ACPI Component Architecture Programmer Reference

OS-Independent Subsystem, Debugger, and Utilities

Revision 1.26

October 30, 2009

Contents

1 Introduction 13

1.1 Document Structure 13
1.2 Rationale and Justification 13
1.3 Reference Documents 14
1.4 Overview of the ACPI Component Architecture 14

2 Architecture Overview 16

2.1 Overview of the ACPICA Subsystem 16
2.1.1 ACPICA Core Subsystem 16
2.1.2 Operating System Services Layer 16
2.1.3 Relationships Between the Host OS, Core Subsystem, and OSL 17
2.1.3.1 Host Operating System Interaction 17
2.1.3.2 OS Services Layer Interaction 18
2.1.3.3 ACPICA Core Subsystem Interaction 18
2.2 Architecture of the ACPICA Core Subsystem 19
2.2.1 ACPI Table Management 19
2.2.2 Early ACPI Table Access 19
2.2.3 AML Interpreter 20
2.2.4 Namespace Management 20
2.2.5 Resource Management 20
2.2.6 ACPI Hardware Management 20
2.2.7 Event Handling 21
2.2.8 Requests from Host OS to ACPICA Subsystem 21
2.3 Architecture of the OS Services Layer (OSL) 22
2.3.1 Types of OSL Services 22
2.3.2 Requests from ACPICA Subsystem to OS 23

3 Design Details 24

3.1 ACPI Namespace Fundamentals 24
3.1.1 Named Objects 24
3.1.2 Scopes 24
3.1.2.1 Example Namespace Scopes, Names, and Objects 24
3.1.3 Predefined Objects 25
3.1.4 Logical Namespace Layout 25
3.2 Execution Model 26
3.2.1 Initialization 26
3.2.2 Memory Allocation 27
3.2.2.1 Caller Allocates All Buffers 27
3.2.2.2 ACPI Allocates Return Buffers 27
3.2.3 Parameter Validation 28
3.2.4 Exception Handling 28
3.2.5 Multitasking and Reentrancy 28
3.2.6 Event Handling 28
3.2.6.1 Fixed Events 29
3.2.6.2 General Purpose Events 29
3.2.6.3 Notify Events 29
3.2.7 Address Spaces and Operation Regions 29
3.2.7.1 Installation of Address Space Handlers 30
3.2.7.2 ACPI-Defined Address Spaces 30
3.3 Policies and Philosophies 31
3.3.1 External Interfaces 31

3.3.1.1 Exception Codes 31
3.3.1.2 Memory Buffers 31
3.3.2 Subsystem Initialization 31
3.3.2.1 ACPI Table Validation 31
3.3.2.2 Required ACPI Tables 32
3.3.3 Major Design Decisions 32
3.3.3.1 Performance versus Code/Data Size 32
3.3.3.2 Object Management – No Garbage Collection 32

4 Implementation Details 33

4.1 Required Host OS Initialization Sequence 33
4.1.1 Bootload and Low Level Kernel Initialization 33
4.1.2 ACPICA Subsystem Initialization 33
4.1.3 Other OS Initialization 33
4.1.4 Device Enumeration, Configuration, and Initialization 34
4.1.5 Final OS Initialization 34
4.2 Required ACPICA Initialization Sequence 34
4.2.1 Global Initialization - AcpiInitializeSubsystem 34
4.2.2 ACPI Table and Namespace Initialization 34
4.2.2.1 AcpiInitializeTables 34
4.2.2.2 AcpiGetTable, AcpiGetTableHeader, AcpiGetTableByIndex 34
4.2.2.3 AcpiLoadTables 35
4.2.2.4 Internal ACPI Namespace Initialization 35
4.2.3 Handler Installation 35
4.2.3.1 Handler Types 35
4.2.4 Hardware Initialization - AcpiEnableSubsystem 36
4.2.4.1 ACPI Hardware and Event Initialization 36
4.2.5 Object Initialization – AcpiIntializeObjects 37
4.2.5.1 ACPI Device Initialization 37
4.2.5.2 Other ACPI Object Initialization 38
4.2.6 Other Operating System ACPI-related Initialization 38
4.2.7 Just-in-time Operation Region Initialization 38
4.2.7.1 SystemMemory Region Initialization 39
4.2.7.2 PCI_Config Region Initialization 39
4.2.8 System Shutdown - AcpiTerminate 39
4.3 Multithreading Support 39
4.3.1 Reentrancy 39
4.3.2 Mutual Exclusion and Synchronization 40
4.3.3 Control Method Execution 40
4.3.3.1 Control Method Blocking 40
4.3.3.2 Control Method Execution Rules 41
4.3.3.3 A Simple Multithreading Model 41
4.3.3.4 A More Complex Multithreading Model 42
4.3.4 ACPI Global Lock Support 43
4.3.4.1 Obtaining The Global Lock 44
4.3.4.2 Releasing the Global Lock 44
4.3.4.3 Global Lock Interrupt Handler 45
4.3.5 Single Thread Environments 45

5 Subsystem Features 46

5.1 AML Interpreter Slack Mode 46
5.2 AML Interpreter Math Mode (32-bit or 64-bit) 46
5.3 Predefined Control Method Validation 46
5.4 I/O Port Protection 47
5.5 Debugging Support 47

5.5.1 Error and Warning Messages 47
5.5.2 Execution Debug Output (ACPI_DEBUG_PRINT Macro) 48
5.5.3 Function Tracing (ACPI_FUNCTION_TRACE Macro) 48
5.5.4 ACPICA Debugger 49
5.6 Environmental Support Requirements 49
5.6.1 Resource Requirements 49
5.6.2 C Library Functions 50
5.6.3 Source Code Organization 51
5.6.4 System Include Files 51
5.6.4.1 Customization to the Target Environment 52

6 Data Types and Interface Parameters 53

6.1 ACPICA Interface Parameters 53
6.1.1 ACPI Names and Pathnames 53
6.1.2 Pointers 53
6.1.3 Buffers 53
6.2 ACPICA Basic Data Types 54
6.2.1 UINT64 and COMPILER_DEPENDENT_UINT64 54
6.2.2 ACPI_PHYSICAL_ADDRESS 54
6.2.3 ACPI_IO_ADDRESS 54
6.2.4 ACPI_SIZE 54
6.2.5 ACPI_INTEGER 54
6.2.6 ACPI_STRING – ASCII String 54
6.2.7 ACPI_BUFFER – Input and Output Memory Buffers 55
6.2.7.1 Input Buffer 55
6.2.7.2 Output Buffer 55
6.2.8 ACPI_STATUS – Interface Exception Return Codes 56
6.2.9 ACPI_HANDLE – Object Handle 56
6.2.9.1 Predefined Handles 56
6.2.10 ACPI_OBJECT_TYPE – Object Type Codes 57
6.2.11 ACPI_OBJECT – Method Parameters and Return Objects 57
6.2.12 ACPI_OBJECT_LIST – List of Objects 58
6.2.13 ACPI_EVENT_TYPE – Fixed Event Type Codes 59
6.2.14 ACPI_TABLE_HEADER – Common ACPI Table Header 59
6.3 ACPI Resource Data Types 59
6.3.1 PCI IRQ Routing Tables 59
6.3.2 Device Resources 60
6.3.2.1 ACPI_RESOURCE_TYPE – Resource Data Types 60
6.4 ACPICA Exception Codes 62

7 Subsystem Configuration 66

7.1 Configuration Files 66
7.2 Component Selection 66
7.2.1 ACPI_DISASSEMBLER 66
7.2.2 ACPI_DEBUGGER 66
7.3 Configurable Data Types 67
7.3.1 ACPI_SPINLOCK 67
7.3.2 ACPI_SEMAPHORE 67
7.3.3 ACPI_MUTEX 67
7.3.4 ACPI_CPU_FLAGS 67
7.3.5 ACPI_THREAD_ID 67
7.3.6 ACPI_CACHE_T 68
7.3.7 ACPI_UINTPTR_T 68
7.4 Subsystem Options 68
7.4.1 ACPI_USE_SYSTEM_CLIBRARY 68

7.4.2 ACPI_USE_STANDARD_HEADERS ... 68
7.4.3 ACPI_DEBUG_OUTPUT ... 68
7.4.4 ACPI_USE_LOCAL_CACHE ... 68
7.4.5 ACPI_DBG_TRACK_ALLOCATIONS ... 69
7.4.6 ACPI_MUTEX_TYPE ... 69
7.4.7 ACPI_MUTEX_DEBUG ... 69
7.4.8 ACPI_SIMPLE_RETURN_MACROS ... 69
7.4.9 ACPI_USE_DO_WHILE_0 ... 70
7.5 Per-Compiler Configuration ... 70
7.5.1 COMPILER_DEPENDENT_INT64 ... 70
7.5.2 COMPILER_DEPENDENT_UINT64 ... 70
7.5.3 ACPI_USE_NATIVE_DIVIDE ... 71
7.5.4 ACPI_DIV_64_BY_32 (Short 64-bit Divide) ... 71
7.5.5 ACPI_SHIFT_RIGHT_64 (64-bit Shift) ... 71
7.5.6 ACPI_EXPORT_SYMBOL ... 72
7.5.7 ACPI_EXTERNAL_XFACE ... 72
7.5.8 ACPI_INTERNAL_XFACE ... 72
7.5.9 ACPI_INTERNAL_VAR_XFACE ... 72
7.5.10 ACPI_SYSTEM_XFACE ... 72
7.5.11 ACPI_PRINTF_LIKE ... 72
7.5.12 ACPI_UNUSED_VAR ... 73
7.6 Per-Machine Configuration ... 73
7.6.1 ACPI_MACHINE_WIDTH ... 73
7.6.2 ACPI_FLUSH_CPU_CACHE ... 73
7.6.3 ACPI_OS_NAME ... 73
7.6.4 ACPI_ACQUIRE_GLOBAL_LOCK ... 74
7.6.5 ACPI_RELEASE_GLOBAL_LOCK ... 74
7.7 Dynamic Configuration ... 75
7.7.1 Interpreter Slack Mode ... 75
7.7.2 ACPI Register Widths ... 75
7.7.3 Serialized Methods ... 76
7.7.4 Wake GPEs ... 76
7.7.5 Creation of_OSI Method ... 76
7.8 Subsystem Configuration Constants ... 76
7.8.1 ACPI_CHECKSUM_ABORT ... 76
7.8.2 ACPI_MAX_LOOP_INTERATIONS ... 76
7.8.3 ACPI_MAX_STATE_CACHE_DEPTH ... 77
7.8.4 ACPI_MAX_PARSE_CACHE_DEPTH ... 77
7.8.5 ACPI_MAX_OBJECT_CACHE_DEPTH ... 77
7.8.6 ACPI_MAX_WALK_CACHE_DEPTH ... 77

8 ACPICA Core Subsystem - External Interface Definition ... 78

8.1 ACPICA Subsystem Initialization and Control ... 78
8.1.1 AcpiInitializeSubsystem ... 78
8.1.2 AcpiInstallInitializationHandler ... 79
8.1.2.1 Interface to User Callback Function ... 79
8.1.3 AcpiEnableSubsystem ... 80
8.1.4 AcpiInitializeObjects ... 81
8.1.5 AcpiSubsystemStatus ... 82
8.1.6 AcpiTerminate ... 82
8.2 ACPI Table Management ... 84
8.2.1 AcpiInitializeTables ... 84
8.2.2 AcpiReallocateRootTable ... 85
8.2.3 AcpiFindRootPointer ... 85
8.2.4 AcpiLoadTables ... 86

