# ACPI Component Architecture Programmer Reference

## Core Subsystem, Debugger, and Utilities

**Revision 1.13**

***May 3, 2002***

# Contents

## Figures

## Tables

# 1 Introduction

## 1.1 Document Structure

This document consists of nine major sections:

1. Introduction: Contains a brief overview of the ACPI Component Architecture (CA) and the interfaces for both the Core Subsystem and OS Services Layers.
2. Design Overview: Summary of the computational and architectural model that is implemented by the ACPI component architecture.
3. Design and Implementation Details: Details concerning design decisions and implementation specifics.
4. Interface Parameters and Data Types: Descriptions of the major data types and data structures that are exposed via the external interfaces. Other related information required to use the ACPI subsystems and interfaces.
5. ACPI CA Core Subsystem Interfaces: Detailed description of the programmatic interfaces that are implemented by the core component of the ACPI Component Architecture.
6. OS Services Layer Interfaces: Detailed description of the programmatic interfaces that must be implemented by OSVs in the layer that interfaces the ACPI CA Core Subsystem to the host operating system.
7. ACPI Debugger: Overview, installation and configuration, and detailed descriptions of the command set
8. Tools and Utilities: A brief overview of the miscellaneous tools and utilities that are part of the Core Subsystem package.
9. Subsystem User Guide: Tips and techniques on how to use the Core Subsystem interfaces, and how to implement the OSL interfaces to host a new operating system.

## 1.2 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 ACPI CA software can be hosted on any operating system by writing a small and relatively simple translation service between the ACPI subsystem and the host operating system (This service is known as the OS Services Layer).

## 1.3 Reference Documents

- *Advanced Configuration and Power Interface Specification*, Revision 1.0b, February 8, 1999
- *Advanced Configuration and Power Interface Specification*, Revision 2.0, July 27, 2000
- *Advanced Configuration and Power Interface Specification*, Revision 2.0a, March 32, 2002

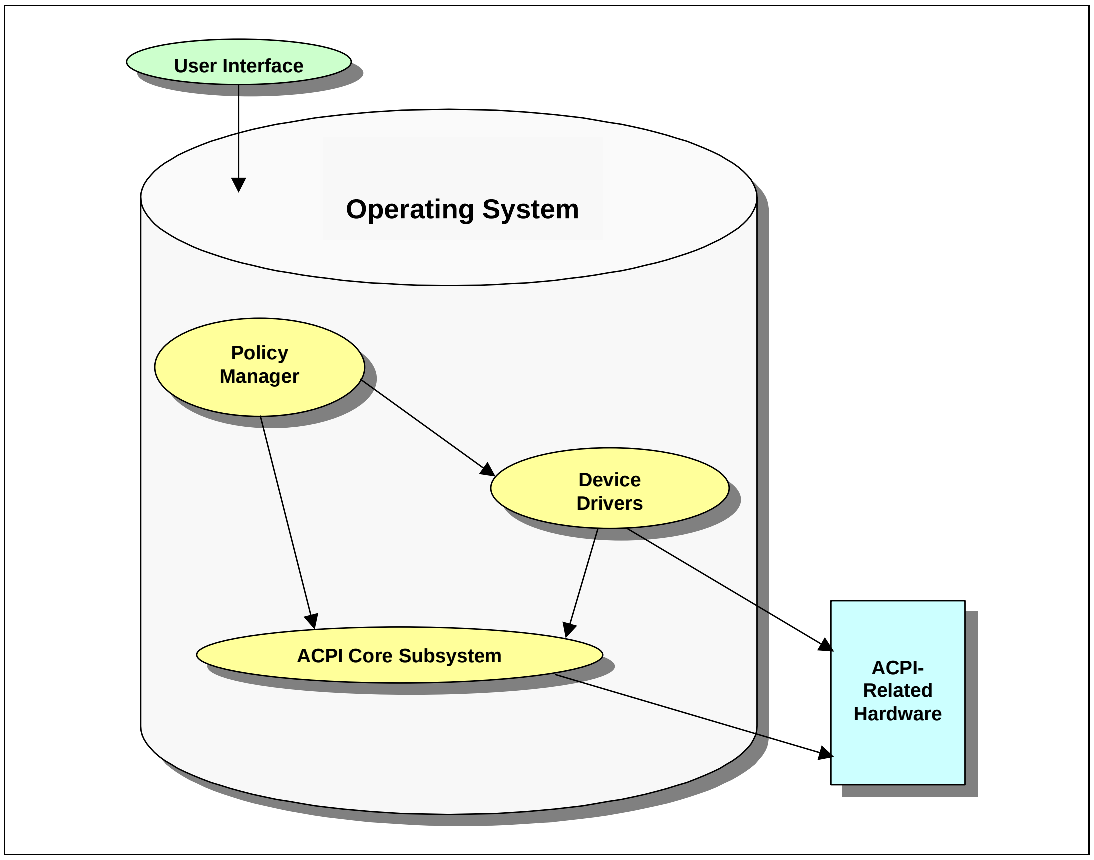
## 1.4 Overview of the ACPI Component Architecture

The ACPI Component Architecture (also referred to by the term “ACPI CA” in this document) defines and implements a group of software components that together create an implementation of the ACPI specification. A major goal of the architecture is to isolate all operating system dependencies to a relatively small translation or conversion layer (the OS Services Layer) so that the bulk of the ACPI CA code is independent of any individual operating system. Therefore, hosting the ACPI CA code on new operating systems requires no source within the CA code itself. The components of the architecture include (from the “top” down):

- 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 Core Subsystem component that provides the fundamental ACPI services (such as the AML interpreter and namespace management).
- An OS Services Layer for each host operating system.

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**



## 1.5 Overview of the ACPI Core Subsystem

The ACPI Subsystem implements the low level or fundamental aspects of the ACPI specification. Included are an AML parser/interpreter, ACPI namespace management, ACPI table and device support, and event handling. Since the ACPI CA core provides low-level system services, it also requires low-level operating system services such as memory management, synchronization, scheduling, and I/O. To allow the Core Subsystem to easily interface to any operating system that provides such services, an *Operating System Services Layer* translates OS requests into the system calls provided by the host operating system. The OS Services Layer is the only component of the ACPI CA that contains code that is specific to a host operating system. Thus, the ACPI Subsystem consists of two major software components:

1. The Acpi Core Subsystem provides the fundamental ACPI services that are independent of any particular operating system.
2. The OS Services Layer (OSL) provides the conversion layer that interfaces the ACPI Core Subsystem to a particular host operating system.

When combined into a single static or 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 ACPI Core Subsystem with the OS Services Layer components into a single module, driver, or load unit.

### 1.5.1 ACPI Core Subsystem

The ACPI Core Subsystem supplies the major building blocks or subcomponents that are required for all ACPI implementations — including an AML interpreter, a namespace manager, ACPI event and resource management, and ACPI hardware support.

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

The Core Subsystem 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 Services Layer*.

The primary user of the services provided by the ACPI Core Subsystem is the OS Services Layer, since it is the OS Services Layer that provides an external interface appropriate for the host operating system. For example, the ACPI subsystem may be packaged as a device driver and the OSL then provides the external OS-defined device driver interfaces that the rest of the OS uses to communicate to the ACPI subsystem.

### 1.5.2 Operating System Services Layer

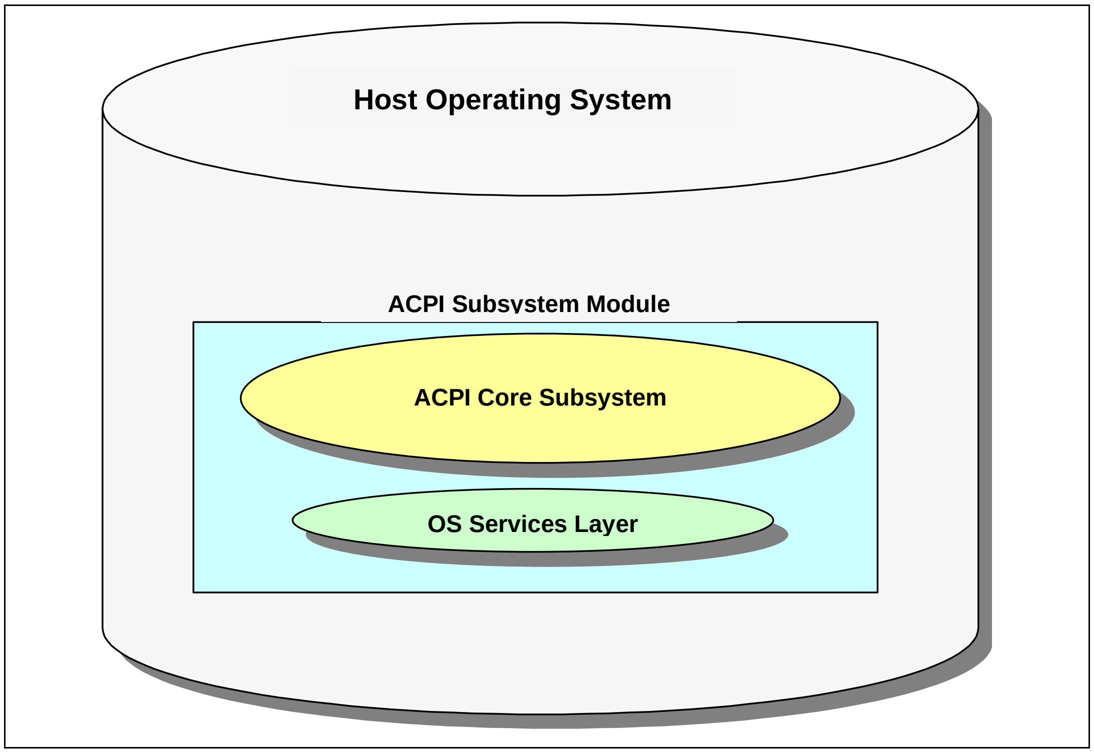
The OS Services Layer (or **OSL**) operates as a bi-directional translation service for both requests from the host OS to the ACPI subsystem, and from the ACPI subsystem to the host OS. These two functions are independent of each other in many ways. In one direction, the OSL translates host OS requests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

Because of its nature, the OS Services Layer must be implemented anew for each supported host operating system. There is a single ACPI Core Subsystem, but there must be an OS Services Layer for each operating system supported by the ACPI component architecture.

The primary function of the OSL in the ACPI Component Architecture is to be the small glue layer that binds the much larger Core Subsystem 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 Core Subsystem 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



## 1.5.3 Relationships between the Host OS, Core Subsystem, and OSL

### 1.5.3.1 Host Operating System Interaction

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

The OSL component "calls up" to the host operating system whenever operating system services are required, either for the OSL itself, or on behalf of the Core Subsystem component. All native calls directly to the host are confined to the OS Services Layer, for obvious reasons.

#### 1.5.3.2 OS Services Layer Interaction

The OS Services Layer 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 OSL-host OS interface receives ACPI requests from the operating system and translates them into one or more requests to the Core Subsystem component. Therefore, the OSL calls the Core Subsystem to implement the host OS interface.

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

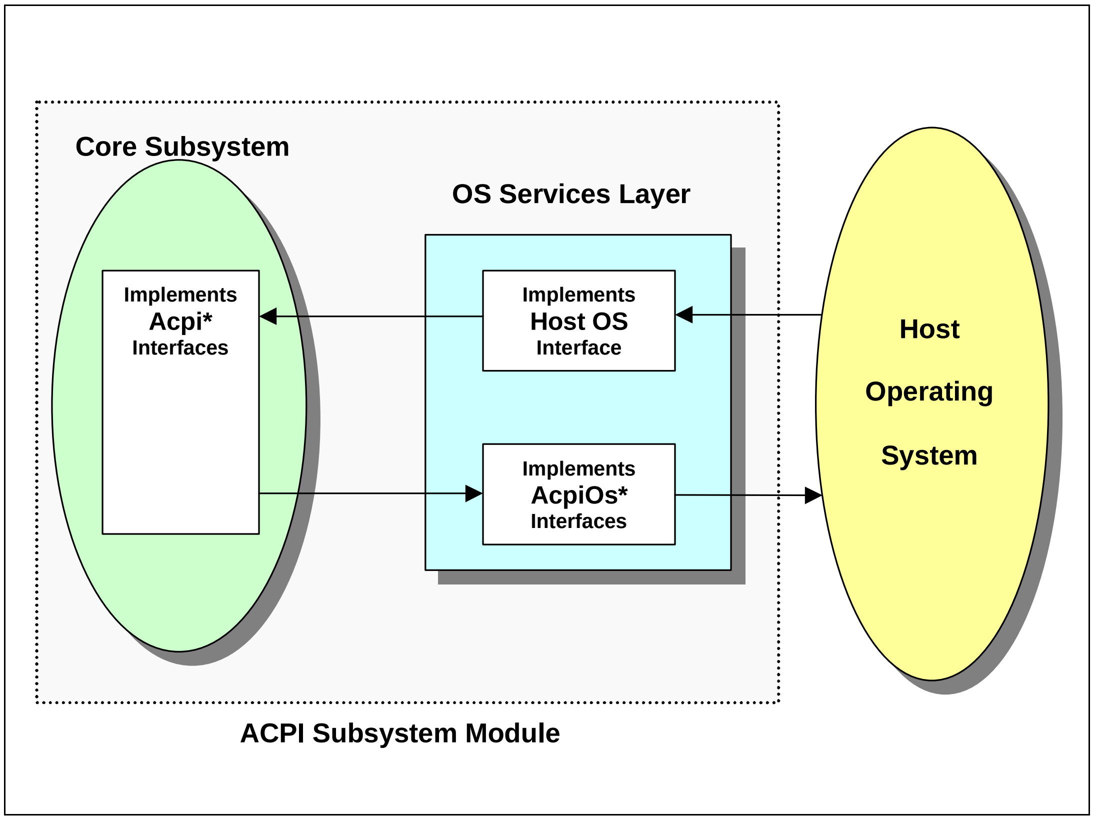
#### 1.5.3.3 Acpi Core Subsystem Interaction

The *Acpi Core Subsystem* 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 Core Subsystem makes requests to the host OS indirectly via the fixed **AcpiOs*** 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 Core Subsystem directly — it accesses services that are provided by the OSL. Also, the Core Subsystem never calls the host directly — instead it makes calls to the **AcpiOs *** interfaces in the OSL. It is this level of indirection in both directions that allows the Core Subsystem to be truly operating system independent.

Figure 3. Interaction between the Architectural Components


Core Subsystem
OS Services Layer
Implements Acpi* Interfaces
Implements Host OS Interface
Implements AcpiOs* Interfaces
Host Operating System
ACPI Subsystem Module

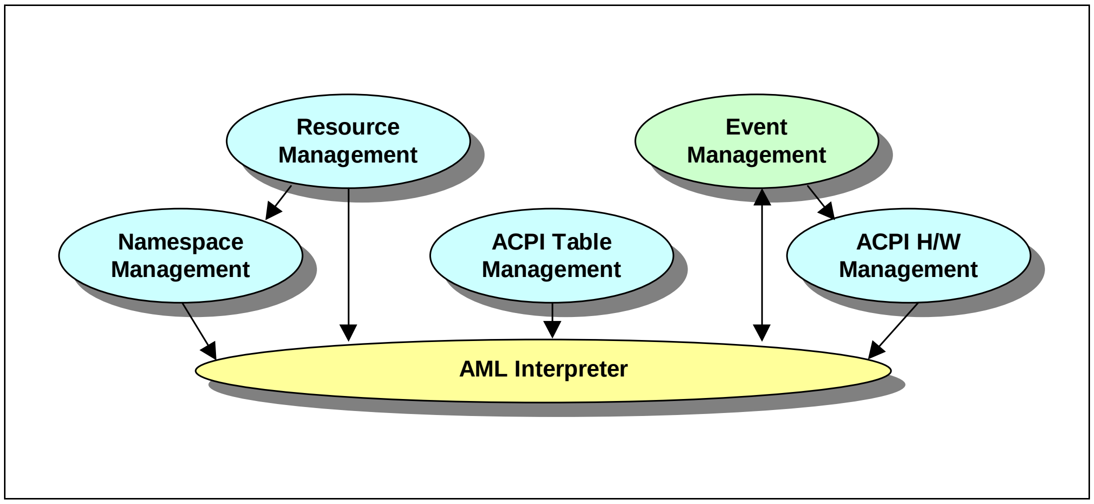
## 1.6 Architecture of the ACPI Core Subsystem

The Core Subsystem 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 ACPI Core Subsystem 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 ACPI Core Subsystem



## 1.6.1 AML Interpreter

The AML interpreter is responsible for the parsing and execution of the AML byte 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

## 1.6.2 ACPI Table Management

This component manages the ACPI tables such as the RSDT, FADT, 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

## 1.6.3 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

### 1.6.4 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

### 1.6.5 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.

### 1.6.6 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)

## 1.7 Architecture of the OS Services Layer (OSL)

The OS Services Layer component of the architecture enables the rehosting or retargeting of the other components to execute under different operating systems, or to even execute in environments where there is no host operating system. In other words, the OSL component provides the glue that joins the other components to a particular operating system and/or environment. The OSL implements interfaces and services using the system calls and utilities that are available from the host OS. Therefore, an OS Services Layer must be written for each target operating system.

The OS Services Layer has several roles.

1. 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 Core Subsystem interfaces.
2. It exposes a set of OS-specific application interfaces. These interfaces translate application requests to calls to the ACPI interfaces
3. The OSL component implements a standard set of interfaces that perform OS dependent functions (such as memory allocation and hardware access) on behalf of the Core Subsystem component. These interfaces are themselves *OS-independent* because they are constant across all OSL implementations. It is the *implementations* of these interfaces that are OS-dependent, because they must use the native services and interfaces of the host operating system.

### 1.7.1 Functional Service Groups

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

The OS Services Layer 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 OSL and ACPI Core Subsystems 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 Services Layer may take the form of a more conventional system call interface, or even simply a local procedure call interface.

Although it is the OS Services Layer that exposes these services to the rest of the operating system, it is very important to note that the OS Services Layer makes use of the services of the lower-level ACPI Core Subsystem to implement its services. It is the intent of the component architecture that the Core Subsystem is a service that is private to the OSL — that is, that only the OSL makes calls to the Core Subsystem.

#### 1.7.1.1 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.

#### 1.7.1.2 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.

#### 1.7.1.3 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.

#### 1.7.1.4 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.

### 1.7.2 Required Functionality

There are three basic functions of the OS Services Layer:

1. Manage the initialization of the entire ACPI subsystem, including both the OSL and ACPI Core Subsystems.
2. Translate requests for ACPI services from the host operating system (and its applications) into calls to the Core Subsystem 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 ACPI Core Subsystem.
3. Implement an interface layer that the Core Subsystem component uses to obtain operating system services. These standard interfaces (defined in this document as the **AcpiOs*** 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 Services Layer 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.

