI System Control Interrupt (SCI). The single SCI multiplexes the ACPI timer, Fixed Events, and General Purpose Events (GPEs). This component also manages dispatch of notification and Address Space/Operation Region events. Services include:

- ACPI mode enable/disable
- ACPI event enable/disable
- Fixed Event Handlers (Installation, removal, and dispatch)
- General Purpose Event (GPE) Handlers (Installation , removal, and dispatch)
- Notify Handlers (Installation, removal, and dispatch)
- Address Space and Operation Region Handlers (Installation, removal, and dispatch)

Architecture of the OS Dependent Component (OSD)

The OS dependent component of the architecture enables the rehosting or retargeting of the OSI component to execute under multiple operating systems. In other words, the OSD component provides the glue that joins the OS-independent component to a particular operating system. The OSD implements interfaces and services using the system calls and utilities that are available from the host OS. Therefore, an OS-dependent component must be written for each target operating system.

The OS dependent component has several roles.

- It acts as the front-end for OS to ACPI requests. It translates OS requests that are received in the native OS format (such as a system call interface, an I/O request/result segment interface, or a device driver interface) into calls to OSI interfaces.
- It exposes a set of OS specific application interfaces. These interfaces translate application requests to ACPI APIs.
- The OSD component implements a standard set of interfaces that perform OS dependent functions (such as memory allocation and hardware access) on behalf of the OSI component. These interfaces are themselves OS-independent because they are constant across all OSD implementations. It is the implementation of these interfaces that are OS-dependent, because they must use the native services and interfaces of the host operating system.

Functional Service Groups

The services provided by the OS dependent component can be categorized into several distinct groups, mostly based upon *when* each of the services are required. There will be boot time functions, device load time functions, run time functions, and asynchronous functions.

The OS-dependent component exposes these services to the software above it via interfaces that can be used by the host operating system, device drivers, and applications. These interfaces are not defined by this document because they are highly dependent on the host OS. For example, if the OS-dependent and OS-independent components are bundled together to form an ACPI device driver, the interfaces to the driver may be in the form of IOCTL requests or some other form of I/O request block. On the other hand, if the ACPI subsystem is integrated into the host operating system as a standard OS subsystem, the interfaces to the OS-dependent component may take the form of a more conventional system call interface, or even simply a local procedure call interface.

Although it is the OS-dependent component that exposes these services to the rest of the operating system, it is very important to note that the OS-dependent component makes use of the

services of the lower-level OS-independent component to implement its services. It is the intent of the component architecture that the OSI is a service that is private to the OSD—that is, that only the OSD makes calls to the OSI.

OS Bootload-time Services

Boot services are those functions that must be executed very early in the OS load process, before most of the rest of the OS initializes. These services include the ACPI subsystem initialization, ACPI hardware initialization, and execution of the _INI control methods for various devices within the ACPI namespace.

Device Driver Load-time Services

For the devices that appear in the ACPI namespace, the operating system must have a mechanism to detect them and load device drivers for them. The Device driver load services provide this mechanism. The ACPI subsystem provides services to assist with device and bus enumeration, resource detection, and setting device resources.

OS Run-time Services

The runtime services include most if not all of the external interfaces to the ACPI subsystem. These services also include event logging and power management functions.

Asynchronous Services

The asynchronous functions include interrupt servicing (System Control Interrupt), Event handling and dispatch (Fixed events, General Purpose Events, Notification events, and Operation Region access events), and error handling.

Required Functionality

There are three basic functions of the OS-dependent component:

1. Manage the initialization of the entire ACPI subsystem, including both the OS-dependent and OS-independent components.
2. Translate requests for ACPI services from the host operating system (and its applications) into calls to the OSI component. This is not necessarily a one-to-one mapping. Very often, a single operating system request may be translated into many calls into the OS-independent component.
3. Implement an interface layer that the OSI component uses to obtain operating system services. These standard interfaces (defined in this EPS as the **Osd*** interfaces) include functions such as memory management and thread scheduling, and must be implemented using the available services of the host operating system.

This section discusses the services and interfaces that the OS dependent component is required to provide. Only the external definition of these interfaces is clearly defined by this document. The actual implementation of the services and interfaces is OS dependent and may be very different for different operating systems.

Requests from the Operating System to the ACPI Subsystem

OS to ACPI requests are by their nature very dependent upon the structure of the operating system. For example, the data format the OS requires to maintain resources will vary greatly from OS to OS. One of the roles of the OS-dependent component is to translate native operating system ACPI requests into calls to the OS-independent component. For example, the OS-dependent component must translate the ACPI resource structure to the native OS resource structure.

The exact ACPI services required (and the requests made to those services) will vary from OS to OS. However, it can be expected that most OS requests will fit into the broad categories of the functional service groups described earlier: boot time functions, device load time functions, and runtime functions.

The flow of OS to ACPI requests is shown in the diagram below.

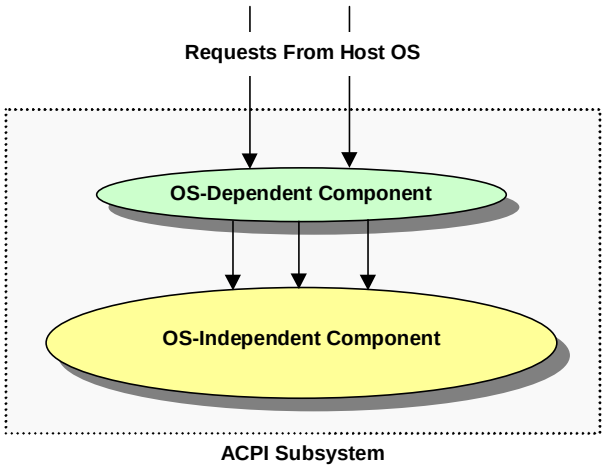

Figure 5. Operating System to ACPI Subsystem Request Flow

Requests from Applications to the ACPI Subsystem

Application level interfaces should be provided in the OS-dependent component to enable the creation of user interfaces for configuration and management of the ACPI system by either the OS vendor or third party software vendors.

The application interfaces must include sufficient functionality that an application will be able to present to the user a clear picture of the ACPI namespace including the interdependencies for enumeration, power, and data.

The type and style of these application interfaces is completely dependent on the architecture of the host operating system and where the ACPI subsystem fits into that architecture. The interfaces may be device driver style interfaces, or system calls into an operating system layer.

Requests from the ACPI Subsystem to the Operating System

ACPI to OS requests are requests for OS services made by the ACPI subsystem. These requests must be serviced (and therefore implemented) in a manner that is appropriate to the host

operating system. These requests include calls for OS dependent functions such as I/O, resource allocation, error logging, and user interaction. The ACPI Component Architecture defines interfaces to the OS-dependent component for this purpose. These interfaces are constant (i.e., they are *OS-independent*), but they must be implemented uniquely for each target OS.

The flow of ACPI to OS requests is shown in the diagram below.

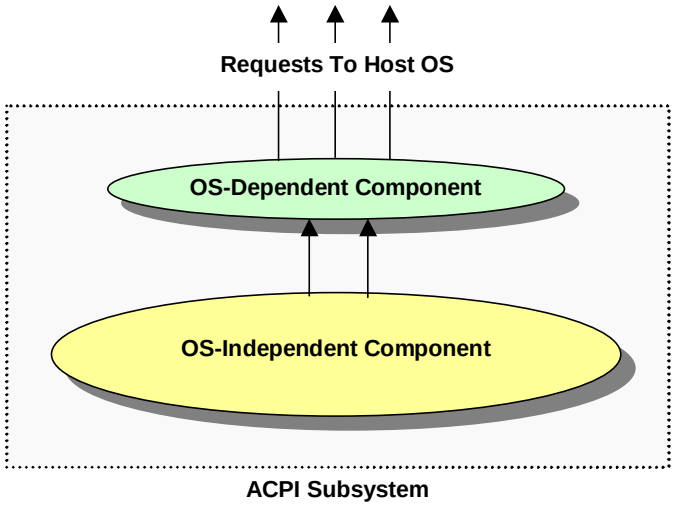

Figure 6. ACPI Subsystem to Operating System Request Flow

Using the ACPI Subsystem

This section contains information about concepts, data types, and data structures that are common to both the OSI and OSD components of the ACPI Subsystem.

Namespace Fundamentals

The *ACPI Namespace* is a large data structure that is constructed and maintained by the OSI component. Constructed primarily from the AML defined within an ACPI Differentiated System Description Table (DSDT), the namespace contains a hierarchy of named ACPI objects.

Named Objects

Each object in the namespace has a fixed 4-character name (32-bits) associated with it. The *root object* is referenced by the backslash as the first character in a pathname. Pathnames are constructed by concatenating multiple 4-character object names with a period as the name separator.

Scopes

The concept of an object *scope* relates directly to the original source ASL that describes and defines an object. An object's scope is defined as all objects that appear between the pair of open and close brackets immediately after the object. In other words, the scope of an object is the container for all of the *children* of that object.

In some of the ACPI CA interfaces, it is convenient to define a scope parameter that is meant to represent this container. For example, when converting an ACPI name into an object handle, the two parameters required to resolve the name are the *name* itself, and a containing *scope* where the name can be found. When the object that matches the name is found within the scope, a handle to that object can be returned.

Example Scopes, Names, and Objects

In the ASL code below, the scope of the object _GPE contains the objects _L08 and _L0A.

```
Scope (\_GPE)
{
    Method (_L08)
    {
          Notify (\_SB.PCI0.DOCK, 1)
    }

    Method (_L0A)
        {
          Store (0, \_SB.PCI0.ISA.EC0.DCS)
        }
}
```

In this example, there are three ACPI namespace *objects*, about which we can observe the following:

- The *names* of the three objects are _GPE, _L08, and _L0A.
- The *child objects* of parent object _GPE are _L08 and _L0A.
- The *absolute pathname* (or *fully qualified pathname*) of object _L08 is **"_GPE._L08"**.
- The *scope* of object _GPE contains both the _L08 and _L0A objects.
- The *objects* _L08 and _L0A have no *scope* associated with them in the internal namespace since they do not define any child objects.
- The *containing scope* of object _L08 is the scope owned by the object _GPE.
- The *parent* of both objects _L08 and _L0A is object _GPE.
- The *type* of both objects _L08 and _L0A is `ACPI_TYPE_Method`.
- The *next object* after object _L08 is object _L0A. In the example _GPE scope, there are no additional objects after object _L0A.
- Since _GPE is a namespace object at the root level (as indicated by the preceding backslash in the name), its parent is the *root object*, and its containing scope is the *root scope*.

Predefined Objects

During initialization of the internal namespace within OSI component, there are several predefined objects that are always created and installed in the namespace, regardless of whether they appear in any of the loaded ACPI tables. These objects and their associated types are shown below.

```
"_GPE",    ACPI_TYPE_Any            // General Purpose Event block
"_PR_",    ACPI_TYPE_Any            // Processor block
"_SB_",    ACPI_TYPE_Any            // System Bus block
"_SI_",    ACPI_TYPE_Any            // System Indicators block
"_TZ_",    ACPI_TYPE_Any            // Thermal Zone block
"_REV",    ACPI_TYPE_Number         // Revision
"_OS_",    ACPI_TYPE_String         // OS Name
"_GL_",    ACPI_TYPE_Mutex          // Global Lock
```

Logical Namespace Layout

In the diagram below, the preceding two sections are combined to show the logical namespace after the predefined objects and the _GPE scope has been entered.

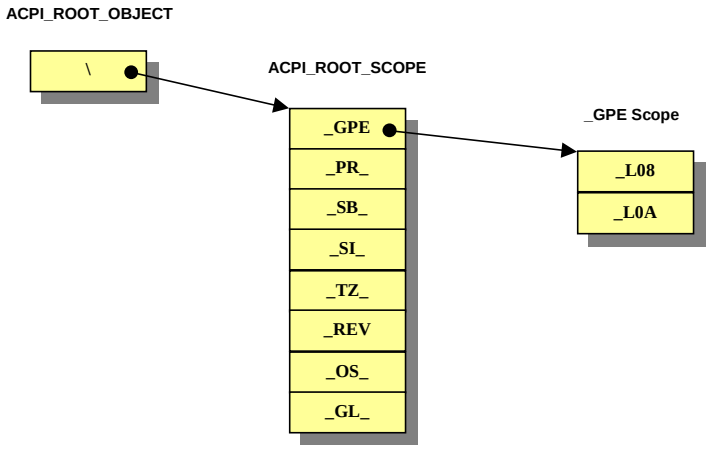

Figure 7. Internal Namespace Structure

Execution Model

Initialization

The initialization of the OS-independent component must be driven entirely by the OS-dependent component. Since it may be appropriate (depending on the requirements of the host OS) to initialize different parts of the OSI at different times, the OSI initialization is a multi-step process that must be coordinated by the OSD. The four main steps are outlined below.

1. Perform a global initialization of the OSI: This initializes the global data and other items within the OSI.
2. Load the ACPI tables. The FACS, DSDT, etc. must be copied (or mapped) into the OSI before the internal namespace can be constructed. The tables may be loaded from the firmware, loaded from a input buffer, or some combination of both.
3. Build the internal namespace: This causes the OSI to parse the DSDT and build an internal namespace from the objects found therein.
4. Enable ACPI mode of the machine. Before ACPI events can occur, the machine must be put into ACPI mode. The OSI installs an interrupt handler for the System Control Interrupts (SCIs) and transitions the hardware from legacy mode to ACPI mode.

Memory Allocation

All interfaces to the OS-independent portion of the ACPI subsystem require the caller (usually the OS-dependent portion) to allocate any required memory. This allows maximum flexibility for the caller since only the caller knows what is the appropriate memory pool to allocate from, whether to statically or dynamically allocate the memory, etc.

It is often the case that the required buffer size is not known by even the ACPI subsystem until after the evaluation of an object or the execution of a control method has been completed. Therefore, the "get size" model of a separate interface to obtain the required buffer size is insufficient. Instead, a model that allows the caller to pre-post a buffer of a large enough size has been chosen. This model is described below.

For ACPI interfaces that use the ACPI_BUFFER data type as an output parameter, the following protocol can be used to determine the exact buffer size required:

1. Set the buffer length field of the ACPI_BUFFER structure to zero, or to the size of a local buffer that is thought to be large enough for the data.
2. Call the Acpi interface.
3. If the return exception code is AE_BUFFER_OVERFLOW, the buffer length field has been set by the interface to the buffer length that is actually required.
4. Allocate a buffer of this length and initialize the length and buffer pointer field of the ACPI_BUFFER structure.
5. Call the Acpi interface again with this valid buffer of the required length.

Alternately, if the caller has some idea of the buffer size required, a buffer can be posted in the original call. If this call fails, only then is a larger buffer allocated. See section 0 ACPI_BUFFER–Input and Output Memory Buffers for additional discussion on using the ACPI_BUFFER data type.

Parameter Validation

Only limited parameter validation is performed on all input parameters passed to the OS-independent component. All calls to the OS-independent code should come from the OS-dependent portion, not directly from user or application code. Therefore, the OS-dependent code is a trusted portion of the kernel code, and should perform all limit and range checks on buffer pointers, strings, and other input parameters before passing them down to the OS-independent code.

The limited parameter validation consists of sanity checking input parameters for non-zero values and nothing more. Any additional parameter validation (such as buffer length validation) must occur in the OSD component.