8.2.5 AcpiGetTableHeader 87
8.2.6 AcpiGetTable 88
8.2.7 AcpiGetTableByIndex 89
8.2.8 AcpiInstallTableHandler 90
8.2.8.1 Interface to Table Event Handlers 90
8.2.9 AcpiRemoveTableHandler 91
8.3 ACPI Namespace Management 93
8.3.1 AcpiEvaluateObject 93
8.3.2 AcpiEvaluateObjectTyped 97
8.3.3 AcpiGetObjectInfo 98
8.3.4 AcpiGetNextObject 101
8.3.5 AcpiGetParent 103
8.3.6 AcpiGetType 103
8.3.7 AcpiGetHandle 104
8.3.8 AcpiGetName 106
8.3.9 AcpiGetDevices 107
8.3.10 AcpiAttachData 108
8.3.11 AcpiDetachData 109
8.3.12 AcpiGetData 110
8.3.13 AcpiInstallMethod 111
8.3.14 AcpiWalkNamespace 113
8.3.14.1 Interface to User Callback Function 114
8.4 ACPI Hardware Management 116
8.4.1 AcpiEnable 116
8.4.2 AcpiDisable 116
8.4.3 AcpiReset 117
8.4.4 AcpiReadBitRegister 118
8.4.5 AcpiWriteBitRegister 119
8.4.6 AcpiRead 120
8.4.7 AcpiWrite 121
8.4.8 AcpiAcquireGlobalLock 121
8.4.9 AcpiReleaseGlobalLock 122
8.4.10 AcpiGetTimerResolution 123
8.4.11 AcpiGetTimerDuration 123
8.4.12 AcpiGetTimer 124
8.5 ACPI Sleep/Wake Support 125
8.5.1 AcpiSetFirmwareWakingVector 125
8.5.2 AcpiSetFirmwareWakingVector64 125
8.5.3 AcpiGetSleepTypeData 126
8.5.4 AcpiEnterSleepStatePrep 127
8.5.5 AcpiEnterSleepState 127
8.5.6 AcpiEnterSleepStateS4Bios 128
8.5.7 AcpiLeaveSleepState 129
8.6 ACPI Fixed Event Management 130
8.6.1 AcpiEnableEvent 130
8.6.2 AcpiDisableEvent 131
8.6.3 AcpiClearEvent 131
8.6.4 AcpiGetEventStatus 132
8.6.5 AcpiInstallFixedEventHandler 133
8.6.5.1 Interface to Fixed Event Handlers 134
8.6.6 AcpiRemoveFixedEventHandler 134
8.7 ACPI General Purpose Event Management 136
8.7.1 AcpiEnableGpe 136
8.7.2 AcpiDisableGpe 137
8.7.3 AcpiClearGpe 138
8.7.4 AcpiSetGpeType 139

8.7.5 AcpiGetGpeStatus 140
8.7.6 AcpiGetGpeDevice 141
8.7.7 AcpiDisableAllGpes 142
8.7.8 AcpiEnableAllRuntimeGpes 142
8.7.9 AcpiInstallGpeBlock 143
8.7.10 AcpiRemoveGpeBlock 144
8.7.11 AcpiInstallGpeHandler 145
8.7.11.1 Interface to General Purpose Event Handlers 146
8.7.12 AcpiRemoveGpeHandler 146
8.8 Miscellaneous Handler Support 148
8.8.1 AcpiInstallNotifyHandler 148
8.8.1.1 Interface to Notification Event Handlers 149
8.8.2 AcpiRemoveNotifyHandler 150
8.8.3 AcpiInstallAddressSpaceHandler 151
8.8.3.1 Interface to Address Space Setup Handlers 152
8.8.3.2 Interface to Address Space Handlers 153
8.8.3.3 Context for the Default PCI Address Space Handler 154
8.8.4 AcpiRemoveAddressSpaceHandler 154
8.8.5 AcpiInstallExceptionHandler 155
8.8.5.1 Interface to Exception Handlers 156
8.9 ACPI Resource Management 157
8.9.1 AcpiGetCurrentResources 157
8.9.2 AcpiGetPossibleResources 158
8.9.3 AcpiSetCurrentResources 159
8.9.4 AcpiGetIRQRoutingTable 160
8.9.5 AcpiGetVendorResource 161
8.9.6 AcpiResourceToAddress64 162
8.9.7 AcpiWalkResources 162
8.9.7.1 Interface to User Callback Function 163
8.10 Memory Management 165
8.10.1 ACPI_ALLOCATE 165
8.10.2 ACPI_ALLOCATE_ZEROED 166
8.10.3 ACPI_FREE 166
8.11 Formatted Output 167
8.11.1 AcpiInfo and ACPI_INFO 167
8.11.2 AcpiWarning and ACPI_WARNING 168
8.11.3 AcpiError and ACPI_ERROR 169
8.11.4 AcpiException and ACPI_EXCEPTION 170
8.11.5 AcpiDebugPrint and ACPI_DEBUG_PRINT 171
8.11.6 AcpiDebugPrintRaw and ACPI_DEBUG_PRINT_RAW 173
8.12 Miscellaneous Utilities 173
8.12.1 AcpiFormatException 173
8.12.2 AcpiDebugTrace 174
8.12.3 AcpiGetSystemInfo 175
8.12.4 AcpiGetStatistics 176
8.12.5 AcpiPurgeCachedObjects 177
8.13 Global Variables 177
8.13.1 AcpiDbgLevel & AcpiDbgLayer 177
8.13.2 AcpiGbl_FADT 178
8.13.3 AcpiCurrentGpeCount 178

9 OS Services Layer - External Interface Definition 179

9.1 Environmental and ACPI Tables 179
9.1.1 AcpiOsInitialize 179
9.1.2 AcpiOsTerminate 180

9.1.3 AcpiOsGetRootPointer 180
9.1.4 AcpiOsPredefinedOverride 181
9.1.5 AcpiOsTableOverride 181
9.2 Memory Management 183
9.2.1 AcpiOsCreateCache 183
9.2.2 AcpiOsDeleteCache 184
9.2.3 AcpiOsPurgeCache 184
9.2.4 AcpiOsAcquireObject 185
9.2.5 AcpiOsReleaseObject 185
9.2.6 AcpiOsMapMemory 186
9.2.7 AcpiOsUnmapMemory 187
9.2.8 AcpiOsGetPhysicalAddress 187
9.2.9 AcpiOsAllocate 188
9.2.10 AcpiOsFree 188
9.2.11 AcpiOsReadable 189
9.2.12 AcpiOsWritable 189
9.3 Multithreading and Scheduling Services 190
9.3.1 AcpiOsGetThreadId 190
9.3.2 AcpiOsExecute 190
9.3.3 AcpiOsSleep 191
9.3.4 AcpiOsStall 192
9.4 Mutual Exclusion and Synchronization 192
9.4.1 AcpiOsCreateMutex 192
9.4.2 AcpiOsDeleteMutex 193
9.4.3 AcpiOsAcquireMutex 193
9.4.4 AcpiOsReleaseMutex 194
9.4.5 AcpiOsCreateSemaphore 194
9.4.6 AcpiOsDeleteSemaphore 195
9.4.7 AcpiOsWaitSemaphore 196
9.4.8 AcpiOsSignalSemaphore 197
9.4.9 AcpiOsCreateLock 197
9.4.10 AcpiOsDeleteLock 198
9.4.11 AcpiOsAcquireLock 198
9.4.12 AcpiOsReleaseLock 199
9.5 Interrupt Handling 199
9.5.1 AcpiOsInstallInterruptHandler 200
9.5.1.1 Interface to OS-independent Interrupt Handlers 201
9.5.2 AcpiOsRemoveInterruptHandler 201
9.6 Memory Access and Memory Mapped I/O 202
9.6.1 AcpiOsReadMemory 202
9.6.2 AcpiOsWriteMemory 203
9.7 Port Input/Output 203
9.7.1 AcpiOsReadPort 204
9.7.2 AcpiOsWritePort 204
9.8 PCI Configuration Space Access 205
9.8.1 AcpiOsReadPciConfiguration 205
9.8.2 AcpiOsWritePciConfiguration 206
9.8.3 AcpiOsDerivePciId 206
9.9 Formatted Output 207
9.9.1 AcpiOsPrintf 207
9.9.2 AcpiOsVprintf 208
9.9.3 AcpiOsRedirectOutput 208
9.10 Miscellaneous 209
9.10.1 AcpiOsValidateInterface 209
9.10.2 AcpiOsGetTimer 209

9.10.3 AcpiOsSignal 210
9.10.4 AcpiOsGetLine 211

10 ACPICA Deployment Guide 212

10.1 Using the ACPICA Core Subsystem Interfaces 212
10.1.1 Initialization Sequence 212
10.1.2 ACPICA Initialization Examples 212
10.1.2.1 Full ACPICA Initialization 212
10.1.2.2 ACPICA Initialization With Early ACPI Table Access 213
10.1.3 Shutdown Sequence 214
10.1.4 Traversing the ACPI Namespace (Low Level) 215
10.1.5 Traversing the ACPI Namespace (High Level) 217
10.2 Implementing the OS Services Layer 218
10.2.1 Parameter Validation 218
10.2.2 Memory Management 218
10.2.3 Scheduling Services 218
10.2.4 Mutual Exclusion and Synchronization 218
10.2.5 Interrupt Handling 218
10.2.6 Stream I/O 219
10.2.7 Hardware Abstraction (I/O, Memory, PCI Configuration) 219

11 Tools and Utilities 220

11.1 iASL Compiler 220
11.2 AcpiExec – User Mode ACPI Execution/Simulation 221
11.3 AcpiXtract – Extract ACPI Tables 221
11.4 AcpiSrc – Convert ACPICA Source Code 221

12 ACPICA Debugger Reference 223

12.1 Overview 223
12.2 Supported Environments 223
12.2.1 The AcpiExec Utility 223
12.3 Debugger Architecture 223
12.4 Configuration and Installation 224
12.5 Command Overview 226
12.6 General Purpose Commands 226
12.6.1 Allocations 226
12.6.2 Dump 226
12.6.3 Exit 227
12.6.4 Help 227
12.6.5 History (! And !!) 227
12.6.6 Level 227
12.6.7 Locks 228
12.6.8 Quit 228
12.6.9 Stats 228
12.6.10 Tables 229
12.6.11 Unload 229
12.7 Namespace Access Commands 229
12.7.1 BusInfo 229
12.7.2 Disassemble 229
12.7.3 Event 230
12.7.4 Find 230
12.7.5 Gpe 230
12.7.6 Gpes 230
12.7.7 Integrity 231

12.7.8 Methods 231
12.7.9 Namespace 231
12.7.10 Notify 231
12.7.11 Object 232
12.7.12 Owner 232
12.7.13 Predefined 233
12.7.14 Prefix 233
12.7.15 References 233
12.7.16 Resources 233
12.7.17 Set N 234
12.7.18 Sleep 234
12.7.19 Terminate 234
12.7.20 Type 234
12.8 Control Method Execution Commands 235
12.8.1 Arguments 235
12.8.2 Breakpoint 235
12.8.3 Call 235
12.8.4 Debug 235
12.8.5 Execute 236
12.8.6 Go 236
12.8.7 Information 236
12.8.8 Into 236
12.8.9 List 237
12.8.10 Locals 237
12.8.11 Results 237
12.8.12 Set 237
12.8.13 Stop 238
12.8.14 Thread 238
12.8.15 Trace 238
12.8.16 Tree 238
12.9 File I/O Commands 239
12.9.1 Close 239
12.9.2 Load 239
12.9.3 Open 239

Figures

Figure 1. The ACPI Component Architecture 15
Figure 2. ACPICA Subsystem Architecture 17
Figure 3. Interaction between the Architectural Components 18
Figure 4. Internal Modules of the ACPICA Core Subsystem 19
Figure 5. Operating System to ACPICA Subsystem Request Flow 22
Figure 6. ACPICA Subsystem to Operating System Request Flow 23
Figure 7. Internal Namespace Structure 26
Figure 8. Global Lock Architecture 44
Figure 9. ACPICA Debugger Architecture 224

Tables

Table 1. C Library Functions Used within the Subsystem 50
Table 2. ACPI Object Type Codes 57
Table 3. Exception Code Values 62

1 Introduction

1.1 Document Structure

This document consists of these 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. ***Architecture Overview*:** Overview of the main architectural components and interface to the host operating system. Summary of the computational and architectural model that is implemented by the ACPI component architecture.
3. ***Design Details*:** Details concerning design decisions and execution model.
4. ***Implementation Details*:** Details concerning implementation specifics.
5. ***Data Types and Interface Parameters*:** Descriptions of the major data types and data structures that are exposed via the external interfaces. Other related information required to use the ACPICA subsystems and interfaces.
6. ***Subsystem Configuration*:** Description of the available configuration options to tailor the subsystem to different compilers and machines, as well as run-time tuning options.
7. ***ACPICA Core Subsystem Interfaces*:** Detailed description of the programmatic interfaces that are implemented by the core component of the ACPI Component Architecture.
8. ***OS Services Layer Interfaces*:** Detailed description of the programmatic interfaces that must be implemented by operating system vendors in the layer that interfaces the ACPICA Core Subsystem to the host operating system.
9. ***ACPICA Deployment 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.
10. ***Tools and Utilities*:** A brief overview of the miscellaneous tools and utilities that are part of the ACPICA package.
11. ***ACPICA Debugger Reference*:** Overview, installation and configuration, and detailed descriptions of the command set.