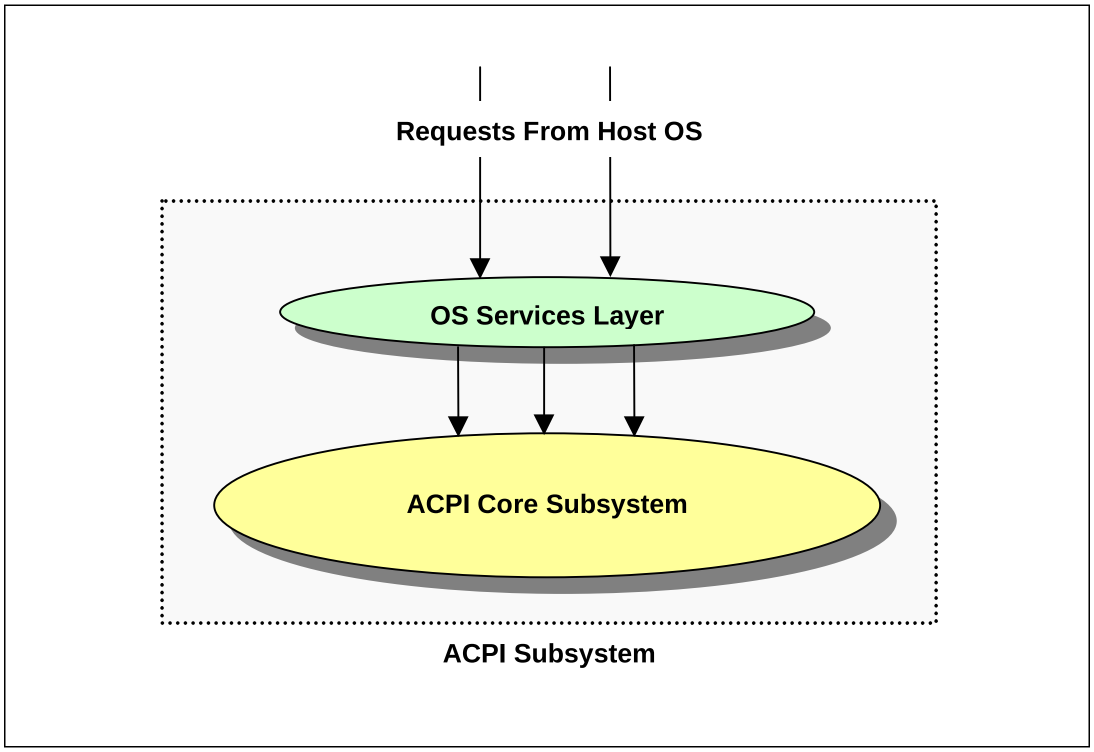
#### 1.7.2.1 Requests from the Operating System to the ACPI Subsystem

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 Services Layer is to translate native operating system ACPI requests into calls to the ACPI Core Subsystem. For example, the OS Services Layer 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**



### 1.7.2.2 Requests from Applications to the ACPI Subsystem

Application level interfaces should be provided in the OS Services Layer 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.

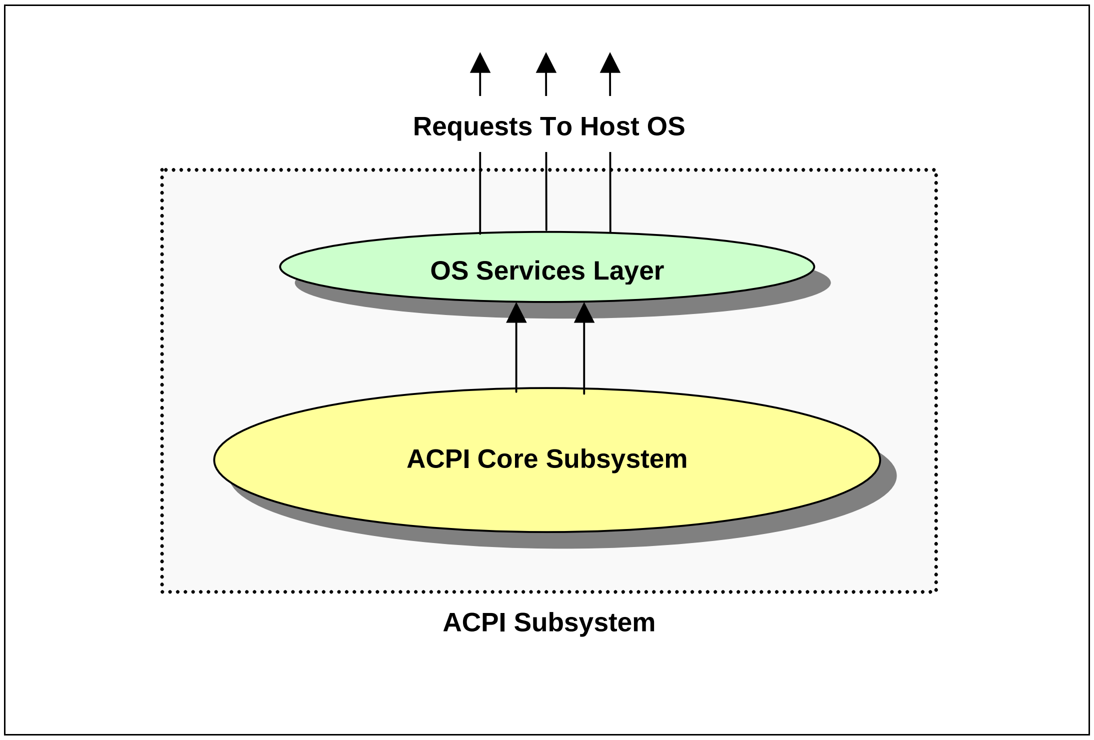
The type and style of these application interfaces is completely dependent on the architecture of the host operating system and where the ACPI subsystem fits into that architecture. The interfaces may be device driver style interfaces, or system calls into an operating system layer.

### 1.7.2.3 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 Services Layer 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**



# 2 Design Overview

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

## 2.1 ACPI Namespace Fundamentals

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

### 2.1.1 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.

### 2.1.2 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*.

## 2.1.3 Predefined Objects

During initialization of the internal namespace within Core Subsystem 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.

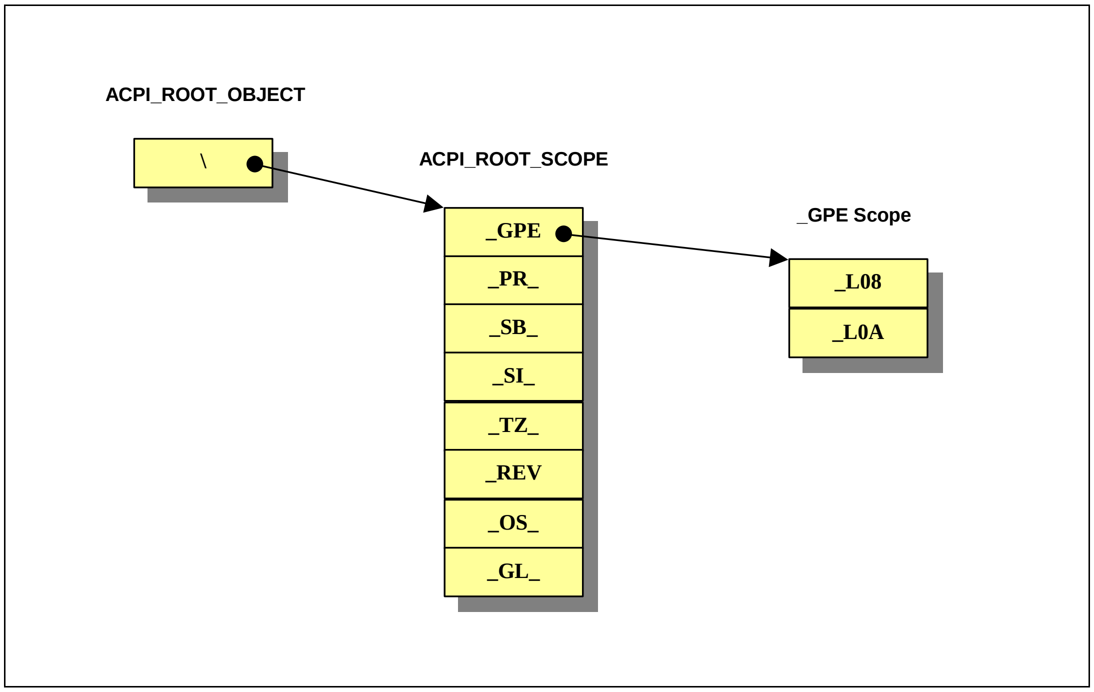
```
"_GPE",    ACPI_TYPE_Any    // General Purpose Event block
"_PR_",    ACPI_TYPE_Any    // Processor block
"_SB_",    ACPI_TYPE_Any    // System Bus block
```

```
"_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
```

### 2.1.4 Logical Namespace Layout

The diagram below shows the logical namespace after the predefined objects and the _GPE scope has been entered.

Figure 7. Internal Namespace Structure



## 2.2 Execution Model

### 2.2.1 Initialization

The initialization of the ACPI Core Subsystem must be driven entirely by the OS Services Layer. Since it may be appropriate (depending on the requirements of the host OS) to initialize different parts of the Core Subsystem at different times, the Core Subsystem initialization is a multi-step process that must be coordinated by the OSL. The four main steps are outlined below.

1. Perform a global initialization of the Core Subsystem – this initializes the global data and other items within the Core Subsystem.
2. Load the ACPI tables – The FACS, FADT,DSDT, etc. must be copied (or mapped) into the Core Subsystem before the internal namespace can be constructed. The tables may be loaded from the firmware, loaded from an input buffer, or some combination of both. The minimum set of ACPI tables includes an FACS, an FADT, and a DSDT.

3. Build the internal namespace – this causes the Core Subsystem 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 Core Subsystem installs an interrupt handler for the System Control Interrupts (SCIs), and transitions the hardware from legacy mode to ACPI mode.

### 2.2.2 Memory Allocation

There are two models of memory allocation that can be used. In the first model, the caller to the ACPI subsystem pre-allocates 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. In the second model, the caller can choose to have the ACPI subsystem allocate memory via the AcpiOsAllocate interface. Although this model is less flexible, it is far easier to use and is sufficient for most environments.

Each memory allocation model is described below.

#### 2.2.2.1 Caller Allocates All Buffers

In this model, the caller preallocates buffers of a large enough size and posts them to the ACPI subsystem via the ACPI_BUFFER data type.

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 4.2.6 - “ACPI_BUFFER – Input and Output Memory Buffers” for additional discussion on using the ACPI_BUFFER data type.

#### 2.2.2.2 ACPI Allocates Return Buffers

In this model, the caller lets the ACPI subsystem allocate return buffers. It is the responsibility of the caller to delete these returned buffers.

For the ACPI interfaces that use the ACPI_BUFFER data type as an output parameter, the following protocol is used to allow the ACPI subsystem to allocate return buffers:

1. Set the buffer length field of the ACPI_BUFFER structure ACPI_ALLOCATE_BUFFER.
2. Call the Acpi interface.
3. If the return exception code is AE_OK, the interface completed successfully and a buffer was allocated. The length of the buffer is contained in the ACPI_BUFFER structure.
4. Delete the buffer by calling AcpiOsFree with the pointer contained in the ACPI_BUFFER structure..

### 2.2.3 Parameter Validation

Only limited parameter validation is performed on all input parameters passed to the ACPI Core Subsystem. All calls to the Core Subsystem code should come from the OSL portion, not directly from user or application code. Therefore, the OSL 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 Core Subsystem 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 OSL component.

### 2.2.4 Exception Handling

All exceptions that occur during the processing of a request to the ACPI Core Subsystem 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 Core Subsystem interfaces. There are no exception handlers associated with either the **Acpi*** or **AcpiOs*** calls.

### 2.2.5 Multitasking and Reentrancy

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 OSL 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 ACPI Core Subsystem 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.

### 2.2.6 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.

#### 2.2.6.1 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.

#### 2.2.6.2 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 **T** is the type of GPE — either **E** for edge-triggered or **L** for level triggered. **xx** is the GPE level in hexadecimal (See the ACPI specification for complete details.) This control method is never 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.

#### 2.2.6.3 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.

### 2.2.7 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 boundaries (window size) and the address space to be accessed by the OpRegion. 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 fields' 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.

#### 2.2.7.1 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 devices 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.

#### 2.2.7.2 ACPI-Defined Address Spaces

The ACPI 2.0a specification defines address spaces for:

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

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.

## 2.3 Policies and Philosophies

This section provides insight into the policies and philosophies that were used during the design and implementation of the ACPI CA Core Subsystem. Many of these policies are a direct interpretation of the ACPI specification. Others are a direct or indirect result of policies and procedures dictated by the ACPI specification. Still others are simply standards that have been agreed upon during the design of the subsystem.

### 2.3.1 External Interfaces

#### 2.3.1.1 Exception Codes

All external interfaces (Acpi*) return an exception code as the function return. Any other return values are returned via pointer(s) passed as parameters. This provides a consistent and simple synchronous exception-handling model.

Since the ACPI CA Core Subsystem is reentrant and supports multiple threads on multiple operating systems, a model where an exception code is stored in the task descriptor (such as the *errno* mechanism) was purposefully avoided to improve portability.

#### 2.3.1.2 Memory Buffers

Memory for return objects, buffers, etc. that is returned via the external interfaces is rarely allocated by the subsystem itself. The model chosen is to force the caller to always pre-allocate memory. This forces the calling software to manage both the creation and deletion of its own buffers — hopefully minimizing memory fragmentation and avoiding memory leaks.  The exception to this is the ACPI_BUFFER type, where the caller can direct the ACPI subsystem to allocate return buffers.

### 2.3.2 Subsystem Initialization

#### 2.3.2.1 ACPI Table Validation

All ACPI tables that are examined by the ACPI core subsystem undergo some minimal validation before they are accepted. This includes all tables found in the RSDT regardless of whether the signature is recognized, and all tables loaded from user buffers. The following validations are performed on each table. A warning is issued for tables that do not pass one or more of these tests:

1. The Table pointer must point to valid physical memory
2. The signature (in the table header) must be 4 ASCII chars, *even if the name is not recognized.*
3. The table must be readable for length specified in the header
4. The table checksum must be valid (with the exception of the FACS, which has no checksum).

Other than this validation, tables that are not recognized by their table header signature are simply ignored.

#### 2.3.2.2 Required ACPI Tables

At the very minimum, the ACPI CA Core Subsystem requires the following ACPI tables:

1. One *Fixed ACPI Description Table* (FADT — signature “FACP”). This table contains important configuration information about the ACPI hardware
2. One *Firmware ACPI Control Structure* (FACS). This table contains the OS-to-firmware interface including the firmware waking vector and the Global Lock.
3. One *Differentiated System Description Table* (DSDT). This table contains the primary AML code for the system.

# 3 Design and Implementation Details

## 3.1 Required Host OS Initialization Sequence

This section describes a generic operating system initialization sequence that includes the ACPI CA subsystem. The ACPI CA subsystem must be loaded very early in the kernel initialization. In fact, ACPI support must be considered to be one of the fundamental startup modules of the kernel. The basic OS requirements of the ACPI subsystem include memory management, synchronization primitives, and interrupt support. As soon as these services are available, ACPI CA should be initialized. Only after ACPI is available can motherboard device enumeration and configuration begin.

In summary, ACPI Tables are descriptions of the hardware, therefore must be loaded into the OS very early.

### 3.1.1 Bootload and Low Level Kernel Initialization

- OS is loaded into memory via bootloader or downloader
- Initialize OS data structures, objects and run-time environment
- Initialize low-level kernel subsystems
- Initialize and enable free space manager
- Initialize and enable synchronization primitives

- Initialize basic interrupt mechanism and hardware
- Initialize and start system timer

### 3.1.2 ACPI CA Subsystem Initialization

- Load ACPI Tables
- Initialize Namespace
- Initialize ACPI Hardware and install SCI interrupt handler
- Initialize ACPI Address Spaces (Default handlers and user handlers)
- Initialize ACPI Objects (_STA, _INI, _HID)
- Find PCI Root Bus(es) and install PCI config space handlers

### 3.1.3 Other OS Initialization

- Remaining non-ACPI Kernel initialization
- Initialize and start System Resource Manager
- Determine processor configuration

### 3.1.4 Device Enumeration, Configuration, and Initialization

- Match motherboard devices to drivers via _HID
- Initialize PCI subsystem: Obtain _PRT interrupt routing table and Initialize PCI routing. PCI driver enumerates PCI bus and loads appropriate drivers.
- Initialize Embedded Controller support/driver
- Initialize SM Bus support/driver
- Load and initialize drivers for any other motherboard devices

### 3.1.5 Final OS Initialization

- Load and initialize any remaining device drivers
- Initialize upper layers of the OS
- Activate user interface

## 3.2 Required ACPI CA Initialization Sequence

This section presents a detailed description of the initialization process for the ACPI CA subsystem. The initialization interfaces are provided at a sufficient granularity to allow customization of the initialization sequence for each host operating system and host environment.

### 3.2.1 ACPI CA Subsystem Initialization

#### 3.2.1.1 AcpiInitializeSubsystem

This mandatory step must be first. It initializes the ACPI CA Subsystem software, including all global variables, tables, and data structures. All elements of the ACPI CA Subsystem are initialized, including the OSL interface layer and the OSPM layer. The interface provided is *AcpiInitializeSubsystem.*

### 3.2.2 ACPI Table and Namespace Initialization

This required phase loads the ACPI tables from the BIOS or elsewhere and initializes the internal ACPI namespace.

#### 3.2.2.1 AcpiLoadFirmwareTables

This interface finds and loads all ACPI tables that are presented to the system by the resident firmware. This is the normal interface used to obtain the ACPI tables on an ACPI-supported platform.

#### 3.2.2.2 AcpiLoadTable

This interface is used to directly load ACPI tables from somewhere (anywhere) other than the BIOS. The table is transferred to the ACPI subsystem via a memory buffer. The *AcpiExec* utility uses this interface to load ACPI tables from a file.

#### 3.2.2.3 Internal ACPI Namespace Initialization

As the various ACPI tables are loaded (installed into the internal data structures of the CA subsystem), the internal ACPI Namespace (database of named ACPI objects) is constructed from those tables. As each table is loaded, the following tasks are automatically performed:

- First pass parse – Load all named ACPI objects into the internal namespace
- Second pass parse – Resolve all forward references within the ACPI table
- First pass parse of all control methods – Sanity check to ensure that the tables can be completely parsed, including the control methods. The resulting parse tree is not stored, since control methods are parsed on the fly every time they are executed. (This task represents minimal CPU overhead, and saves huge amounts of memory that would be consumed by storing parse trees.)
- Lock the namespace so that GPEs will not cause control methods to run

### 3.2.3 Handler Installation

Once the namespace has been constructed, the OS should install any handlers that it may require during execution of the ACPI CA subsystem. The purpose of installing these handlers at this point in the initialization process is so that the handlers are in place before execution of any control methods is allowed – thereby insuring that any custom handlers will not miss any of the events that they are intended to handle. Any handlers installed in this phase will override the default handlers.