Exception Handling

All exceptions that occur during the processing of a request to the OS-independent component are translated into the appropriate ACPI_STATUS return code and bubbled up to the original caller.

All exception handling is performed inline by the caller to the OSI interfaces. There are no exception handlers associated with either the **Acpi*** or **Osd*** calls.

Multitasking and Reentrancy

NOTE: *The current implementation of the OSI is single threaded only*.

All components of the ACPI subsystem are intended to be fully reentrant and support multiple threads of execution. To achieve this, there are several mutual exclusion OSD interfaces that must

be properly implemented with the native host OS primitives to ensure that mutual exclusion and synchronization can be performed correctly. Although dependent on the correct implementation of these interfaces, the OS-independent component is otherwise fully reentrant and supports multiple threads throughout the component, with the exception of the AML interpreter, as explained below.

Because of the constraints of the ACPI specification, there is a major limitation on the concurrency that can be achieved within the AML interpreter portion of the subsystem. The specification states that at most one control method can be actually executing AML code at any given time. If a control method blocks (an event that can occur only under a few limited conditions), another method may begin execution. However, it can be said that the specification precludes the concurrent execution of control methods. Therefore, the AML interpreter itself is essentially a single-threaded component of the ACPI subsystem. Serialization of both internal and external requests for execution of control methods is performed and managed by the front-end of the interpreter.

Event Handling

The term *Event Handling* is used somewhat loosely to describe the class of asynchronous events that can occur during the execution of the ACPI subsystem. These events include:

- System Control Interrupts (SCIs) that are generated by both the ACPI Fixed and General Purpose Events.
- Notify events that are generated via the execution of the ASL *Notify* keyword in a control method.
- Events that are caused by accesses to an address space or operation region during the execution of a control method.

Each of these events and the support for them in the ACPI subsystem are described in more detail below.

Fixed Events

Incoming Fixed Events can be handled by the default ACPI subsystem event handlers, or individual handlers can be installed for each event. Only device drivers or system services should install such handlers.

General Purpose Events

Incoming General Purpose Events (GPEs) are usually handled by executing a control method that is associated with a particular GPE. According to the ACPI specification, each GPE level may have a method associated with it whose name is of the form _Txx, where xx is the GPE level in hexadecimal (See the ACPI specification for complete details.) This control method is not executed in the context of the SCI interrupt handler, but is instead queued for later execution by the host operating system.

In addition to this mechanism, individual handlers for GPE levels may be installed. It is not required that a handler be installed for a GPE level, and in fact, currently the only device that requires a dedicated GPE handler is the ACPI Embedded Controller. A device driver for the Embedded Controller would install a handler for the GPE that is dedicated to the EC.

If a GPE handler is installed for a given GPE, the handler is invoked first, then the associated control method (if any) is queued for execution.

Notify Events

An ACPI Notify Event occurs as a result of the execution of a *Notify* opcode during the execution of a control method. A notify event occurs on a particular ACPI object, and this object must be a device or thermal zone. If a handler is installed for notifications on a particular device, this handler is invoked during the execution of the *Notify* opcode, in the context of the thread that is executing the control method.

Notify handlers should be installed by device drivers and other system services that know about the particular device or thermal zone on which notifications will be received.

Address Spaces and Operation Regions

ASL source code and the corresponding AML code use the *Address Space* mechanism to access data that is out of the direct scope of the ASL. For example, Address Spaces are used to access the CMOS RAM and the ACPI Embedded Controller. There are several pre-defined Address Spaces that may be accessed and user-defined Address Spaces are allowed.

The Operating System software (which includes the AML Interpreter) allows access to the various address spaces via the ASL *Operation Region* (OpRegion) construct. An OpRegion is a named window into an address space. During the creation of an OpRegion, the ASL programmer defines both the the boundaries (window size) and the address space to be accessed by the OprRegion. Specific addresses within the access window can then be defined as named *fields* to simplify their use.

The AML Interpreter is responsible for translating ASL/AML references to named *Fields* into accesses to the appropriate Address Space. The interpreter resolves locations within an address space using the field's address within an OpRegion and then the OpRegion's offset within the address space. The resolved address, address access width, and function (read or write) are then passed to the address space handler who is responsible for performing the actual physical access of the address space.

Installation of Address Space Handlers

At runtime, the ASL/AML code cannot access an address space until a handler has been installed for that address space. An ACPI CA user can either install the default address space handlers or install user defined address space handlers using the *AcpiInstallAddressSpaceHandler* interface.

Each Address Space is "owned" by a particular device such that all references to that address space within the *scope* of the device will be handled by that device's address space handler. This mechanism allows multiple address space/operation region handlers to be installed for the same *type* of address space, each mutually exclusive by virtue of being governed by the ACPI address space scoping rules. For example, picture a platform with two SMBus devices, one an embedded controller based SMBus; the other a PCI based SMBus. Each SMBus must expose its own address space to the ASL without disrupting the function of the other. In this case, there may be two device drivers and two distinctly different address space handlers, one for each type of SMBus. This mechanism can be employed in a similar manner for the other predefined address spaces. For example, the PCI Configuration space for each PCI bus is unique to that bus. Creation of a region within the scope of a PCI bus must refer only to that bus.

Address space handlers must be installed on a named object in the ACPI namespace or on the special object ACPI_ROOT_OBJECT. This is required to maintain the scoping rules of address

space access. Address handlers are installed for the namespace object representing the device that "owns" that address space. Per ASL rules, regions that access that address space must be declared in the ASL within the scope of that namespace object.

It is the responsibility of the ACPI CA user to enumerate the namespace and install address handlers as needed.

ACPI-Defined Address Spaces

The ACPI 1.0b specification defines address spaces for:

- System Memory
- System I/O
- PCI Configuration Space
- System Management Bus (SMBus)
- Embedded Controller

The ACPI CA subsystem implements default address space handlers for the following ACPI defined address spaces:

- System Memory
- System I/O
- PCI Configuration Space

Default address space handlers can be installed by supplying the special value ACPI_DEFAULT_HANDLER as the handler address when calling the *AcpiInstallAddressSpaceHandler* interface.

The other predefined address spaces (Embedded Controller and SMBus) have no default handlers and will not be accessible without OS provided handlers. This is typically the role of the Embedded Controller and SMBus device drivers.

Environmental Support Requirements

This section describes the environmental requirements of the ACPI subsystem. This includes the external functions and header files that the subsystem uses, as well as the resources that are consumed from the host operating system.

Required C Library Functions

In order to make the OS-independent component as portable and truly OS-independent as possible, there is only extremely limited use of standard C library functions within the OSI component itself. The calls are limited to those that can generate code in-line or link to small, independent code modules. Below is a comprehensive list of the C library functions that are called by the OSI code.

```
sprintf
memcpy
memset
strcat
strcmp
strcpy
strlen
```

```
strncmp
strncat
strncpy
strtoul
va_list
va_start
va_end
```

If a kernel-level C library is unavailable for any reason, these functions could be implemented in the OSD with little difficulty.

System Include Files

The following include files (header files) are useful for users of both the **Acpi*** and **Osd*** interfaces:

- acpiexcep.h The ACPI_STATUS exception codes
- acpiosd.h The prototypes for all of the **Osd*** interfaces
- acpisubsys.h The prototypes for all of the **Acpi*** interfaces
- acpitypes.h Common data types used across all interfaces

Customization to the Target Environment

The use of header files that are external to the ACPI subsystem is confined to a single header file named *environment.h*. These external include files consist of several of the standard C library headers:

- stdio.h
- stdlib.h
- stdarg.h
- string.h

When generating the OSI component from source, the environment.h header may be modified if the filenames above are not appropriate for generation on the target system. For example, some environments use a different set of header files for the kernel-level C library versus the user-level C library. Use of C library routines within the OSI component has been kept to a minimum in order to enhance portability and to ensure that the OSI will run as a kernel-level component in most operating systems.

ACPI Subsystem Interface Parameters

ACPI Names and Pathnames

As defined in the ACPI Specification, all ACPI object *names* (the names for all ACPI objects such as control methods, regions, buffers, packages, etc.) are exactly four ASCII characters long. The ASL compiler automatically pads names out to four characters if an input name in the ASL source is shorter. (The padding character is the underscore.) Since all ACPI names are always of a fixed length, they can be stored in a single 32-bit integer to simplify their use.

Pathnames are null-terminated ASCII strings that reference named objects in the ACPI namespace. A pathname can be composed of multiple 4-character ACPI names separated by a

period. In addition, two special characters are defined. The backslash appearing at the start of a pathname indicates to begin the search at the root of the namespace. A carat in the pathname directs the search to traverse upwards in the namespace by one level. The ACPI namespace is defined in the ACPI specification. The ACPI CA subsystem honors all of the naming conventions that are defined in the ACPI specification.

Frequently in this document, pathnames are referred to as "fully qualified pathname" or "absolute pathname" or "relative pathname". A pathname is fully qualified if it begins with the backslash character ('\') since it defines the complete path to an object from the root of the namespace. All other pathnames are relative since they specify a path to an object from somewhere in the namespace besides the root.

The ACPI specification defines special search rules for single segment (4-character) or standalone names. These rules are intended to apply to the execution of AML control methods that reference named ACPI objects. The ACPI CA OSI component implements these rules fully for the execution of control methods. It does not implement the so-called "parent tree" search rules for the external interfaces in order to avoid object reference ambiguities.

Pointers

Many of the interfaces defined here pass pointers as parameters. It is the responsibility of the caller to ensure that all pointers passed to the ACPI CA subsystem are valid and addressable. The interfaces only verify that pointers are non-NULL. If a pointer is any value other than NULL, it will be assumed to be a valid pointer and will be used as such.

Buffers

It is the responsibility of the caller to ensure that all input and output buffers supplied to the OSI component are at least as long as the length specified in the ACPI_BUFFER structure, readable, and writable in the case of output buffers. The OSI does not perform addressability checking on buffer pointers, nor does it perform range validity checking on the buffers themselves. In the ACPI Component Architecture, it is the responsibility of the OS-dependent component to validate all buffers passed to it by application code, create aliases if necessary to address buffers, and ensure that all buffers that it creates locally are valid. In other words, the OS-independent component *trusts* the OS-dependent component to validate all buffers.

ACPI Subsystem Data Types

ACPI_STRING–ASCII String

The ACPI_STRING data type is a conventional "char *" null-terminated ASCII string. It is used whenever a full ACPI pathname or other variable-length string is required. This data type was defined to strongly differentiate it from the ACPI_NAME data type.

ACPI_BUFFER–Input and Output Memory Buffers

Many of the ACPI CAinterfaces require buffers to be passed into them and/or buffers to be returned from them. A common structure is used for all input and output buffers across the interfaces. The buffer structure below is used for both input and output buffers. The OSI component never allocates memory for return buffers by design—this allows the caller complete flexibility in where and how memory is allocated. This is especially important in kernel level code.

```
typedef struct
{
    UINT32          Length;         // Length in bytes of the buffer;
    void            *Pointer;       // pointer to buffer

} ACPI_BUFFER;
```

Input Buffer

An input buffer is defined to be a buffer that is filled with data by the user (caller) before it is passed in as a parameter to one of the ACPI interfaces. When passing an input buffer to one of the OSI interfaces, the user creates an ACPI_BUFFER structure and initializes it with a pointer to the actual buffer and the length of the valid data in the buffer. Since the memory for the actual ACPI_BUFFER structure is small, it will typically be dynamically allocated on the CPU stack. For example, a user may allocate a 4K buffer for common storage. The buffer may be reused many times with data of various lengths. Each time the number of bytes of *significant* data contained in the buffer is entered in the Length field of the ACPI_BUFFER structure before an OSI interface is called.

Output Buffer

An output buffer is defined to be a buffer that is filled with data by an ACPI interface before it is returned to the caller. When the ACPI_BUFFER structure is used as an output buffer the caller must always initialize the structure by placing a value in the Length field that indicates the maximum size of the buffer that is pointed to by the *Pointer* field. The length is used by the ACPI interface to ensure that there is sufficient user provided space for the return value.

If the buffer that is passed in by the caller is too small, the ACPI interfaces that require output buffers will indicate the failure by returning the error code AE_BUFFER_OVERFLOW. The interfaces will never attempt to put more data into the caller's buffer than is specified by the Length field of the ACPI_BUFFER structure. The caller may recover from this failure by examining the Length field of the ACPI_BUFFER structure. The interface will place the *required* length in this field in the event that the buffer was too small.

During normal operation, the ACPI interface will copy data into the buffer. It will indicate to the caller the length of data in the buffer by setting the Length field of the ACPI_BUFFER to the actual number of bytes placed in the buffer.

Therefore, the Length field is both an input and output parameter. On input, it indicates the size of the buffer. On output, it either indicates the actual amount of data that was placed in the buffer (if the buffer was large enough), or it indicates the buffer size that is required (if the buffer was too small) and the exception is set to AE_BUFFER_OVERFLOW.

ACPI_HANDLE–Object Handle

References to ACPI objects managed by the OSI component are made via the ACPI_HANDLE data type. A handle to an object is obtained by creating an attachment to the object via the *AcpiPathnameToHandle* or *AcpiNameToHandle* primitives. The concept is similar to opening a file and receiving a connection–after the pathname has been resolved to an object handle, no additional internal searching is performed whenever additional operations are needed on the object.

References to object scopes also use the ACPI_HANDLE type. This allows objects and scopes to be used interchangeably as parameters to Acpi interfaces. In fact, a scope handle is actually a handle to the *first object* within the scope.

Predefined Handles

One predefined handle is provided in order to simplify access to the ACPI namespace:

- **ACPI_ROOT_OBJECT**: A handle to the root object of the namespace. All objects contained within the root scope are children of the root object.

ACPI_OBJECT_TYPE–Object Type Codes

Each ACPI object that is managed by the ACPI subsystem has a **type** associated with it. The valid ACPI object types are defined as follows:

```
ACPI_TYPE_Any
ACPI_TYPE_Number
ACPI_TYPE_String
ACPI_TYPE_Buffer
ACPI_TYPE_Package
ACPI_TYPE_FieldUnit
ACPI_TYPE_Device
ACPI_TYPE_Event
ACPI_TYPE_Method
ACPI_TYPE_Mutex
ACPI_TYPE_Region
ACPI_TYPE_Power
ACPI_TYPE_Processor
ACPI_TYPE_Thermal
ACPI_TYPE_Alias
ACPI_OBJECT_TYPE_MAX
```

ACPI_OBJECT–Method Parameters and Return Objects

The general purpose ACPI_OBJECT is used to pass parameters to control methods, and to receive results from the evaluation of namespace objects. The point of this data structure is to provide a common object that can be used to contain multiple ACPI data types.

When passing parameters to a control method, each parameter is contained in an ACPI_OBJECT. All of the parameters are then grouped together in an ACPI_OBJECT_LIST.