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

1.3 Reference Documents

ACPI documents are available at http://www.acpi.info

- *Advanced Configuration and Power Interface Specification*, Revision 1.0, December 1, 1996
- *Advanced Configuration and Power Interface Specification*, Revision 1.0a, July 1, 1998
- *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
- *Advanced Configuration and Power Interface Specification*, Revision 2.0b, October 11, 2002
- *Advanced Configuration and Power Interface Specification*, Revision 2.0c, August 23, 2003
- *Advanced Configuration and Power Interface Specification*, Revision 3.0, September 2, 2004
- *Advanced Configuration and Power Interface Specification*, Revision 3.0a, December 30, 2005
- *Advanced Configuration and Power Interface Specification*, Revision 3.0b, October 10, 2006
- *Advanced Configuration and Power Interface Specification*, Revision 4.0, June 16, 2009
- *iASL Compiler User Reference*

1.4 Overview of the ACPI Component Architecture

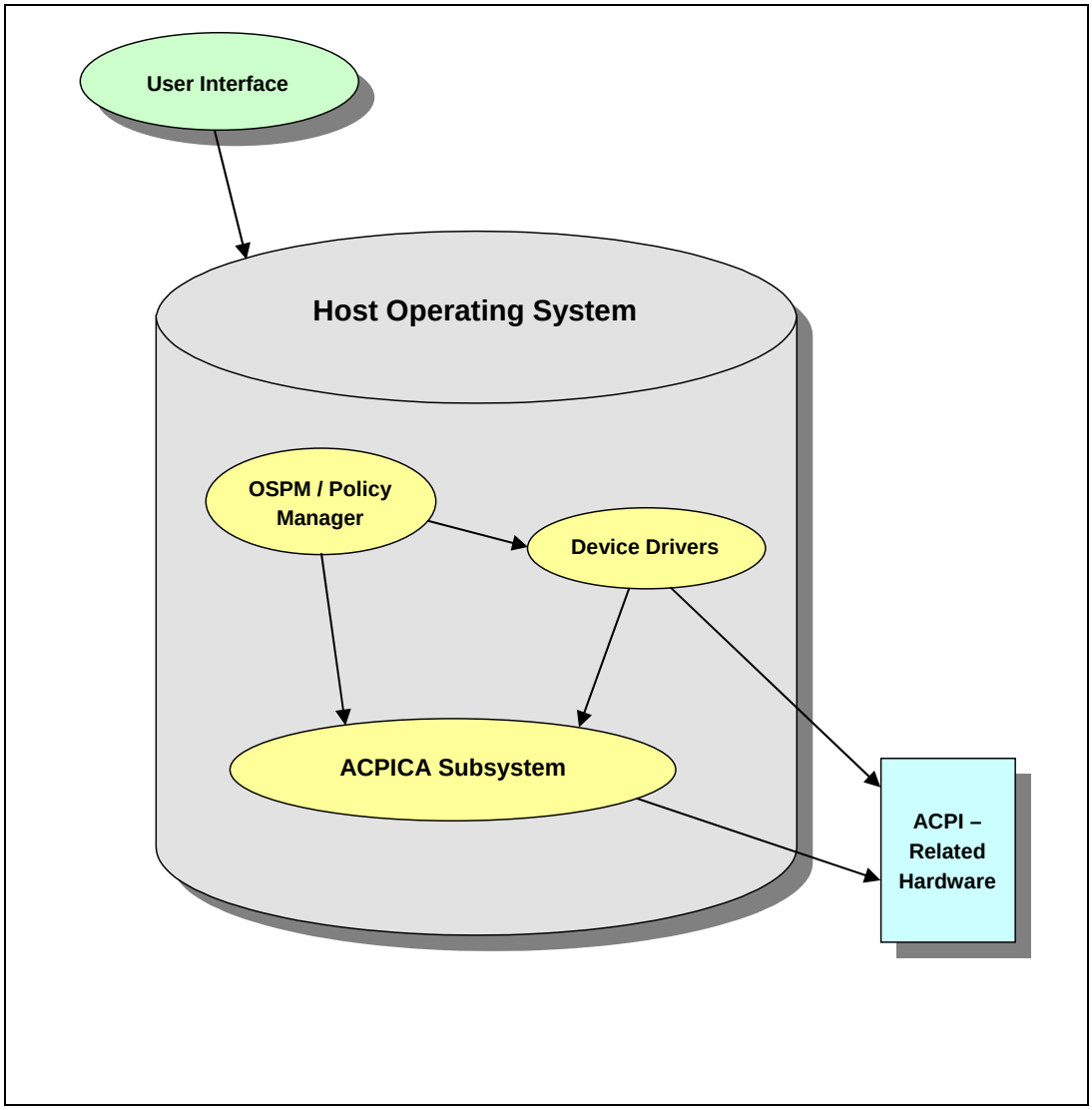
The ACPI Component Architecture (also referred to by the term “ACPICA” in this document) defines and implements a group of software components that together create an implementation of the ACPI specification. A major goal of the architecture is to isolate all operating system dependencies to a relatively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

- An OS-independent ACPICA Core Subsystem component that provides the fundamental ACPI services such as the AML interpreter and namespace management.
- An OS-dependent OS Services Layer for each host operating system to provide OS support for the ACPICA Core Subsystem.
- An ASL compiler-disassembler for translating ASL code to AML byte code and for disassembling existing binary ACPI tables back to ASL source code.
- Several ACPI utilities for executing the interpreter in ring 3 user space, extracting binary ACPI tables from the output of the AcpiDump utility, and translating the ACPICA source code to Linux/Unix format.

This document describes the ACPICA Subsystem, OS services layer, and utilities. The iASL compiler is documented in the *iASL Compiler User Reference*.

In the diagram below, the ACPICA subsystem is shown in relation to the host operating system, device driver, OSPM software, and the ACPI hardware.

Figure 1. The ACPI Component Architecture



2 Architecture Overview

2.1 Overview of the ACPICA Subsystem

The ACPICA 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 ACPICA 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 ACPICA-to-OS requests into the system calls provided by the host operating system. The OS Services Layer is the only component of the ACPICA that contains code that is specific to a host operating system.

Thus, the ACPICA Subsystem consists of two major software components:

1. The ACPICA 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 ACPICA 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 *ACPICA Subsystem*. Throughout this document, the term “ACPICA Subsystem” refers to the combination of the ACPICA Core Subsystem with the OS Services Layer components into a single module, driver, or load unit.

2.1.1 ACPICA Core Subsystem

The ACPICA 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 ACPICA Core Subsystem is to provide an abstraction level high enough such that the host operating system does not need to understand or know about the very low-level ACPI details. For example, all AML code is hidden from the host. 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 ACPICA Core Subsystem are the host OS device drivers and power/thermal management software.

2.1.2 Operating System Services Layer

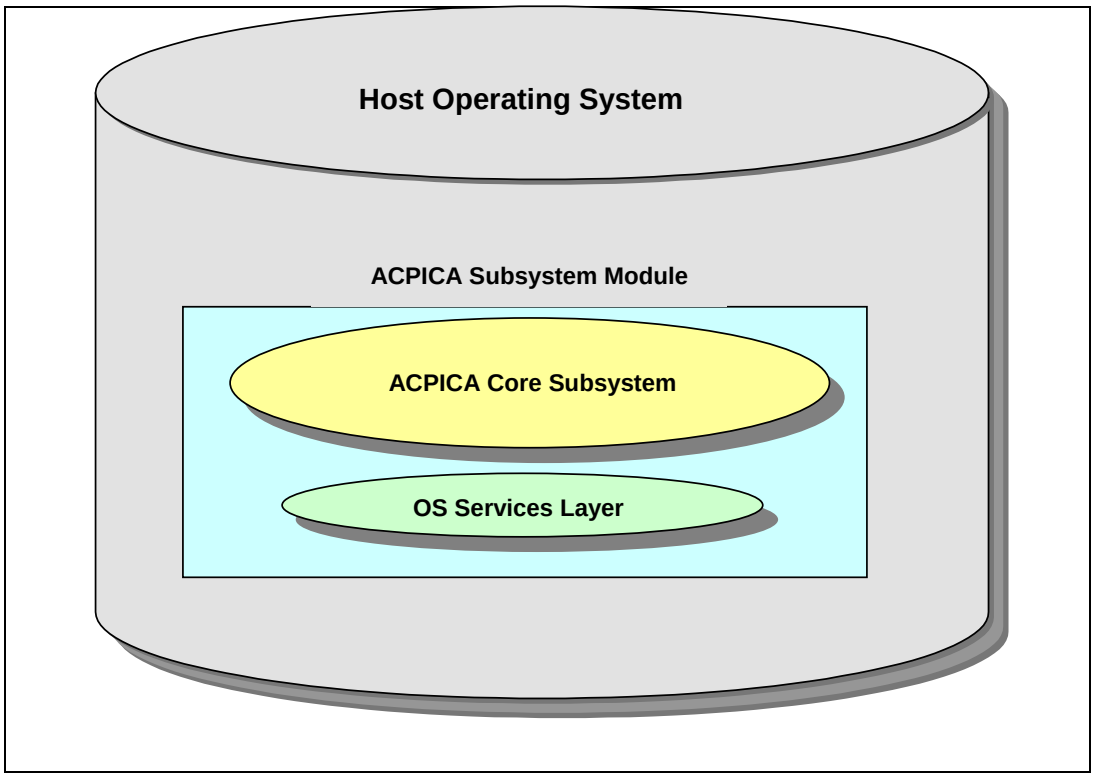
The OS Services Layer (or ***OSL***) operates as a translation service for requests from the ACPICA core subsystem back to the host OS. The OSL implements a generic set of OS service interfaces by using the primitives available from the host OS.

Because of its nature, the OS Services Layer must be implemented anew for each supported host operating system. There is a single ACPICA 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 ACPICA 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 ACPI 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. ACPICA Subsystem Architecture



2.1.3 Relationships Between the Host OS, Core Subsystem, and OSL

2.1.3.1 Host Operating System Interaction

The Host Operating System makes direct calls to the **Acpi*** interfaces within the ACPICA Core Subsystem to request ACPI services.

Whenever the ACPICA Core Subsystem requires operating system services, it makes calls the OS Services Layer. 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 (OS-dependent) calls made directly to the host are confined to the OS Services Layer. The core ACPICA code contains no operating system-specific code.