#### 3.2.3.1 AcpiInstallAddressSpaceHandler

This function is used to install address space handlers to override the default address space handlers (for the predefined address spaces) or install handlers for custom address spaces.

#### 3.2.3.2 AcpiInstallFixedEventHandler

This function is used to install handlers for ACPI Fixed Events and General Purpose Events (GPEs).

#### 3.2.3.3 AcpiInstallGpeHandler

This function is used to install handlers for ACPI General Purpose Events (GPEs).

#### 3.2.3.4 AcpiInstallNotifyHandler

This function is used to install handlers for ACPI device notifications.

### 3.2.4 Subsystem Initialization Completion

#### 3.2.4.1 AcpiEnableSubsystem

This single interface performs the functions described in the sections below. To summarize the actions performed by this call:

- Initialize ACPI hardware and ACPI events
- Enter ACPI Mode
- Initialize ACPI device objects
- Install handlers for the PCI Root Bridge(s)
- Initialize all Operation Regions (Address Spaces) and Fields

#### 3.2.4.2 ACPI Hardware and Event Initialization

This step sets up the ACPI hardware, initializes the ACPI Event handling, and puts the system into ACPI mode if necessary. This step is optional when running in "hardware-independent" mode – when there is no access to hardware by the ACPI subsystem (For example, the *AcpiDump* and *AcpiExec* utilities run in this mode.)

The ACPI hardware must be initialized and an SCI interrupt handler must be installed before it is architecturally safe to evaluate ACPI objects and execute control methods, for the following reasons:

1. *Any* ACPI named object (predefined or otherwise) can be implemented as a control method and there is no way to safely make any assumptions about which objects are and are not implemented as control methods. This is dependent on the individual AML on each platform.
2. Because control methods can access the ACPI hardware, cause SCIs, and most interesting of all, *can block while waiting for an SCI to be serviced*, it is inherently unsafe and architecturally incorrect to attempt to execute control methods without first initializing the hardware and installing an SCI interrupt handler

This step is only optional when running in “hardware-independent” mode. Otherwise it is required to setup the ACPI hardware and System Control Interrupt handling. ACPI mode is entered if the machine is in legacy mode. IF the machine is already in ACPI mode (such as an IA-64 machine), no action is required.

- Initialize the ACPI hardware
- Initialize the SCI, GPE, and FixedEvent handling
- Enter ACPI mode

After this step, control methods can be executed because the hardware is now initialized and the subsystem is able to take ACPI-related interrupts (*System Control Interrupts or SCIs*). The execution of any control method (including the **_REG** methods) can cause the generation of an SCI – therefore, the hardware must be initialized before control methods may be run. Additional ACPI subsystem initialization that requires control method execution can now be completed.

#### 3.2.4.3 Just-in-time Address Space Initialization

This phase includes just-in-time initialization for any Operation Regions or Fields (*and some new types in ACPI 2.0, TBD)* that are accessed by the control methods executed here. For example, if a **_REG** method for a PCIConfig address space accesses a SystemMemory Operation Region, the definition of that particular SystemMemory region is fully evaluated at that time. (Operation Regions and CreateField ASL statements can contain executable AML code and therefore the initialization of the objects must be deferred until the CA subsystem and ACPI hardware are both initialized).

Therefore, Address Spaces are initialized *in the order in which they are accessed*, not in the order that they are declared in the ASL source code.

When any Address Space is initialized, the associated **_REG** method (if any) is executed as well.

#### 3.2.4.4 ACPI Device Initialization

This step initializes device objects found within the ACPI namespace. The PCI configuration space handlers are setup in this phase. **Note:** The initialization of the device objects entails running the **_INI** method on all devices that are present as indicated by the **_STA** method. This is *not* an actual initialization of the device hardware – this is left to the actual device drivers for the hardware.

The **_STA**, **_INI**, and **_HID** methods are run on all ACPI objects of type *Device* found within the namespace (that are ready and available.)

Traverse the entire namespace and run these methods on each and every device found within: **_STA**, **_INI**, **_HID** (in this order.) Any operation regions accessed by these methods will be automatically initialized by the just-in-time address space initialization mechanism.

If the **_HID** method indicates the presence of a PCI Root Bridge (if it returns an HID value of PNP0A03), perform PCI Configuration Space initialization on the bridge. Install the PCI address space handler on the bridge (and on all descendents) and run the **_ADR**, **_SEG**, and **_BBN** methods to obtain the PCI device, function and bus numbers. Then run the associated **_REG** method to indicate the availability of the region.

Note that this sequence of events (run the **_STA**, **_INI**, and **_HID** methods on all devices) is the correct (and the only proper) method to detect the presence of the PCI root bridge or bridges.

#### 3.2.4.5 Other ACPI Object Initializatoin

This step initializes the remaining AML Operation Regions and Fields that were not initialized during the device and address space initialization.

Operation Regions and CreateField ASL statements can contain executable AML code and therefore the initialization of the objects must be deferred until the CA subsystem and ACPI hardware are both initialized. Some of this initialization may have been completed during the earlier steps. This step completes that initialization.

This final pass through the loaded ACPI tables will execute all AML code outside of the control methods that has not already been executed on-demand during the previous phases. The purpose is to initialize the Field and OpRegion objects by executing all CreateField, OperationRegion code in the AML. ACPI 2.0 has additional elements that will need to be initialized this way (*Not yet implemented.)*

#### 3.2.4.6 Other Operating System ACPI-related Initialization

All external ACPI  interfaces are available.

- Enumerate devices using the **_HID** method
- Load, configure, and install device drivers
- Device Drivers install handlers for other address spaces such as SmBus, EC, and custom address spaces
- The PCI driver enumerates PCI devices and loads PCIConfig handlers for PCI-to-PCI-bridge devices (which causes the associated child PCI bus_REG methods to run, etc. RON's comment).

### 3.2.5 System Shutdown

#### 3.2.5.1 [AcpiTerminate]

This step frees all dynamically allocated resources back to the host operating system  The subsystem may be restarted at Phase One after this step completes.

## 3.3 Multithreading Support

### 3.3.1 Reentrancy

All external interfaces to the ACPI CA Core Subsystem are fully reentrant. There are limitations to the amount of concurrency allowed during control method execution, but these limitations are transparent to the calling threads — in the sense that threads that attempt to execute control methods will block until the interpreter becomes available.

### 3.3.2 Control Method Execution

Most of the multithread support within the ACPI subsystem is implemented using traditional locks and mutexes around critical (shared) data areas. However, the AML interpreter design is different in that the ACPI specification defines a special threading behavior for the execution of control methods. The design implements the following portion of the ACPI specification that defines a partially multithreaded AML interpreter in four sentences:

> *A control method can use other internal, or well-defined, control methods to accomplish the task at hand, which can include defined control methods provided by the operating software. Interpretation of a Control Method is not preemptive, but can block. When a control method does block, the operating software can initiate or continue the execution of a different control method. A control method can only assume that access to global objects is exclusive for any period the control method does not block.*

#### 3.3.2.1 Control Method Blocking

First of all, how can a control method block?  This is a fairly exhaustive list of the possibilities:

1. Executes the **Sleep()** ASL opcode
2. Executes the **Acquire()** ASL opcode and the request cannot be immediately satisfied
3. Executes the **Wait()** ASL opcode and the request cannot be immediately satisfied
4. Attempts to acquire the **Global Lock** (via OpRegion access, etc), but must wait
5. Attempts to execute a control method that is serialized and already executing (or is blocked), or has reached its concurrency limit
6. Invokes the host debugger via a write to the debug object or executes the **BreakPoint()** ASL opcode
7. Accesses an **Operation Region** which results in a dispatch to a user-installed handler that blocks on I/O or other long-term operation
8. A **Notify** AML opcode results in a dispatch to a user-installed handler that blocks in a similar way

#### 3.3.2.2 Control Method Execution Rules

Here are some Control Method execution “rules” that the ACPI CA multithread support is built upon. These rules are not always stated explicitly in the ACPI specification — some of them are inferred.

1. A Control Method will run to completion (as far as the interpreter is concerned - this doesn’t include thread preemption and interrupt handling by the OS) unless it blocks (i.e. a control method will not be arbitrarily preempted *by the interpreter*.)
2. If a Control Method blocks, the next Control Method in the queue will be executed. When the original (blocked) control method becomes ready, it will **not** preempt the executing method. Instead, it will be placed back on the execution queue (*We could place the method at the tail or the head of the execution queue, or leave this decision to the OSL implementers*).
3. Methods can be *serialized* (non-reentrant) or reentrant. A thread will block if an attempt is made to execute (either via direct invocation or indirectly via a method call) a serialized method that is already executing (or is blocked).
4. The “implicit” synchronization supported by OpRegions and mentioned in the ACPI specification seems to depend entirely on the non-preemptive control method execution model (see above.)

### 3.3.2.3 A Simple Multithreading Model

The actual mechanisms to block a thread are simple and are already in place on the OSL side:

1. **Sleep()** - directly implemented via ***AcpiOsSleep()***, will block the caller and free the processor.
2. **Acquire()** - implemented via an **AcpiOsSemaphore**.
3. **Wait()** - implemented via an **AcpiOsSemaphore**.
4. **Global Lock** - implemented via an **AcpiOsSemaphore** and the interrupt caused by the release of the lock.
5. Concurrency limit - we could put a queue at each method (high overhead), or simply requeue the thread (perhaps in a high-priority queue if we implement one).
6. Host Debugger - These are simply **AcpiOs** calls that we assume will block for a long time.
7. Operation Region Handler blocks on some OS primitive
8. Notify handler blocks in the same manner as (7).

These mechanisms are sufficient to implement the blocking, but this isn't enough to implement the execution semantics of "no preemption unless the method does something to block itself". This requires additional support. I will take a stab at a multithread model here; please feel free to modify or comment.

1. True concurrent control method execution is not allowed. Although the interpreter is "reentrant" in the sense that more than one thread can call into the interpreter, only one thread at any given time (systemwide) can be **actively** interpreting a control method. All other control methods (and the threads that are executing them) must be either blocked or awaiting execution/resumption.
2. Therefore, we can put a mutex around the entire interpreter and only allow a thread access to the interpreter when there are no other accessing threads.
3. The implication and result is that when an executing control method blocks, it is defined to have stopped accessing the interpreter, and is no longer executing within the interpreter.
4. If any interrupt handler needs interpreter services (such as the EC driver and the **_Qxx** control methods), it must schedule a thread for execution. When it runs, this thread calls the interpreter to execute the method.

The algorithm below implements the model described above:

```
AmlExecuteControlMethod ()
   Acquire (Global Interpreter Lock)
      If <the method does anything that might block>
      Check if it will block (such as wait on a semaphore with a zero
      timeout, or grab global lock)
      If <we know or the method will block or still think that it might
      block>
      (such as sleep, acquire-no-units, wait-no-event, global lock not
      available, reached concurrency limit) - and perhaps before we
      dispatch to a user OpRegion or Notify handler)
         Release (Global Interpreter Lock)     (Allow another thread to
         execute a method)
         Execute the blocking call (AcpiOsSleep or AcpiOsWaitSemaphore)
         Acquire (Global Interpreter Lock)     (Must re-enter the
         interpreter, can't preempt running thread!)
Release (Global Interpreter Lock)     (Finished with this method, free
the interpreter)
```

#### 3.3.2.4 A More Complex Multithreading Model

This extension to the model shown above adds a mechanism to implement a “priority” system where all executing and blocked Control Methods have a higher priority than methods that are queued and have never executed yet. This allows the interpreter some control over the scheduling of threads that are executing control methods, without relying directly on an OS-defined priority mechanism. In other words, it provides an OS-dependent way to schedule threads the way we want.

Two semaphores are used, call them an “Outer Gate” and an “Inner Gate”. A thread must pass through both gates before it can begin execution. Once inside both gates, it releases the outer gate, allowing a thread in to wait at the inner gate. When the first thread completes execution of the method, it releases the inner gate, allowing the next thread to proceed. If at any time during execution a thread must block, it releases the inner gate, blocks, then re-acquires the inner gate when it resumes execution.

The maximum length of the queue at the inner gate will never exceed **<the number of blocked threads (running a method)> + 1** (the last thread allowed in through the outer gate).

In the typical (blocking) case, T1 blocks allowing T2 to run. T1 unblocks and eventually waits on the inner gate. T2 eventually completes and signals the inner gate. T1 now runs to completion. All of this happens regardless of the number of threads waiting at the outer gate - therefore, it gives priority to threads that are already running a method.

The algorithm below implements the modified model described above:

```
AmlExecuteControlMethod ()
     Acquire (Outer Lock)
     Acquire (Inner Lock) (Must acquire both locks to begin execution)
     Release (Outer Lock) (Allow one thread into the outer lock)
     If <the method does anything that might block>
        Check if it will block (such as wait on a semaphore with a zero
        timeout)
        If <we know or the method will block or still think that it might
        block>
        (such as sleep, acquire-no-units, wait-no-event, global lock not
        available, reached concurrency limit) - and perhaps before we
        dispatch to a user OpRegion or Notify handler)
           Release (Inner Lock)      (Allow another thread to begin
           execution of a method)
           Execute the blocking call (AcpiOsSleep, AcpiOsWaitSemaphore,
           etc.)
           Acquire (Inner Lock)      (Must re-enter the interpreter since
           we cannot preempt running thread!)
     Release (Inner Lock)     (Finished with this method, free the
     interpreter)
```

***Note:*** It is not so important that the threads free the locks in **reverse** order as it is that they all unlock the locks **in the same order**. Since they are all executing the same code, this behavior is ensured.

While the simple multithreading model will be *sufficient*, the more complex model allows a more “fair” allocation of resources under heavy load. The outstanding question is whether there will ever be enough concurrent use of the AML interpreter to justify the complexity of the second model.

### 3.3.3 Global Lock Support

The ACPI *Global Lock* is intended to be a mutual exclusion mechanism that allows both the host operating system and the resident firmware to access common hardware and data structures. It is not intended to be a mutual exclusion mechanism between threads implemented by the host OS.

The one and **only** purpose of the Global Lock is to provide synchronization between the resident firmware (SMI BIOS, etc.) and all other software on the platform.

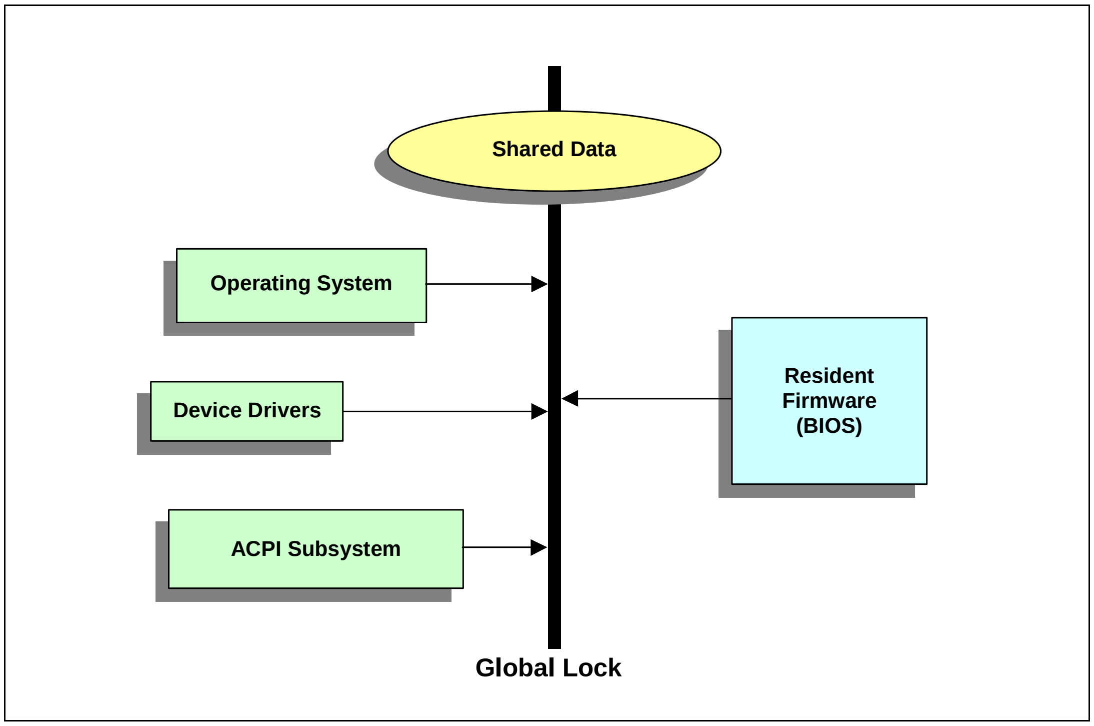
The following assumptions are made about interaction between the OS and firmware concerning the ACPI Global Lock:

- When the firmware owns the global lock, the OS queues up all requests to acquire the global lock
- When the firmware releases the global lock, the OS grabs it and releases (satisfies) all queued requests
- When the last thread calls the OS to release the global lock (now all of the **acquires** have performed a matching **release**), the OS does the actual hardware release.

With this algorithm, it is possible to “starve” the firmware for arbitrary lengths of time, but this is not considered to be a major problem.

The diagram below shows the global lock in relation to the BIOS and other system software.

**Figure 8. Global Lock Architecture**



### 3.3.3.1 Obtaining The Global Lock

```
Gbl_GlobalLockThreadCount++;
If (Gbl_WeHaveTheGlobalLock)
{
     return; /* All done! */
}
If (AcquireHardwareGlobalLock())
{
     Gbl_WeHaveTheGlobalLock = TRUE;
     return; /* All done! */
}
```

```
AmlExitInterpreter ();
AcpiOsWaitSemaphore (GlobalLockSemaphore, WAIT_FOREVER);
AmlEnterInterpreter ();
```

#### 3.3.3.2 Releasing the Global Lock

```
Gbl_GlobalLockThreadCount--;
If (Gbl_GlobalLockThreadCount == 0)
{
     Gbl_WeHaveTheGlobalLock = FALSE;
     ReleaseHardwareGlobalLock ();
}
```

#### 3.3.3.3 Global Lock Interrupt Handler

```
/* We get an SCI when the firmware releases the lock */
AcquireHardwareGlobalLock ()
Gbl_WehaveTheGlobalLock = TRUE;
For <all threads waiting on the lock> (Gbl_GlobalLockThreadCount)
{
     AcpiOsSignalSemaphore (GlobalLockSemaphore);
}
```

### 3.3.4 Single Thread Environments

Both the design and implementation of the ACPI CA Core Subsystem is targeted primarily for inclusion within the kernel of a multitasking operating system. However, it is possible to generate and operate the subsystem within a single threaded environment — with either a primitive operating system or loader, or even standalone with no additional system software other than a few device drivers.

The successful operation of the ACPI CA in any environment depends upon the correct implementation of the OSL layer underneath it. This requirement is no different for a single threaded environment, but some special considerations must be made:

The primary mechanism used for mutual exclusion and multithread synchronization throughout the ACPI subsystem is the OSL *Semaphore*. Since this mechanism is not required in a single threaded environment, it is sufficient to implement these interfaces to simply always return an AE_OK exception code.