When receiving a result from the evaluation of a namespace object, an ACPI_OBJECT is returned in an ACPI_BUFFER structure. This allows variable length objects such as ACPI Packages to be returned in the buffer. The first item in the buffer is always the base ACPI_OBJECT.

```
typedef union AcpiObj
{
    UINT32                  Type;       // An ACPI_OBJECT_TYPE

    struct      /* ACPI_TYPE_Number */
    {
        UINT32              Type;
        UINT32              Value;      // The actual number
    } Number;

    struct      /* ACPI_TYPE_String */
    {
        UINT32              Type;
        UINT32              Length;     // Length of string, without null
        UINT8               *Pointer;   // points to the string value
    } String;

    struct      /* ACPI_TYPE_Buffer */
    {
        UINT32              Type;
        UINT32              Length;     // # of bytes in buffer
        UINT8               *Pointer;   // points to the buffer
    } Buffer;

    struct      /* ACPI_TYPE_Package */
    {
        UINT32              Type;
        UINT32              Count;      // # of elements in package
        union AcpiObj       *Elements;  // Pointer to an array of Objects
    } Package;

} ACPI_OBJECT, *PACPI_OBJECT;
```

ACPI_OBJECT_LIST–List of Objects

This object is used to pass parameters to control methods via the *AcpiEvaluateMethod* interface. The *Count* is the number of ACPI objects pointed to by the *Pointer* field. In other words, the *Pointer* field must point to an array that contains *Count* ACPI objects.

```
typedef struct AcpiObjList
{
    UINT32                 Count;
    ACPI_OBJECT            *Pointer;

} ACPI_OBJECT_LIST, *PACPI_OBJECT_LIST;
```

ACPI_EVENT_TYPE–Fixed Event Type Codes

The ACPI fixed events are defined in the ACPI specification. The event codes below are used to install handlers for the individual events.

```
EVENT_PMTIMER                // Power Management Timer rollover
EVENT_NOT_USED               // Reserved
EVENT_GLOBAL                 // Global Lock released
EVENT_POWER_BUTTON           // Power Button (pressed)
EVENT_SLEEP_BUTTON           // Sleep Button (pressed)
EVENT_RTC                    // Real Time Clock alarm
EVENT_GENERAL                // General Event
ACPI_EVENT_MAX
```

ACPI_TABLE_TYPE–ACPI Table Type Codes

The following ACPI tables are supported by the ACPI CA subsystem. The table type codes below are used to load, unload, or get a copy of the individual tables.

```
TABLE_RSDP              // Root System Description Pointer
TABLE_APIC              // Multiple APIC Description Table
TABLE_DSDT              // Differentiated System Description Table
TABLE_FACP              // Fixed ACPI Description Table
TABLE_FACS              // Firmware ACPI Control Structure
TABLE_PSDT              // Persistent System Description Table
TABLE_RSDT              // Root System Description Table
TABLE_SSDT              // Secondary System Description Table
TABLE_SBDT              // Smart Battery Description Table
ACPI_TABLE_MAX
```

ACPI_TABLE_HEADER–Common ACPI Table Header

```
typedef struct  /* ACPI common table header */
{
    char                    Signature [4];      /* Identifies type of table
*/
    UINT32                  Length;             /* Length of table, in
bytes,
                                                 * including header */
    UINT8                   Revision;           /* Specification minor
version # */
    UINT8                   Checksum;           /* To make sum of entire
table == 0 */
    char                    OemId [6];          /* OEM identification */
    char                    OemTableId [8];     /* OEM table identification
*/
    UINT32                  OemRevision;        /* OEM revision number */
    char                    AslCompilerId [4];  /* ASL compiler vendor ID */
    UINT32                  AslCompilerRevision;/* ASL compiler revision
number */

} ACPI_TABLE_HEADER;
```

ACPI_STATUS–Interface Exception Return Codes

Each of the external ACPI interfaces return an exception code of type ACPI_STATUS as the function return value, as shown in the example below:

```
ACPI_STATUS                 Status;

Status = AcpiLoadNamespace ();
if (Status != AE_OK)
{
   // Exception handling code here
}
```

ACPI Resource Data Types

The ACPI resource data types are used by the ACPI CA resource interfaces.

PCI IRQ Routing Tables

The PCI IRQ routing tables are retrieved by using the AcpiGetIrqRoutingTable API. This API returns the routing table in the ACPI_BUFFER provided by the caller. Upon return, the *Length* field of the ACPI_BUFFER will indicate the amount of the buffer used to store the PCI IRQ routing tables. If AE_BUFFER_OVERFLOW is the returned status, the *Length* indicates the size of the buffer needed to contain the routing table.

The ACPI_BUFFER *Pointer* points to a buffer of at least *Length* size. The buffer contains a series of PCI_ROUTING_TABLE entries, each of which contains both a *Length* member and a *Data* member. The *Data* member is a PRT_ENTRY. The *Length* member specifies the length of the PRT_ENTRY and can be used to walk the PCI_ROUTING_TABLE entries. By incrementing a buffer walking pointer by *Length* bytes, the pointer will reference each succeeding table element. The final PCI_ROUTING_TABLE entry will contain no Data and have a *Length* member of zero.

Each PRT_ENTRY contains the Address, Pin, Source and Source Index information as described in Chapter 6 of the ACPI Specification. While all structure members are UINT32 types, the valid portion of both the *Pin* and *SourceIndex* members are only UINT8 wide. Although the *Source* member is defined as UINT8 Source[1], it can be de-referenced as a null-terminated string.

```
typedef struct          /* a single IRQ table entry */
{
    UINT32    Address;     /* PCI Address of device */
    UINT32    Pin;         /* PCI Pin (0=INTA, 1=INTB, 2=INTC, 3=INTD # */
    UINT32    SourceIndex; /* index of resource of allocating device */
    UINT8     Source[1];   /* Null terminated Name of device that allocates
*/
                           /* this interrupt  */
} PRT_ENTRY;

typedef struct             /* An IRQ table entry packed in the return buffer
*/
{
    UINT32    Length;      /* Length of this PRT_ENTRY */
    PRT_ENTRY Data;        /* The PRT Entry data */
} PCI_ROUTING_TABLE;
```

Device Resources

Device resources are returned by executing the _CRS and _PRS control methods by using the AcpiGetCurrentResources and AcpiGetPossibleResources APIs. Device resources are needed to properly execute the _SRS control method using the AcpiSetCurrentResources API.

These API's require an ACPI_BUFFER parameter. If the *Length* member of the ACPI_BUFFER is set to zero then the AcpiGet* API's will return an ACPI_STATUS of AE_BUFFER_OVERFLOW with *Length* set to the size buffer needed to contain the resource descriptors. If the *Length* member is non-zero and *Pointer* in non-NULL, it is assumed that *Pointer* points to a memory buffer of at least *Length* size. Upon return, the *Length* member will indicate the amount of the buffer used to store the resource descriptors.

RESOURCE_TYPE – Resource data types

The following resource types are supported by the ACPI CA subsystem. The resource types that follow are use in the resource definitions used in the resource handling APIs: AcpiGetCurrentResources, AcpiGetPossibleResources and AcpiSetCurrentResources.

1. Irq
2. Dma
3. StartDependentFunctions
4. EndDependentFunctions
5. Io
6. FixedIo
7. VendorSpecific
8. EndTag
9. Memory24
10. Memory32
11. FixedMemory32
12. Address16
13. Address32
14. ExtendedIrq

```
typedef union  /* union of all resources */
{
    IRQ_RESOURCE                        Irq;
    DMA_RESOURCE                        Dma;
    START_DEPENDENT_FUNCTIONS_RESOURCE  StartDependentFunctions;
    IO_RESOURCE                         Io;
    FIXED_IO_RESOURCE                   FixedIo;
    VENDOR_RESOURCE                     VendorSpecific;
    MEMORY24_RESOURCE                   Memory24;
    MEMORY32_RESOURCE                   Memory32;
    FIXED_MEMORY32_RESOURCE             FixedMemory32;
    ADDRESS16_RESOURCE                  Address16;
    ADDRESS32_RESOURCE                  Address32;
    EXTENDED_IRQ_RESOURCE               ExtendedIrq;
} RESOURCE_DATA;


typedef struct _resource_tag
{
    RESOURCE_TYPE           Id;
    UINT32                  Length;
    RESOURCE_DATA           Data;
} RESOURCE;
```

The ACPI_BUFFER *Pointer* points to a buffer of at least *Length* size. The buffer is filled with a series of RESOURCE entries, each of which begins with an *Id* that indicates the type of resource descriptor, a *Length* member and a *Data* member that is a RESOURCE_DATA union. The RESOURCE_DATA union can be any of fourteen different types of resource descriptors. The *Length* member will allow the caller to walk the RESOURCE entries. By incrementing a buffer walking pointer by *Length* bytes, the pointer will reference each succeeding table element. The final element in the list of RESOURCE entries will have an *Id* of EndTag. An EndTag entry contains no additional data.

When walking the RESOURCE entries, the *Id* member determines how to interpret the structure. For example, if the *Id* member evaluates to StartDependentFunctions, then the *Data* member is two 32-bit values, a CompatibilityPriority value and a PerformanceRobustness value. These values are interpreted using the constant definitions, which are found in acpitypes.h, GOOD_CONFIGURATION, ACCEPTABLE_CONFIGURATION or

SUB_OPTIMAL_CONFIGURATION. The interpretation of these constant definitions is discussed in the Start Dependent Functions section of the ACPI specification.

For another, more complex example, let us consider a RESOURCE entry with an *Id* member that evaluates to Address32, then the *Data* member is a ADDRESS32_RESOURCE structure. The ADDRESS32_RESOURCE structure contains fourteen members that map to the data discussed in the DWORD Address Space Descriptor section of the ACPI specification. The *Data.Address32.ResourceType* member is interpreted using the constant definitions MEMORY_RANGE, IO_RANGE or BUS_NUMBER_RANGE. This value also effects the interpretation of the *Data.Address32. Attribute* structure because it contains type specific information.

The General Flags discussed in the ACPI specification are interpreted and given separate members within the ADDRESS32_RESOURCE structure. Each of the bits in the General Flags that describe whether the maximum and minimum addresses is fixed or not, whether the address is subtractively or positively decoded and whether the resource simply consumes or both produces and consumes a resource are represented by the members *MaxAddressFixed*, *MinAddressFixed*, *Decode* and *ProducerConsumer* respectively.

The *Attribute* member is interpreted based upon the *ResourceType* member. For example, if the *ResourceType* is MEMORY_RANGE, then the *Attribute* member contains two 16-bit values, a *Data.Address32.Attribute.Memory.CacheAttribute* value and a *ReadWriteAttribute* value.

The *Data.Address32.Granularity, MinAddressRange, MaxAddressRange, AddressTranslationOffset* and *AddressLength* members are simply interpreted as UINT32 numbers.

The optional *Data.Address32.ResourceSourceIndex* is valid only if the *ResourceSourceStringLength* is non-zero. Although the *ResourceSource* member is defined as UINT8 ResourceSource[1], it can be de-referenced as a null-terminated string whose length is *ResourceSourceStringLength*.

Exception Codes

A common and consistent set of return codes is used throughout the ACPI subsystem. For example, all of the public ACPI interfaces return the exception AE_BAD_PARAMETER when an invalid parameter is detected.

The exception codes are contained in the public acpiexcep.h file.

The entire list of available exception codes is given below, along with a generic description of each code. See the description of each public primitive for a list of possible exceptions, along with specific reason(s) for each exception.

```
AE_OK                   // No error
AE_PENDING              // Internal use only
AE_AML_ERROR            // AML parsing error
AE_RETURN_VALUE         // Internal use only
AE_ERROR                // Unspecified error
AE_NO_ACPI_TABLES       // ACPI tables could not be found
AE_NO_NAMESPACE         // A namespace has not been loaded
AE_NO_MEMORY            // Insufficient dynamic memory
AE_BAD_HEADER           // Invalid field in an ACPI table header
```

```
AE_BAD_CHECKSUM          // An ACPI table checksum is not correct
AE_BAD_PARAMETER         // A parameter is out of range or invalid
AE_BAD_CHARACTER         // An invalid character was found in a name
AE_NOT_FOUND             // The name was not found in the namespace
AE_NOT_EXIST             // A required entity does not exist
AE_EXIST                 // An entity already exists
AE_TYPE                  // The object type is incorrect
AE_NULL_ENTRY            // The requested object does not exist
AE_BUFFER_OVERFLOW       // The buffer provided is too small
AE_STACK_OVERFLOW        // An internal stack overflowed
AE_STACK_UNDERFLOW       // An internal stack underflowed
AE_NOT_IMPLEMENTED       // The feature is not implemented
AE_VERSION_MISMATCH      // An incompatible version was detected
AE_BAD_SIGNATURE         // An ACPI table has an unrecognized signature
AE_SUPPORT               // The feature is not supported
AE_SHARE                 // There was a sharing violation
AE_LIMIT                 // A predefined limit was exceeded
AE_TIME                  // A time limit or timeout expired
AE_TERMINATE             // Request to terminate the current operation
AE_DEPTH                 // Maximum search depth has been reached
AE_UNKNOWN_STATUS        // An unknown status code was encountered
```

OS-Independent Component - External Interface Definition

This section contains documentation for the specific interfaces exported by the OS-Independent component to provide ACPI services to the OSD. The interfaces are broken into groups based upon their functionality. These groups are closely related to the internal modules of the OSI described earlier in this document. These interfaces are intended for use by the OSD only. The host OS does not call these interfaces directly. All interfaces to the OS-independent component are prefixed by the letters **Acpi**.

• *Global Initialization, Shutdown, and Status*

• AcpiInitialize

Initialize the OS-independent component of the ACPI Subsystem.

ACPI_STATUS
AcpiInitialize (void)

PARAMETERS

None

RETURN

Status	Exception code indicates success or reason for failure.

EXCEPTIONS

AE_OK	The OSI was successfully initialized.
AE_ERROR	The system is not capable of supporting ACPI mode.
AE_NO_MEMORY	Insufficient dynamic memory to complete the ACPI initialization.

Functional Description:

This function initializes the OS-independent part of the ACPI subsystem. It must be called once before any of the other **Acpi*** interfaces are called.

- **AcpiTerminate**

Shutdown the OS-independent component of the ACPI Subsystem.

ACPI_STATUS
AcpiTerminate (void)

PARAMETERS

None

RETURN

Status	Exception code indicates success or reason for failure.

EXCEPTIONS

AE_OK	The OSI component was successfully shutdown.
AE_ERROR	Execution error

Functional Description:

This function performs a shutdown of the OS-independent portion of the ACPI subsystem. The namespace tables are unloaded, and all resources are freed to the host operating system. This function should be called prior to unloading the ACPI subsystem. In more detail, the terminate function performs the following:

- Free all memory associated with the ACPI tables (either allocated or mapped memory).
- Free all internal objects associated with the namespace.
- Free all internal namespace tables.
- Free all OS resources associated with mutual exclusion.

- **AcpiGetSystemInfo**

Get global ACPI-related system information.

ACPI_STATUS
AcpiGetSystemInfo (
ACPI_BUFFER ***OutBuffer)**

PARAMETERS

OutBuffer	A pointer to a location where the system information is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The system information list was successfully returned.
AE_BAD_PARAMETER	At least one of the following is true: • The OutBuffer pointer is `NULL`. • The *Pointer* field of *OutBuffer* is `NULL`.
AE_BUFFER_OVERFLOW	The *Length* field of *OutBuffer* indicates that the buffer is too small to hold the system information. Upon return, the *Length* field contains the minimum required buffer length.