2.1.3.2 OS Services Layer Interaction

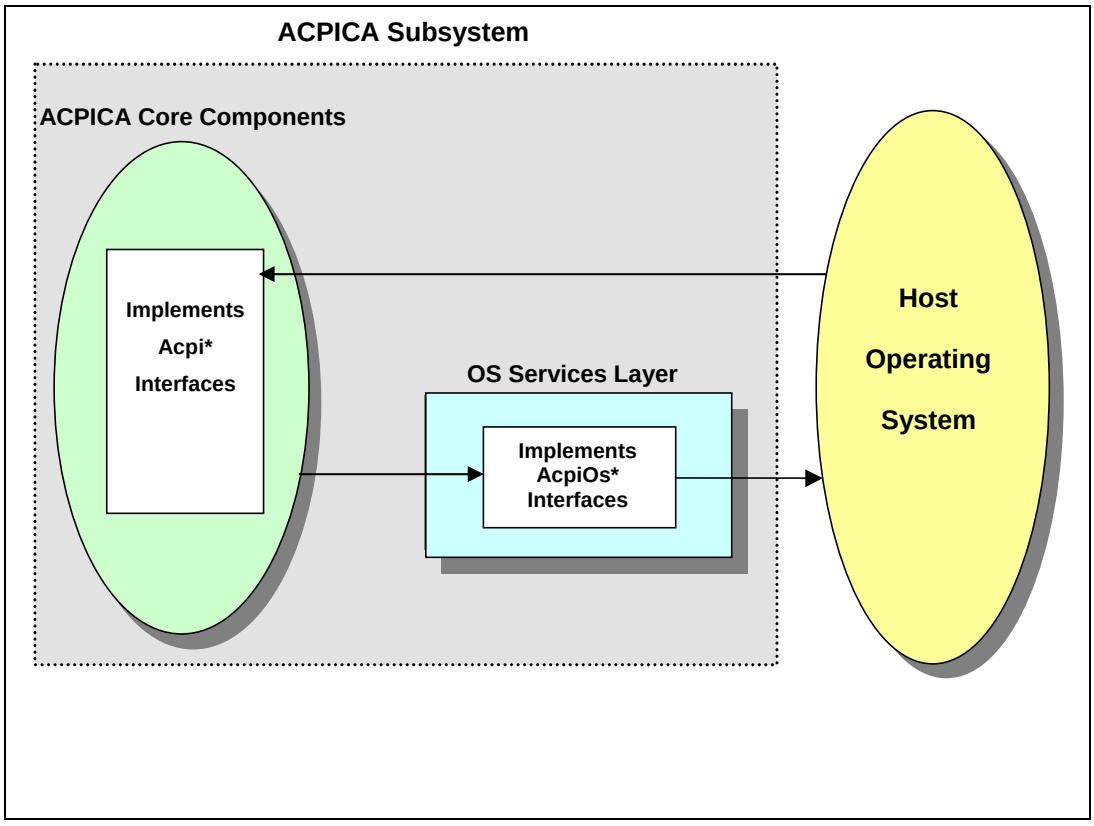
The OS Services Layer provides operating-system dependent implementations of the predefined **AcpiOs*** interfaces. These interfaces provide common operating system services to the Core Subsystem such as memory allocation, mutual exclusion, hardware access, and I/O. The Core Subsystem component uses these interfaces 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 *** interfaces.

2.1.3.3 ACPICA Core Subsystem Interaction

The *ACPICA Core Subsystem* implements a set of external interfaces that can be directly called from the host OS. These **Acpi*** interfaces provide the actual ACPI services for the host. 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 Core Subsystem never calls the host directly -- instead it makes calls to the **AcpiOs *** interfaces in the OSL. This provides the ACPICA code with OS-independence.

Figure 3. Interaction between the Architectural Components



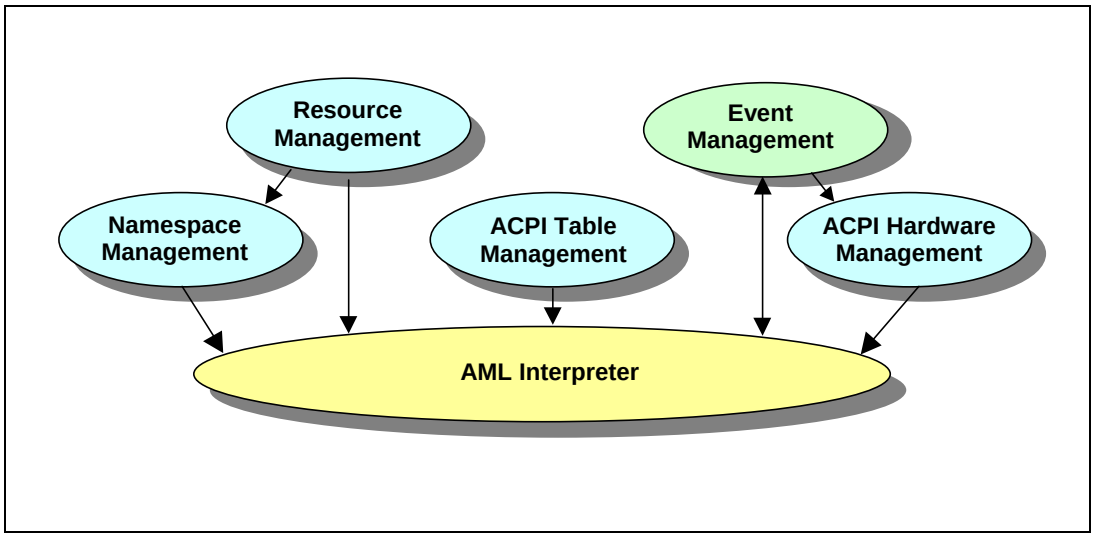
2.2 Architecture of the ACPICA Core Subsystem

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 ACPICA 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 ACPICA 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 ACPICA Core Subsystem



2.2.1 ACPI Table Management

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

- ACPI Table Verification
- ACPI Table installation and removal
- Access to all available ACPI tables

2.2.2 Early ACPI Table Access

In many cases, certain ACPI tables are required by the host OS very early during system/kernel initialization. For example, the ECDT (Embedded Controller Boot Resources Table) and MADT (Multiple APIC Description Table) may be required before hardware elements can be initialized

properly. This initialization and thus these ACPI tables may be required before the kernel dynamic memory (and virtual memory) is available.

To support this need, the ACPICA Table Manager component is designed as a standalone service that can be initialized and used independently from the rest of the ACPICA core subsystem. It can be executed with no need for any dynamic memory, and only the need for a single memory mapping at any given time.

2.2.3 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:

- ACPI Table Parsing
- AML Control Method Execution
- Evaluation of Namespace Objects

2.2.4 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 ACPI tables
- Device Enumeration
- Namespace Access
- Access to ACPI data and tables

2.2.5 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

2.2.6 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
- ACPI mode enable/disable
- Global Lock support
- Sleep Transitions support (S-states)

2.2.7 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 event enable/disable (Fixed Events, GPEs)
- 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)

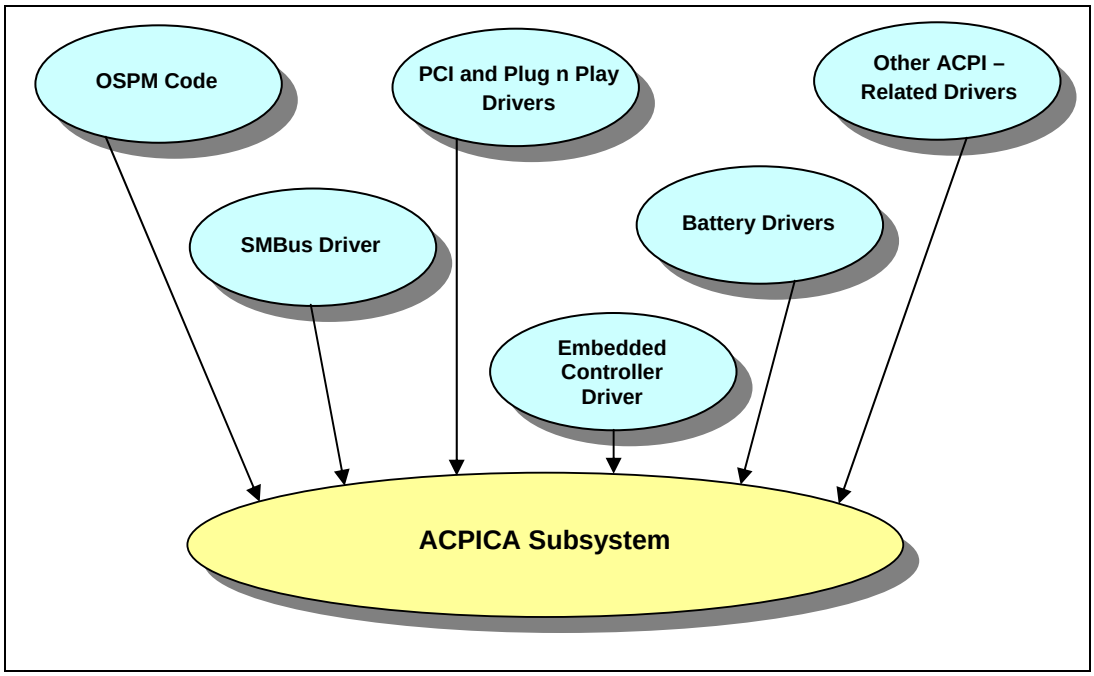
2.2.8 Requests from Host OS to ACPICA Subsystem

The host operating system can make direct calls to the Acpi* external interfaces to request ACPI services.

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 ACPICA requests is shown in the diagram below.

Figure 5. Operating System to ACPICA Subsystem Request Flow



2.3 Architecture of the OS Services Layer (OSL)

The OS Services Layer component of the architecture enables the rehosting or retargeting of the ACPICA 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 ACPICA 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 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.

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.

2.3.1 Types of OSL Services

The services provided for the ACPICA Core Subystem by the OS Services Layer can be categorized into the following groups:

- **Environmental** – global initialization and environment setup.
- **Memory Management** – dynamic memory allocation and memory mapping.
- **Multitasking Support** – scheduling and asynchronous execution.

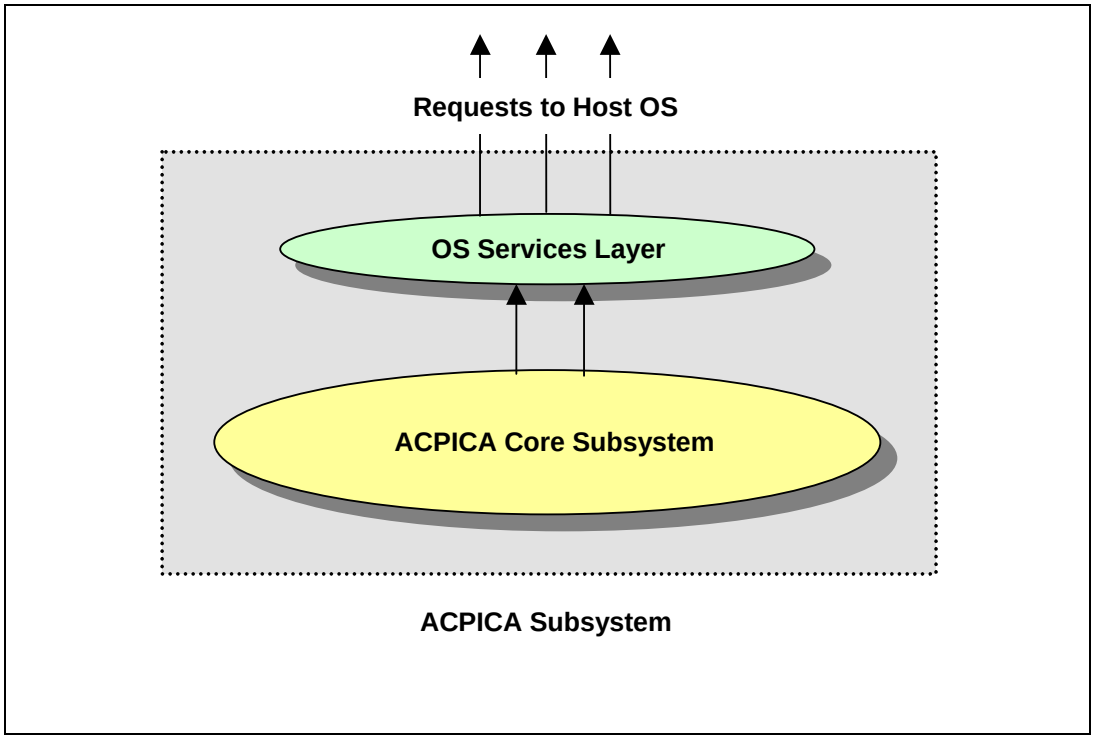
- **Mutual Exclusion and Synchronization** – Mutexes, Semaphores, and Spin Locks.
- **Interrupt handling** – interrupt handlers.
- **Address Spaces** – memory, I/O port, and PCI configuration space access.
- **Stream I/O** – support for console I/O with printf-like functions. This provides error, warning, debug, and trace output from the subsystem.

2.3.2 Requests from ACPICA Subsystem to OS

ACPI to OS requests are requests for OS services made by the ACPICA 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. ACPICA Subsystem to Operating System Request Flow



3 Design Details

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

3.1 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.

3.1.1 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.

3.1.2 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 ACPICA 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.

3.1.2.1 Example Namespace 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 *scope* of control methods _L08 and _L0A contain executable AML code.
- 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* (or *peer 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*.