When used within an OS kernel at ring 0, the ACPI debugger requires a dedicated thread to perform command line processing. Since this mechanism is not required in a single threaded environment, it can be configured out during generation of the subsystem.

If defined, the “ACPI_APPLICATION” switch disables all multithread support throughout the ACPI core subsystem.

## 3.4 Debugging Support

Two styles of debugging are supported with the debugging tools available with the ACPI Subsystem:

1. Extraordinary amounts of trace and debug output can be generated from debug output and trace statements that are embedded in the debug version of the ACPI subsystem. This data can be used to track down problems after the fact. So much data can be generated that the debug output can be selectively enabled on a per-subcomponent basis and even a finer granularity of the type of debug statement can be selected.

2. An AML debugger is provided that has the ability to single step control methods to examine the results of individual AML opcodes, and to change the values of local variables and method arguments if necessary.

## 3.4.1 Function Tracing (ACPI_FUNCTION_TRACE Macro)

Most of the functions within the subsystem use the ACPI_FUNCTION_TRACE macro upon entry and the return_ACPI_STATUS macro upon exit. For the debug version of the subsystem, if the function trace debug level is enabled, the ACPI_FUNCTION_TRACE macro displays the name of the module and function and the current call nesting level. Upon exit, the return_ACPI_STATUS macro again displays the name of the function, the call nesting level, and the return status code of the call.

The next few lines show examples of the function tracing. On each invocation of the ACPI_FUNCTION_TRACE macro, we see the module name and line number, followed by the call nesting level (2 digits), followed by the name of the actual procedure entered. Some versions of the ACPI_FUNCTION_TRACE macro allow one of the function parameters to be displayed as well.

```
Executing \BITZ
nsobject-0356 [07] NsGetAttachedObject   : ----Entry 004A2CC8
nsobject-0373 [07] NsGetAttachedObject   : ----Exit- 004A2728
dswscope-0186 [07] DsScopeStackPush      : ----Entry
 utalloc-0235 [07] UtAcquireFromCache    : 004A1DC8 from State Cache
  utmisc-0711 [08] UtPushGenericState    : ----Entry
  utmisc-0719 [08] UtPushGenericState    : ----Exit-
dswscope-0223 [07] DsScopeStackPush      : ----Exit- AE_OK
dsmthdat-0274 [07] DsMethodDataInitArgs  : ----Entry 004A1438
dsmthdat-0655 [08] DsStoreObjectToLocal  : ----Entry
dsmthdat-0657 [08] DsStoreObjectToLocal  : Opcode=104 Idx=0 Obj=004A2F08
```

The function entry and exit macros have the ability to generate huge amounts of output data. However, this is often the best way to determine the actual execution path taken by subsystem. If the problem being debugged can be narrowed to a single control method, tracing can be enabled for that method only, thus reducing the amount of debug data generated.

## 3.4.2 Execution Debug Output (ACPI_DEBUG_PRINT Macro)

The ACPI_DEBUG_PRINT macro is used throughout the source code of the ACPI core subsystem to selectively print debug messages. Over 900 invocations of the ACPI_ DEBUG_PRINT are scattered throughout the ACPI subsystem source. This macro is compiled out entirely for non-debug versions of the subsystem.

Output from ACPI_ DEBUG_PRINT can be enabled at two levels: on a per-subcomponent level (Namespace manager, Parser, Interpreter, etc.), and on a per-type level (informational, warnings, errors, and more.) There are two global variables that set these output levels:

1. **DebugLayer** Bitfield that enables/disables debug output from entire subcomponents within the ACPI subsystem.
2. **DebugLevel** Bitfield that enables/disables the various debug output levels

The example below shows some of the debug output from a namespace search. None of the output of the function tracing is shown here, but the enter/exit traces would appear interspersed with the other debug output.

```
nsutils-0346:  NsInternalizeName: returning [00821F30] (abs) "\BITZ"
nsaccess-0424: NsLookup: Searching from root [007F09B4]
nsaccess-0477: NsLookup: Multi Name (1 Segments, Flags=0)
nsaccess-0494: NsLookup: [BITZ/]
nssearch-0166: NsSearchOnly: Searching \/    [007F09B4]
nssearch-0168: NsSearchOnly: For BITZ (type 0)
nssearch-0239: NsSearchOnly: Name BITZ (actual type 8) found at 007FC384
nseval-0302:   NsEvaluateByName: \BITZ [007FC384] Value 007FE0C0
```

### 3.4.3 ACPI Debugger

Provided as a subcomponent of the ACPI Core Subsystem, the ACPI/AML Debugger provides the capability to display subsystem data structures and objects (such as the namespace and associated internal object), and to debug the execution of control methods (including single step and breakpoint support.) By using only two OSL interfaces, *AcpiOsGetLine* for input and *AcpiOsPrint* for output, the debugger can operate standalone or as an extension to a host debugger.

The debugger provides a more active debugging environment where data can be examined and altered during the execution of control methods.

## 3.5 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.

### 3.5.1 Resource Requirements

Static Memory: TBD: Code/data for both debug and non-debug versions

Dynamic Memory: TBD: (Tables, namespace, objects)

System Objects: TBD: (Semaphores)

### 3.5.2 C Library Functions

In order to make the ACPI Core Subsystem as portable and truly OS-independent as possible, there is only extremely limited use of standard C library functions within the Core Subsystem 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 used by the Core Subsystem code.

**Table 1. C Library Functions Used within the Subsystem**

| |
|---|
| `sprintf` |
| `memcpy` |
| `memset` |
| `strcat` |
| `strcmp` |
| `strcpy` |
| `strlen` |
| `strncmp` |
| `strncat` |
| `strncpy` |
| `strstr` |
| `strtoul` |
| `strupr` |
| `toupper` |
| `tolower` |
| `va_list` |
| `va_start` |
| `va_end` |

If "SYSTEM_CLIB_FUNCTIONS" is defined during the compilation of the subsystem, the subsystem must be linked to a local C library to resolve these Clib references. If SYSTEM_CLIB_FUNCTIONS is not set, the subsystem will automatically link to local implementations of these functions. Note that the local implementations are written in portable ANSI C, and may not be as efficient as local assembly code implementations of the same functions. Therefore, it is recommended that the local versions of the C library functions be used if at all possible.

### 3.5.3 System Include Files

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

| | |
|---|---|
| • acexcep.h | The ACPI_STATUS exception codes |
| • acpiosxf.h | The prototypes for all of the **AcpiOs*** interfaces |
| • acpixf.h | The prototypes for all of the **Acpi*** interfaces |
| • actypes.h | Common data types used across all interfaces |

#### 3.5.3.1 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 *acenv.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 Core Subsystem component from source, the acenv.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 Core Subsystem component has been kept to a minimum in order to enhance portability and to ensure that the Core Subsystem will run as a kernel-level component in most operating systems.

# 4 Interface Parameters and Data Types

## 4.1 ACPI Subsystem Interface Parameters

### 4.1.1 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 Core Subsystem 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.

### 4.1.2 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.

### 4.1.3 Buffers

It is the responsibility of the caller to ensure that all input and output buffers supplied to the Core Subsystem 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 Core Subsystem 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 Services Layer 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 ACPI Core Subsystem *trusts* the OS Services Layer to validate all buffers.

When the length field of ACPI_BUFFER is set to ACPI_ALLOCATE_BUFFER before a call that returns data in an output buffer, the core subsystem will allocate a return buffer on behalf of the caller. It is the responsibility of the caller to free this buffer when it is no longer needed.

## 4.2 ACPI Subsystem Data Types

### 4.2.1 UINT64 and COMPILER_DEPENDENT_UINT64

Beginning with the ACPI version 2.0 specification, the width of integers within the AML interpreter are defined to be 64 bits on all platforms (both 32- and 64-bit). The implementation of this requirement requires the deployment of 64-bit integers across the entire ACPI Core Subsystem. Since there is (currently) no standard method of defining a 64-bit integer in the C language, the COMPILER_DEPENDENT_UINT64 macro is used to allow the UINT64 typedef to be defined by each host compiler. The UINT64 data type is used at the Acpi* interface level for both physical memory addresses and ACPI (interpreter) integers.

### 4.2.2 ACPI_PHYSICAL_ADDRESS

The width of all *physical* addresses is fixed at 64 bits, regardless of the platform or operating system. Logical addresses (pointers) remain the natural width of the machine (i.e. 32 bit pointers on 32-bit machines, 64-bit pointers on 64-bit machines.) This allows for a full 64 bit address space on 64-bit machines as well as “extended” physical addresses (above 4Gbytes) on 32-bit machines.

### 4.2.3 ACPI_POINTER

This data type is a union that allows either a physical address or logical pointer to be specified. A flags field defines the pointer type.

### 4.2.4 ACPI_INTEGER

This is the data type that directly corresponds to the ACPI-defined *Integer* data type. Beginning with ACPI 2.0, the width of this data type is 64 bits on all platforms.

### 4.2.5 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.

### 4.2.6 ACPI_BUFFER – Input and Output Memory Buffers

Many of the ACPI CA interfaces 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 Core Subsystem component only allocates memory for return buffers if requested to do so — 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;
```

#### 4.2.6.1 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 Core Subsystem 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 Core Subsystem interface is called.

#### 4.2.6.2 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 either

1. 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.

2. Initializing the Length field to ACPI_ALLOCATE_BUFFER to cause the ACPI subsystem to allocate a buffer.

If a buffer that was 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 (unless ACPI_ALLOCATE_BUFFER is used). 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 either the size of the buffer or an indication to the ACPI subsystem to allocate a return buffer on behalf of the caller. 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.

### 4.2.7 ACPI_HANDLE – Object Handle

References to ACPI objects managed by the Core Subsystem 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.

#### 4.2.7.1 Predefined Handles

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

1. **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.

### 4.2.8 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:

Table 2. ACPI Object Type Codes

| |
|---|
| 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_BufferField |
| ACPI_TYPE_DdbHandle |
| ACPI_TYPE_DebugObject |

```
ACPI_OBJECT_TYPE_MAX
```

### 4.2.9 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
{
     ACPI_OBJECT_TYPE       Type;       // Object Type
     struct  /* ACPI_TYPE_Number */
     {
       ACPI_OBJECT_TYPE     Type;
       ACPI_INTEGER         Value;      // The actual number (64 bits)
     } Number;
     struct  /* ACPI_TYPE_String */
     {
       ACPI_OBJECT_TYPE     Type;
       UINT32               Length;     // Length of string without null
       NATIVE_CHAR          *Pointer;   // points to the string value
     } String;
     struct  /* ACPI_TYPE_Buffer */
     {
       ACPI_OBJECT_TYPE     Type;
       UINT32               Length;    // # of bytes in buffer
       UINT8                *Pointer;  // points to the buffer
     } Buffer;
     struct  /* ACPI_TYPE_Any */
     {
       ACPI_OBJECT_TYPE     Type;
       ACPI_HANDLE          Handle;     // object reference
     } Reference;
     struct  /* ACPI_TYPE_Package */
     {
       ACPI_OBJECT_TYPE     Type;
       UINT32               Count;      // # of elements in package
       union AcpiObj        *Elements;  // Pointer to array of Objects
     } Package;
     struct  /* ACPI_TYPE_Processor */
     {
       ACPI_OBJECT_TYPE    Type;
       UINT32              ProcId;
       ACPI_IO_ADDRESS     PblkAddress;
       UINT32              PblkLength;
     } Processor;
     struct  /* ACPI_TYPE_Power */
     {
       ACPI_OBJECT_TYPE    Type;
       UINT32              SystemLevel;
       UINT32              ResourceOrder;
     } PowerResource;
} ACPI_OBJECT, *PACPI_OBJECT;
```

### 4.2.10 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;
```

### 4.2.11 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           // TBD
ACPI_EVENT_MAX
```

### 4.2.12 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 3. ACPI Table Type Codes**

| | |
|---|---|
| TABLE_RSDP | // Root System Description Pointer |
| TABLE_DSDT | // Differentiated System Description Table |
| TABLE_FADT | // 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 |

### 4.2.13 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;
```

### 4.2.14 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 = AcpiLoadTables (RsdpPhysicalAddress);
if (Status != AE_OK)
{
      // Exception handling code here
}
```

## 4.3 ACPI Resource Data Types

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

### 4.3.1 PCI IRQ Routing Tables

The *AcpiGetIrqRoutingTable* interface retrieves the PCI IRQ routing tables. This interface 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 the returned status is AE_BUFFER_OVERFLOW, 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;
```

## 4.3.2 Device Resources

Device resources are returned by indirectly executing the _CRS and _PRS control methods via the *AcpiGetCurrentResources* and *AcpiGetPossibleResources* interfaces. These device resources are needed to properly execute the _SRS control method using the *AcpiSetCurrentResources* interface.

These interfaces require an ACPI_BUFFER parameter. If the *Length* member of the ACPI_BUFFER is set to zero, the **AcpiGet*** interfaces 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.

### 4.3.2.1 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 interfaces: *AcpiGetCurrentResources*, *AcpiGetPossibleResources* and *AcpiSetCurrentResources*.

1. Irq
3. Dma
4. StartDependentFunctions
5. EndDependentFunctions
6. Io
7. FixedIo
8. VendorSpecific
9. EndTag
10. Memory24
11. Memory32
12. FixedMemory32
13. Address16
14. Address32
15. 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 that are found in actypes.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, Chapter 6.

As another, more complex example, consider a RESOURCE entry with an *Id* member that evaluates to Address32, then the *Data* member is an 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, Chapter 6. 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*.

## 4.4 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 acexcep.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.

**Table 4. Exception Code Values**

| | |
|---|---|
| AE_OK | // No error |
| /* Environmental Exceptions */ | |
| 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_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_OBJECT | // A required object was missing |
| 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_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_UNKNOWN_STATUS | // An unknown status code was encountered |

**Table 4. Exception Code Values (Cont'd)**

| | |
|---|---|
| `/* Programmer Exceptions */` | |
| `AE_BAD_PARAMETER` | `// A parameter is out of range or invalid` |
| `AE_BAD_CHARACTER` | `// An invalid character was found in a name` |
| `AE_BAD_PATHNAME` | `// An invalid character was found in a pathname` |
| `AE_BAD_DATA` | `// A package or buffer contained incorrect data` |
| `AE_BAD_ADDDRESS` | `// An invalid physical address` |
| `/* ACPI Table Exceptions */` | |
| `AE_BAD_SIGNATURE` | `// An ACPI table has an unrecognized signature` |
| `AE_BAD_HEADER` | `// Invalid field in an ACPI table header` |
| `AE_BAD_CHECKSUM` | `// An ACPI table checksum is not correct` |
| `AE_BAD_VALUE` | `// An invalid value was found in a table */` |
| `/* AML Exceptions */` | |
| `AE_AML_ERROR` | `// Unspecified AML error` |
| `AE_AML_PARSE` | `// Invalid AML could not be parsed` |
| `AE_AML_BAD_OPCODE` | `// Invalid AML opcode encountered` |
| `AE_AML_NO_OPERAND` | `// An operand is missing (such as a method ret val)` |
| `AE_AML_OPERAND_TYPE` | `// An operand of an incorrect type was encountered` |
| `AE_AML_OPERAND_VALUE` | `// The operand had an inappropriate or invalid value` |
| `AE_AML_UNINITIALIZED_LOCAL` | `// Method tried to use an uninitialized local var` |
| `AE_AML_UNINITIALIZED_ARG` | `// Method tried to use an uninitialized argument` |
| `AE_AML_UNINITIALIZED_ELEMENT` | `// Method tried to use an empty package element` |
| `AE_AML_NUMERIC_OVERFLOW` | `// Overflow during BCD conversion or other` |
| `AE_AML_REGION_LIMIT` | `// Tried to access beyond a defined Operation Region` |
| `AE_AML_BUFFER_LIMIT` | `// Tried to access beyond the end of a buffer` |
| `AE_AML_PACKAGE_LIMIT` | `// Tried to access beyond the end of a package` |
| `AE_AML_DIVIDE_BY_ZERO` | `// Bad divide` |
| `AE_AML_BAD_NAME` | `// An ACPI name contained invalid character(s)` |
| `AE_AML_NAME_NOT_FOUND` | `// Could not resolve a named reference` |
| `AE_AML_INTERNAL` | `// An internal error within the interpreter` |

**Table 4. Exception Code Values (Cont'd)**

```
/* Internal Exceptions used for control */
AE_CTRL_RETURN_VALUE            // Internal use – method has returned a
                                   value
AE_CTRL_PENDING                 // Internal use – method is calling
                                   another method
AE_CTRL_TERMINATE               // Request to terminate the current
                                   operation
AE_CTRL_TRUE                    // Internal use – predicate result
AE_CTRL_FALSE                   // Internal use – predicate result
AE_CTRL_DEPTH                   // Maximum search depth has been reached
```

# 5 ACPI Core Subsystem - External Interface Definition

This section contains documentation for the specific interfaces exported by the ACPI Core. The interfaces are grouped based upon their functionality. These groups are closely related to the internal modules (or sub-components) of the Core Subsystem described earlier in this document. These interfaces are intended to be used by the OSL only. The host OS does not call these interfaces directly. All interfaces to the ACPI Core Subsystem are prefixed by the letters **Acpi**.

## 5.1 Subsystem Configuration

There are two methods of configuring the OS-independent ACPI Core Subsystem. The first is the compile-time configuration through the use of compiler switches. The second configuration method is via run-time global variables which are statically initialized from the configuration header file (This is really a combination of static compile-time configuration and run-time configuration).

### 5.1.1 Compile-time Configuration

The subsystem is configured at compile time via various compiler switches that are described below.

#### 5.1.1.1 ACPI_DEBUG

This switch enables the DEBUG_PRINT macro and various other debugging support within the core subsystem. The code for the DEBUG_PRINT macro is only generated when the ACPI_DEBUG switch is set;  otherwise the macro is defined to be null and thus all debug code is compiled out.

#### 5.1.1.2 ACPI_APPLICATION

This switch should be set when the entire ACPI subsystem is to be run as an application on top of an operating system instead of a driver integrated with the kernel.

#### 5.1.1.3 PARSER_ONLY

This switch is used by applications that only use the AML parser, not the interpreter. An example application is the *AcpiDump* utility that simply disassembles the AML code, it does not attempt to interpret the code.

#### 5.1.1.4 SYSTEM_CLIB_FUNCTIONS

This switch allows the use of a system-supplied C library for the Clib functions used by the subsystem. If this switch is not set, the subsystem uses its own implementations of these functions. Use of a system C library (when available) may be more efficient in terms of reused system code and efficiency of the function implementations.

#### 5.1.1.5 _IA16

This switch sets the 16-bit data types and allows compilation with 16-bit compilers. The default architecture width  is 32 bits.

#### 5.1.1.6 _IA64

This switch sets the 64-bit data types and allows compilation with 64-bit compilers. The default architecture width is 32-bits. Some 64-bit compilers may automatically set this switch

### 5.1.2 Run-time Configuration

The run-time configuration begins with constants that are specified in the *config.h* header file. These constants may be modified at either compile time by changing the constants in *config.h*, or at run-time by changing the contents of the global variables where these constants are stored.

#### 5.1.2.1 ACPI_OS_NAME

This is the string associated with and returned by the _OS_ named object. Change the string to the appropriate product name and/or product ID.

#### 5.1.2.2 MAX_STATE_CACHE_DEPTH

The maximum number of objects in the generic state object cache used to avoid recursive calls within the subsystem. These are small objects, but are used frequently. A larger cache will improve the performance of the entire subsystem (loading tables, parsing methods, and executing methods.)

#### 5.1.2.3 MAX_PARSE_CACHE_DEPTH

The maximum number of objects in the parse object cache. These are the objects used to build parse trees. A larger cache will improve the execution performance of control methods (when the parse just-in-time strategy is used) by improving the time to parse the AML.

#### 5.1.2.4 MAX_OBJECT_CACHE_DEPTH

The maximum number of objects in the interpreter operand object cache. These objects are used during control methods to pass the operands for individual AML opcodes to the interpreter. A larger cache will improve the performance of control method execution

#### 5.1.2.5 MAX_WALK_CACHE_DEPTH

The maximum number of objects in the parse tree walk object cache. These are relatively large objects (about 512 bytes) that are used to contain the entire state of a control method during its execution. Each nested control method requires an additional walk object. Since only one object is required per control method, it is not necessary to cache a large number of these objects. A few cached walk objects are sufficient to increase the performance of control method execution and reduce memory fragmentation.

## 5.2 Global Initialization, Shutdown, and Status

### 5.2.1 AcpiInitializeSubsystem

Initialize all ACPI components.

**ACPI_STATUS**
**AcpiInitializeSubsystem (**
**void)**

**PARAMETERS**

None

**RETURN**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The subsystem 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 entire ACPI subsystem, including the OS Services Layer. It must be called once before any of the other **Acpi*** interfaces are called.

### 5.2.2 AcpiEnableSubsystem

**Complete the ACPI Subsystem initialization and enable ACPI operations.**