Functional Description:

This function obtains information about the current state of the ACPI system. It will return system information in the *OutBuffer* structure. Upon completion the *Length* field of *OutBuffer* will indicate the number of bytes copied into the *Pointer* field of the *OutBuffer* buffer. This routine will never return a partial resource structure.

If the function fails an appropriate status will be returned and the value of *OutBuffer* is undefined.

The structure that is returned in *OutBuffer* is defined as follows:

```
typedef struct _AcpiSysInfo
{
    UINT32                  Flags;
    UINT32                  TimerResolution;
    UINT32                  Reserved1;
    UINT32                  Reserved2;
    UINT32                  DebugLevel;
    UINT32                  DebugLayer;

} ACPI_SYSTEM_INFO;
```

Where:

Flags	Static information about the system: SYS_MODE_ACPI Acpi mode supported on this system. SYS_MODE_LEGACY Legacy mode supported.
TimerResolution	Resolution of the ACPI Power Management Timer. Either 24 or 32 bits of resolution.

Future versions of this document may add and consolidate other ACPI info such as: Timer resolution, Acpi and legacy mode capabilities, supported sleep states, current mode, etc.

- **AcpiFormatException**

Return the ASCII name of an ACPI exception code.

ACPI_STATUS
AcpiFormatException (

ACPI_STATUS	**Exception**
ACPI_BUFFER	***OutBuffer)**

PARAMETERS

Exception	The ACPI exception code to be translated.
OutBuffer	A pointer to a location where the exception code name is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The exception code name was was successfully returned.
AE_BAD_PARAMETER	At least one of the following is true: • The OutBuffer pointer is NULL. • The *Pointer* field of *OutBuffer* is NULL. • The *Exception* is not valid.
AE_BUFFER_OVERFLOW	The *Length* field of *OutBuffer* indicates that the buffer is too small to hold the exception name. Upon return, the *Length* field contains the minimum required buffer length.

Functional Description:

This function converts an ACPI exception code into a human-readable string. It will return the exception name string in the *OutBuffer* structure. Upon completion the *Length* field of *OutBuffer* will indicate the number of bytes copied into the *Pointer* field of the *OutBuffer* buffer. This routine will never return a partial string.

- ***ACPI Table Manipulation***

- **AcpiLoadFirmwareTables**

Load all available ACPI tables from the firmware.

ACPI_STATUS
AcpiLoadFirmwareTables (
void)

PARAMETERS

None.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The table was successfully loaded and a handle returned.
AE_BAD_CHECKSUM	The computed table checksum does not match the checksum in the table.
AE_BAD_HEADER	The table header is invalid or is not a valid type.
AE_NO_ACPI_TABLES	The ACPI tables (RSDT, DSDT, FACP, etc.) could not be found in physical memory.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.

Functional Description:

This function loads all available ACPI tables that are posted by the firmware. The *Root System Description Pointer* (RSDP) is found in low physical memory (within the first megabyte), and the remaining tables are found via pointers contained in the *Root System Description Table* (RSDT).

If the operation fails an appropriate status will be returned and the value of *OutTableHandle* is undefined.

- **AcpiLoadTable**

Load an ACPI table from a buffer.

ACPI_STATUS
AcpiLoadTable (
ACPI_TABLE_HEADER *Table)

PARAMETERS

Table	A pointer to a buffer containing the entire table to be loaded.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The table was successfully loaded and a handle returned.
AE_BAD_CHECKSUM	The computed table checksum does not match the checksum in the table.
AE_BAD_HEADER	The table header is invalid.
AE_BAD_PARAMETER	At least one of the following is true: • The *Table* pointer is NULL.
AE_BAD_SIGNATURE	The signature field in the table header is not one of the supported table types.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.

Functional Description:

This function is loads a single ACPI table from the caller's buffer. The buffer must contain an entire ACPI Table including a valid header. The header fields are verified, and the call will fail if it is determined that the table is invalid.

The table type (DSDT, FACS, etc.) is determined from the signature in the table header. See the ACPI_TABLE_TYPE data type for the supported table types.

Any previously loaded table of the same table type is automatically unloaded before the new table is installed.

If the call fails an appropriate status will be returned and the value of *OutTableHandle* is undefined.

- **AcpiUnloadTable**

Unload a previously loaded ACPI table.

ACPI_STATUS
AcpiUnloadTable (
ACPI_TABLE_TYPE **Type)**

PARAMETERS

Type	The type of the table to be unloaded. This must be a table loaded by either the *AcpiLoadTable* or the *AcpiLoadFirmware* functions.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The table was successfully unloaded.
AE_BAD_PARAMETER	The *Type* is invalid.
AE_NOT_EXIST	There is no table of this type currently loaded.

Functional Description:

This function unloads a previously loaded table. The table may have been loaded from the firmware or from a call to the *AcpiLoadTable* interface. For table types that allow multiple table (SSDT, PSDT), all tables of the given type are unloaded.

- **AcpiGetTableHeader**

Get the header portion of a loaded ACPI table.

ACPI_STATUS
AcpiGetTableHeader (

ACPI_TABLE_TYPE	**TableType,**
UINT32	**Instance**
ACPI_TABLE_HEADER	***OutTableHeader)**

PARAMETERS

TableType	One of the defined ACPI table types.
Instance	For table types that support multiple tables, the instance of the table to be returned. For table types that support only a single table, this parameter must be set to one.
OutTableHeader	A pointer to a location where the table header is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The table header was successfully located and returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *TableType* is invalid. • The *OutTableHeader* pointer is `NULL`. • The table type only supports single tables, and the *Instance* is not one.
AE_NOT_EXIST	There is no table of this type currently loaded, or the table of the specified *Instance* is not loaded.
AE_TYPE	The table *Type* is not supported (RSDP).

Functional Description:

This function obtains the header of an installed ACPI table. The header contains a length field that can be used to determine the size of the buffer needed to contain the entire table. This function is not valid for the RSDP table since it does not have a standard header and is fixed length.

For table types that support more than one table, the *Instance* parameter is used to specify which table header of the given type should be returned. For table types that only support single tables, the *Instance* parameter must be set to one.

If the operation fails an appropriate status will be returned and the contents of *OutTableHeader* are undefined.

- **AcpiGetTable**

Get a loaded ACPI table.

ACPI_STATUS
AcpiGetTable (

ACPI_TABLE_TYPE	**TableType,**
UINT32	**Instance**
ACPI_BUFFER	***OutBuffer)**

PARAMETERS

TableType	One of the defined ACPI table types.
Instance	For table types that support multiple tables, the instance of the table to be returned. For table types that support only a single table, this parameter must be set to one.
OutBuffer	A pointer to location where the table is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The table was successfully located and returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *TableType* is invalid. • The *OutBuffer* pointer is NULL. • The *Pointer* field of *OutBuffer* is NULL. • The table type only supports single tables, and the *Instance* is not one.
AE_BUFFER_OVERFLOW	The *Length* field of *OutBuffer* indicates that the buffer is too small to hold the table. Upon return, the *Length* field contains the minimum required buffer length.
AE_NOT_EXIST	There is no table of this type currently loaded, or the table of the specified *Instance* is not loaded.

Functional Description:

This function obtains an installed ACPI table. The caller supplies an *OutBuffer* large enough to contain the entire ACPI table. The caller should call the *AcpiGetTableHeader* function first to determine the buffer size needed. Upon completion the *Length* field of *OutBuffer* will indicate the number of bytes copied into the *Pointer* field of the *OutBuffer* buffer. This table will be a complete table including the header.

For table types that support more than one table, the *Instance* parameter is used to specify which table of the given type should be returned. For table types that only support single tables, the *Instance* parameter must be set to one.

If the operation fails an appropriate status will be returned and the contents of *OutBuffer* are undefined.

• *ACPI Namespace Access*

• AcpiLoadNamespace

Load the internal ACPI namespace.

ACPI_STATUS
AcpiLoadNamespace (
void)

PARAMETERS

None.

RETURN

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The ACPI namespace was successfully loaded and initialized.
AE_NO_MEMORY	Insufficient memory to build the internal namespace.

Functional Description:

This function loads the internal ACPI namespace. The namespace is loaded from the DSDT.

• AcpiEvaluateObject

Evaluate an ACPI namespace object and return the result.

ACPI_STATUS
AcpiEvaluateObject (

ACPI_HANDLE	**Object,**
ACPI_STRING	***Pathname,**
ACPI_OBJECT_LIST	***MethodParams,**
ACPI_BUFFER	***ReturnBuffer)**

PARAMETERS

Object	One of the following: • A handle to the object to be evaluated.

	• A handle to a parent object that is a prefix to the pathname. • A NULL handle if the pathname is fully qualified.
Pathname	Pathname of namespace object to evaluate. May be either an **absolute** path or a path **relative** to the *Object*.
MethodParams	If the object is a control method, this is a pointer to a list of parameters to pass to the method. This pointer may be NULL if no parameters are being passed to the method or if the object is not a method.
ReturnBuffer	A pointer to a location where the return value of the object evaluation (if any) is placed. If this pointer is NULL, no value is returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The object was successfully evaluated.
AE_AML_ERROR	An error occurred during the parsing of the AML code.
AE_BAD_CHARACTER	An invalid character was found in the *Pathname* parameter.
AE_BAD_PATHNAME	The path contains at least one ACPI name that is not exactly four characters long.
AE_BAD_PARAMETER	At least one of the following is true: • Both the *Object* and *Pathname* parameters are NULL. • The *Object* handle is NULL, but the *Pathname* is not absolute. • The *Pathname* is relative but the *Object* is invalid. • The *Pointer* field of the *ReturnBuffer* is NULL.
AE_BUFFER_OVERFLOW	The *Length* field of the *ReturnBuffer* is too small to hold the actual returned object. Upon return, the *Length* field contains the minimum required buffer length.
AE_ERROR	An unspecified error occurred.
AE_NO_MEMORY	Insufficient dynamic memory to complete the request.
AE_NOT_FOUND	The object referenced by the combination of the *Object* and *Pathname* was not found within the namespace.
AE_STACK_OVERFLOW	An internal stack overflow occurred because of an error in the AML, or because control methods or objects are nested too deep.
AE_STACK_UNDERFLOW	An internal stack underflow occurred during evaluation.
AE_TYPE	The object is of a type that cannot be evaluated.

Functional Description:

This function locates and evaluates objects in the namespace. This interface has two modes of operation, depending on the type of object that is being evaluated:

1) If the target object is a control method, the method is executed and the result (if any) is returned.

2) If the target is not a control method, the current "value" of that object is returned. The type of the returned value corresponds to the type of the object; for example, the object (and the corresponding returned result) may be a number, a string, or a buffer.

Specifying a Target Object: The target object may be any valid named ACPI object. To specify the object, a valid *Object*, a valid *Pathname*, or both may be provided. However, at least one of these parameters must be valid.

If the *Object* is NULL, the *Pathname* must be a fully qualified (absolute) namespace path.

If the *Object* is non-NULL, the *Pathname* may be either:

1) A path relative to the *Object* handle (a *relative* pathname as defined in the ACPI specification)

2) An absolute pathname. In this case, the *Object* handle is ignored.

Parameters to Control Methods: If the object to be evaluated is a control method, the caller can supply zero or more parameters that will be passed to the method when it is executed. The *MethodParams* parameter is a pointer to an ACPI_OBJECT_LIST that in turn is a counted array of ACPI_OBJECTs. If *MethodParams* is NULL, then no parameters are passed to the control method. If the *Count* field of *MethodParams* is zero, then the entire parameter is treated exactly as if it is a NULL pointer. If the object to be evaluated is not a control method, the *MethodParams* field is ignored.

Receiving Evaluation Results: The *ReturnObject* parameter optionally receives the results of the object evaluation. If this parameter is NULL, the evaluation results are not returned and are discarded. If there is no result from the evaluation of the object and no error occurred, the *Length* field of the *ReturnObject* parameter is set to zero.

Unsupported Object Types: The object types that cannot be evaluated are the following: ACPI_TYPE_Device.

EXAMPLES

Example 1: Executing the control method with an absolute path, two input parameters, with no return value expected:

```
ACPI_OBJECT_LIST       Params;
ACPI_OBJECT            Obj[2];


/* Initialize the parameter list */

Params.Count = 2;
Params.Pointer = &Obj;

/* Initialize the parameter objects */

Obj[0].Type = ACPI_TYPE_String;
Obj[0].String.Pointer = "ACPI User";

Obj[1].Type = ACPI_TYPE_Number;
Obj[1].Number.Value = 0x0E00200A;

/* Execute the control method */

Status = AcpiEvaluateObject (NULL,"\_SB.PCI0._TWO" , &Params, NULL);
```

Example 2: Before executing a control method that returns a result, we must declare and initialize an ACPI_BUFFER to contain the return value:

```
ACPI_BUFFER            Results;
ACPI_OBJECT            Obj;


/* Initialize the return buffer structure */

Results.Length = sizeof (Obj);
Results.Pointer = &Obj;
```

The three examples that follow are functionally identical.

Example 3: Executing a control method using an absolute path. In this example, there are no input parameters, but a return value is expected.

```
Status = AcpiEvaluateObject (NULL,"\_SB.PCI0._STA" , NULL, &Results);
```

Example 4: Executing a control method using a relative path. A return value is expected.

```
Status = AcpiPathnameToHandle ("\_SB.PCI0", &Object)
Status = AcpiEvaluateObject (Object, "_STA" , NULL, &Results);
```

Example 5: Executing a control method using a relative path. A return value is expected.

```
Status = AcpiPathnameToHandle ("\_SB.PCI0._STA", &Object)
Status = AcpiEvaluateObject (Object, NULL, NULL, &Results);
```

- **AcpiGetObjectInfo**

Get information about an ACPI-related device.

ACPI_STATUS
AcpiGetObjectInfo (
ACPI_HANDLE **Object,**
ACPI_DEVICE_INFO ***OutInfo)**

PARAMETERS

Object	A handle to an ACPI object for which information is to be returned.
OutInfo	A pointer to a location where the device info is returned.

RETURN

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	Device info was successfully returned. See the ACPI_DEVICE_INFO structure for valid returned fields.
AE_BAD_PARAMETER	At least one of the following is true: • The *Object* handle is invalid. • The *OutInfo* pointer is NULL.
AE_TYPE	The *Device* handle does not refer to an object of type ACPI_TYPE_Device.

Functional Description:

This function obtains information about an object contained within the ACPI namespace. The information returned is a composite of static internal information and the results of evaluating the following standard ACPI device methods and objects on behalf of the device:

Type — The ACPI object type of the object

Name — The 4-character ACPI name of the object

_HID — The hardware ID of the object.

_UID — The Unique ID of the object.

_ADR — The address of the object (bus and device specific).

_STA — The current status of the object/device.

Returned Data Format: The device information is returned in the ACPI_DEVICE_INFO structure that is defined as follows:

```
typedef struct
{
    ACPI_OBJECT_TYPE        Type;
    UINT32                  Name;
    UINT32                  Valid;
    char                    HardwareId [9];
    char                    UniqueId [9];
    UINT32                  Address;
    UINT32                  CurrentStatus;

} ACPI_DEVICE_INFO;
```

Where:

Type	Is the object type number
Name	The 4-character ACPI name of the object
Valid	A bitfield that indicates which of the remaining fields are valid.
HardwareId	The result of evaluating _HID for this object.
UniqueId	The result of evaluating _UID for this object.
Address	The result of evaluating _ADR for this object.
CurrentStatus	The result of evaluating _STA method for this object.