3.1.3 Predefined Objects

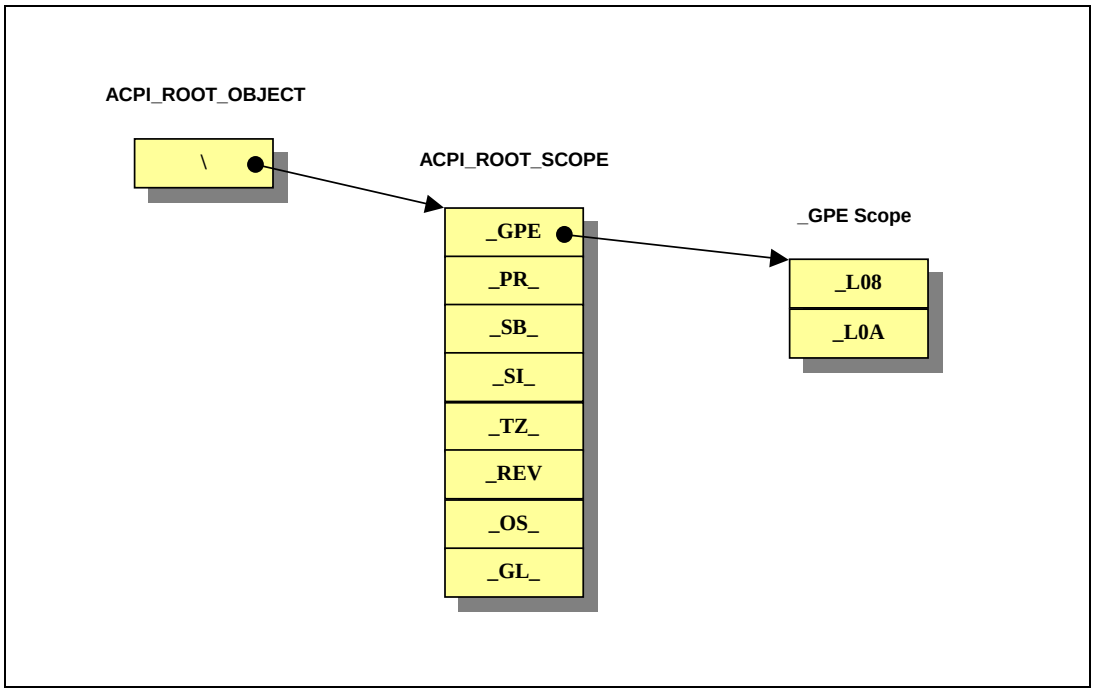
During initialization of the internal namespace within Core Subsystem component, there are several predefined objects that are always created and installed in the namespace, regardless of whether they appear in any of the loaded ACPI tables. These objects and their associated types are shown below.

```
"_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  // Supported ACPI specification revision
"_OS_",     ACPI_TYPE_STRING  // OS Name
"_GL_",     ACPI_TYPE_MUTEX   // Global Lock
"_OSI",     ACPI_TYPE_METHOD  // Query OS Interfaces
```

3.1.4 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



3.2 Execution Model

3.2.1 Initialization

The initialization of the ACPICA Subsystem must be driven entirely by the host operating system. Since it may be appropriate (depending on the requirements of the host OS) to initialize different parts of the ACPICA Subsystem at different times, this initialization is split into a multi-step process. The four main steps are outlined below.

1. Perform a global initialization of the ACPICA Subsystem – this initializes the global data and other items within the subsystem.
2. Initialize the table manager and load the ACPI tables – The FADT, FACS, DSDT, and SSDTs must be acquired and mapped 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 RSDT/XSDT, FADT, FACS, and a DSDT. Any SSDTs are optional. All other ACPI tables defined by the ACPI specification are not directly used by the ACPICA subsystem, but they are available to ACPI-related device drivers via the table manager external interfaces. These tables include the MADT, ECDT, etc.
3. Build the internal namespace – this causes ACPICA to parse the DSDT and any SSDTs 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 ACPICA Subsystem installs an interrupt handler for the System Control Interrupts (SCIs), and transitions the hardware from legacy mode to ACPI mode.

3.2.2 Memory Allocation

There are two models of memory allocation that can be used. In the first model, the caller to the ACPICA 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 ACPICA 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.

3.2.2.1 Caller Allocates All Buffers

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

It is often the case that the required buffer size is not known by even the ACPICA 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 6.2.7 - “ACPI_BUFFER – Input and Output Memory Buffers” for additional discussion on using the ACPI_BUFFER data type.

3.2.2.2 ACPI Allocates Return Buffers

In this model, the caller lets the ACPICA 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 ACPICA 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.

3.2.3 Parameter Validation

Only limited parameter validation is performed on all input parameters passed to the ACPICA Core Subsystem. Therefore, the host OS 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 null pointers and out-of-range values and nothing more. Any additional parameter validation (such as buffer length validation) must occur before the host calls the ACPICA code.

3.2.4 Exception Handling

All exceptions that occur during the processing of a request to the ACPICA Core Subsystem are returned in an ACPI_STATUS return code and bubbled up to the original caller. Names for the ACPICA exceptions are all prefixed with “AE_”. For example, AE_OK indicates successful completion of a request.

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.

3.2.5 Multitasking and Reentrancy

All components of the ACPICA 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 ACPICA 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 ACPICA subsystem. Serialization of both internal and external requests for execution of control methods is performed and managed by the front-end of the interpreter.

3.2.6 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 ACPICA 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 ACPICA subsystem are described in more detail below.

3.2.6.1 Fixed Events

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

3.2.6.2 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 **_Exx** for edge-triggered or **_Lxx** 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.

GPE Block Devices are also supported. These GPE blocks may be installed and removed dynamically as necessary. The ACPICA Core Subsystem provides centralized GPE handling and dispatch, and provides the necessary interfaces to install and remove GPE Block Devices.

3.2.6.3 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.

3.2.7 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.

3.2.7.1 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 ACPICA 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 ACPICA user to enumerate the namespace and install address handlers as needed.

3.2.7.2 ACPI-Defined Address Spaces

The ACPI specification defines address spaces for:

- System Memory
- System I/O
- PCI Configuration Space
- System Management Bus (SMBus)
- Embedded Controller
- CMOS
- PCI Bar Target
- IPMI (ACPI 4.0)

The ACPICA 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 (such as 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, SMBus, and other ACPI-related device drivers.

3.3 Policies and Philosophies

This section provides insight into the policies and philosophies that were used during the design and implementation of the ACPICA 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.

3.3.1 External Interfaces

3.3.1.1 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 ACPICA 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.

3.3.1.2 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 ACPICA subsystem to allocate return buffers.

3.3.2 Subsystem Initialization

3.3.2.1 ACPI Table Validation

All ACPI tables that are examined by the ACPICA 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.

3.3.2.2 Required ACPI Tables

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

1. One *Fixed ACPI Description Table* (FADT — signature "FACP"). This table contains configuration information about the ACPI hardware and pointers to the FACS and DSDT tables.
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.
4. Optional are one or more *Secondary System Description Tables* (SSDTs) that contain additional AML code. All SSDTs found in the RSDT/XSDT root table are loaded during the table/namespace initialization. Other SSDTs and OEM tables can be loaded at runtime via the *Load* or *LoadTable* AML operators.

3.3.3 Major Design Decisions

3.3.3.1 Performance versus Code/Data Size

The ACPICA subsystem is optimized to minimize code and data size at the expense of performance. The relatively static internal namespace data structure has been optimized to minimize non-paged kernel memory use, and control method execution parse trees are freed immediately upon method termination.

3.3.3.2 Object Management – No Garbage Collection

Creation and deletion of all internal objects are managed such that garbage collection is never required or performed. Objects are deleted deterministicatlly when they are no longer needed. This is achieved through the use of object reference counts and object trees.

Internal object caches allow the reuse of commonly used objects without burdening the OS free space manager. This greatly improves the performance of the entire subsystem.

4 Implementation Details

4.1 Required Host OS Initialization Sequence

This section describes a generic operating system initialization sequence that includes the ACPICA subsystem. The ACPICA 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 ACPICA subsystem include memory management, synchronization primitives, and interrupt support. As soon as these services are available, ACPICA 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.

4.1.1 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 ACPI Table Manager if early ACPI table access is required
- Initialize and enable free space manager
- Initialize and enable synchronization primitives
- Initialize basic interrupt mechanism and hardware
- Initialize and start system timer

4.1.2 ACPICA Subsystem Initialization

- Initialize ACPICA Table Manager and 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)
- Find PCI Root Bus(es) and install PCI config space handlers

4.1.3 Other OS Initialization

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

4.1.4 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

4.1.5 Final OS Initialization

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

4.2 Required ACPICA Initialization Sequence

This section presents a detailed description of the initialization process for the ACPICA 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.

4.2.1 Global Initialization - AcpiInitializeSubsystem

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

4.2.2 ACPI Table and Namespace Initialization

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

4.2.2.1 AcpiInitializeTables

This function initializes the ACPICA Table Manager. This component is independent of the rest of the ACPICA core subsystem and may be initialized and executed at any time during kernel initialization, even before dynamic/virtual memory is available. This allows the ACPI tables to be acquired very early in the kernel initialization process. Some ACPI tables are required during early kernel initialization/configuration -- such as the SLIT (System Locality Distance Information Table), SRAT (System Resource Affinity Table), and MADT (Multiple APIC Description Table.)

4.2.2.2 AcpiGetTable, AcpiGetTableHeader, AcpiGetTableByIndex

These functions may be used by the host OS and device drivers to obtain individual ACPI tables as necessary. The only ACPI tables that are consumed by the ACPICA subsystem are the FADT, FACS, DSDT, and any SSDTs. All other ACPI tables present on the platform must be consumed by

the host OS and device drivers. For example, the ECDT (Embedded Controller Boot Resources Table) is used by the host-dependent Embedded Controller device driver.

4.2.2.3 AcpiLoadTables

This interface creates the internal ACPI namespace data structure from the DSDT and SSDTs found in the RSDT/XSDT root table. All SSDTs found in the root table are loaded. Other SSDTs may be loaded by AML code at runtime via the AML Load operator. OEM tables that appear in the RSDT/XSDT can only be loaded via the AML LoadTable operator.

4.2.2.4 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

4.2.3 Handler Installation

Once the namespace has been constructed, the OS should install any handlers that it may require during execution of the ACPICA 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 any default handlers.

4.2.3.1 Handler Types

The following handler installation interfaces are available

Initialization Handler: *AcpiInstallInitializationHandler*

This function is used to install a global handler for ACPICA initialization events. Currently, the handler is called after the execution of every device _INI method.

Table Event Handler: *AcpiInstallTableHandler*

This function is used to install a global handler for ACPI table load/unload events.

AML Exception Handler: *AcpiInstallExceptionHandler*

This function is used to install a global handler for AML run-time exceptions.

Address Space Handlers: *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. These handlers are invoked to implement Operation Region requests.

Fixed Event Handlers: *AcpiInstallFixedEventHandler*

This function is used to install handlers for ACPI Fixed Events.

General-Purpose Event Handlers: *AcpiInstallGpeHandler*

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

Notify Handlers: *AcpiInstallNotifyHandler*

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

4.2.4 Hardware Initialization - AcpiEnableSubsystem

This step continues the subsystem initialization and is more hardware oriented. It first puts the system into ACPI mode, then installs the default Operation Region handlers, initializes the event manager, and installs the SCI and Global Lock handlers.

During the event manager initialization, fixed events are initialized and enabled. GPEs are initialized, but are not enabled at this time.

To summarize the actions performed by this call:

- Enter ACPI Mode.
- Install default operation region handlers for the following address spaces that must always be available: SystemMemory, SystemIO, PCI_Config, and DataTable.
- Initialize ACPI Fixed and General Purpose events (not enabled at this time, however.)
- Install the SCI and Global Lock interrupt handlers.

4.2.4.1 ACPI Hardware and Event Initialization

This step puts the system into ACPI mode if necessary, sets up the ACPI hardware, initializes the ACPI Event handling, and installs the ACPI interrupt handlers. This step is optional when running in “hardware-independent” mode – when there is no access to hardware by the ACPICA subsystem (For example, the *AcpiExec* utility runs 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 ACPI interrupts (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 the 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.

When this step is complete, control methods may 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 ACPICA subsystem initialization that requires control method execution can now be completed.