```
ACPI_STATUS
AcpiEnableSubsystem (
    UINT32                          Flags)
```

**PARAMETERS**

| | |
|---|---|
| Flags | Specifies how the subsystem should be initialized. Must be one of these manifest constants: |
| | **ACPI_FULL_INITIALIZATION** – Perform completed initialization. This is the normal use of this interface |
| | **ACPI_NO_ADDRESS_SPACE_INIT**. Do not install the default address space handlers. For debug purposes only. |
| | **ACPI_NO_HARDWARE_INIT**. Do not initialize the ACPI hardware. For hardware-independent mode only. |
| | **ACPI_NO_EVENT_INIT**. Do not install an SCI handler and do not initialize ACPI events. For hardware independent mode only |
| | **ACPI_NO_ACPI_ENABLE**. Do not attempt to enter ACPI mode. For hardware-independent mode only. |
| | **ACPI_NO_DEVICE_INIT**. Do not attempt to run the _INI methods on devices in the ACPI namespace. |
| | **ACPI_NO_PCI_INIT**. When running _INI methods, do not attempt to recognize and initialize the PCI Root Bridges. |
| | **ACPI_NO_OBJECT_INIT**. Do not run the final initialization pass to complete initialization of all address spaces and Fields. |

**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 completes initialization of the ACPI Subsystem.

### 5.2.3 AcpiTerminate

**Shutdown all ACPI Components.**

**ACPI_STATUS**
**AcpiTerminate (**
**void)**

**PARAMETERS**

None

**RETURN**

| | |
|---|---|
| Status | Exception code indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The subsystem was successfully shutdown. |
| AE_ERROR | TBD!! |

**Functional Description:**

This function performs a shutdown of the Core Subsystem 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:

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

### 5.2.4 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 *Length* field of *OutBuffer* is not ACPI_ALLOCATE_BUFFER, but 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. |

TBD: Add and consolidate other ACPI info such as: Timer resolution, Acpi and legacy mode capabilities, supported sleep states, current mode, etc.

### 5.2.5 AcpiFormatException

**Return the ASCII name of an ACPI exception code**

**const char ***
**AcpiFormatException (**
**ACPI_STATUS Status)**

**PARAMETERS**

| | |
|---|---|
| Status | The ACPI status/exception code to be translated. |

**RETURN VALUE**

| | |
|---|---|
| Exception String | A pointer to the formatted exception string. |

**EXCEPTIONS**

None

**Functional Description:**

This function converts an ACPI exception code into a human-readable string. It returns the exception name string as the function return value. The string is a const value that does not require deletion by the caller.

## 5.3 Memory Management

The ACPI core subsystem provides memory management services that are built upon the memory management services exported by the OS services layer. If enabled (in debug mode), the core memory manager tracks and logs each allocation to detect the following conditions:

1) Detect attempts to release (free) an allocated memory block more than once.

2) Detect memory leaks by keeping a list of all outstanding allocated memory blocks. This list can be examined at any time; however, the best time to find memory leaks is after the subsystem is shutdown -- any remaining allocations represent leaked blocks.

Do not mix memory manager calls. In other words, if the Acpi* memory manager is used to allocate memory, do not free memory via the OS Services Layer (AcpiOsFree), via the C library (free), or directly call the OS memory management primitives.

### 5.3.1 AcpiAllocate

**Allocate memory from the dynamic memory pool.**

**void ***
**AcpiAllocate (**
**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.

### 5.3.2 AcpiCallocate

**Allocate and initialize memory.**

**void ***
**AcpiCallocate (**
**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.

### 5.3.3 AcpiFree

**Free previously allocated memory.**

```
void
AcpiFree (
    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 *AcpiAllocate* or *AcpiCallocate*.

## 5.4 ACPI Table Manipulation

### 5.4.1 AcpiGetFirmwareTable

**Obtain a firmware-supplied ACPI table**

```
ACPI_STATUS
AcpiGetFirmwareTable  (
    ACPI_STRING             TableSignature,
    UINT32                  TableInstance,
    UINT32                  Flags,
    ACPI_TABLE_HEADER       **Table)
```

**PARAMETERS**

| | |
|---|---|
| TableSignature | A string containing the ACPI-defined ASCII signature of the desired table |
| TableInstance | If multiple instances of the table are allowed. |
| Flags | Current addressing mode of the processor – whether paging is currently enabled or not – one of these manifest constants:<br>ACPI_PHYSICAL_ADDRESSING<br>ACPI_LOGICAL_ADDRESSING |
| Table | A pointer to where the address of the requested ACPI table is returned. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The requested table was found and returned. |
| AE_NO_ACPI_TABLES | A valid RSDP could not be located. |
| AE_NO_MEMORY | Insufficient dynamic memory to complete the operation. |

**Functional Description:**

This function locates and returns one of the ACPI tables that are supplied by the system firmware. On IA-32 systems, this involves scanning within the first megabyte of physical memory for the RSDP signature.

This function may be called at any time, even before the ACPI subsystem has been initialized. This allows early access to ACPI tables -- even before the system virtual memory manager has been started.

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

## 5.4.2 AcpiFindRootPointer

**Locate the RSDP via memory scan**

```
ACPI_STATUS
AcpiFindRootPointer (
    UINT32              Flags,
    ACPI_POINTER        *RsdpPhysicalAddress)
```

**PARAMETERS**

| | |
|---|---|
| Flags | Current addressing mode of the processor – whether paging is currently enabled or not – one of these manifest constants:<br><br>ACPI_PHYSICAL_ADDRESSING<br><br>ACPI_LOGICAL_ADDRESSING |
| RsdpPhysicalAddress | A pointer to where the physical address of the ACPI RSDP table will be returned. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The RSDP was found and returned. |
| AE_NO_ACPI_TABLES | A valid RSDP could not be located. |

| | |
|---|---|
| AE_NO_MEMORY | Insufficient dynamic memory to complete the operation. |

**Functional Description:**

This function locates and returns the ACPI Root System Description Pointer by scanning within the first megabyte of physical memory for the RSDP signature.  This is mechanism is only applicable to IA-32 systems.

This interface should only be called from the OSL function *AcpiOsGetRootPointer* if the memory scanning mechanism is appropropriate for the current platform.

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

### 5.4.3 AcpiLoadTables

Load core ACPI tables and build an internal ACPI namespace

**ACPI_STATUS**
**AcpiLoadTables  (**
    **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, FADT, etc.) could not be found in physical memory. |
| AE_NO_MEMORY | Insufficient dynamic memory to complete the operation. |

**Functional Description:**

This function loads the ACPI tables that are pointed to by the RSDP/RSDT and installs them into the internal ACPI namespace database. The *Root System Description Pointer* (RSDP) points to the *Root System Description Table* (RSDT), and the remaining ACPI tables are found via pointers contained in RSDT.

The minimum required set of ACPI tables that will allow the ACPI CA core subsystem to initialize consists of the following:

- RSDT/XSDT
- FADT
- FACS
- DSDT

Only tables that are used directly by the ACPI subsystem are loaded. Other tables (such as the MADT, SRAT, etc.) are obtained and consumed by different kernel subsystems and/or device drivers.

If the operation fails an appropriate status will be returned.

### 5.4.4 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 and installs it into the internal ACPI namespace database. 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.

### 5.4.5 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.

## 5.4.6 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.

## 5.4.7 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 *Length* field of *OutBuffer* is not ACPI_ALLOCATE_BUFFER, but 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.

## 5.5 ACPI Namespace Access

### 5.5.1 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:<br>• A handle to the object to be evaluated.<br>• A handle to a parent object that is a prefix to the pathname.<br>• 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 unspecified error occurred during the parsing of the AML code. |
| AE_AML_PARSE | The control method could not be parsed due to invalid AML code. |
| AE_AML_BAD_OPCODE | An invalid opcode was encountered in the AML code. |
| AE_AML_NO_OPERAND | An required operand was missing. This could be caused by a method that does not return any object. |
| AE_AML_OPERAND_TYPE | An operand object is not of the required ACPI type. |
| AE_AML_OPERAND_VALUE | An operand object has an invalid value |
| AE_AML_UNINITIALIZED_LOCAL | A method attempted to access a local variable that was not initialized. |
| AE_AML_UNINITIALIZED_ARG | A method attempted to access an argument that was not part of the argument list, or was not passed into the method properly. |
| AE_AML_UNITIALIZED_ELEMENT | A method attempted to use (dereference) a reference to an element of a package object that is empty (uninitialized). |
| AE_AML_NUMERIC_OVERFLOW | An overflow occurred during a numeric conversion (Such as BCD conversion.) |
| AE_AML_REGION_LIMIT | A method attempted to access beyond the end of an Operation Region defined boundary. |
| AE_ AML_BUFFER_LIMIT | A method attempted to access beyond the end of a Buffer object. |
| AE_ AML_PACKAGE_LIMIT | A method attempted to access beyond the end of a Package object. |
| AE_ AML_DIVIDE_BY_ZERO | A method attempted to execute a divide instruction with a zero divisor. |
| AE_AML_BAD_NAME | A name contained within the AML code has one or more invalid characters. |
| AE_AML_NAME_NOT_FOUND | A name reference within the AML code could not be found and therefore could not be resolved. |
| AE_AML_INTERNAL | An error that is internal to the ACPI CA subsystem occurred. |
| AE_BAD_CHARACTER | An invalid character was found in the Pathname parameter. |

| | |
|---|---|
| AE_BAD_DATA | Bad or invalid data was found in a package object. |
| 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:<br>• Both the *Object* and *Pathname* parameters are `NULL.`<br>• The *Object* handle is `NULL`, but the *Pathname* is not absolute.<br>• The *Pathname* is relative but the *Object* is invalid.<br>• The *Length* field of *OutBuffer* is not ACPI_ALLOCATE_BUFFER, but the *Pointer* field of *OutBuffer* 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_NULL_OBJECT | A required object was missing. This is an internal error. |
| 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. Others TBD.

Exceptional Conditions: Any exceptions that occur during the execution of a control method result in the immediate termination of the control methods. All nested control methods are also terminated, up to and including the parent method.

**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);
```

## 5.5.2 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.<br>• 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.

### 5.5.3 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:<br>• The *Parent* handle is invalid.<br>• The *Child* handle is invalid.<br>• 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.

## 5.5.4 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:<br>• The *Child* handle is invalid.<br>• 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.

### 5.5.5 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:<br>• The *Object* handle is invalid.<br>• 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.

### 5.5.6 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.

### 5.5.7 AcpiGetName

**Get the name of an ACPI object.**

```
ACPI_STATUS
AcpiGetName (
    ACPI_HANDLE            Object,
    UINT32                 NameType
    ACPI_BUFFER            *OutName)
```

**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:<br>• ACPI_FULL_PATHNAME – return a complete pathname (from the namespace root) to the object<br>• 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:<br>• The Parent handle is invalid.<br>• The Object handle is invalid.<br>• The OutName pointer is NULL.<br>• The *Length* field of *OutName* is not ACPI_ALLOCATE_BUFFER, but 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
```

## 5.5.8 AcpiAttachData

**Attach user data to an ACPI namespace object**

```
ACPI_STATUS
AcpiAttachData (
    ACPI_HANDLE               Object,
    ACPI_OBJECT_HANDLER       Handler
    void                      *Data)
```

**PARAMETERS**

| | |
|---|---|
| Object | A handle to an object to which the data will be attached. |
| Handler | A pointer to a function that is called when the namespace object is deleted: |
| Data | A pointer to arbitrary user data. The pointer is stored in the namespace with the namespace object. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The data was successfully attached. |
| AE_BAD_PARAMETER | At least one of the following is true:<br>• The Object handle is invalid.<br>• The *Handler* pointer is NULL.<br>• The *Data* pointer is NULL. |
| AE_NO_MEMORY | |
| AE_NO_NAMESPACE | The namespace has not been successfully loaded. |

**Functional Description:**

This function allows arbitrary data to be associated with a namespace object.

### 5.5.9 AcpiDetachData

**Remove a previously data attachment to a namespace object**

```
ACPI_STATUS
AcpiAttachData (
    ACPI_HANDLE             Object,
    ACPI_OBJECT_HANDLER     Handler)
```

**PARAMETERS**

| | |
|---|---|
| Object | A handle to an object to which the data will be attached. |
| Handler | A pointer to a function that is called when the namespace object is deleted. This must be the same pointer used when the original call to *AcpiAttachData* was used. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The data was successfully detached. |
| AE_BAD_PARAMETER | At least one of the following is true: • The Object handle is invalid. • The *Handler* pointer is NULL. |
| AE_NO_MEMORY | |
| AE_NO_NAMESPACE | The namespace has not been successfully loaded. |

**Functional Description:**

This function removes a previous association between user data and a namespace object.

## 5.5.10 AcpiGetData

**Retrieve data that was associated with a namespace object**

```
ACPI_STATUS
AcpiGetData (
    ACPI_HANDLE              Object,
    ACPI_OBJECT_HANDLER      Handler
    void                     **Data)
```

**PARAMETERS**

| | |
|---|---|
| Object | A handle to an object to from which the attached data will be returned. |
| Handler | A pointer to a function that is called when the namespace object is deleted: This must be the same pointer used when the original call to *AcpiAttachData* was used. |
| Data | A pointer to where the arbitrary user data pointer will be returned. The pointer is stored in the namespace with the namespace object. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The data was successfully returned. |
| AE_BAD_PARAMETER | At least one of the following is true:<br>• The Object handle is invalid.<br>• The *Handler* pointer is NULL.<br>• The *Data* pointer is NULL. |
| AE_NO_MEMORY | |
| AE_NO_NAMESPACE | The namespace has not been successfully loaded. |

**Functional Description:**

This function retrieves data that was previously associated with a namespace object.

## 5.5.11 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,
    ACPI_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.

### 5.5.11.1 Interface to User Callback Function

**Interface to the user function that is invoked from AcpiWalkNamespace.**

```
typedef
ACPI_STATUS  (*ACPI_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*.

## 5.6 ACPI Resource Management

### 5.6.1 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 *Length* field of *OutBuffer* is not ACPI_ALLOCATE_BUFFER, but 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.

### 5.6.2 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:<br>• The *Device* handle is invalid.<br>• The *OutBuffer* pointer is `NULL`.<br>• The *Length* field of *OutBuffer* is not ACPI_ALLOCATE_BUFFER, but 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.

### 5.6.3 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:<br>• The *Device* handle is invalid.<br>• The *InBuffer* pointer is `NULL`.<br>• The *Pointer* field of *InBuffer* is `NULL`.<br>• 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.

### 5.6.4 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:<br>• The Device handle is invalid.<br>• The OutBuffer pointer is NULL.<br>• The *Length* field of *OutBuffer* is not ACPI_ALLOCATE_BUFFER, but 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.

## 5.7 ACPI Event Management

### 5.7.1 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.

### 5.7.2 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.

## 5.7.3 AcpiEnableEvent

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

```
ACPI_STATUS
AcpiEnableEvent (
    UINT32                  Event,
    UINT32                  Type,
    UINT32                  Flags)
```

**PARAMETERS**

| | |
|---|---|
| Event | The fixed event or GPE to be enabled. For GPEs, this must be a number from 0 to 255 and also must be a valid GPE on the current platform. For Fixed Events, this parameter must be one of the following manifest constants:<br><br>ACPI_EVENT_PMTIMER<br><br>ACPI_EVENT_GLOBAL<br><br>ACPI_EVENT_POWER_BUTTON<br><br>ACPI_EVENT_SLEEP_BUTTON<br><br>ACPI_EVENT_RTC |
| Type | The type of event, one of these manifest constants:<br><br>ACPI_EVENT_FIXED<br><br>ACPI_EVENT_GPE |
| Flags | For GPE events, specify if the event should also be enabled for wake events.<br><br>ACPI_EVENT_WAKE_ENABLE |

**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:<br>• The *Event* is invalid.<br>• 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.

## 5.7.4 AcpiDisableEvent

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

```
ACPI_STATUS
AcpiDisableEvent (
    UINT32                  Event,
    UINT32                  Type,
    UINT32                  Flags)
```

**PARAMETERS**

| | |
|---|---|
| Event | The fixed event or GPE to be disabled. For GPEs, this must be a number from 0 to 255 and also must be a valid GPE on the current platform. For Fixed Events, this parameter must be one of the following manifest constants:<br>ACPI_EVENT_PMTIMER<br>ACPI_EVENT_GLOBAL<br>ACPI_EVENT_POWER_BUTTON<br>ACPI_EVENT_SLEEP_BUTTON<br>ACPI_EVENT_RTC |
| Type | The type of event, one of these manifest constants:<br>ACPI_EVENT_FIXED<br>ACPI_EVENT_GPE |
| Flags | For GPE events, specify if the event should only be disabled for wake events.<br>ACPI_EVENT_WAKE_DISABLE |

**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:<br>• The *Event* is invalid.<br>• 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.

## 5.7.5 AcpiClearEvent

**Clear a pending ACPI Event (Fixed Events and General Purpose Events)**

```
ACPI_STATUS
AcpiClearEvent (
    UINT32                 Event,
    UINT32                 Type)
```

**PARAMETERS**

| | |
|---|---|
| Event | The fixed event or GPE to be cleared. For GPEs, this must be a number from 0 to 255 and also must be a valid GPE on the current platform. For Fixed Events, this parameter must be one of the following manifest constants:<br>ACPI_EVENT_PMTIMER<br>ACPI_EVENT_GLOBAL<br>ACPI_EVENT_POWER_BUTTON<br>ACPI_EVENT_SLEEP_BUTTON<br>ACPI_EVENT_RTC |
| Type | The type of event, one of these manifest constants:<br>ACPI_EVENT_FIXED<br>ACPI_EVENT_GPE |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The event was successfully cleared. |
| AE_BAD_PARAMETER | At least one of the following is true:<br>• The *Event* is invalid.<br>• The *Type* is invalid. |

**Functional Description:**

This function clears (zeros the status bit for) a single ACPI event. Both Fixed Events and General Purpose Events may be cleared with this interface.

## 5.7.6 AcpiGetEventStatus

**Obtain the status of an ACPI Event (Fixed Events and General Purpose Events)**

```
ACPI_STATUS
AcpiGetEventStatus (
    UINT32                  Event,
    UINT32                  Type,
    ACPI_EVENT_STATUS       *EventStatus)
```

**PARAMETERS**

| | |
|---|---|
| Event | The fixed event or GPE to be cleared. For GPEs, this must be a number from 0 to 255 and also must be a valid GPE on the current platform. For Fixed Events, this parameter must be one of the following manifest constants:<br>ACPI_EVENT_PMTIMER<br>ACPI_EVENT_GLOBAL<br>ACPI_EVENT_POWER_BUTTON<br>ACPI_EVENT_SLEEP_BUTTON<br>ACPI_EVENT_RTC |
| Type | The type of event, one of these manifest constants:<br>ACPI_EVENT_FIXED<br>ACPI_EVENT_GPE |
| EventStatus | Where the event status is returned. The following bits may be set:<br>ACPI_EVENT_FLAG_ENABLED<br>ACPI_EVENT_FLAG_WAKE_ENABLED |

ACPI_EVENT_FLAG_SET

**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.
- The *EventStatus* pointer is NULL or invalid

**Functional Description:**

This function obtains the current status of a single ACPI event. Status for both Fixed Events and General Purpose Events may be obtained with this interface.

### 5.7.7 AcpiInstallFixedEventHandler

**Install a handler for ACPI Fixed Events.**

```
ACPI_STATUS
AcpiInstallFixedEventHandler (
    ACPI_EVENT_TYPE          Event,
    ACPI_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.