The fields of the structure that are valid because the corresponding method or object has been successfully found under the device are indicated by the values of the *Valid* bitfield via the following constants:

```
ACPI_VALID_HID
ACPI_VALID_UID
ACPI_VALID_ADR
ACPI_VALID_STA
```

Each bit should be checked before the corresponding value in the structure can be considered valid. **None** of the methods/objects that are used by this interface are *required* by the ACPI specification. Therefore, there is no guarantee that all or even any of them are available for a particular device. Even if none of the methods are found, the interface will return an AE_OK status — but none of the bits set in the *Valid* field return structure will be set.

Both the _HID and _UID values can be of either type STRING or NUMBER in the ACPI tables. In order to provide a consistent data type in the external interface, these values are always

returned as NULL terminated strings, regardless of the original data type in the source ACPI table. A data type conversion is performed if necessary.

- **AcpiGetNextObject**

Get a handle to the next child ACPI object of a parent object.

ACPI_STATUS
AcpiGetNextObject (

ACPI_OBJECT_TYPE	**Type,**
ACPI_HANDLE	**Parent,**
ACPI_HANDLE	**Child,**
ACPI_HANDLE	***OutHandle)**

PARAMETERS

Type	The desired type of the next object.
Parent	A handle to a parent object to be searched for the next child object.
Child	A handle to a child object. The **next** child object of the parent object that matches the *Type* will be returned. Use the value of NULL to get the first child of the parent.
OutHandle	A pointer to a location where a handle to the next child object is to be returned. If this pointer is NULL, the child object handle is not returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The next object was successfully found and returned.
AE_BAD_PARAMETER	At least one of the following is true:

- The *Parent* handle is invalid.
- The *Child* handle is invalid.
- The *Type* parameter refers to an invalid type

AE_NOT_FOUND	The child object parameter is the last object of the given type within the parent—a next child object was not found. If *Child* is NULL, this exception means that the parent object has no children.

Functional Description:

This function obtains the next child object of the parent object that is of type *Type*. Both the *Parent* and the *Child* parameters are optional. The behavior for the various combinations of *Parent* and *Child* is as follows:

1. If the *Child* is non-NULL, it is used as the starting point (the *current object*) for the search.
2. If the *Child* is NULL and the *Parent* is non-NULL, the search is performed starting at the beginning of the scope.

3. If both the *Parent* and the *Child* parameters are NULL, the search begins at the start of the namespace (the search begins at the *Root Object*).

If the search fails, an appropriate status will be returned and the value of *OutHandle* is undefined.

This interface is appropriate for use within a loop that looks up a group of objects within the internal namespace. However, the *AcpiWalkNamespace* primitive implements such a loop and may be simpler to use in your application; see the description of this interface for additional details.

- **AcpiGetParent**

Get a handle to the parent object of an ACPI object.

ACPI_STATUS
AcpiGetParent (
ACPI_HANDLE **Child,**
ACPI_HANDLE ***OutParent)**

PARAMETERS

Child	A handle to an object whose parent is to be returned.
OutParent	A pointer to a location where the handle to the parent object is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The parent object was successfully found and returned.
AE_BAD_PARAMETER	At least one of the following is true:

- The *Child* handle is invalid.
- The *OutParent* pointer is NULL.

AE_NULL_ENTRY	The referenced object has no parent. (Entries at the root level do not have a parent object.)

Functional Description:

This function returns a handle to the parent of the *Child* object. If an error occurs, a status code is returned and the value of *OutParent* is undefined.

- **AcpiGetType**

Get the type of an ACPI object.

ACPI_STATUS
AcpiGetType (
ACPI_HANDLE **Object,**
ACPI_OBJECT_TYPE ***OutType)**

PARAMETERS

Object	A handle to an object whose type is to be returned.
OutType	A pointer to a location where the object type is to be returned.

RETURN

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The object type was successfully returned.
AE_BAD_PARAMETER	At least one of the following is true:

- The *Object* handle is invalid.
- The *OutType* pointer is NULL.

Functional Description:

This function obtains the type of an ACPI namespace object. See the definition of the ACPI_OBJECT_TYPE for a comprehensive listing of the available object types.

- **AcpiGetHandle**

Get the object handle associated with an ACPI name.

ACPI_STATUS
AcpiGetHandle (
ACPI_HANDLE **Parent,**
ACPI_STRING ***Pathname,**
ACPI_HANDLE ***OutHandle)**

PARAMETERS

Parent	A handle to the parent of the object specified by *Pathame*. In other words, the *Pathame* is relative to the *Parent*. If *Parent* is NULL, the pathname must be a fully qualified pathname.
Pathname	A name or pathname to an ACPI object (a NULL terminated ASCII string). The string can be either a single segment ACPI name or a multiple segment ACPI pathname (with path separators).
OutHandle	A pointer to a location where a handle to the object is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The pathname was successfully associated with an object and the handle was returned.
AE_BAD_CHARACTER	An invalid character was found in the pathname.
AE_BAD_PATHNAME	The path contains at least one ACPI name that is not exactly four characters long.
AE_BAD_PARAMETER	At least one of the following is true: • The *Pathname* pointer is NULL. • The *Pathname* does not begin with a backslash character. • The *OutHandle* pointer is NULL.
AE_NO_NAMESPACE	The namespace has not been successfully loaded.
AE_NOT_FOUND	One or more of the segments of the pathname refers to a non-existent object.

Functional Description:

This function translates an ACPI pathname into an object handle. It locates the object in the namespace via the combination of the *Parent* and *Pathame* parameters. Only the specified *Parent* object will be searched for the name—this function will not perform a walk of the namespace tree (See *AcpiWalkNamespace*).

The pathname is relative to the *Parent*. If the parent object is NULL, the *Pathname* must be fully qualified (absolute), meaning that the path to the object must be a complete path from the root of the namespace, and the pathname must begin with a backslash ('\').

Multiple instances of the same name under a given parent (within a given scope) are not allowed by the ACPI specification. However, if more than one instance of a particular name were to appear under a single parent in the ACPI DSDT, only the first one would be successfully loaded into the internal namespace. The second attempt to load the name would collide with the first instance of the name, and the second instance would be ignored.

If the operation fails an appropriate status will be returned and the value of *OutHandle* is undefined.

- **AcpiGetName**

Get the name of an ACPI object.

ACPI_STATUS
AcpiGetName (

ACPI_HANDLE	**Object,**
UINT32	**NameType**
ACPI_BUFFER	***OutNname)**

PARAMETERS

Object	A handle to an object whose name or pathname is to be returned.
NameType	The type of name to return, must be one of these manifest constants: • ACPI_FULL_PATHNAME – return a complete pathname (from the namespace root) to the object • ACPI_SINGLE_NAME – return a single segment ACPI name for the object (4 characters, null terminated).
OutName	A pointer to a location where the fully qualified and NULL terminated name or pathname is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The full pathname associated with the handle was successfully retrieved and returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *Parent* handle is invalid. • The *Object* handle is invalid. • The *OutName* pointer is NULL. • The *Pointer* field of *OutName* is NULL.
AE_BUFFER_OVERFLOW	The *Length* field of *OutName* indicates that the buffer is too small to hold the actual pathname. Upon return, the *Length* field contains the minimum required buffer length.
AE_NO_NAMESPACE	The namespace has not been successfully loaded.

Functional Description:

This function obtains the name that is associated with the *Object* parameter. The returned name can be either a full pathname (from the root, with path segment separators) or a single segment, 4-character ACPI name. This function and *AcpiGetHandle* are complementary functions, as shown in the examples below.

EXAMPLES

Example 1: The following operations:

```
Status = AcpiGetName (Handle, ACPI_FULL_PATHNAME, &OutName)
Status = AcpiGetHandle (NULL, OutName.BufferPtr, &OutHandle))
```

Yield this result:

```
Handle == OutHandle;
```

Example 2: If *Name* is a 4-character ACPI name, the following operations:

```
Status = AcpiGetHandle (Parent, Name, &OutHandle))
Status = AcpiGetName (OutHandle, ACPI_SINGLE_NAME, &OutName)
```

Yield this result:

```
Name == OutName.BufferPtr
```

- **AcpiWalkNamespace**

Traverse a portion of the ACPI namespace to find objects of a given type.

ACPI_STATUS
AcpiWalkNamespace (

ACPI_OBJECT_TYPE	**Type,**
ACPI_HANDLE	**StartObject,**
UINT32	**MaxDepth,**
WALK_CALLBACK	**UserFunction,**
Void	***UserContext)**
Void	****ReturnValue**

PARAMETERS

Type	The type of object desired.
StartObject	A handle to an object where the namespace walk is to begin. The constant `ACPI_ROOT_OBJECT` indicates to start the walk at the root of the namespace (walk the entire namespace.)
MaxDepth	The maximum number of levels to descend in the namespace during the walk.
UserFunction	A pointer to a user-written function that is invoked for each matching object that is found during the walk. (See the interface specification for the user function below.)
UserContext	A value that will be passed as a parameter to the user function each time it is invoked.
ReturnValue	A pointer to a location where the (void *) return value from the *UserFunction* is to be placed if the walk was terminated early. Otherwise, `NULL` is returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The walk was successful. Termination occurred from completion of the walk or by the user function, depending on the value of the return parameter.
AE_BAD_PARAMETER	At least one of the following is true: • The *MaxDepth* is zero. • The *UserFunction* address is `NULL`. • The *StartObject* handle is invalid. • The *Type* is invalid.

Functional Description:

This function performs a modified depth-first walk of the namespace tree, starting (and ending) at the object specified by the *StartObject* handle. The *UserFunction* is invoked whenever an object that matches the type parameter is found. If the user function returns a non-zero value, the search is terminated immediately and this value is returned to the caller.

The point of this procedure is to provide a generic namespace walk routine that can be called from multiple places to provide multiple services; the user function can be tailored to each task — whether it is a print function, a compare function, etc.

- **Interface to User Callback Function**

Interface to the user function that is invoked from AcpiWalkNamespace.

typedef
ACPI_STATUS (*WALK_CALLBACK) (

ACPI_HANDLE	**ObjHandle,**
UINT32	**NestingLevel,**
Void	***Context,**
Void	****ReturnValue)**

PARAMETERS

ObjHandle	A handle to an object that matches the search criteria.
Nesting Level	Depth of this object within the namespace (distance from the root)
Context	The *UserContext* value that was passed as a parameter to the *AcpiWalkNamespace* function.
ReturnValue	A pointer to a location where the return value (if any) from the user function is to be stored.

RETURN VALUE

Status	AE_OK	Continue the walk
	AE_TERMINATE	Stop the walk immediately
	AE_DEPTH	Go no deeper into the namespace tree
	All others	Abort the walk with this exception code

Functional Description:

This function is called from *AcpiWalkNamespace* whenever a object of the desired type is found. The walk can be modified by the exception code returned from this function. AE_TERMINATE will abort the walk immediately, and *AcpiWalkNamespace* will return AE_OK to the original caller. AE_DEPTH will prevent the walk from progressing any deeper down the current branch of the namespace tree. AE_OK is the normal return that allows the walk to continue normally. All other exception codes will cause the walk to terminate and the exception is returned to the original caller of *AcpiWalkNamespace.*

• *ACPI Resource Management*

• AcpiGetCurrentResources

Get the current resource list associated with an ACPI-related device.

ACPI_STATUS
AcpiGetCurrentResources (

ACPI_HANDLE	**Device,**
ACPI_BUFFER	***OutBuffer)**

PARAMETERS

Device	A handle to a device object for which the current resources are to be returned.
OutBuffer	A pointer to a location where the current resource list is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The resource list was successfully returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *Device* handle is invalid. • The *OutBuffer* pointer is NULL. • The *Pointer* field of *OutBuffer* is NULL.
AE_BUFFER_OVERFLOW	The *Length* field of *OutBuffer* indicates that the buffer is too small to hold the resource list. Upon return, the *Length* field contains the minimum required buffer length.
AE_TYPE	The *Device* handle refers to an object that is not of type ACPI_TYPE_Device.

Functional Description:

This function obtains the current resources for a specific device. The caller must first acquire a handle for the desired device. The resource data is placed in the buffer pointed contained in the OutBuffer structure. Upon completion the *Length* field of *OutBuffer* will indicate the number of bytes copied into the *Pointer* field of the *OutBuffer* buffer. This routine will never return a partial resource structure.

If the function fails an appropriate status will be returned and the value of *OutBuffer* is undefined.

- ## AcpiGetPossibleResources

Get the possible resource list associated with an ACPI-related device.

ACPI_STATUS
AcpiGetPossibleResources (

ACPI_HANDLE	**Device,**
ACPI_BUFFER	***OutBuffer)**

PARAMETERS

Device	A handle to a device object for which the possible resources are to be returned..
OutBuffer	A pointer to a location where the possible resource list is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The resource list was successfully returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *Device* handle is invalid. • The *OutBuffer* pointer is NULL. • The *Pointer* field of *OutBuffer* is NULL.
AE_BUFFER_OVERFLOW	The *Length* field of *OutBuffer* indicates that the buffer is too small to hold the resource table. Upon return, the *Length* field contains the minimum required buffer length.
AE_TYPE	The *Device* handle refers to an object that is not of type ACPI_TYPE_Device.

Functional Description:

This function obtains the list of the possible resources for a specific device. The caller must first acquire a handle for the desired device. The resource data is placed in the buffer contained in the *OutBuffer* structure. Upon completion the *Length* field of *OutBuffer* will indicate the number of bytes copied into the *Pointer* field of the *OutBuffer* buffer. This routine will never return a partial resource structure.

If the function fails an appropriate status will be returned and the value of *OutBuffer* is undefined.

- **AcpiSetCurrentResources**

Set the current resource list associated with an ACPI-related device.

ACPI_STATUS
AcpiSetCurrentResources (

ACPI_HANDLE	**Device,**
ACPI_BUFFER	***InBuffer)**

PARAMETERS

Device	A handle to a device object for which the current resource list is to be set.
InBuffer	A pointer to an ACPI_BUFFER containing the resources to be set for the device.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The resources were set successfully.
AE_BAD_PARAMETER	At least one of the following is true: • The *Device* handle is invalid. • The *InBuffer* pointer is NULL. • The *Pointer* field of *InBuffer* is NULL. • The *Length* field of *InBuffer* is zero.
AE_TYPE	The *Device* handle refers to an object that is not of type ACPI_TYPE_Device.

Functional Description:

This function sets the current resources for a specific device. The caller must first acquire a handle for the desired device. The resource data is passed to the routine the buffer pointed to by the *InBuffer* variable.

- **AcpiGetIRQRoutingTable**

Get the ACPI Interrupt Request (IRQ) Routing Table for an ACPI-related device.

ACPI_STATUS
AcpiGetIRQRoutingTable (

ACPI_HANDLE	**Device,**
ACPI_BUFFER	***OutBuffer)**

PARAMETERS

Device	A handle to a device object for which the IRQ routing table is to be returned.

OutBuffer	A pointer to a location where the IRQ routing table is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The system information list was successfully returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *Device* handle is invalid. • The *OutBuffer* pointer is NULL. • The *Pointer* field of *OutBuffer* is NULL.
AE_BUFFER_OVERFLOW	The *Length* field of *OutBuffer* indicates that the buffer is too small to hold the IRQ table. Upon return, the *Length* field contains the minimum required buffer length.
AE_TYPE	The *Device* handle refers to an object that is not of type ACPI_TYPE_Device.