4.2.5 Object Initialization – AcpiIntializeObjects

This step completes the initialization of all objects within the loaded namespace, then initializes and enables the runtime general-purpose events:

- Initialize all Operation Regions. This step runs all Operation Region _REG methods for the address spaces with default handlers – SystemMemory, SystemIO, PCI_Config, and DataTable. Note: Operation Regions that are declared within control methods are not initialized until actual execution of the method.
- Finish initialization of complex objects (Operation Regions, BufferFields, Buffers, BankFields, and Packages) that contain executable AML code within the declaration.
- Initialize all Device, Processor and Thermal objects within the namespace by executing the**_STA** and **_INI** methods on each of these objects.
- Initialize the FADT-defined GPE blocks.
- Execute all **_PRW** methods within the namespace. These methods identify and define the GPEs that are used for wake events. These types of GPEs are never enabled at runtime, they are only enabled as the system enters a sleep state.
- Enable all runtime GPEs. The GPEs can only be enabled after the **_REG**, **_STA**, and **_INI** methods have been run. This ensures that all Operation Regions and all Devices have been initialized and are ready.

4.2.5.1 ACPI Device Initialization

During this step, all Device, Processor, and Thermal objects found within the ACPI namespace are initialized. The **_INI** method is executed for 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 entire namespace is traversed and the**_STA** and **_INI** methods are run on all ACPI objects of type *Device*, *Processor*, and *Thermal* found therein. Any operation regions accessed by these methods will be automatically initialized by the just-in-time address space initialization mechanism. The initialization is performed via the following steps:

- A namespace analysis is performed to identify all subtrees that contain devices that have a corresponding **_INI** method. This greatly enhances the speed of this step and can reduce operating system boot time. If there is no **_INI** method for a given device, then no attempt is made to execute the **_STA** method for the device.
- If the device has an **_INI** method, attempt to execute the **_STA** method for the device.

- If **_STA** does not exist within the scope of the device, the device is assumed to be both *present* and *functional* – as per the ACPI specification.
- If the **_STA** flags indicate the device is *not present* but *functioning*, do not run **_INI** on the device, but continue to examine the children of the device.
- If the **_STA** flags indicate the device is *not present* and *not functioning*, do not examine the children of this device – abort the walk of this subtree of the namespace.
- If the **_STA** flags indicate that the device is *present*, then attempt to execute the **_INI** method for the device.
- The global initialization handler is called after the execution of every **_INI** method.

4.2.5.2 Other ACPI Object Initialization

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.)*

4.2.6 Other Operating System ACPI-related Initialization

All external ACPI interfaces are available and the host OS can perform the following initialization steps:

- Enumerate devices using the **_HID** method
- Load, configure, and install device drivers
- Device Drivers install handlers for other address spaces such as SMBus, EC, IPMI, 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.)

4.2.7 Just-in-time Operation Region Initialization

This phase includes just-in-time initialization for any Operation Regions, Packages, Buffers, or Fields 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.

4.2.7.1 SystemMemory Region Initialization

For each operation region within the SystemMemory address space, a memory mapped window of maximum size **ACPI_SYSMEM_REGION_WINDOW_SIZE** is maintained, in an attempt to minimize the overhead of mapping entire operation regions if they are very large.

When a request is received that is outside of the current window, the existing mapping is deleted and a new mapping that can service the request is created.

This mapping feature is implemented in the default handler for the SystemMemory address space.

4.2.7.2 PCI_Config Region Initialization

For these operation regions, the namespace is searched upwards from the region to find the corresponding PCI Root Bridge.

If a **_HID** or **_CID** method under a device object indicates the presence of a PCI Root Bridge (an ID value of PNP0A03 or PNP0A08 for PCI Express), perform PCI Configuration Space initialization on the bridge. Install the PCI address space handler on the bridge (and on all descendents) and run the **_REG** method for the device if it is present. Then execute the **_ADR**, **_SEG**, and **_BBN** methods (in the bridge scope) to obtain the PCI Device, Function, Segment, and Bus numbers. Finally, run the associated **_REG** method to indicate the availability of the region.

- The initial PCI Device and Function values are obtained from the **_ADR** method.
- The initial PCI Segment number is obtained from the **_SEG** method.
- The initial PCI Bus number is obtained from the **_BBN** method.
- The final PCI ID consisting of Device, Function, Segment, and Bus is obtained by calling the AcpiOsDerivePciId OSL interface. This allows the host OS to make any adjustments to the PCI ID as required.

When accessing a PCI_Config operation region, all I/O from/to the PCI confituration space is performed via the OSL interfaces AcpiOsReadPciConfiguration and AcpiOsWritePciConfiguration.

4.2.8 System Shutdown - AcpiTerminate

This step frees all dynamically allocated resources back to the host operating system The ACPICA subsystem may be re-initialized and restarted from the beginning anytime after this step completes.

4.3 Multithreading Support

4.3.1 Reentrancy

All external interfaces to the ACPICA 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 simply block until the interpreter becomes available.

4.3.2 Mutual Exclusion and Synchronization

Three different types of synchronization objects are used by the ACPICA code:

1. **Mutex objects**. These objects are used for high-level mutual exclusion within the ACPICA core and AML interpreter and to implement the ASL Mutex operators, as well as the ACPI Global Lock. If there are no mutex primitives available in the host OS, they can be implemented with semaphore objects (binary semaphores.)

2. **Semaphore objects**. These objects are used for synchronization and to implement the ASL Event operators.

3. **Spin Locks**. These objects are only used at interrupt level (in interrupt handlers).

4.3.3 Control Method Execution

Most of the multithread support within the ACPICA 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 these 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.*

4.3.3.1 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 Operation Region 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

4.3.3.2 Control Method Execution Rules

Here are some Control Method execution "rules" that the ACPICA 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 Operation Regions and mentioned in the ACPI specification seems to depend entirely on the non-preemptive control method execution model (see above.)

4.3.3.3 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 **AcpiOsMutex**.
3. **Wait ()** - implemented via an **AcpiOsSemaphore**.
4. **Global Lock** - implemented via an **AcpiOsMutex** 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 re-queue 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 (system wide) 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)
```

4.3.3.4 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.

4.3.4 ACPI 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.

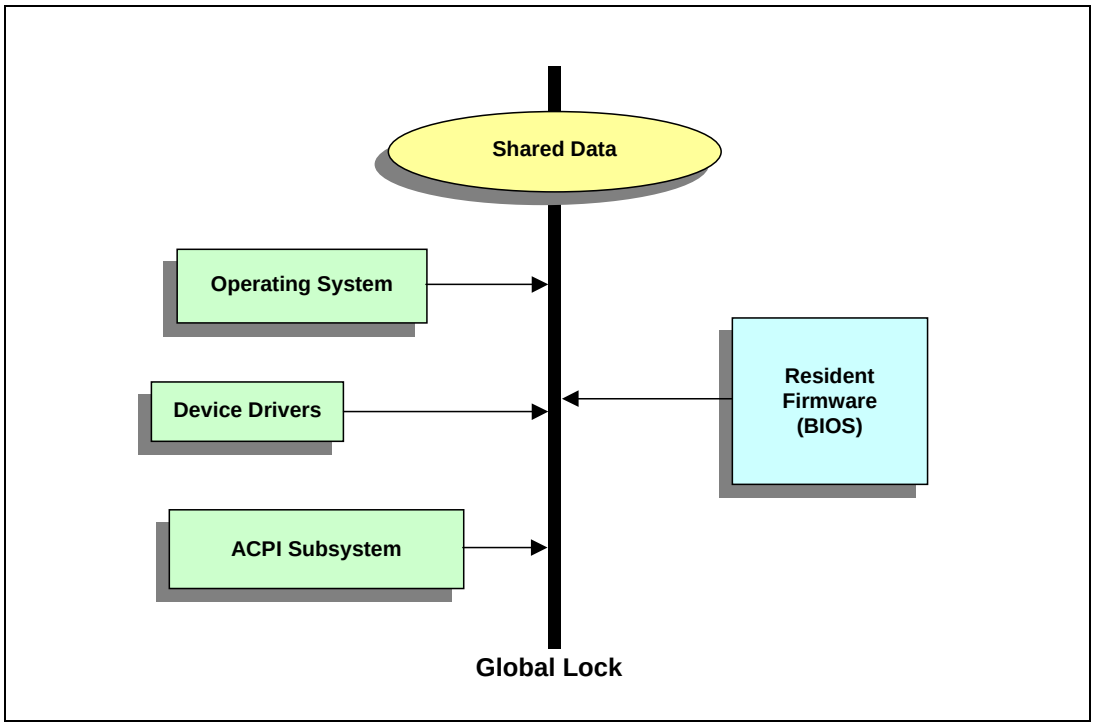
The ACPICA subsystem manages the global lock in the following manner:

- When the firmware owns the global lock, ACPICA queues up all requests to acquire the global lock.
- When the firmware releases the global lock, ACPICA satisfies all queued requests one at a time. A separate hardware acquire and release is performed for each thread that has requested the lock.

This algorithm prevents starvation of the global lock if many OS threads are requesting it. The BIOS has the opportunity to acquire the lock after each requesting thread releases it.

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

Figure 8. Global Lock Architecture



4.3.4.1 Obtaining The Global Lock

```
/* Only one thread can acquire the lock at a time */

Acquire the internal global lock mutex
If (AcquireHardwareGlobalLock())
{
     GlobalLockAquired = TRUE;
     return; /* All done! */
}

/* Must wait until the BIOS releases the lock and generates interrupt */

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

4.3.4.2 Releasing the Global Lock

```
If global lock is not acquired
     Error, return;

ReleaseHardwareGlobalLock ();
If Pending bit set
     Write the GBL_RLS bit to the control register

GlobalLockAquired = FALSE;
Release the internal global lock mutex
```

4.3.4.3 Global Lock Interrupt Handler

```
/* We get an SCI when the firmware releases the lock */

AcquireHardwareGlobalLock ()
If (Global Locak was acquired)
{
       GlobalLockAcquired = TRUE;
       AcpiOsSignalSemaphore (GlobalLockSemaphore);
}
```

4.3.5 Single Thread Environments

Both the design and implementation of the ACPICA 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 ACPICA 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 mechanisms used for mutual exclusion and multithread synchronization throughout the ACPICA subsystem are the OSL *Spinlock*, *Mutex*, and *Semaphore*. Since these mechanisms are 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.

5 Subsystem Features

5.1 AML Interpreter Slack Mode

When enabled, this mode provides better compatibility with other existing ACPI implementation(s) by ignoring certain errors and improper AML sequences. It also enables the *Implicit Return* feature.

Implicit Return Value: This feature will automatically return the result of the last AML operation in a control method, in the absence of an explicit Return() operator. Since other ACPI implementations have implemented this feature by default, there are many existing machines whose ASL/AML depends on this behavior.

Operation Region Range Checking: Allow access beyond the end of of a region. The default behavior is to strictly limit access to the end of the operation region. Typically, access beyond the end of the region occurs when the access data width causes the overrun. For example, a one-byte operation region and a field with DWORD access. Normally, access to the field will cause an error. This option will allow the access to continue.

Uninitialized Method Locals and Arguments: Allow access to uninitialized Locals and Arguments as if they were initialized to an Integer object with a value of zero. If this feature is not enabled, an error is generated an the method is aborted.

Source Operand Types for Store Operator: Allow objects of any type to be the source for the ASL/AML Store operator. The ACPI specification restricts the source operand to be one of a subset of the available ACPI object types. This option overrides the ACPI specification and allows source operands of any type.

Unresolved References within Packages: Allow references within Package objects to go unresolved with no error or warning. A NULL package element is inserted instead. This is another compatibility issue with other AML interpreters, and there are existing machines that depend on this feature.

5.2 AML Interpreter Math Mode (32-bit or 64-bit)

The integer size used by the AML interpreter is variable and is dynamically set via the DSDT that is loaded. For ACPI 1.0 DSDTs with a version number of 1, the integer width used is always 32-bits for backward compatibility. For ACPI 2.0 and later DSDTs with a version number larger than 1, full 64-bit integer math is used.

5.3 Predefined Control Method Validation

For the predefined control methods (methods that are defined in the ACPI specification and whose names begin with a single underscore), the ACPICA subsystem performs a validation on the return value, if any. There are nearly 200 such methods.

The input number of arguments and the type of the return object is validated against the ACPI specification. If the method returns a package, the length of the package as well as the individual elements of the package are validated. A warning message is issued if there are any problems found.

This feature is useful in finding problems with objects returned by BIOS AML code immediately upon execution of the method -- before the ACPI-related device drivers run into them.