### 5.7.7.1 Interface to Fixed Event Handlers

**Definition of the handler interface for Fixed Events.**

```
typedef
UINT32 (*ACPI_EVENT_HANDLER) (
    void                        *Context)
```

**PARAMETERS**

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

**RETURN VALUE**

|  |  |
|---|---|
| ??? | TBD. |

**Functional Description:**

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

3) This function executes in the context of an interrupt handler.

## 5.7.8 AcpiRemoveFixedEventHandler

**Remove an ACPI Fixed Event handler.**

```
ACPI_STATUS
AcpiRemoveFixedEventHandler (
    ACPI_EVENT_TYPE             Event,
    ACPI_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*.

### 5.7.9 AcpiInstallGpeHandler

**Install a handler for ACPI General Purpose Events.**

```
ACPI_STATUS
AcpiInstallGpeHandler (
    UINT32                  GpeNumber,
    ACPI_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:<br>• The *GpeNumber* is invalid.<br>• 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

### 5.7.9.1 Interface to General Purpose Event Handlers

**Definition of the handler interface for General Purpose Events.**

```
typedef
void (*ACPI_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.

4) This function executes in the context of an interrupt handler.

## 5.7.10 AcpiRemoveGpeHandler

**Remove an ACPI General-Purpose Event handler.**

```
ACPI_STATUS
AcpiRemoveGpeHandler (
    UINT32                      GpeNumber,
    ACPI_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*.

## 5.7.11 AcpiInstallNotifyHandler

**Install a handler for notification events on an ACPI object.**

```
ACPI_STATUS
AcpiInstallNotifyHandler (
    ACPI_HANDLE              Object,
    UINT32                   Type,
    ACPI_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)

#### 5.7.11.1 Interface to Notification Event Handlers

**Definition of the handler interface for Notification Events.**

```
typedef
void (*ACPI_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.

5) This function **does not** execute in the context of an interrupt handler.

### 5.7.12 AcpiRemoveNotifyHandler

**Remove a handler for ACPI notification events.**

```
ACPI_STATUS
AcpiRemoveNotifyHandler (
    ACPI_HANDLE                 Object,
    UINT32                      Type,
    ACPI_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:<br>ACPI_SYSTEM_NOTIFY – Notifications 0x00 to 0x7F<br>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:<br>• The Object handle is invalid.<br>• The Handler pointer is NULL.<br>• 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.*

### 5.7.13 AcpiInstallAddressSpaceHandler

**Install handlers for ACPI Operation Region events.**

```
ACPI_STATUS
AcpiInstallAddressSpaceHandler (
    ACPI_HANDLE                 Device,
    UINT32                      SpaceId,
    ACPI_ADR_SPACE_HANDLERHandler,
    ACPI_ADR_SPACE_SETUP        Setup,
    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:<br><br>ACPI_TYPE_Device,<br><br>ACPI_TYPE_Processor,<br><br>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. |
| Setup | Address of a start/stop initialization/termination function that is called when the region first becomes available and also if and when it becomes unavailable. |
| 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:<br><br>• The Device handle does not refer to an object of type Device, Processor, ThermalZone, or  the root object.<br><br>• 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 that 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.

### 5.7.13.1 Interface to Address Space Setup Handlers

**Definition of the setup (Address Space start/stop) handler interface for Operation Region Events.**

```
typedef
void (*ACPI_ADR _SPACE_SETUP) (
    ACPI_HANDLE                 RegionHandle,
    UINT32                      Function
    Void                        *HandlerContext)
    Void                        **ReturnContext)
```

**PARAMETERS**

| | |
|---|---|
| RegionHandle | Handle to the region that is initializing or terminating |
| Function | The type of function to be performed; must be one of the following manifest constants:<br>ACPI_REGION_ACTIVATE (init)<br>ACPI_REGION_DEACTIVATE (terminate) |
| HandlerContext | An address space specific Context value. Typically this is the context that was passed as a parameter to the AcpiInstallAddressSpaceHandler function. |
| ReturnContext | An address space specific Context value. This context subsumes the HandlerContext, and this is the context value that is passed to the actual address space handler routine |

**RETURN VALUE**

None

**Functional Description:**

This handler is installed via *AcpiInstallAddressSpaceHandler*. It is invoked to both initialize and terminate the operation region handling code. The setup handler is first invoked with a function value of ACPI_REGION_ACTIVATE upon the first access to the region from AML code. It is called again with a function value of ACPI_REGION_DEACTIVATE just before the address space handler is removed.

6) This function **does not** execute in the context of an interrupt handler.

### 5.7.13.2 Interface to Address Space Handlers

**Definition of the handler interface for Operation Region Events.**

```
typedef
void (*ACPI_ADR _SPACE_HANDLER) (
    UINT32                      Function,
    UINT32                      Address,
    UINT32                      BitWidth,
    ACPI_INTEGER                *Value,
    Void                        *Context)
```

**PARAMETERS**

| | |
|---|---|
| Function | The type of function to be performed; must be one of the following manifest constants:<br><br>ADDRESS_SPACE_READ<br><br>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.

7) This function **does not** execute in the context of an interrupt handler.

#### 5.7.13.3 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.

### 5.7.14 AcpiRemoveAddressSpaceHandler

**Remove an ACPI Operation Region handler.**

```
ACPI_STATUS
AcpiRemoveAddressSpaceHandler (
    UINT32                     SpaceId,
    ACPI_ADR _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:<br>• The SpaceId is invalid.<br>• The Handler pointer is NULL.<br>• 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*.

## 5.8 ACPI Hardware Management

### 5.8.1 AcpiGetRegister

**Get the contents of an ACPI bit-defined Register.**

```
ACPI_STATUS
AcpiGetRegister (
    UINT32              RegisterId,
    UINT32              *Value,
    UINT32              Flags)
```

**PARAMETERS**

| | |
|---|---|
| RegisterId | The ID of the desired register, one of the following manifest constants:<br>ACPI_BITREG_TIMER_STATUS<br>ACPI_BITREG_BUS_MASTER_STATUS<br>ACPI_BITREG_GLOBAL_LOCK_STATUS<br>ACPI_BITREG_POWER_BUTTON_STATUS<br>ACPI_BITREG_SLEEP_BUTTON_STATUS<br>ACPI_BITREG_RT_CLOCK_STATUS<br>ACPI_BITREG_WAKE_STATUS<br>ACPI_BITREG_TIMER_ENABLE<br>ACPI_BITREG_GLOBAL_LOCK_ENABLE<br>ACPI_BITREG_POWER_BUTTON_ENABLE<br>ACPI_BITREG_SLEEP_BUTTON_ENABLE<br>ACPI_BITREG_RT_CLOCK_ENABLE<br>ACPI_BITREG_WAKE_ENABLE<br>ACPI_BITREG_SCI_ENABLE<br>ACPI_BITREG_BUS_MASTER_RLD<br>ACPI_BITREG_GLOBAL_LOCK_RELEASE<br>ACPI_BITREG_SLEEP_TYPE_A<br>ACPI_BITREG_SLEEP_TYPE_B<br>ACPI_BITREG_SLEEP_ENABLE<br>ACPI_BITREG_ARB_DISABLE. |
| Value | A pointer to a location where the data is to be returned. |
| Flags | Indicates whether the ACPI hardware should be locked or not. If calling this interface with interrupts disabled, use: ACPI_MTX_DO_NOT_LOCK. Otherwise, use ACPI_MTX_LOCK. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The register was read successfully. |
| AE_BAD_PARAMETER | Invalid RegisterId. |
| Other | The function failed at the operating system level. |

**Functional Description:**

This function reads the bit register specified in the RegisterId. The value returned is normalized to bit zero. Can be used with interrupt enabled or disabled.

## 5.8.2 AcpiSetRegister

**Get the contents of an ACPI bit-defined Register.**

```
ACPI_STATUS
AcpiSetRegister (
    UINT32                     RegisterId,
    UINT32                     Value,
    UINT32                     Flags)
```

**PARAMETERS**

| | |
|---|---|
| RegisterId | The ID of the desired register, one of the following manifest constants:<br>ACPI_BITREG_TIMER_STATUS<br>ACPI_BITREG_BUS_MASTER_STATUS<br>ACPI_BITREG_GLOBAL_LOCK_STATUS<br>ACPI_BITREG_POWER_BUTTON_STATUS<br>ACPI_BITREG_SLEEP_BUTTON_STATUS<br>ACPI_BITREG_RT_CLOCK_STATUS<br>ACPI_BITREG_WAKE_STATUS<br>ACPI_BITREG_TIMER_ENABLE<br>ACPI_BITREG_GLOBAL_LOCK_ENABLE<br>ACPI_BITREG_POWER_BUTTON_ENABLE<br>ACPI_BITREG_SLEEP_BUTTON_ENABLE<br>ACPI_BITREG_RT_CLOCK_ENABLE<br>ACPI_BITREG_WAKE_ENABLE<br>ACPI_BITREG_SCI_ENABLE<br>ACPI_BITREG_BUS_MASTER_RLD<br>ACPI_BITREG_GLOBAL_LOCK_RELEASE<br>ACPI_BITREG_SLEEP_TYPE_A<br>ACPI_BITREG_SLEEP_TYPE_B<br>ACPI_BITREG_SLEEP_ENABLE<br>ACPI_BITREG_ARB_DISABLE. |
| Value | The data to be written. |

| | |
|---|---|
| Flags | Indicates whether the ACPI hardware should be locked or not. If calling this interface with interrupts disabled, use: ACPI_MTX_DO_NOT_LOCK. Otherwise, use ACPI_MTX_LOCK. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The register was read successfully. |
| AE_BAD_PARAMETER | Invalid RegisterId. |
| Other | The function failed at the operating system level. |

**Functional Description:**

This function reads the bit register specified in the RegisterId. The value written must be normalized to bit zero before calling. Can be used with interrupt enabled or disabled.

### 5.8.3 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_NO_ACPI_TABLES | The FACS is not loaded or could not be found |

**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.

## 5.8.4 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. |
| AE_NO_ACPI_TABLES | The FACS is not loaded or could not be found |

**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.

## 5.8.5 AcpiGetSleepTypeData

**Get the SLP_TYP data for the requested sleep state.**

```
ACPI_STATUS
AcpiGetSleepTypeData (
    UINT8                               SleepState,
    UINT8                               *SleepTypeA,
    UINT8                               *SleepTypeB)
```

**PARAMETERS**

| | |
|---|---|
| SleepState | The SleepState value (0 through 5) for which the SLP_TYPa and SLP_TYPb values will be returned. |
| SleepTypeA | A pointer to a location where the value of SLP_TYPa will be returned. |
| SleepTypeB | A pointer to a location where the value of SLP_TYPb will be returned. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | Both SLP_TYP values were returned successfully. |
| AE_BAD_PARAMETER | Either SleepState has an invalid value, or one of the SleepType pointers is invalid. |
| AE_AML_NO_OPERAND | Could not locate one or more of the SLP_TYP values. |
| AE_AML_OPERAND_TYPE | One or more of the SLP_TYP objects was not a numeric type. |

**Functional Description:**

This function returns the SLP_TYP object for the requested sleep state.

### 5.8.6 AcpiEnterSleepStatePrep

**Prepare to enter a system sleep state (S1-S5).**

```
ACPI_STATUS
AcpiEnterSleepStatePrep (
    UINT8                       SleepState)
```

**PARAMETERS**

| | |
|---|---|
| SleepState | The sleep state to prepare to enter. Must be in the range 1 through 5. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The PTS and GTS methods were successfully run |
| Other | Exception from AcpiEvaluateObject. |

**Functional Description:**

Prepare to enter a system sleep state. .

This function evaluates the _PTS and _GTS methods.

### 5.8.7 AcpiEnterSleepState

**Enter a system sleep state (S1-S5).**

```
ACPI_STATUS
AcpiEnterSleepState (
    UINT8                     SleepState)
```

**PARAMETERS**

| | |
|---|---|
| SleepState | The sleep state to enter. Must be in the range 1 through 5. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The sleep state (S1) was successfully entered. |
| Other | Hardware access exception. |

**Functional Description:**

This function only returns for transitions to the S1 state or when an error occurs. Sleep states S2-S4 use the firmware waking vector during wakeup.

This function must be called with interrupts disabled.

### 5.8.8 AcpiLeaveSleepState

**Leave (cleanup) a system sleep state (S1-S5).**

```
ACPI_STATUS
AcpiLeaveSleepState (
    UINT8                     SleepState)
```

**PARAMETERS**

| | |
|---|---|
| SleepState | The sleep state to leave. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The cleanup was successful. |
| Other | Hardware access exception. |

**Functional Description:**

Perform cleanup after leaving a sleep stae. .

## 5.8.9 AcpiAcquireGlobalLock

**Acquire the ACPI Global Lock.**

```
ACPI_STATUS
AcpiAcquireGlobalLock (
    UINT32                  Timeout,
    UINT32                  *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.

## 5.8.10 AcpiReleaseGlobalLock

**Release the ACPI Global Lock.**

**ACPI_STATUS**
**AcpiReleaseGlobalLock (**
**UINT32 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*.

## 5.8.11 AcpiGetTimer

**Get the current value of the ACPI Power Management 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 PT Timer (in ticks).

### 5.8.12 AcpiGetTimerResolution

**Get the resolution of the ACPI Power Management Timer.**

```
ACPI_STATUS
AcpiGetTimerResolution (
    UINT32                    *OutValue)
```

**PARAMETERS**

| | |
|---|---|
| OutValue | A pointer to where the current value of the PM Timer resolution is to be returned. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The PM Timer resolution was successfully retrieved and returned. |
| AE_BAD_PARAMETER | The OutValue pointer is NULL. |

**Functional Description:**

This function returns the PM Timer resolution – either 24 (for 24-bit) or 32 (for 32-bit timers).

### 5.8.13 AcpiGetTimerDuration

**Calculates the time elapsed (in microseconds) between two values of the ACPI Power Management Timer.**

```
ACPI_STATUS
AcpiGetTimer (
    UINT32                    StartTicks,
    UINT32                    EndTicks,
    UINT32                    *OutValue)
```

**PARAMETERS**

| | |
|---|---|
| StartTicks | The value of the PM Timer at the start of a time measurement (obtained by calling AcpiGetTimer). |

| | |
|---|---|
| EndTicks | The value of the PM Timer at the end of a time measurement (obtained by calling AcpiGetTimer). |
| OutValue | A pointer to where the elapsed time (in microseconds) is to be returned. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The time elapsed was successfully calculated and returned. |
| AE_BAD_PARAMETER | The OutValue pointer is NULL. |

**Functional Description:**

This function calculates and returns the time elapsed (in microseconds) between StartTicks and EndTicks, taking into consideration the PM Timer frequency, resolution, and counter rollovers.

- AcpiWriteRegister

# 6 OS Services Layer - External Interface Definition

This section contains the definitions of the interfaces that must be exported by the OS Services Layer. The ACPI Core Subsystem requires that all of these interfaces be present. All interfaces to the OS Services Layer *that are intended for use by the ACPI Core Subsystem* are prefixed by the letters "**AcpiOs".**

## 6.1 Environmental

### 6.1.1 AcpiOsInitialize

**Initialize the OSL subsystem**

**void ***
**AcpiOsInitialize (void)**

**PARAMETERS**

None

**RETURN VALUE**

None

**Functional Description:**

This function allows the OSL to initialize itself. It is called during initialization of the ACPI subsystem.

### 6.1.2 AcpiOsTerminate

**Terminate the OSL subsystem**

**void ***
**AcpiOsTerminate (void)**

**PARAMETERS**

None

**RETURN VALUE**

None

**Functional Description:**

This function allows the OSL to cleanup and terminate. It is called during termination of the ACPI subsystem.

### 6.1.3 AcpiOsGetRootPointer

**Obtain the Root ACPI table pointer (RSDP)**

```
ACPI_STATUS
AcpiOsGetRootPointer (
    UINT32                  Flags,
    ACPI_POINTER            **Address)
```

**PARAMETERS**

| | |
|---|---|
| Flags | Current addressing mode of the processor – whether paging is currently enabled or not – one of these manifest constants:<br><br>ACPI_PHYSICAL_ADDRESSING<br><br>ACPI_LOGICAL_ADDRESSING |
| Address | Where the pointer to the RSDP table is returned. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

This function returns the physical address of the.ACPI RSDP (Root System Description Pointer) table. The mechanism used to obtain this pointer is platform and/or OS dependent. There are two primary methods used to obtain this pointer and thus implement this interface:

1) On IA-32 platforms, the RSDP is obtained by searching the first megabyte of physical memory for the RSDP signature (“RSD PTR “). On these platforms, this interface should be implemented via a call to the *AcpiFindRootPointer* interface.