Functional Description:

This function obtains the IRQ routing table for a specific bus. It does so by attempting to execute the _PRT method contained in the scope of the device whose handle is passed as a parameter.

If the function fails an appropriate status will be returned and the value of *OutBuffer* is undefined.

• *ACPI Event Management*

• AcpiEnable

Put the system into ACPI mode.

ACPI_STATUS
AcpiEnable (void)

PARAMETERS

None

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	ACPI mode was successfully enabled.
AE_ERROR	Either ACPI mode is not supported by this system (legacy mode only), the SCI interrupt handler could not be installed, or the system could not be transitioned into ACPI mode.

AE_NO_ACPI_TABLES	The ACPI tables have not been successfully loaded.

Functional Description:

This function enables ACPI mode on the host computer system. It ensures that the system control interrupt (SCI) is properly configured, disables SCI event sources, installs the SCI handler, and transfers the system hardware into ACPI mode.

- **AcpiDisable**

Take the system out of ACPI mode.

ACPI_STATUS
AcpiDisable (void)

PARAMETERS

None

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	ACPI mode was successfully disabled.
AE_ERROR	The system could not be transitioned out of ACPI mode.

Functional Description:

This function disables ACPI mode on the host computer system. It returns the system hardware to original ACPI/legacy mode, disables all events, and removes the SCI interrupt handler.

- **AcpiEnableEvent**

Enable an ACPI Event (Fixed Events and General Purpose Events).

ACPI_STATUS
AcpiEnableEvent (

UINT32	**Event,**
UINT32	**Type)**

PARAMETERS

Event	The fixed event or GPE to be enabled.
Type	The type of event, one of these manifest constants: EVENT_FIXED EVENT_GPE

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The event was successfully enabled.
AE_BAD_PARAMETER	At least one of the following is true: • The *Event* is invalid. • The *Type* is invalid.

Functional Description:

This function is enables a single ACPI event. Both Fixed Events and General Purpose Events may be enabled with this interface.

- **AcpiDisableEvent**

Disable an ACPI Event (Fixed Events and General Purpose Events).

ACPI_STATUS
AcpiDisableEvent (

UINT32	**Event,**
UINT32	**Type)**

PARAMETERS

Event	The fixed event or GPE to be disabled.
Type	The type of event, one of these manifest constants: EVENT_FIXED EVENT_GPE

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The event was successfully disabled.
AE_BAD_PARAMETER	At least one of the following is true: • The *Event* is invalid. • The *Type* is invalid.

Functional Description:

This function disables a single ACPI event. Both Fixed Events and General Purpose Events may be disabled with this interface.

- **AcpiInstallFixedEventHandler**

Install a handler for ACPI Fixed Events.

ACPI_STATUS
AcpiInstallFixedEventHandler (

ACPI_EVENT_TYPE	**Event,**
FIXED_EVENT_HANDLER	**Handler,**
void	***Context)**

PARAMETERS

Event	The fixed event to be managed by this handler.
Handler	Address of the handler to be installed.
Context	A context value that will be passed to the handler as a parameter.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully installed.
AE_BAD_PARAMETER	At least one of the following is true: • The *Event* is invalid. • The *Handler* pointer is `NULL`.
AE_ERROR	The fixed event enable register could not be written.
AE_EXIST	A handler for this event is already installed.

Functional Description:

This function installs a handler for a predefined fixed event.

- **Interface to Fixed Event Handlers**

Definition of the handler interface for Fixed Events.

typedef
UINT32 (*FIXED_EVENT_HANDLER) (

void	***Context)**

PARAMETERS

Context	The *Context* value that was passed as a parameter to the *AcpiInstallFixedEventHandler* function.

RETURN VALUE

None

Functional Description:

This handler is installed via *AcpiInstallFixedEventHandler*. It is called whenever the particular fixed event it was installed to handle occurs.

NOTE: This function executes in the context of an interrupt handler.

- **AcpiRemoveFixedEventHandler**

Remove an ACPI Fixed Event handler.

ACPI_STATUS
AcpiRemoveFixedEventHandler (

ACPI_EVENT_TYPE	**Event,**
FIXED_EVENT_HANDLER	**Handler)**

PARAMETERS

Event	The fixed event whose handler is to be removed.
Handler	Address of the previously installed handler.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully removed.
AE_BAD_PARAMETER	At least one of the following is true: • The *Event* is invalid. • The *Handler* pointer is NULL. • The *Handler* address is not the same as the one that is installed.
AE_ERROR	The fixed event enable register could not be written.
AE_NOT_EXIST	There is no handler installed for this event.

Functional Description:

This function removes a handler for a predefined fixed event that was previously installed via a call to *AcpiInstallFixedEventHandler*.

- **AcpiInstallGpeHandler**

Install a handler for ACPI General Purpose Events.

ACPI_STATUS
AcpiInstallGpeHandler (

UINT32	**GpeNumber,**
GPE_HANDLER	**Handler,**
void	***Context)**

PARAMETERS

GpeNumber	A zero based Gpe number. Gpe numbers start with GPE register bank zero, and continue sequentially through GPE bank one.
Handler	Address of the handler to be installed.
Context	A context value that will be passed to the handler as a parameter.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully installed.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeNumber* is invalid. • The *Handler* pointer is `NULL`.
AE_EXIST	A handler for this general purpose event is already installed.

Functional Description:

This function installs a handler for a general purpose event

- **Interface to General Purpose Event Handlers**

Definition of the handler interface for General Purpose Events.

typedef
void (*GPE_HANDLER) (

Void	***Context)**

PARAMETERS

Context	The *Context* value that was passed as a parameter to the *AcpiInstallGpeHandler* function.

RETURN VALUE

None

Functional Description:

This handler is installed via *AcpiInstallGpeHandler*. It is called whenever the particular general purpose event it was installed to handle occurs.

NOTE: This function executes in the context of an interrupt handler.

- **AcpiRemoveGpeHandler**

Remove an ACPI General Purpose Event handler.

ACPI_STATUS
AcpiRemoveGpeHandler (

UINT32	**GpeNumber,**
GPE_HANDLER	**Handler)**

PARAMETERS

GpeNumber	A zero based Gpe number. Gpe numbers start with GPE register bank zero, and continue sequentially through GPE bank one.
Handler	Address of the previously installed handler.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully removed.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeNumber* is invalid. • The *Handler* pointer is NULL. • The *Handler* address is not the same as the one that is installed
AE_NOT_EXIST	There is no handler installed for this general purpose event.

Functional Description:

This function removes a handler for a general purpose event that was previously installed via a call to *AcpiInstallGpeHandler*.

- **AcpiInstallNotifyHandler**

Install a handler for notification events on an ACPI object.

ACPI_STATUS
AcpiInstallNotifyHandler (

ACPI_HANDLE	**Object,**
UINT32	**Type,**
NOTIFY_HANDLER	**Handler,**
void	***Context)**

PARAMETERS

Object	Handle to the object for which notify events will be handled. Notifies on this object will be dispatched to the handler. If ACPI_ROOT_OBJECT is specified, the handler will become a global handler that receives all (systemwide) notifications of the *Type* specified. Otherwise, this object must be one of the following types: ACPI_TYPE_Device ACPI_TYPE_Processor ACPI_TYPE_Power ACPI_TYPE_ThermalZone
Type	Specifies the type of notifications that are to be received by this handler: ACPI_SYSTEM_NOTIFY – Notifications 0x00 to 0x7F ACPI_DEVICE_NOTIFY – Notifications 0x80 to 0xFF
Handler	Address of the handler to be installed.
Context	A context value that will be passed to the handler as a parameter.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully installed.
AE_BAD_PARAMETER	At least one of the following is true: • The *Object* handle is invalid. • The *Type* is not a valid value. • The *Handler* pointer is `NULL`.
AE_EXIST	A handler for notifications on this object is already installed.
AE_TYPE	The type of the *Object* is not one of the supported object types.

Functional Description:

This function installs a handler for notify events on an ACPI object. According to the ACPI specification, the only objects that can receive notifications are Devices and Thermal Zones.

A global handler for each notify type may be installed by using the ACPI_ROOT_OBJECT constant as the object handle. When a notification is received, it is first dispatched to the global handler (if there is one), and then to the device-specific notify handler (if there is one)

- **Interface to Notification Event Handlers**

Definition of the handler interface for Notification Events.

typedef
void (*NOTIFY_HANDLER) (

ACPI_HANDLE	**Device**
UINT32	**Value,**
void	***Context)**

PARAMETERS

Device	The handle for the device on which the notify occurred.
Value	The **notify** value that was passed as a parameter to the AML notify operation.
Context	The *Context* value that was passed as a parameter to the *AcpiInstallNotifyHandler* function.

RETURN VALUE

None

Functional Description:

This handler is installed via *AcpiInstallNotifyHandler*. It is called whenever a **notify** occurs on the target object. If the handler is installed as a global notification handler, it is called for every notify of the type specified when it was installed.

NOTE: this function **does not** execute in the context of an interrupt handler.

- **AcpiRemoveNotifyHandler**

Remove a handler for ACPI notification events.

ACPI_STATUS
AcpiRemoveNotifyHandler (

ACPI_HANDLE	**Object,**
UINT32	**Type,**
NOTIFY_HANDLER	**Handler)**

PARAMETERS

Object	Handle to the object for which a notify handler will be removed. If ACPI_ROOT_OBJECT is specified, the global handler of the *Type* specified is removed. Otherwise, this object must be one of the following types: ACPI_TYPE_Device

	ACPI_TYPE_Processor ACPI_TYPE_Power ACPI_TYPE_ThermalZone
HandlerType	Specifies the type of notify handler to be removed: ACPI_SYSTEM_NOTIFY – Notifications 0x00 to 0x7F ACPI_DEVICE_NOTIFY – Notifications 0x80 to 0xFF
Handler	Address of the previously installed handler.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully removed.
AE_BAD_PARAMETER	At least one of the following is true: • The *Object* handle is invalid. • The *Handler* pointer is `NULL`. • The *Handler* address is not the same as the one that is installed
AE_NOT_EXIST	There is no handler installed for notifications on this object.
AE_TYPE	The type of the Object is not one of the supported object types

Functional Description:

This function removes a handler for notify events that was previously installed via a call to *AcpiInstallNotifyHandler*.

- **AcpiInstallAddressSpaceHandler**

Install a handler for ACPI Operation Region events.

ACPI_STATUS
AcpiInstallAddressSpaceHandler (

ACPI_HANDLE	**Device,**
UINT32	**SpaceId,**
ADDRESS_SPACE_HANDLER	**Handler,**
void	***Context)**

PARAMETERS

Device	Handle for the device for which a address space handler will be installed. This object may be specified as the ACPI_ROOT_OBJECT to request global scope. Otherwise, this object must be one of the following types: ACPI_TYPE_Device, ACPI_TYPE_Processor, ACPI_TYPE_ThermalZone
SpaceId	The ID of the Address Space or Operation Region to be managed by this handler.

Handler	Address of the handler to be installed if the special value ACPI_DEFAULT_HANDLER is used the handler supplied with by the ACPI CA for that address space will be installed.
Context	A context value that will be passed to the handler as a parameter.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully installed.
AE_BAD_PARAMETER	At least one of the following is true: • The *Device* handle does not refer to an object of type *Device, Processor, ThermalZone,* or the root object. • The *SpaceId* is invalid. • The *Handler* pointer is NULL.
AE_EXIST	A handler for this address space or operation region is already installed.
AE_NOT_EXIST	ACPI_DEFAULT_HANDLER was specified for an address space which has no default handler.
AE_NO_MEMORY	There was insufficient memory to install the handler.

Functional Description:

This function installs a handler for an Address Space.

- **Interface to Address Space Handlers**

Definition of the handler interface for Operation Region Events.

typedef
void (*ADDRESS_SPACE_HANDLER) (

UINT32	**Function,**
UINT32	**Address,**
UINT32	**BitWidth,**
UINT32	***Value,**
Void	***Context)**

PARAMETERS

Function	The type of function to be performed; must be one of the following manifest constants: ADDRESS_SPACE_READ ADDRESS_SPACE_WRITE
Address	A space-specific address where the operation is to be performed.
BitWidth	The width of the operation, typically 8, 16, 32, or 64.
*Value	A pointer to the value to be written (WRITE), or where the value that was read should be returned (READ).

Context	An address space specific *Context* value. Typically this is the context that was passed as a parameter to the *AcpiInstallAddressSpaceHandler* function.

RETURN VALUE

None

Functional Description:

This handler is installed via *AcpiInstallAddressSpaceHandler*. It is invoked whenever AML code attempts to access the target Operation Region.

NOTE: this function **does not** execute in the context of an interrupt handler.

- **Context for the Default PCI Address Space Handler**

Definition of the context required for installation of the default PCI address space handler.

UINT32 PCIContext

Where PCIContext contains the PCI bus number and the PCI segment number. With the bus number in the low 16 bits and the segment number in the high 16 bits.

- **AcpiRemoveAddressSpaceHandler**

Remove an ACPI Operation Region handler.

ACPI_STATUS
AcpiRemoveAddressSpaceHandler (
UINT32 SpaceId,
ADDRESS_SPACE_HANDLER Handler)

PARAMETERS

SpaceId	The ID of the Address Space or Operation Region whose handler is to be removed.
Handler	Address of the previously installed handler.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully removed.
AE_BAD_PARAMETER	At least one of the following is true: • The *SpaceId* is invalid. • The *Handler* pointer is `NULL`. • The *Handler* address is not the same as the one that is installed

AE_NOT_EXIST	There is no handler installed for this address space or operation region.

Functional Description:

This function removes a handler for an Address Space or Operation Region that was previously installed via a call to *AcpiInstallAddressSpaceHandler*.

• *ACPI Hardware Management*

• AcpiSetFirmwareWakingVector

Set the ROM BIOS wake vector.

ACPI_STATUS
AcpiSetFirmwareWakingVector (
void ***Vector)**

PARAMETERS

Vector	The physical address to be stored in the waking vector.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The vector was set successfully.
AE_BAD_PARAMETER	Unrecognized parameter

Functional Description:

This function sets the firmware (ROM BIOS) wake vector.

If the function fails an appropriate status will be returned and the value of the waking vector will be undisturbed.

- **AcpiGetFirmwareWakingVector**

Get the current value of the ROM BIOS wake vector.

ACPI_STATUS
AcpiGetFirmwareWakingVector (
void ****OutVector)**

PARAMETERS

OutVector	A pointer to a location where the current vector (physical address) is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The vector was successfully returned.
AE_BAD_PARAMETER	The OutVector pointer is NULL.

Functional Description:

This function obtains the BIOS wake vector. This address is returned as a (void *) physical address.

If the function fails an appropriate status will be returned and the value of the *OutVector* location will be undetermined.

- **AcpiAcquireGlobalLock**

Acquire the ACPI Global Lock.