5.4 I/O Port Protection

The ACPICA subsystem protects certain I/O ports from access via the AML code. Some ports are always illegal, and some ports are illegal based upon the strings that the BIOS has requested via the **_OSI** predefined control method. When an I/O request is made to a blocked port, the **AE_AML_ILLEGAL_ADDRESS** exception is returned.

The current list of protected ports is as follows:

```
{"DMA",     0x0000, 0x000F, ACPI_OSI_WIN_XP},       /* DMA controller 1 */
{"PIC0",    0x0020, 0x0021, ACPI_ALWAYS_ILLEGAL},   /* Interrupt Controller */
{"PIT1",    0x0040, 0x0043, ACPI_OSI_WIN_XP},       /* System Timer 1 */
{"PIT2",    0x0048, 0x004B, ACPI_OSI_WIN_XP},       /* System Timer 2 failsafe */
{"RTC",     0x0070, 0x0071, ACPI_OSI_WIN_XP},       /* Real-time clock */
{"CMOS",    0x0074, 0x0076, ACPI_OSI_WIN_XP},       /* Extended CMOS */
{"DMA1",    0x0081, 0x0083, ACPI_OSI_WIN_XP},       /* DMA 1 page registers */
{"DMA1L",   0x0087, 0x0087, ACPI_OSI_WIN_XP},       /* DMA 1 Ch 0 low page */
{"DMA2",    0x0089, 0x008B, ACPI_OSI_WIN_XP},       /* DMA 2 Ch 2 low page */
{"DMA2L",   0x008F, 0x008F, ACPI_OSI_WIN_XP},       /* DMA 2 low page refresh */
{"ARBC",    0x0090, 0x0091, ACPI_OSI_WIN_XP},       /* Arbitration control */
{"SETUP",   0x0093, 0x0094, ACPI_OSI_WIN_XP},       /* System board setup */
{"POS",     0x0096, 0x0097, ACPI_OSI_WIN_XP},       /* POS channel select */
{"PIC1",    0x00A0, 0x00A1, ACPI_ALWAYS_ILLEGAL},   /* Cascaded PIC */
{"IDMA",    0x00C0, 0x00DF, ACPI_OSI_WIN_XP},       /* ISA DMA */
{"ELCR",    0x04D0, 0x04D1, ACPI_ALWAYS_ILLEGAL},   /* PIC edge/level registers */
{"PCI",     0x0CF8, 0x0CFF, ACPI_OSI_WIN_XP}        /* PCI configuration space */
```

ACPI_ALWAYS_ILLEGAL: These ports are always blocked.

ACPI_OSI_WIN_XP: These ports are legal unless the BIOS AML has invoked **_OSI** with the XP string “Windows 2001” or any Windows string representing a release of Windows later than XP. Performed for Windows compatibility, this means that these ports are illegal on most modern x86 machines.

5.5 Debugging Support

Two styles of debugging are supported with the debugging tools available with the ACPICA 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 ACPICA 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.

5.5.1 Error and Warning Messages

There are several macros used throughout the ACPICA subsystem to format and print error and warning messages. In addition to the input message, each of these macros automatically print the module name, line number, and current ACPICA version number.

These macros are conditionally compiled and can be removed if desired by defining **ACPI_NO_ERROR_MESSAGES** during subsystem compilation. However, they are used only for serious issues in order to limit their overhead.

ACPI_ERROR – Displays an error message.

ACPI_EXCEPTION – Displays an error message with a decoded ACPI_STATUS exception.

ACPI_WARNING – Displays a warning message.

ACPI_INFO – Information message only.

The current statistics for the use of these macros within the ACPICA source is as follows:

ACPI_ERROR	284 invocations
ACPI_EXCEPTION	55 invocations
ACPI_WARNING	41 invocations
ACPI_INFO	8 invocations

5.5.2 Execution Debug Output (ACPI_DEBUG_PRINT Macro)

The ACPI_DEBUG_PRINT macro is used throughout the source code of the ACPICA Core Subsystem to selectively print debug messages. Over 350 invocations of the ACPI_DEBUG_PRINT are scattered throughout the ACPICA 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. **AcpiDbgLayer** Bit field that enables/disables debug output from entire subcomponents within the ACPICA subsystem.
2. **AcpiDbgLevel** Bit field 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
```

5.5.3 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.

5.5.4 ACPICA Debugger

Provided as a subcomponent of the ACPICA Core Subsystem, the 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.

5.6 Environmental Support Requirements

This section describes the environmental requirements of the ACPICA 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.

5.6.1 Resource Requirements

Static Memory - example Code and Data Size: These are the sizes for the OS-independent acpica.lib produced by the Microsoft Visual C++ 6.0 32-bit compiler. The debug version of the code includes the debug output trace mechanism and has a much larger code and data size.

```
Non-Debug Version:  81.2K Code, 17.0K Data,  98.2K Total
Debug Version:     155.8K Code, 49.1K Data, 204.9K Total
```

Dynamic Memory: The size of the internal ACPI namespace is dependent on the size of the loaded ACPI tables – DSDT and any SSDTs – and the number of named ACPI objects they create at table load time. All resources used during control method execution are freed at control method termination.

5.6.2 C Library Functions

In order to make the ACPICA 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

isalpha
isdigit
isprint
isspace
isupper
isxdigit
memcmp
memcpy
memset
strcat
strcmp
strcpy
strlen
strncat
strncmp
strncpy
strstr
strtoul
strupr
tolower
toupper
va_end
va_list
va_start

If **ACPI_USE_SYSTEM_CLIBRARY** is defined during the compilation of the subsystem, the subsystem must be linked to a local C library to resolve these Clib references. If **ACPI_USE_SYSTEM_CLIBRARY** 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.

5.6.3 Source Code Organization

The ACPICA source code as released is organized as below. At the top level, there are separate directories for the ACPICA documentation, generation tools, and the actual C source code. The source code itself is organized into a separate directory for each major ACPICA component, tool, or test.

```
acpica
     documents                    // Acpica documentation
     generate                     // Source generation tools:
          lint                    // PC-lint files
          linux                   // Linux makefiles
          msvc                    // Microsoft VC++ 6.0 makefiles
          release                 // Release utilities
          unix                    // Generic Unix/gcc makefiles
     source                       // Entire ACPICA source code tree:
          common                  // Common files
          compiler                // iASL compiler
          components              // Main ACPICA components:
               debugger           // AML Debugger
               disassembler       // AML Disassembler
               dispatcher         // AML Interpreter dispatcher
               events             // ACPI Event Manager (GPEs etc.)
               executer           // Main AML Interpreter
               hardware           // ACPI Hardware Manager
               namespace          // ACPI Namespace Manager
               parser             // AML Interpreter parser
               resources          // ACPI Resource Manager
               tables             // ACPI Table Manager
               utilities          // Miscellaneous utilities
          include                 // Most ACPICA includes
               platform           // Platform-specific files
          os_specific             // OS-specific files
               service_layers     // Various OSLs
          tools                   // ACPICA tools/utilities:
               acpibin            // Binary file utility
               acpiexec           // ACPI user space executer
               acpisrc            // Source translation utility
               acpixtract         // Table extraction utility
               examples           // ACPICA example code
          tests                   // ACPICA test suites:
               aapits             // ACPICA interface tests
               aslts              // ASL test suite
               misc               // Miscellaneous ASL tests
```

5.6.4 System Include Files

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

- acpi.h — Includes all of the files below.
- 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

5.6.4.1 Customization to the Target Environment

The use of header files that are external to the ACPICA subsystem is confined to a single header file named *acenv.h*. These external include files are used only if the following symbols are defined:

- **ACPI_USE_SYSTEM_CLIBRARY**
- **ACPI_USE_STANDARD_HEADERS**

Several of the standard C library headers are used:

- stdarg.h
- stdlib.h
- string.h
- ctype.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.

6 Data Types and Interface Parameters

6.1 ACPICA Interface Parameters

6.1.1 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 ACPICA 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 ACPICA 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.

6.1.2 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 ACPICA 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.

6.1.3 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 ACPICA 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.

6.2 ACPICA Basic Data Types

6.2.1 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 ACPICA 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.

6.2.2 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.

6.2.3 ACPI_IO_ADDRESS

Similar to ACPI_PHYSICAL_ADDRESS, except it is used for I/O addresses.

6.2.4 ACPI_SIZE

This data type is 32-bits or 64 bits depending on the platform. It is used in leiu of the C library **size_t**, which cannot be guaranteed to be available.

6.2.5 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.

6.2.6 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.

6.2.7 ACPI_BUFFER – Input and Output Memory Buffers

Many of the ACPICA 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 of the buffer in bytes;
     void          *Pointer;     // pointer to buffer
} ACPI_BUFFER;
```

6.2.7.1 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.

6.2.7.2 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 ACPICA 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 ACPICA 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.

6.2.8 ACPI_STATUS – Interface Exception Return Codes

Most 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 = AcpiInitializeSubsystem ();
if (ACPI_FAILURE (Status))
{
     // Exception handling code here
}
```

6.2.9 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.

6.2.9.1 Predefined Handles

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

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

6.2.10 ACPI_OBJECT_TYPE – Object Type Codes

Each ACPI object that is managed by the ACPICA 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_INTEGER
ACPI_TYPE_STRING
ACPI_TYPE_BUFFER
ACPI_TYPE_PACKAGE
ACPI_TYPE_FIELD_UNIT
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_BUFFER_FIELD
ACPI_TYPE_DDB_HANDLE
ACPI_TYPE_DEBUG_OBJECT

6.2.11 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 acpi_object
{
    ACPI_OBJECT_TYPE    Type;   /* See definition of AcpiNsType for values */
    struct
    {
        ACPI_OBJECT_TYPE    Type;       /* ACPI_TYPE_INTEGER */
        ACPI_INTEGER        Value;      /* The actual number */
    } Integer;
```

```
        struct
        {
            ACPI_OBJECT_TYPE    Type;       /* ACPI_TYPE_STRING */
            UINT32              Length;     /* # of bytes in string, minus null */
            char                *Pointer;   /* points to the string value */
        } String;

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

        struct
        {
            ACPI_OBJECT_TYPE    Type;       /* ACPI_TYPE_PACKAGE */
            UINT32              Count;      /* # of elements in package */
            union acpi_object   *Elements;  /* Pointer to array of ACPI_OBJECTs */
        } Package;

        struct
        {
            ACPI_OBJECT_TYPE    Type;       /* ACPI_TYPE_LOCAL_REFERENCE */
            ACPI_OBJECT_TYPE    ActualType; /* Type associated with the Handle */
            ACPI_HANDLE         Handle;     /* object reference */
        } Reference;

        struct
        {
            ACPI_OBJECT_TYPE    Type;       /* ACPI_TYPE_PROCESSOR */
            UINT32              ProcId;
            ACPI_IO_ADDRESS     PblkAddress;
            UINT32              PblkLength;
        } Processor;

        struct
        {
            ACPI_OBJECT_TYPE    Type;       /* ACPI_TYPE_POWER */
            UINT32              SystemLevel;
            UINT32              ResourceOrder;
        } PowerResource;

    } ACPI_OBJECT;
```

6.2.12 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;
```

6.2.13 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.

```
ACPI_EVENT_PMTIMER        // Power Management Timer rollover
ACPI_EVENT_GLOBAL         // Global Lock released
ACPI_EVENT_POWER_BUTTON   // Power Button (pressed)
ACPI_EVENT_SLEEP_BUTTON   // Sleep Button (pressed)
ACPI_EVENT_RTC            // Real Time Clock alarm
```

6.2.14 ACPI_TABLE_HEADER – Common ACPI Table Header

This is the header used for most of the BIOS-provided ACPI tables.

```
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;
```

6.3 ACPI Resource Data Types

These data types are used by the ACPICA resource interfaces.

6.3.1 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 “char Source[4]”, it can be de-referenced as a null-terminated string.