2) On IA-64 platforms, the RSDP is obtained from the EFI (Extended Firmware Interface). The pointer in the EFI information block that is passed to the OS at OS startup.

### 6.1.4 AcpiOsTableOverride

**Allow the host OS to override a firmware ACPI table**

```
ACPI_STATUS
AcpiOsTableOverride (
    ACPI_TABLE_HEADER          *ExistingTable,
    ACPI_TABLE_HEADER          **NewTable)
```

**PARAMETERS**

| | |
|---|---|
| ExistingTable | A pointer to the existing ACPI table header. |
| NewTable | Where the pointer to the replacement table is returned. The OSL returns NULL if no replacement is provided. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

This function allows the host to override the ACPI table that was found in the firmware. NOTE: Currently, only the DSDT can be replaced. The OS can examine the header for table signature and version number and decide to replace it if desired.

## 6.2 Memory Management

These interfaces provide an OS-independent memory management interface.

### 6.2.1 AcpiOsMapMemory

**Map physical memory into the caller's address space.**

```
ACPI_STATUS
AcpiOsMapMemory (
    ACPI_PHYSICAL_ADDRESS      PhysicalAddress,
    ACPI_SIZE                  Length,
    Void                       **LogicalAddress)
```

**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. |
| LogicalAddress | Where the pointer to the mapped memory is returned. Only valid if the return status is AE_OK |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**EXCEPTIONS**

| | |
|---|---|
| AE_OK | The physical address was successfully mapped to the returned logical address. |
| AE_BAD_ADDRESS | The physical address does not refer to valid memory on the system. |
| AE_NO_MEMORY | There was insufficient memory to allocate the necessary page tables to complete the operation. |

**Functional Description:**

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

### 6.2.2 AcpiOsUnMapMemory

**Remove a physical to logical memory mapping.**

```
void
AcpiOsUnMapMemory (
    Void                        *LogicalAddress,
    ACPI_SIZE                   Length)
```

**PARAMETERS**

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

**RETURN VALUE**

None

**Functional Description:**

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

### 6.2.3 AcpiOsGetPhysicalAddress

**Translate a logical address to a physical address.**

**ACPI_STATUS**
**AcpiOsGetPhysicalAddress (**

| | |
|---|---|
| **Void** | ***LogicalAddress,** |
| **ACPI_PHYSICAL_ADDRESS** | ***PhysicalAddress)** |

**PARAMETERS**

| | |
|---|---|
| LogicalAddress | The logical address to be translated. |
| PhysicalAddress | The physical memory address of the logical address. |

**RETURN VALUE**

| | |
|---|---|
| AE_OK | The logical address translation was successfully. |
| AE_ERROR | An error occurred in the translation system call. |
| AE_BAD_PARAMETER | One or both of the parameters are NULL, no translation was attempted. |

**Functional Description:**

This function translates a logical address to its physical address location.

### 6.2.4 AcpiOsAllocate

**Allocate memory from the dynamic memory pool.**

**void ***
**AcpiOsAllocate (**

| | |
|---|---|
| **ACPI_SIZE** | **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 is not assumed to be initialized to any particular value or values.

## 6.2.5 AcpiOsFree

**Free previously allocated memory.**

```
void
AcpiOsFree (
    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 *AcpiOsAllocate*.

## 6.2.6 AcpiOsReadable

**Check if a memory region is readable**

```
BOOLEAN
AcpiOsReadable (
    void                            *Memory
    UINT32                          Length)
```

**PARAMETERS**

| | |
|---|---|
| Memory | A pointer to the memory region to be checked. |
| Length | The length of the memory region, in bytes. |

**RETURN VALUE**

| | |
|---|---|
| TRUE | If the entire memory region is readable without faults |
| FALSE | If one or more bytes within the region are unreadable |

**Functional Description:**

This function validates that a pointer to a memory region is valid and the entire region is readable. Used to validate input parameters to the ACPI subsystem.

### 6.2.7 AcpiOsWritable

**Check if a memory region is writable (and readable)**

```
BOOLEAN
AcpiOsWritable (
    void                          *Memory,
    UINT32                        Length)
```

**PARAMETERS**

| | |
|---|---|
| Memory | A pointer to the memory region to be checked. |
| Length | The length of the memory region, in bytes. |

**RETURN VALUE**

| | |
|---|---|
| TRUE | If the entire memory region is both readable and writable without faults |
| FALSE | If one or more bytes within the region are unreadable or unwritable |

**Functional Description:**

This function validates that a pointer to a memory region is valid and the entire region is both writable and readable. Used to validate input parameters to the ACPI subsystem..

## 6.3 Multithreading and Scheduling Services

### 6.3.1 AcpiOsGetThreadId

**Obtain the ID of the currently executing thread**

```
UINT32
AcpiOsGetThreadId (
    void)
```

**PARAMETERS**

None

**RETURN VALUE**

| | |
|---|---|
| ThreadId | A unique value that represents the ID of the currently executing thread. For single threaded implementations, a constant integer is acceptable. The value 0xFFFFFFFF (-1) is reserved and must not be returned by this interface. |

**Functional Description:**

This function returns the ID of the currently executing thread. The value must be non-zero and must be unique to the executing thread.

### 6.3.2 AcpiOsQueueForExecution

**Schedule a procedure for deferred execution.**

```
ACPI_STATUS
AcpiOsQueueForExecution (
    UINT32                      Priority,
    OSL_EXECUTION_CALLBACK      Function,
    Void                        *Context);)
```

**PARAMETERS**

| | |
|---|---|
| Priority | Requested priority of the execution – one of these manifest constants:<br>OSL_PRIORITY_HIGH<br>OSL_PRIORITY_MED<br>OSL_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:<br>• The *Priority* is invalid.<br>• The *Function* pointer is `NULL`. |

**Functional Description:**

This function queues a procedure for later scheduling and execution.

## 6.3.3 AcpiOsSleep

**Suspend the running task (course granularity).**

```
ACPI_STATUS
AcpiOsSleep (
    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 | TBD!!!! |
| 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.

## 6.3.4 AcpiOsStall

**Suspend the running task (fine granularity).**

```
ACPI_STATUS
AcpiOsSleepUsec (
    UINT32                  Microseconds)
```

**PARAMETERS**

| | |
|---|---|
| Microseconds | The amount of time to delay 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_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.

## 6.4 Mutual Exclusion and Synchronization

Thread synchronization and locking.

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.

### 6.4.1 AcpiOsCreateSemaphore

**Create a semaphore.**

```
ACPI_STATUS
AcpiOsCreateSemaphore (
    UINT32                      MaxUnits,
    UINT32                      InitialUnits,
    ACPI_HANDLE                 *OutHandle)
```

**PARAMETERS**

| | |
|---|---|
| MaxUnits | The maximum number of units this semaphore will be required to accept. |
| 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:<br>• The *InitialUnits* is invalid.<br>• The *OutHandle* pointer is `NULL`. |
| AE_NO_MEMORY | Insufficient memory to create the semaphore. |

**Functional Description:**

Create a standard semaphore. The *MaxUnits* parameter allows the semaphore to be tailored to specific uses. For example, a *MaxUnits* value of one indicates that the semaphore is to be used as a *mutex*. The underlying OS object used to implement this semaphore may be different than if *MaxUnits* is greater than one (thus indicating that the semaphore will be used as a general purpose semaphore.) The ACPI Core Subsystem creates semaphores of both the mutex and general-purpose variety.

### 6.4.2 AcpiOsDeleteSemaphore

**Delete a semaphore.**

```
ACPI_STATUS
AcpiOsDeleteSemaphore (
    ACPI_HANDLE                 Handle)
```

**PARAMETERS**

| | |
|---|---|
| Handle | A handle to a semaphore object that was returned by a previous call to *AcpiOsCreateSemaphore*. |

**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.

### 6.4.3 AcpiOsWaitSemaphore

**Wait for units from a semaphore.**

```
ACPI_STATUS
AcpiOsWaitSemaphore (
    ACPI_HANDLE                 Handle,
    UINT32                      Units,
    UINT32                      Timeout)
```

**PARAMETERS**

| | |
|---|---|
| Handle | A handle to a semaphore object that was returned by a previous call to *AcpiOsCreateSemaphore*. |

| | |
|---|---|
| Units | The number of units the caller is requesting. |
| Timeout | How long the caller is willing to wait for the requested units. The timeout is specified in milliseconds. A value of 0xFFFFFFFF (-1) indicates that the calling thread is willing to wait forever. |

**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.

Implementation notes:

1. The implementation of this interface must support timeout values of zero. This is frequently used to determine if a call to the interface with an actual timeout value would block. In this case, AcpiOsWaitSemaphore must return either an E_OK if the units were obtained immediately, or an AE_TIME to indicate that the requested units are not available. Single threaded OSL implementations should always return AE_OK for this interface.
2. The implementation must also support arbitrary timed waits in order for ASL functions such as *Wait()* to work properly.

### 6.4.4 AcpiOsSignalSemaphore

**Send units to a semaphore.**

```
ACPI_STATUS
AcpiOsSignalSemaphore (
    ACPI_HANDLE          Handle,
    UINT32               Units)
```

**PARAMETERS**

| | |
|---|---|
| Handle | A handle to a semaphore object that was returned by a previous call to *AcpiOsCreateSemaphore*. |
| 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. |
| AE_LIMIT | The semaphore has already been signaled MaxUnits times. No more units can be accepted. |

**Functional Description:**

Send the requested number of units to a semaphore. Single threaded OSL implementations should always return AE_OK for this interface.

## 6.5 Interrupt Handling

Interrupt handler installation and removal.

### 6.5.1 AcpiOsInstallInterruptHandler

**Install a handler for a hardware interrupt level.**

```
ACPI_STATUS
AcpiOsInstallInterruptHandler (
    UINT32                  InterruptLevel,
    OSL_HANDLER             Handler,
    void                    *Context)
```

**PARAMETERS**

| | |
|---|---|
| InterruptLevel | 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:<br>• The *InterruptNumber* is invalid.<br>• 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.

#### 6.5.1.1 Interface to OS-independent Interrupt Handlers

**Definition of the interface for OS-independent interrupt handlers.**

```
typedef
UINT32 (*OSL_HANDLER) (
    Void                          *Context)
```

**PARAMETERS**

| | |
|---|---|
| Context | The Context value that was passed as a parameter to the AcpiOsInstallInterruptHandler 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 OSL interrupt handler “wrapper” that exists within the OS Services Layer. It is the responsibility of the OS Services Layer to manage the installed interrupt handler(s), and dispatch interrupts to the handler(s) appropriately.

### 6.5.2 AcpiOsRemoveInterruptHandler

**Remove an interrupt handler.**

```
ACPI_STATUS
AcpiOsRemoveInterruptHandler (
    UINT32                        InterruptNumber,
    OSL_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:<br>• The *InterruptNumber* is invalid.<br>• The *Handler* pointer is `NULL`.<br>• 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.

## 6.6 Stream I/O

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

### 6.6.1 AcpiOsPrintf

**Formatted stream output.**

```
INT32
AcpiOsPrintf (
    OSL_FILE            *Stream,
    const char          *Format,
    …                   <variable argument list>)
```

**PARAMETERS**

| | |
|---|---|
| Stream | Open stream to write to. NULL is defined to be the debugger output channel. |
| Format | A standard print 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 OSL stream.

### 6.6.2 AcpiOsVprintf

**Formatted stream output.**

```
INT32
AcpiOsVprintf (
    OSL_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 OSL stream via the va_list argument format.

## 6.7 Address Space Access: Port Input/Output

These interfaces allow the OS Services Layer 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 Services Layer itself, or these calls may be translated into additional OS calls — such as calls to a Hardware Abstraction Component.

### 6.7.1 AcpiOsReadPort

**Read a value from an input port.**

```
ACPI_STATUS
AcpiOsReadPort (
    ACPI_IO_ADDRESS             Addresss,
    ACPI_INTEGER                *Value,
    UINT32                      Width)
```

**PARAMETERS**

| | |
|---|---|
| Address | Hardware I/O port address to read from. |
| Value | A pointer to a location where the data is to be returned. |
| Width | The port width in bits, either 8, 16, 32, or 64. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

This function reads data from the specified input port. The data is zero extended to fill the 32-bit return value even if the bit width of the port is less than 32.

### 6.7.2 AcpiOsWritePort

**Write a value to an output port.**

```
ACPI_STATUS
AcpiOsWritePort (
    ACPI_IO_ADDRESS          Addresss,
    ACPI_INTEGER             Value,
    UINT32                   Width)
```

**PARAMETERS**

| | |
|---|---|
| Address | Hardware I/O port address to read from. |
| Value | The value to be written. |
| Width | The port width in bits, either 8, 16, 32, or 64. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

This function writes data to the specified input port. If the bit width of the port is less than 32, only the lower significant bits of the Value are written.

## 6.8 Address Space Access: Memory and Memory Mapped I/O

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

These calls are used by the ACPI CA for small amounts of data transfer only, such as memory mapped I/O. For large transfers (such as reading the ACPI tables), the ACPI CA code will call *AcpiOsMapMemory* instead.

### 6.8.1 AcpiOsReadMemory

**Read a value from a memory location**

```
ACPI_STATUS
AcpiOsReadMemory (
    ACPI_PHYSICAL_ADDRESS     Address,
    ACPI_INTEGER              *Value,
    UINT32                    Width)
```

**PARAMETERS**

| | |
|---|---|
| Address | Memory address to be read. |
| Value | A pointer to a location where the data is to be returned. |
| Width | The memory width in bits, either 8, 16, 32, or 64. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

This function is used to read a data from the specified memory location. The data is zero extended to fill the 32-bit return value even if the bit width of the location is less than 32.

### 6.8.2 AcpiOsWriteMemory

**Write a value to a memory location.**

```
ACPI_STATUS
AcpiOsWriteMemory (
    ACPI_PHYSICAL_ADDRESS     Address,
    ACPI_INTEGER              Value,
    UINT32                    Width)
```

**PARAMETERS**

| | |
|---|---|
| Address | Memory address where data is to be written. |
| Value | Data to be written to the memory location. |
| Width | The memory width in bits, either 8, 16, 32, or 64. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

This function writes data to the specified memory location. If the bit width of the memory location is less than 32, only the lower significant bits of the Value are written.

## 6.9 Address Space Access: PCI Configuration Space

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

### 6.9.1 AcpiOsReadPciConfiguration

**Read a value from a PCI configuration register.**

```
ACPI_STATUS
AcpiOsReadPciConfiguration (
    ACPI_PCI_ID              PciId,
    UINT32                   Register,
    ACPI_INTEGER             *Value,
    UINT32                   Width)
```

**PARAMETERS**

| | |
|---|---|
| PciId | The full PCI configuration space address, consisting of a segment number, bus number, device number, and function number. |
| Register | The PCI register address to be read from. |
| Value | A pointer to a location where the data is to be returned. |
| Width | The register width in bits, either 8, 16, 32, or 64. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

This function reads data from the specified PCI configuration port. The data is zero extended to fill the 32-bit return value even if the bit width of the location is less than 32.

## 6.9.2 AcpiOsWritePciConfiguration

**Write a value to a PCI configuration register.**

**ACPI_STATUS**
**AcpiOsWritePciConfiguration (**

| | |
|---|---|
| **ACPI_PCI_ID** | **PciId,** |
| **UINT32** | **Register,** |
| **ACPI_INTEGER** | **Value,** |
| **UINT32** | **Width)** |

**PARAMETERS**

| | |
|---|---|
| PciId | The full PCI configuration space address, consisting of a segment number, bus number, device number, and function number. |
| Register | The PCI register address to be written to. |
| Value | Data to be written. |
| Width | The register width in bits, either 8, 16, 32, or 64. |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

This function writes data to the specified PCI configuration port. If the bit width of the register is less than 32, only the lower significant bits of the Value are written.

## 6.10 Miscellaneous

### 6.10.1 AcpiOsSignal

**Break to the debugger or display a breakpoint message**

**ACPI_STATUS**
**AcpiOsSignal (**

| | |
|---|---|
| **UINT32** | **Function,** |
| **void** | ***Info)** |

**PARAMETERS**

| | |
|---|---|
| Function | Signal to be sent to the host operating system – one of these manifest constants: |

ACPI_SIGNAL_FATAL

ACPI_SIGNAL_BREAKPOINT

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

This function is used to pass various signals and notifications to the host operating system.

#### 6.10.1.1 ACPI_SIGNAL_FATAL

This signal corresponds to the AML **Fatal** opcode. It is sent to the host OS only when this opcode is encountered in the AML stream. The host OS may or may not return control from this signal.

The definition of the Info structure for this signal is as follows:

```
typedef struct AcpiFatalInfo
{
    UINT32                  Type;
    UINT32                  Code;
    UINT32                  Argument;

} ACPI_SIGNAL_FATAL_INFO;
```

#### 6.10.1.2 ACPI_SIGNAL_BREAKPOINT

This signal corresponds to the AML **Breakpoint** opcode. The OSL implements a “Breakpoint” operation as appropriate for the host OS. If in debug mode, this interface may cause a break into the host kernel debugger.

The definition of the Info structure for this signal is as follows:

```
char                 *BreakpointMessage;
```

### 6.10.2 AcpiOsGetLine

**Get a input line of data**

**ACPI_STATUS**
**AcpiOsGetLine (**
**char** ***Buffer)**

**PARAMETERS**

| | |
|---|---|
| Message | A message string related to the breakpoint |

**RETURN VALUE**

| | |
|---|---|
| Status | Exception code that indicates success or reason for failure. |

**Functional Description:**

The purpose of this function is to support the ACPI Debugger, and it is therefore optional depending on whether ACPI debugger support is desired.

# 7 ACPI Debugger

## 7.1 Overview

The ACPI/AML Debugger is an optional subcomponent of the ACPI CA Core Subsystem. It can be operated standalone or in conjunction with (or as an extension of) a native kernel debugger. The debugger provides the ability to load ACPI tables, dump internal data structures, execute control methods, disassemble control methods, single step control methods, and set breakpoints within control methods.

## 7.2 Supported Environments

The debugger can be executed in a ring 0 (kernel) or ring 3 (application) environment. The following combinations of debugger and front-end (user-interface) are supported:

- Ring 0 Debugger, Ring 0 Front-End: In this case, the front-end is a host kernel debugger, and the Debugger operates as an extension to the host debugger.
- Ring 0 Debugger, Ring 3 Front-End. In this mode, the front-end is a ring 3 application that obtains the command lines from the user and sends them to the debugger executing in Ring 0. The actual mechanism used for this communication is dependent on the host operating system.
- Ring 3 Debugger, Ring 3 Front-End. In this mode, the entire ACPI CA subsystem (including the debugger) resides in a Ring 3 application. A single thread can be used for the user interface, debugger, and AML control method execution.