ACPI_STATUS
AcpiAcquireGlobalLock (
UINT32 **Timeout,**
ACPI_HANDLE ***OutHandle)**

PARAMETERS

Timeout	The maximum time (in *System Ticks*) the caller is willing to wait for the global lock.
OutHandle	A pointer to where a handle to the lock is to be returned. This handle is required to release the global lock.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The global lock was successfully acquired.
AE_BAD_PARAMETER	The *OutHandle* pointer is NULL.
AE_TIME	The global lock could not be acquired within the specified time limit.

Functional Description:

This function obtains exclusive access to the single system-wide ACPI Global Lock. The purpose of the global lock is to ensure exclusive access to resources that must be shared between the operating system and the firmware.

- **AcpiReleaseGlobalLock**

Release the ACPI Global Lock.

ACPI_STATUS
AcpiReleaseGlobalLock (
ACPI_HANDLE **Handle)**

PARAMETERS

Handle	The handle that was obtained when the Global Lock was acquired. This allows different threads to acquire and release the lock, as long as they share the handle.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The global lock was successfully released
AE_BAD_PARAMETER	The *Handle* is invalid.

Functional Description:

This function releases the global lock. The releasing thread may be different from the thread that acquired the lock. However, the Handle must be the same handle that was returned by *AcpiAcquireGlobalLock*.

- **AcpiGetTimer**

Get the current value of the ACPI timer.

ACPI_STATUS
AcpiGetTimer (
UINT32 ***OutValue)**

PARAMETERS

OutValue	A pointer to where the current value of the ACPI Timer is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The current value of the timer was successfully retrieved and returned.
AE_BAD_PARAMETER	The *OutValue* pointer is `NULL`.

Functional Description:

This function returns the current value of the ACPI timer.

- **ACPI Register Access**

- AcpiReadEnableBit
- AcpiReadStatusBit
- AcpiClearStatusBit
- AcpiReadRegister
- AcpiWriteRegister

- **Legacy Mode Support**

  - AcpiSetLegacyMode
  - AcpiLegacyModeStatus
  - AcpiLegacyModeCapabilities

- **Sleep Transitions Support (S-states)**
  - AcpiSleepStatesSupported
  - AcpiSleep
  - AcpiWakeup

OS-Dependent Component - External Interface Definition

This section contains the definitions of the interfaces that must be exported by the OS-dependent component. The OS-independent component requires all of these interfaces. All interfaces to the OS-dependent component *that are intended for use by the OS-Independent Component* of the ACPI subsystem are prefixed by the letters **Osd**.

• *Memory Management*

These interfaces provide an OS-independent memory management interface for use by the OSI component.

• OsdMapMemory

Map physical memory into the caller's address space.

void *
OsdMapMemory (

Void	***PhysicalAddress,**
UINT32	**Length)**

PARAMETERS

PhysicalAddress	A full physical address of the memory to be mapped into the caller's address space.
Length	The amount of memory to be mapped starting at the given physical address.

RETURN VALUE

Memory	A valid pointer to the mapped memory. NULL is returned if the operation could not be completed.

Functional Description:

This function maps a physical address into the caller's address space. A logical pointer is returned.

- **OsdUnMapMemory**

Remove a physical to logical memory mapping.

void
OsdUnMapMemory (
Void ***LogicalAddress,**
UINT32 **Length)**

PARAMETERS

LogicalAddress	The logical address that was returned from a previous call to *OsdMapMemory*.
Length	The amount of memory that was mapped. This value must be identical to the value used in the call to *OsdMapMemory*.

RETURN VALUE

None

Functional Description:

This function deletes a mapping that was created by *OsdMapMemory*.

- **OsdAllocate**

Allocate memory from the dynamic memory pool.

void *
OsdAllocate (
UINT32 **Size)**

PARAMETERS

Size	Amount of memory to allocate.

RETURN VALUE

Memory	A pointer to the allocated memory. A NULL pointer is returned on error.

Functional Description:

This function dynamically allocates memory. The returned memory cannot be assumed to be initialized to any particular value or values.

- **OsdCallocate**

Allocate and initialize memory.

void *
OsdCallocate (
UINT32 **Size)**

PARAMETERS

Size	Amount of memory to allocate.

RETURN VALUE

Memory	A pointer to the allocated memory. A NULL pointer is returned on error.

Functional Description:

This function dynamically allocates and initializes memory. The returned memory is guaranteed to be initialized to all zeros.

- **OsdFree**

Free previously allocated memory.

void
OsdFree (
void ***Memory)**

PARAMETERS

Memory	A pointer to the memory to be freed.

RETURN VALUE

None

Functional Description:

This function frees memory that was previously allocated via *OsdAllocate* or *OsdCallocate*.

- ***Scheduling Services***

- **OsdQueueForExecution**

Schedule a procedure for deferred execution.

ACPI_STATUS
OsdQueueForExecution (

UINT32	**Priority,**
OSD_EXECUTION_CALLBACK	**Function,**
Void	***Context);)**

PARAMETERS

Priority	Requested priority of the execution – one of these manifest constants: OSD_PRIORITY_HIGH OSD_PRIORITY_MED OSD_PRIORITY_LO
Function	Address of the procedure to execute.
Context	A context value to be passed to the called procedure.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The procedure was successfully queued for execution by the host operating system. This does not indicate that the procedure has actually executed, however.
AE_BAD_PARAMETER	At least one of the following is true: • The *Priority* is invalid. • The *Function* pointer is NULL.

Functional Description:

This function queues a procedure for later scheduling and execution. .

- **OsdSleep**

Suspend the running task (course granularity).

ACPI_STATUS
OsdSleep (
UINT32 **Seconds,**
UINT32 **Milliseconds);)**

PARAMETERS

Seconds	The number of whole seconds to sleep.
Milliseconds	The number of partial seconds to sleep, in milliseconds.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The running thread slept for the time specified.
AE_BAD_PARAMETER	Unrecognized parameter
AE_ERROR	The running thread did not sleep because of a host OS error.

Functional Description:

This function sleeps for the specified time. Execution of the running thread is suspended for this time. The sleep granularity is one millisecond.

- **OsdSleepUsec**

Suspend the running task (fine granularity).

ACPI_STATUS
OsdSleepUsec (
UINT32 **Microseconds)**

PARAMETERS

Micrososeconds	The amount of time to sleep in microseconds.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The running thread slept for the time specified.
AE_BAD_PARAMETER	Unrecognized parameter
AE_ERROR	The running thread did not sleep because of a host OS error.

Functional Description:

This function sleeps for the specified time. Execution of the running thread is suspended for this time. The sleep granularity is one microsecond.

- ***Mutual Exclusion and Synchronization***

Thread synchronization and locking.

NOTE: These interfaces **MUST** perform parameter validation of the input handle to at least the extent of detecting a null handle and returning the appropriate exception.

3.6.1 OsdCreateSemaphore

Create a semaphore.

ACPI_STATUS
OsdCreateSemaphore (

UINT32	**InitialUnits,**
ACPI_HANDLE	***OutHandle)**

PARAMETERS

InitialUnits	The initial number of units to be assigned to the semaphore.
OutHandle	A pointer to a location where a handle to the semaphore is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The semaphore was successfully created.
AE_BAD_PARAMETER	At least one of the following is true: • The *InitialUnits* is invalid. • The *OutHandle* pointer is NULL.
AE_NO_MEMORY	Insufficient memory to create the semaphore.

Functional Description:

Create a standard semaphore.

3.6.2 OsdDeleteSemaphore

Delete a semaphore.

ACPI_STATUS
OsdDeleteSemaphore (
ACPI_HANDLE **Handle)**

PARAMETERS

Handle	A handle to a semaphore object that was returned by a previous call to *OsdCreateSemaphore*.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The semaphore was successfully deleted.
AE_BAD_PARAMETER	The *Handle* is invalid.

Functional Description:

Delete a semaphore.

3.6.3 OsdWaitSemaphore

Wait for units from a semaphore.

ACPI_STATUS
OsdWaitSemaphore (
ACPI_HANDLE **Handle,**
UINT32 **Units,**
UINT32 **Timeout)**

PARAMETERS

Handle	A handle to a semaphore object that was returned by a previous call to *OsdCreateSemaphore*.
Units	The number of units the caller is requesting.
Timeout	How long the caller is willing to wait for the units.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The requested units were successfully received.
AE_BAD_PARAMETER	The *Handle* is invalid.
AE_TIME	The units could not be acquired within the specified time limit.

Functional Description:

Wait for the specified number of units from a semaphore.

OsdSignalSemaphore

Send units to a semaphore.

ACPI_STATUS
OsdSignalSemaphore (

ACPI_HANDLE	**Handle,**
UINT32	**Units)**

PARAMETERS

Handle	A handle to a semaphore object that was returned by a previous call to *OsdCreateSemaphore*.
Units	The number of units to send to the semaphore.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The semaphore was successfully signaled.
AE_BAD_PARAMETER	The *Handle* is invalid.

Functional Description:

Send the requested number of units to a semaphore.

• *Interrupt Handling*

Interrupt handler installation and removal.

• OsdInstallInterruptHandler

Install a handler for a hardware interrupt level.

ACPI_STATUS
OsdInstallInterruptHandler (

UINT32	**InterruptLevel,**
OSD_HANDLER	**Handler,**
void	***Context)**

PARAMETERS

InterruptNumber	Interrupt level that the handler will service.
Handler	Address of the handler.
Context	A context value that is passed to the handler when the interrupt is dispatched.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully installed.
AE_BAD_PARAMETER	At least one of the following is true: • The *InterruptNumbe*r is invalid. • The *Handler* pointer is `NULL`.
AE_EXIST	A handler for this interrupt level is already installed.

Functional Description:

This function installs an interrupt handler for a hardware interrupt level. The ACPI driver must install an interrupt handler to service the SCI (System Control Interrupt) which it owns. The interrupt level for the SCI interrupt is obtained from the ACPI tables.

- **Interface to OS-Independent Interrupt Handlers**

Definition of the interface for OS-independent interrupt handlers.

typedef
UINT32 (*OSD_HANDLER) (
Void ***Context)**

PARAMETERS

Context	The Context value that was passed as a parameter to the OsdInstallInterruptHandler function.

RETURN VALUE

HandlerActionTaken	The handler should return one of the following manifest constants: INTERRUPT_HANDLED INTERRUPT_NOT_HANDLED INTERRUPT_ERROR

Functional Description:

The OS-independent interrupt handler must be called from an OS-dependent interrupt handler "wrapper" that exists within the OS-dependent component. It is the responsibility of the OS-dependent component to manage the installed interrupt handler(s), and dispatch interrupts to the handler(s) appropriately.

- **OsdRemoveInterruptHandler**

Remove an interrupt handler.

ACPI_STATUS
OsdRemoveInterruptHandler (
UINT32 **InterruptNumber,**
OSD_HANDLER **Handler)**

PARAMETERS

InterruptNumber	Interrupt number that the handler is currently servicing.
Handler	Address of the handler that was previously installed.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully removed.
AE_BAD_PARAMETER	At least one of the following is true:

	• The *InterruptNumber* is invalid. • The *Handler* pointer is NULL. • The *Handler* address is not the same as the one that is installed
AE_NOT_EXIST	There is no handler installed for this interrupt level.

Functional Description:

Remove a previously installed hardware interrupt handler.

• *Stream I/O*

These interfaces provide formatted stream I/O. Used by the OSI mainly for debug output, these functions may be redirected to whatever output device or file is appropriate for the host operating system.

• OsdPrintf

Formatted stream output.

INT32
OsdPrintf (

OSD_FILE	***Stream,**
const char	***Format,**
…	**)**

PARAMETERS

Stream	Open stream to write to. NULL is defined to be the debugger output channel.
Format	A standard *printf* format string.
…	Variable parameter list.

RETURN VALUE

Count	Number of parameters successfully printed. –1 on error.

Functional Description:

This function provides formatted output to an open OSD stream.

- **OsdVprintf**

Formatted stream output.

INT32
OsdVprintf (

OSD_FILE	***Stream,**
const char	***Format,**
va_list	**Args)**

PARAMETERS

Stream	Open stream to write to. NULL is defined to be the debugger output channel.
Format	A standard *printf* format string.
Args	A variable parameter list.

RETURN VALUE

Count	Number of parameters successfully printed. –1 on error.

Functional Description:

This function provides formatted output to an open OSD stream via the va_list argument format.

• *Hardware Abstraction*

These interfaces allow the OS-dependent component to implement hardware I/O services in any manner that is acceptable to the host OS. The actual hardware I/O instructions may execute within the OS-dependent component itself, or these calls may be translated into additional OS calls — such as calls to a Hardware Abstraction Component.

- **OsdIn8**

Read 8 bits from an input port.

UINT8
OsdIn8 (

UINT16	**InPort)**

PARAMETERS

InPort	Hardware I/O port address to read from.

RETURN VALUE

Value	The data returned by the read operation.

Functional Description:

This function is used to read a BYTE (8 bits) from the specified input port.

- **OsdIn16**

Read 16 bits from an input port.

UINT16
OsdIn16 (
UINT16 **InPort)**

PARAMETERS

InPort Hardware I/O port address to read from.

RETURN VALUE

Value The data returned by the read operation.

Functional Description:

This function reads a WORD (16 bits) from the specified input port.

- **OsdIn32**

Read 32 bits from an input port.

UINT32
OsdIn32 (
UINT16 **InPort)**

PARAMETERS

InPort Hardware I/O port address to read from.

RETURN VALUE

Value The data returned by the read operation.

Functional Description:

This function reads a DWORD (32 bits) from the specified input port.

- **OsdOut8**

Write 8 bits to an output port.

void
OsdOut8 (
UINT16 **OutPort,**
UINT8 **Value)**

PARAMETERS

OutPort	Hardware I/O port address where data is to be written.
Value	Data to be written to the I/O port.

RETURN VALUE

None

Functional Description:

This function writes a BYTE (8 bits) to the specified output port.

- **OsdOut16**

Write 16 bits to an output port.

void
OsdOut16 (
UINT16 **OutPort,**
UINT16 **Value)**

PARAMETERS

OutPort	Hardware I/O port address where data is to be written.
Value	Data to be written to the I/O port.

RETURN VALUE

None

Functional Description:

This function writes a WORD (16 bits) to the specified output port.

- **OsdOut32**

Write 32 bits to an output port.

void
OsdOut32 (

UINT16	**OutPort,**
UINT32	**Value)**

PARAMETERS

OutPort	Hardware I/O port address where data is to be written.
Value	Data to be written to the I/O port.

RETURN VALUE

None

Functional Description:

This function writes a DWORD (32 bits) to the specified output port.

- **OsdReadPciCfgByte**

Read 8 bits from a PCI configuration register.

ACPI_STATUS
OsdReadPciCfgByte (

UINT32	**SegBus,**
UINT32	**DeviceFunction,**
UINT32	**Register,**
UINT8	***Value)**

PARAMETERS

SegBus	The PCI Segment number (Hiword) and Bus ID (Lowword).
DeviceFunction	The PCI device number (Hiword) and function number (Lowword).
Register	The PCI register address to be read from.
Value	A pointer to a location where the data is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

Functional Description:

This function reads a BYTE (8 bits) from the specified PCI configuration port.

- **OsdReadPciCfgWord**

Read 16 bits from a PCI configuration register.

ACPI_STATUS
OsdReadPciCfgWord (

UINT32	**SegBus,**
UINT32	**DeviceFunction,**
UINT32	**Register,**
UINT16	***Value)**

PARAMETERS

SegBus	The PCI Segment number (Hiword) and Bus ID (Lowword).
DeviceFunction	The PCI device number (Hiword) and function number (Lowword).
Register	The PCI register address to be read from.
Value	A pointer to a location where the data is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

Functional Description:

This function reads a WORD (16 bits) from the specified PCI configuration port.

- **OsdReadPciCfgDword**

Read 32 bits from a PCI configuration register.

ACPI_STATUS
OsdReadPciCfgDword (

UINT32	**SegBus,**
UINT32	**DeviceFunction,**
UINT32	**Register,**
UINT32	***Value)**

PARAMETERS

SegBus	The PCI Segment number (Hiword) and Bus ID (Lowword).
DeviceFunction	The PCI device number (Hiword) and function number (Lowword).
Register	The PCI register address to be read from.
Value	A pointer to a location where the data is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

Functional Description:

This function reads a DWORD (32 bits) from the specified PCI configuration port.

- **OsdWritePciCfgByte**

Write 8 bits to a PCI configuration register.

ACPI_STATUS
OsdWritePciCfgByte (

UINT32	**SegBus,**
UINT32	**DeviceFunction,**
UINT32	**Register,**
UINT8	**Value)**

PARAMETERS

SegBus	The PCI Segment number (Hiword) and Bus ID (Lowword).
DeviceFunction	The PCI device number (Hiword) and function number (Lowword).
Register	The PCI register address to be read from.
Value	Data to be written.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

Functional Description:

This function writes a BYTE (8 bits) to the specified PCI configuration port.

- **OsdWritePciCfgWord**

Write 16 bits to a PCI configuration register.

ACPI_STATUS
OsdWritePciCfgWord (

UINT32	**SegBus,**
UINT32	**DeviceFunction,**
UINT32	**Register,**
UINT16	**Value)**

PARAMETERS

SegBus	The PCI Segment number (Hiword) and Bus ID (Lowword).
DeviceFunction	The PCI device number (Hiword) and function number (Lowword).
Register	The PCI register address to be read from.
Value	Data to be written.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

Functional Description:

This function writes a WORD (16 bits) to the specified PCI configuration port.

- **OsdWritePciCfgDword**

Write 32 bits to a PCI configuration register.

ACPI_STATUS
OsdWritePciCfgDword (

UINT32	**SegBus,**
UINT32	**DeviceFunction,**
UINT32	**Register,**
UINT32	**Value)**

PARAMETERS

SegBus	The PCI Segment number (Hiword) and Bus ID (Lowword).
DeviceFunction	The PCI device number (Hiword) and function number (Lowword).
Register	The PCI register address to be read from.
Value	Data to be written.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

Functional Description:

This function writes a DWORD (32 bits) to the specified PCI configuration port.

User Guide

Using the ACPI OSI Interfaces

Initialization Sequence

In order to allow the most flexibility for the host operating system, there is no single interface that initializes the entire ACPI subsystem. Instead, the subsystem is initialized in stages, at the times that are appropriate for the host OS. The following example shows the sequence of initialization calls that must be made; it is up to the host interface (OS-dependent component) to make these calls when they are appropriate.

1. Initialize the OSI Component:

```
Status = AcpiInitialize ();
```

2. Load the ACPI tables from the firmware:

```
Status = AcpiLoadFirmwareTables ();
```

3. Build the internal namespace from the loaded ACPI tables:

```
Status = AcpiLoadNamespace ();
```

4. Put the system into ACPI mode:

```
Status = AcpiEnable ();
```

Shutdown Sequence

The OS-independent component does not absolutely require a shutdown before the system terminates. It does not hold any cached data that must be flushed before shutdown. However, if the ACPI subsystem is to be unloaded at any time during system operation, the subsystem should be shutdown so that resources that are held internally can be released back to the host OS. These resources include memory segments, an interrupt handler, and the ACPI hardware itself. To shutdown the OS-independent component, the following calls should be made:

1. Put the system back into legacy mode:

```
Status = AcpiDisable ();
```

2. Unload the namespace and free all resources:

```
Status = AcpiTerminate ();
```

Traversing the ACPI Namespace (Low Level)

This example demonstrates traversal of the APCI namespace using the low-level Acpi* primitives. The code is in fact the implementation of the higher-level *AcpiWalkNamespace* interface, and therefore this example has two purposes:

1. Demonstrate how the low-level namespace interfaces are used.
2. Provide an understanding of how the namespace walk interface works.

```
ACPI_STATUS
AcpiWalkNamespace (
    ACPI_OBJECT_TYPE        Type,
    ACPI_HANDLE             StartHandle,
    UINT32                  MaxDepth,
    WALK_CALLBACK           UserFunction,
    void                    *Context,
    void                    **ReturnValue)
{
    ACPI_HANDLE             ObjHandle = 0;
    ACPI_HANDLE             Scope;
    ACPI_HANDLE             NewScope;
    void                    *UserReturnVal;
    UINT32                  Level = 1;



/* Parameter validation */

    if ((Type > ACPI_TYPE_MAX) ||
        (!MaxDepth)             ||
        (!UserFunction))
    {
        return_ACPI_STATUS (AE_BAD_PARAMETER);
    }

    /* Special case for the namespace root object */

    if (StartObject == ACPI_ROOT_OBJECT)
    {
        StartObject = Gbl_RootObject;
    }


    /* Null child means "get first object" */

    ParentHandle    = StartObject;
    ChildHandle     = 0;
    ChildType       = ACPI_TYPE_Any;
    Level           = 1;

    /*
     * Traverse the tree of objects until we bubble back up to where we
     * started. When Level is zero, the loop is done because we have
     * bubbled up to (and passed) the original parent handle
(StartHandle)
     */

    while (Level > 0)
    {
```

```
        /* Get the next typed object in this scope.  Null returned if not
found */

        Status = AE_OK;
        if (ACPI_SUCCESS (AcpiGetNextObject (ACPI_TYPE_Any, ParentHandle,
                          ChildHandle, &ChildHandle)))
        {
            /* Found an object, Get the type if we are not searching for
ANY */

            if (Type != ACPI_TYPE_Any)
            {
                AcpiGetType (ChildHandle, &ChildType);
            }

            if (ChildType == Type)
            {
                /* Found a matching object, invoke the user callback
function */

                Status = UserFunction (ChildHandle, Level, Context,
ReturnValue);
                switch (Status)
                {
                case AE_OK:
                case AE_DEPTH:
                    break;                          /* Just keep going */

                case AE_TERMINATE:
                    return_ACPI_STATUS (AE_OK);     /* Exit now, with OK
status */
                    break;

                default:
                    return_ACPI_STATUS (Status);    /* All others are
valid exceptions */
                    break;
                }
            }

            /*
             * Depth first search:  Attempt to go down another
             * level in the namespace if we are allowed to. Don't go any
further if we
             * have reached the caller specified maximum depth or if the
user function
             * has specified that the maximum depth has been reached.
             */

            if ((Level < MaxDepth) && (Status != AE_DEPTH))
            {
                if (ACPI_SUCCESS (AcpiGetNextObject (ACPI_TYPE_Any,
ChildHandle,
                                                     0, NULL)))
                {
                    /* There is at least one child of this object, visit
the object */

                    Level++;
                    ParentHandle    = ChildHandle;
                    ChildHandle     = 0;
```

```
                }
            }
        }

        else
        {
            /*
             * No more children in this object (AcpiGetNextObject
failed),
             * go back upwards in the namespace tree to the object's
parent.
             */
            Level--;
            ChildHandle = ParentHandle;
            AcpiGetParent (ParentHandle, &ParentHandle);
        }
    }


    return_ACPI_STATUS (AE_OK); /* Complete walk, not terminated by user
function */
}
```

Traversing the ACPI Namespace (High Level)

This example demonstrates the use of the *AcpiWalkNamespace* interface and other **Acpi*** interfaces. It shows how to properly invoke *AcpiWalkNamespace* and write a callback routine.

This code searches for all device objects in the namespace under the system bus (where most, if not all devices usually reside.) The callback function always returns NULL, meaning that the walk is not terminated until the entire namespace under the system bus has been traversed.

Part 1: This is the top-level procedure that invokes *AcpiWalkNamespace*.

```
DisplaySystemDevices (void)
{
ACPI_HANDLE             SysBusHandle;


AcpiNameToHandle (0, NS_SYSTEM_BUS, &SysBusHandle);

printf ("Display of all devices in the namespace:\n");

AcpiWalkNamespace (ACPI_TYPE_Device, SysBusHandle, INT_MAX,
                       DisplayOneDevice, NULL, NULL);
}
```

Part 2: This is the callback routine that is repeatedly invoked from *AcpiWalkNamespace*.

```
void *
DisplayOneDevice (
    ACPI_HANDLE             ObjHandle,
    UINT32                  Level,
    void                    *Context)
{
    ACPI_STATUS             Status;
    ACPI_DEVICE_INFO        Info;
    ACPI_BUFFER             Path;
    Char                    Buffer[256];
```

```
    Path.Length = sizeof (Buffer);
    Path.Pointer = Buffer;

    /* Get the full path of this device and print it */

    Status = AcpiHandleToPathname (ObjHandle, &Path);
    if (ACPI_SUCCESS (Status))
    {
        printf ("%s\n", Path.Pointer));
    }

    /* Get the device info for this device and print it */

    Status = AcpiGetDeviceInfo (ObjHandle, &Info);
    if (ACPI_SUCCESS (Status))
    {
        printf ("    HID: %.8X, ADR: %.8X, Status: %x\n",
                Info.HardwareId, Info.Address, Info.CurrentStatus));
    }

    return NULL;
}
```

Implementing the OS-Dependent Component

Parameter Validation

In all implementations of the OS-dependent component, the interfaces should adhere to this EPS in the actual interface parameters as well as the returned exception codes. This means that the parameter validation is not optional and that the OS-independent layer depends on correct exception codes returned from the OSD.

Memory Management

Implementation of the memory allocation functions should be straightforward. If the host operating system has several kernel-level memory pools that can be used for allocation, it may be useful to know some of the dynamic memory requirements of the OS-independent component.

During initialization, the ACPI tables are copied into local memory segments. Some of these tables (especially the DSDT) can be fairly large, up to about 64K. The namespace is built from multiple small memory segments, each of a fixed (but configurable) length. The default namespace table length is 16 entries times about 32 bytes each for a total of 512 bytes per table and per allocation.

During operation, many internal objects are created and deleted while servicing requests. The size of an internal object is about 32 bytes, and this is the primary run-time memory request size.

The implementation of *OsdCallocate* must absolutely ensure that the memory segment that is returned is initialized to all zeros. Much of the OS-independent code depends on this initialization and will cease to function if this function is not correctly implemented.

Interrupt Handling

In order to support the OS-independent interrupt handler that is implemented in the OSI, the OSD must provide a local interrupt handler whose interface conforms to the requirements of the host operating system. This local interrupt handler is a wrapper for the OS-independent handler; it is the actual handler that is installed for the given interrupt level. The task of this wrapper is to handle incoming interrupts and dispatch them to the OS-independent handler via the OS-independent handler interface. When the OSI handler returns, the wrapper performs any necessary cleanup and exits the interrupt.

Stream I/O

The *OsdPrintf* and *OsdVprintf* functions can usually be implemented using a kernel-level debug print facility. Kernel printf functions usually output data to a serial port or some other special debug facility. If there is more than one type of debug print routine, use one that can be called from within an interrupt handler so that Fixed Events and General Purpose events can be traced.

Hardware Abstraction

The intent of the hardware I/O interfaces is to allow these calls to be translated into calls or macros provided by the host OS for this purpose. However, if the host does not provide a hardware abstraction service, these functions can be implemented simply and directly via I/O machine instructions.

References

A number of the references are Intel Confidential. These references are available through an Intel-approved source or an Intel representative.

ACPI1 — *Advanced Configuration and Power Interface Specification, Revision 1.0b* http://www.teleport.com/~acpi/.

ACPI2 — *PCI Hot-plug and ACPI Guidelines, Revision 0.1*. This document can be obtained from Intel Corporation.

ACPI CA — *Unix Developer's Interface Guide for IA-64 Servers, Revision 1.0,* Appendix A: ACPI Component Architecture: ACPI Subsystem.

DIG64 — *Developer's Interface Guide for IA-64 Servers, Revision 1.0,* http://dig64.org/.

EFI1 — *Extensible Firmware Interface Specification, Revision 0.91*. http://developer.intel.com/technology/efi.

ELF1 — *ELF-64 Object File Format Specification.* http://www.sco.com/developer/gabi/contents.html.

ELF2 — *Processor-Specific ELF Supplement for IA-64*. This document can be obtained from Intel Corporation.

SAB1 — *System V Application Binary Interface Specification* http://www.sco.com/developer/devspecs/.

SAB2 — *System V Application Binary Interface Intel 386 Architecture Processor Supplement Specification* http://www.sco.com/developer/devspecs/.

SAB3 — *System V ABI Updates* http://www.sco.com/developer/devspecs/.

SRC1 — *IA-64 Software Conventions and Runtime Architecture Guide* http://developer.intel.com/design/ia64/devinfo.htm.

UDI1 — *UDI Specifications, Revision 1.0* http://www.project-udi.org.

UDI2 *UDI Core Specification, Revision 1.0* http://www.project-udi.org.

UDI3 *UDI PCI Bus Binding Specification Revision 1.0* http://www.project-udi.org.

UDI4 *UDI SCSI Driver Specification, Revision 1.0* http://www.project-udi.org.

UDI5 *UDI Network Driver Specification, Revision 1.0* http://www.project-udi.org.

UDI6 *UDI Physical I/O Specification Revision 1.0* http://www.project-udi.org.

USB1 *Open USBDI Specification* http://www.usb.org/.

Glossary

ACPI (Advanced Configuration and Power Interface)

A specification that defines a new interface to the system board that enables the operating system to implement operating system-directed power management and system configuration.

ACPI Subsystem

The combination of the operating system independent and operating system dependent components which together comprise the ACPI "driver" or subsystem.

AML

ACPI Machine Language. Pseudo code, or byte code, capable of being executed by a virtual machine or interpreter.

AML Interpreter

AML byte code, created by the ASL compiler, is stored in a hierarchical database called the ACPI name space. The AML interpreter executes AML code. The interpreter is the biggest sub-component of the ACPI subsystem. It parses the ACPI tables to create the name space, and executes control methods stored in the name space during run-time.

BIOS

Basic Input Output System (System firmware component).

Boot

The period of time from the end of POST to the OS disconnect from EFI Boot services.

Control Method

ASL control routines used to manipulate the ACPI platform. These routines are capable of performing I/O, and logical and arithmetic operations.

Device Manager

ACPI sub-component containing interfaces for managing ACPI compatible devices.

Event Manager

An event manager is an OSI sub-component containing functionality for managing the ACPI event mechanisms.

IHV

Independent Hardware Vendor.

ISA

Industry Standard Architecture (Expansion bus on PC XT/AT machines).

LUN

Logical Unit Number (SCSI device address component).

MPS

Multi-Processor Specification.

OS

Operating system.

OSV

Operating system vendor.

PnP

Plug and Play.

POST

Power-On Self-Test. This code is executed at power-on prior to launching EFI.

ROM

Read Only Memory.

UDIG-compliant system

When a system adheres to all required guidelines, the system is referred to as a UDIG-compliant system.

Index