### 7.2.1 The AcpiExec Utility

An example of the Ring3/Ring3 model of execution is the Windows-based *AcpiExec* utility. This Windows application includes the entire ACPI CA subsystem (including the Debugger) and allows the user to load ACPI tables from files and execute methods contained in the tables.

Of course, hardware and memory access from Ring 3 is very limited.

## 7.3 Debugger Architecture

The ACPI debugger consists of the following architectural elements:

- A command line interpreter that receives entire command lines from the host, parses them into commands and parameters, and dispatches the request to the appropriate handler for the command.
- A group of modules that implement the various debugger commands.

- A group of callback routines that are invoked by the interpreter/dispatcher during the execution control methods. These callbacks enable the single stepping of control methods and the display of arguments to each executed control method.

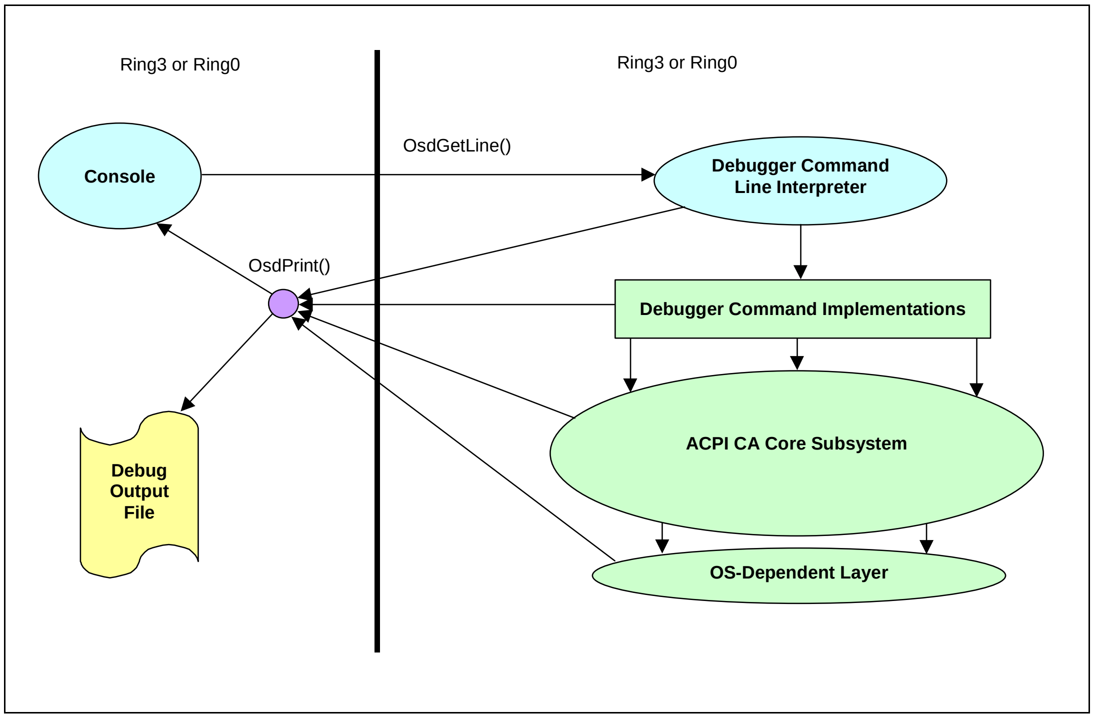
When executing in a Ring 0 environment, the debugger initialization creates a separate thread for the debugger CLI. This threads performs the following tasks until the debugger is shut down:

1. Wait for a command line by calling the *AcpiOsGetLine* interface
2. Execute the command

All output from the debugger is via the AcpiOsPrint and AcpiOsVprintf interfaces.

The overall architecture of the ACPI Debugger is shown in the diagram below. Note how the Debugger CLI uses the *AcpiOsGetLine* interface to obtain user command lines, and how output from the entire debugger and ACPI subsystem can be directed to the console, a file, or both via the implementation of the *AcpiOsPrint* interface within the OSL layer. Also note how the debugger and ACPI subsystem can reside in a different protection ring than the user console support and file I/O support.

**Figure 9. ACPI Debugger Architecture**



## 7.4 Configuration and Installation

The basic idea behind the debugger thread is that it receives a command line from *somewhere* and then asynchronously executes it. The command line can come from a ring 3 application (a debugger front-end), or it can come from the resident kernel debugger (you would install a debugger extension that forwards command lines to the debugger.)

With this in mind, there are several key components of the debugger:

1. **DbInitialize** – Initializes the debugger semaphores and creates the debugger thread, DbExecuteThread
2. **DbCommandDispatch** – This is the actual command execution code
3. **DbExecuteThread** – Waits for a command to become available (as indicated by the MTX_DEBUG_CMD_READY mutex), executes the command,(via DbCommandDispatch), then signals command completion via the MTX_DEBUG_CMD_COMPLETE mutex.
4. **DbUserCommands** – An example command loop that must execute in its own thread (this is the *caller* thread, not a thread that is part of the debugger). This loop obtains a command line via AcpiOsGetLine, puts it into the LineBuf buffer, and signals the DbExecuteThread that a command line is available. It is not necessary to use this procedure, however, if command lines become available from somewhere besides AcpiOsGetLine.
5. **DbSingleStep** – Called from the dispatcher just before an AML opcode is executed. Implements its own command loop that obtains command lines from either the MTX_DEBUG_CMD_READY mutex (multi-thread mode), or by calling AcpiOsGetLine directly (single thread mode). Drops out of the loop when the control method is aborted or is allowed to continue running (perhaps just to the next opcode…)

This is the basic thread model and handshake with the outside world. To integrate the debugger into a specific environment, it is your responsibility to get command lines to the DbExecuteThread via the LineBuf and the MTX_DEBUG_CMD_READY mutex. Alternatively, you can just call the DbCommandDispatch directly if you don't need an asynchronous debugger thread. Additional explanation follows.

The AcpiExec Ring3 application uses DbUserCommands to process command lines (DbUserCommands is actually called from aemain.c). However, if integrating with a kernel debugger, you will probably want to implement your own mechanism instead of using the DbUserCommands loop. I would imagine this would entail the following:

1. Install a small extension to the kernel debugger that receives command lines intended for that extension.
2. Copy the command line to the LineBuf.
3. Signal the DbExecuteThread that a command is available. (MTX_DEBUG_CMD_READY).
4. Wait for the command to complete (MTX_DEBUG_CMD_COMPLETE).
5. Return to the kernel debugger.

If you don't need the extra debugger thread, you can simply execute commands in the caller's context:

1. Install a small extension to the kernel debugger that receives command lines intended for that extension.
2. Copy the command line to the LineBuf.
3. Call DbCommandDispatch to execute the command directly.
4. Return to the kernel debugger.

The behavior of the debugger can be configured as follows (via the *config.h* header):

```
#define DEBUGGER_THREADING        DEBUGGER_SINGLE_THREADED
```

This sets the single thread mode of the debugger.

```
#define DEBUGGER_THREADING        DEBUGGER_MULTI_THREADED
```

This sets the multi-thread mode of the debugger.

Basically, in multithread mode, we just wait for some other thread to fill the LineBuf with a command and signal the semaphore. In single thread mode, we explicitly call AcpiOsGetLine to get a command line.

## 7.5 Debugger Commands

There are three classes of commands supported by the debugger:

1. The **General-Purpose** commands are available in all modes of the debugger. These commands provide the basic functionality of loading tables, dumping internal data structures, and starting the execution of control methods.
2. The **Control Method Execution** Commands are available only during the single-step execution of control methods. These commands allow the display and modification of method arguments and local variables, control method disassemble, and the setting of method breakpoints
3. The **File I/O** Commands are available only if a filesystem is available to the debugger.

### 7.5.1 General Purpose Debugger Commands

#### 7.5.1.1 Allocations

**Memory allocation status**

**SYNTAX**

- Allocations

This command dumps the current status of the dynamic memory allocations, as maintained by the ACPI subsystem debug memory allocation tracking mechanism. Primarily used to detect memory leaks, the mechanism tracks the allocation and freeing of each memory block, and maintains statistics on the amount of memory allocated, the number of allocations, etc.

#### 7.5.1.2 Debug

**Single step a control method**

**SYNTAX**

- Debug <Namepath> [Arg0, Arg1,…]

Begin execution of a control method in single step mode. Each AML opcode and its associated operand(s) is disassembled and displayed before execution. A single carriage return (Enter) single steps to the next AML opcode. The values of the arguments and the value of the return value (if any) are displayed for each opcode.

#### 7.5.1.3 Dump

**Display objects and memory**

**SYNTAX**

- Dump <Address>|<Namepath> [Byte|Word|Dword|Qword]

A generic command to dump all internal ACPI objects and memory. The operand can be a namespace name, a pointer to an ACPI object, or a pointer to random memory in the current address space. The command determines the type of ACPI object and decodes it into the appropriate fields

#### 7.5.1.4 Event

**Generate an ACPI Event**

**SYNTAX**

- Event <Fixed|Gpe> <Value>

Generate an ACPI event to test event handling <NOT IMPLEMENTED>

#### 7.5.1.5 Execute

**Execute a control method**

**SYNTAX**

- Execute <Namepath> [Arg0, Arg1,…]

Execute a control method. This command begins execution of the named method and lets it run to completion without single stepping. The return result if any is displayed after execution completes.

#### 7.5.1.6 Exit

**Terminate**

**SYNTAX**

- Exit

Terminate the ACPI subsystem and exit the debugger.

#### 7.5.1.7 Find

**Find names in the Namespace**

**SYNTAX**

- Find <name>

Find an ACPI name or names within the current ACPI namespace. All names that match the given name are displayed as they are found in the namespace. Names are up to four characters, and wildcards are supported. A '?' in the name will match any character. Thus, the wildcarded name "A???" will match all names in the namespace that begin with the letter "A".

#### 7.5.1.8 Help

**Get help**

**SYNTAX**

- Help

Displays a help screen with the syntax of each command and a short description of each.

#### 7.5.1.9 History (! And !!)

**Command line recall**

**SYNTAX**

History

! <Command Number>

!!

last few commands. The "!" command can be used to select and re-execute a particular command from the numbered command buffer, or the "!!" command can be used to simply re-execute the immediately previous command.

#### 7.5.1.10 Level

**Set debug level**

**SYNTAX**

- Allocations

Sets the global debug output level of the ACPI subsystem for both output directed to a file and output to the console.

#### 7.5.1.11 Methods

**List all control methods**

**SYNTAX**

- Methods

Displays a list of all control methods (and their full pathnames) that are contained within the current ACPI namespace. (Alias for the command "Object Methods".)

#### 7.5.1.12 Namespace

**List the namespace**

**SYNTAX**

- Namespace [<Address> | <Namepath>] [Depth]

Dump all or a portion of the current ACPI namespace. If given with no parameter, this command displays the entire namespace, one named object per line with information about each object. If given the name of an object or a pointer to an object, it displays the subtree rooted by that object.

#### 7.5.1.13 Notify

**Generate a Notify**

**SYNTAX**

- Notify <Namepath> <Value>

Generates a notify on the specified device. This means that the notify handler for the device is invoked with the parameters specified.

#### 7.5.1.14 Object

**Display typed objects**

**SYNTAX**

- Object <Object Type>

Display objects within the namespace of the requested type.

#### 7.5.1.15 Prefix

**Get or Set current prefix**

**SYNTAX**

- Prefix [<NamePath>]

Sets the pathname prefix that is prepended to namestrings entered into the debug and execute commands. This command is the equivalent of the “CD” command.

#### 7.5.1.16 Quit

**Terminate**

**SYNTAX**

- Quit

Terminate the current execution mode. If executing (single stepping) a control method, the method is immediately aborted with an exception and the debugger returns to the normal command line mode. If no control method is executing, the ACPI subsystem is terminated and the debugger exits.

### 7.5.1.17 Stats

**Namespace statistics**

**SYNTAX**

- Stats

Display namespace statistics that were gathered when the namespace was loaded. This includes information about the number of objects and their types, the amount of dynamic memory required, and the number of search operations performed on the namespace database.

### 7.5.1.18 Terminate

**Shutdown ACPI subsystem**

**SYNTAX**

- Terminate

Shutdown the ACPI subsystem, but don't exit the debugger. This command is useful to find memory leaks in the form of objects left over after the subsystem deletes the entire namespace and all known internal objects. Any objects left over after shutdown are displayed and may be examined.

### 7.5.1.19 Thread

**Execute a control method with multiple threads**

**SYNTAX**

- Thread <number of threads> <number of loops> <Pathname>

Create the specified number of threads to execute the control method at <Pathname>. Each thread will execute the method <number of loops> times. The command waits until all threads have completed before returning.

### 7.5.1.20 Unload

**Unload table**

**SYNTAX**

- Unload <Table ID>

Unload an ACPI Table <Not implemented>

### 7.5.2 Control Method Execution Commands

During single stepping of a control method, the following commands are available. The debugger enters a slightly different command mode (as indicated by the '%' prompt) when single stepping a control method to indicate that these commands are now available

#### 7.5.2.1 Arguments

**Display Method arguments**

**SYNTAX**

Arguments

Args

Display all arguments to the currently executing control method

#### 7.5.2.2 Breakpoint

**Set control method breakpoint**

**SYNTAX**

- Breakpoint <AML Offset>

Set a breakpoint at the AML offset given. When execution reaches this offset, execution is stopped and the debugger is entered.

#### 7.5.2.3 Call

**Run to next call**

**SYNTAX**

- Call

Step execution of the current control method until the next method invocation (call) is encountered.

#### 7.5.2.4 Go

**Run method to next breakpoint**

**SYNTAX**

- Go

Cease single step mode and let the control method run freely until either a breakpoint is reached or the method terminates.

#### 7.5.2.5 Information

**Info about a control method**

**SYNTAX**

- Information

#### 7.5.2.6 Into

**Step into call**

**SYNTAX**

- Into

Step into a control method invocation instead of over the call. The default single step behavior is to step **over** control method calls, meaning that the call is executed and single stepping resumes after the call returns. Use this command to single step the execution of a called control method.

#### 7.5.2.7 List

**Disassemble AML code**

**SYNTAX**

- List [<Opcode count>]

Disassemble the AML code of the current control method from the current AML offset for the length given. Useful for  finding interesting places to set breakpoints.

#### 7.5.2.8 Locals

**Display method local variables**

**SYNTAX**

- Locals

Display the current values of all of the local variables for the current control method. When stepping into a control method invocation, the locals of the newly invoked method are displayed during the time that method is single stepped.

#### 7.5.2.9 Results

**Display method result stack**

**SYNTAX**

- Results

Display the current contents of the “Result Stack” for the control method.

#### 7.5.2.10 Set

**Set arguments or locals**

**SYNTAX**

- Set  Arg|Local <ID> <Value>

Set the value of any of a method's arguments or local variables

#### 7.5.2.11 Stop

**Stop method**

**SYNTAX**

- Stop

Terminate the currently executing control method

#### 7.5.2.12 Tree

**Display calling tree**

**SYNTAX**

- Tree

Display the calling tree of the current method.

### 7.5.3 File I/O Commands

#### 7.5.3.1 Close

**Close debug output file**

**SYNTAX**

- Close

Close the debug output file, if one is currently open. Using Exit or Quit to terminate the debugger will automatically close any open file.

#### 7.5.3.2 Load

**Load ACPI table**

**SYNTAX**

- Load <Filename>

Load an ACPI table into the namespace from a file.

#### 7.5.3.3 Open

**Open debug output file**

**SYNTAX**

- Open <Filename>

Open a file for debug output.

# 8 Tools and Utilities

## 8.1 AcpiDump

This utility is a DOS-based table disassembler and table extractor. The 16-bit version can be put on a DOS boot diskette and used to extract the ACPI tables from memory and store them to a disk file.

## 8.2 AcpiExec

This Windows-based utility can be used to load any ACPI tables from file(s), execute control methods, single step control methods, inspect the ACPI namespace, etc. When generated from source, it contains the entire ACPI core subsystem including the ACPI Debugger.

## 8.3 WDM Driver and Test Application

The WDM driver contains the ACPI Core Subsystem and the Debugger. The Ring3 test application can be used to communicate with the ACPI Debugger. Control methods can be executed or single stepped, the namespace can be dumped and inspected, etc. All commands of the ACPI Debugger are available, as well as commands unique to the test application for the execution of the various Acpi* interfaces to the core subsystem.

# 9 Subsystem User Guide

## 9.1 Using the ACPI Core Subsystem Interfaces

### 9.1.1 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 Services Layer) to make these calls when they are appropriate.

1. Initialize all ACPI Code:

   ```
   Status = AcpiInitializeSubsystem ();
   ```

2. Load the ACPI tables from the firmware and build the internal namespace:

   ```
   Status = AcpiLoadTables ();
   ```

3. Complete initialization and put the system into ACPI mode:

   ```
   Status = AcpiEnableSubsystem ();
   ```

### 9.1.2 Shutdown Sequence

The ACPI Core Subsystem 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 ACPI Core Subsystem, the following calls should be made:

1. Unload the namespace and free all resources:

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

### 9.1.3 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 */
    }
```

## 9.1.4 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.

<u>Part 1</u>: 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);
}
```

<u>Part 2</u>: 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;
    }
```

## 9.2 Implementing the OS Services Layer

### 9.2.1 Parameter Validation

In all implementations of the OS Services Layer, the interfaces should adhere to the descriptions in the document as far as the actual interface parameters as well as the returned exception codes. This means that the parameter validation is not optional and that the Core Subsystem layer depends on correct exception codes returned from the OSL.

### 9.2.2 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 ACPI Core Subsystem.

During initialization, the ACPI tables are either mapped from BIOS memory or 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.

Several internal caches are used within the core subsystem to minimize the number of requests to the memory manager.

### 9.2.3 Scheduling Services

The intent of the *AcpiOsQueueForExecution* interface is to schedule another thread. It makes no difference whether this is a new thread created at the time this call is made, or simply a thread that is allocated out of a pool of system threads. Only the ACPI Debugger creates a permanent thread.

### 9.2.4 Mutual Exclusion and Synchronization

In a single thread environment, the semaphore interfaces can simply return AE_OK. In a multiple thread environment, these interfaces must be implemented with real blocking semaphores since the mutual exclusion support in the core subsystem relies *completely* upon the proper implementation of this mechanism and these interfaces.

### 9.2.5 Interrupt Handling

In order to support the OS-independent interrupt handler that is implemented within the Core Subsystem, the OSL 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 handler returns, the wrapper performs any necessary cleanup and exits the interrupt.

### 9.2.6 Stream I/O

The *AcpiOsPrintf* and *AcpiOsVprintf* 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.

### 9.2.7 Hardware Abstraction (I/O, Memory, PCI Configuration)

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.

This page intentionally left blank.