```
typedef struct acpi_pci_routing_table
{
    UINT32        Length;
    UINT32        Pin;            /* PCI Pin */
    ACPI_INTEGER  Address;        /* PCI Address of device */
    UINT32        SourceIndex;    /* Index of resource, allocating dev */
    char          Source[4];      /* pad to 64 bits so sizeof() works */

} ACPI_PCI_ROUTING_TABLE;
```

6.3.2 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.

6.3.2.1 ACPI_RESOURCE_TYPE – Resource Data Types

The following resource types are supported by the ACPICA 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. Address64
16. ExtendedAddress64
17. ExtendedIrq
18. GenericRegister

```
typedef union acpi_resource_data /* union of all resources */
{
    ACPI_RESOURCE_IRQ                       Irq;
    ACPI_RESOURCE_DMA                       Dma;
    ACPI_RESOURCE_START_DEPENDENT           StartDpf;
    ACPI_RESOURCE_IO                        Io;
    ACPI_RESOURCE_FIXED_IO                  FixedIo;
    ACPI_RESOURCE_VENDOR                    Vendor;
    ACPI_RESOURCE_VENDOR_TYPED              VendorTyped;
    ACPI_RESOURCE_END_TAG                   EndTag;
    ACPI_RESOURCE_MEMORY24                  Memory24;
    ACPI_RESOURCE_MEMORY32                  Memory32;
    ACPI_RESOURCE_FIXED_MEMORY32            FixedMemory32;
    ACPI_RESOURCE_ADDRESS16                 Address16;
    ACPI_RESOURCE_ADDRESS32                 Address32;
    ACPI_RESOURCE_ADDRESS64                 Address64;
    ACPI_RESOURCE_EXTENDED_ADDRESS64        ExtAddress64;
    ACPI_RESOURCE_EXTENDED_IRQ              ExtendedIrq;
    ACPI_RESOURCE_GENERIC_REGISTER          GenericReg;

} ACPI_RESOURCE_DATA;

typedef struct acpi_resource
{
    UINT32                          Type;
    UINT32                          Length;
    ACPI_RESOURCE_DATA              Data;

} ACPI_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*.

6.4 ACPICA Exception Codes

A common and consistent set of return codes is used throughout the ACPICA 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 3. Exception Code Values

Exception Name	Typical Meaning
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_ALREADY_EXISTS	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_SUPPORT	The feature is not supported

Exception Name	Typical Meaning
AE_LIMIT	A predefined limit was exceeded
AE_TIME	A time limit or timeout expired
AE_ACQUIRE_DEADLOCK	Internal error – attempt was made to acquire a mutex in improper order
AE_RELEASE_DEADLOCK	Internal error – attempt was made to release a mutex in improper order
AE_NOT_ACQUIRED	An attempt to release a mutex or the Global Lock without a previous acquire
AE_ALREADY_ACQUIRED	Internal error – attempt was made to acquire a mutex twice
AE_NO_HARDWARE_RESPONSE	Hardware did not respond after an I/O operation
AE_NO_GLOBAL_LOCK	There is no hardware Global Lock
AE_ABORT_METHOD	A control method was aborted
AE_SAME_HANDLER	Attempt was made to install the same handler that is already installed.
AE_OWNER_ID_LIMIT	There are no more Owner IDs available for ACPI tables or control methods
Programmer Exceptions (ACPI external interfaces)	
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_HEX_CONSTANT	Invalid character in a Hex constant
AE_BAD_OCTAL_CONSTANT	Invalid character in an Octal constant
AE_BAD_DECIMAL_CONSTANT	Invalid character in a Decimal constant
AE_MISSING_ARGUMENTS	To few arguments were passed to a control method
AE_BAD_ADDRESS	A null I/O address was passed as a parameter to AcpiRead or AcpiWrite
ACPI Table Exceptions	
AE_BAD_SIGNATURE	An ACPI table has an invalid 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
AE_INVALID_TABLE_LENGTH	The FADT or FACS has improper length
AML (Interpreter) Exceptions	
AE_AML_BAD_OPCODE	Invalid AML opcode encountered
AE_AML_NO_OPERAND	An operand is missing (such as a method that did not return a required value)

Exception Name	Typical Meaning
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 variable
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 the end of an 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	During execution of AML Divide operator
AE_AML_BAD_NAME	An ACPI name contains invalid character(s)
AE_AML_NAME_NOT_FOUND	Could not resolve a named reference
AE_AML_INTERNAL	An internal error within the interpreter
AE_AML_INVALID_SPACE_ID	An Operation Region SpaceID is invalid
AE_AML_STRING_LIMIT	String is longer than 200 characters
AE_AML_NO_RETURN_VALUE	A method did not return a required value
AE_AML_METHOD_LIMIT	A control method reached the maximum reentrancy limit of 255
AE_AML_NOT_OWNER	A thread tried to release a mutex that it does not own
AE_AML_MUTEX_ORDER	Mutex SyncLevel release mismatch
AE_AML_MUTEX_NOT_ACQUIRED	Attempt to release a mutex that was not previously acquired
AE_AML_INVALID_RESOURCE_TYPE	Invalid resource type in resource list
AE_AML_INVALID_INDEX	Invalid Argx or Localx (x too large)
AE_AML_REGISTER_LIMIT	Bank value or Index value beyond range of register
AE_AML_NO_WHILE	Break or Continue without a While
AE_AML_ALIGNMENT	Non-aligned memory transfer on platform that does not support this
AE_AML_NO_RESOURCE_END_TAG	No End Tag in a resource list
AE_AML_BAD_RESOURCE_VALUE	Invalid value of a resource element
AE_AML_CIRCULAR_REFERENCE	Two references refer to each other
AE_AML_BAD_RESOURCE_LENGTH	The length of a Resource Descriptor in the AML was incorrect
AE_AML_ILLEGAL_ADDRESS	A memory, I/O, or PCI configuration address was invalid

Exception Name	Typical Meaning
AE_AML_INFINITE_LOOP	An AML While loop appears to have been stuck infinitely and the method was aborted
Internal Exceptions used for control	
AE_CTRL_RETURN_VALUE	A Method returned a value
AE_CTRL_PENDING	Method is calling another method
AE_CTRL_TERMINATE	Terminate the executing method
AE_CTRL_TRUE	An If or While predicate result
AE_CTRL_FALSE	An If or While predicate result
AE_CTRL_DEPTH	Maximum search depth has been reached
AE_CTRL_END	An If or While predicate is false
AE_CTRL_TRANSFER	Transfer control to called method
AE_CTRL_BREAK	A Break has been executed
AE_CTRL_CONTINUE	A Continue has been executed
AE_CTRL_PARSE_CONTINUE	Used to skip over bad opcodes
AE_CTRL_PARSE_PENDING	Used to implement AML While loops

7 Subsystem Configuration

There are several methods of configuring the OS-independent ACPICA Core Subsystem:

1. Selection of individual ACPICA components.

2. Configuration of platform-specific data types.

3. Per-machine configuration for machine-specific dependencies.

4. Per-compiler configuration for compiler dependencies.

5. Other compile-time configuration through the use of compiler switches.

6. Run-time global variables which are statically initialized from the configuration header file.

7.1 Configuration Files

The ACPICA subsystem has three types of configuration header files to allow the subsystem to be tailored to the particular machine and compiler, as well as allowing for the tuning of subsystem constants.

These three include files perform the subsystem configuration:

- An include file that is specific to the particular compiler being used to compile the ACPICA subsystem provides macros and defines that must be implemented on a per-compiler basis. These files appear in the *include/platform* directory.
- An include file that is specific to the particular machine being targeted for the ACPICA subsystem provides macros and defines that must be implemented on a per-machine basis. These files appear in the *include/platform* directory.
- A global include file, *acconfig.h* allows for the tailoring and tuning of various subsystem constants and options. This file appears in the *include* directory

7.2 Component Selection

7.2.1 ACPI_DISASSEMBLER

This switch enables the AML Disassembler component, which is usually used in conjunction with the ACPI Debugger.

7.2.2 ACPI_DEBUGGER

This switch enables the ACPICA Debugger component. It also enables the various object dumping routines.

7.3 Configurable Data Types

The configurable data types are used to help tailor the ACPICA subsystem to a particular operation system or compiler. Any changes from the default values should be specified in a system-dependent header file under the *include/platform* directory.

7.3.1 ACPI_SPINLOCK

This type is an OS-dependent handle for a spinlock. It is returned by the *AcpiOsCreateLock* interface, and passed as a parameter to the *AcpiOsAcquireLock* and *AcpiOsReleaseLock* interfaces. The default value for **ACPI_SPINLOCK** is (void *). It can be changed to whatever type the host OS uses for spinlocks.

7.3.2 ACPI_SEMAPHORE

This type is an OS-dependent handle for a semaphore. It is returned by the *AcpiOsCreateSemaphore* interface, and passed as a parameter to the *AcpiOsWaitSemaphore* and *AcpiOsSignalSemaphore* interfaces. The default value for **ACPI_SEMAPHORE** is (void *). It can be changed to whatever type the host OS uses for semaphore objects.

7.3.3 ACPI_MUTEX

This type is an OS-dependent handle for a mutex. It is returned by the *AcpiOsCreateMutex* interface, and passed as a parameter to the *AcpiOsAcquireMutex* and *AcpiOsReleaseMutex* interfaces. The default value for **ACPI_MUTEX** is (void *). It can be changed to whatever type the host OS uses for mutex objects.

If mutex objects are not supported by the host operating system, use the **ACPI_MUTEX_TYPE** with the **ACPI_BINARY_SEMAPHORE** option (described later). This option causes mutexes to be automatically implemented via **ACPI_SEMAPHORE** objects, and the OSL mutex interfaces are not required.

7.3.4 ACPI_CPU_FLAGS

This type is used for the value returned from *AcpiOsAcquireLock*, and the value passed as a parameter to *AcpiOsReleaseLock.* It can be configured to whatever type the host OS uses for CPU flags that need to be saved and restored across the acquisition and release of a spinlock. The default value is **ACPI_SIZE**.

7.3.5 ACPI_THREAD_ID

This type is an OS-dependent Thread ID that is returned by the *AcpiOsGetThreadId* interface. The default type for **ACPI_THREAD_ID** is **ACPI_SIZE**, but it is configurable since some operating systems implement a thread ID as a pointer.

7.3.6 ACPI_CACHE_T

This type is used for the value returned from *AcpiOsCreateCache*. It is used as a parameter to the various OSL cache interfaces to identify a cache object for operating systems that implement a cache manager. If the local ACPICA cache memory manager is used (configured), the value for this type is **ACPI_MEMORY_LIST**. Otherwise, the value is OS-dependent.

7.3.7 ACPI_UINTPTR_T

This type is introduced to assist compilation of ACPICA under a C99 compiler that implements the **uintptr_t** type. It is used for casting of pointers to eliminate compiler warnings. The default value for the non-C99 case is (void *).

7.4 Subsystem Options

These defines are used to customize the ACPICA Subsystem at compile time by selecting or disabling various features.

7.4.1 ACPI_USE_SYSTEM_CLIBRARY

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.

7.4.2 ACPI_USE_STANDARD_HEADERS

This switch allows the use of standard C library headers that are provided by the host. The following C library headers are used:

```
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

7.4.3 ACPI_DEBUG_OUTPUT

This switch enables all debug facilities within ACPICA. This includes the **ACPI_DEBUG_PRINT** output statements, the **ACPI_FUNCTION_TRACE** tracing statements, and the various object dumping routines. If disabled, all of these macros evaluate to NULL and no code is produced.

7.4.4 ACPI_USE_LOCAL_CACHE

This switch enable the local ACPICA cache manager code. The use of a cache can improve the ACPICA performance considerably, since it frequently allocations and deallocates objects of identical size. If the host OS provides a similar cache manager, the ACPICA cache manager is not needed.

7.4.5 ACPI_DBG_TRACK_ALLOCATIONS

This switch enables the ACPICA cache statistics mechanism, and is only applicable if the local ACPICA cache manager is enabled (**ACPI_USE_LOCAL_CACHE**.) When enabled, information about each cache is saved, including the total memory allocated/freed, total requests, cache hits/misses, etc. This information can be displayed via the ACPICA Debugger.

7.4.6 ACPI_MUTEX_TYPE

This macro is used to define the type of mutex support desired. Either native (host OS) mutexes may be used, or binary semaphores may be used. The default behavior is to use binary semaphores.

The **ACPI_MUTEX_TYPE** must be one of the two following values:

ACPI_BINARY_SEMAPHORE (default)

Use this value if the host OS does not support **mutex** objects. If set, this switch enables the automatic use of macros that implement the mutex interfaces via binary semaphores, and the various mutex interfaces do not need to be implemented in the OSL.

ACPI_OSL_MUTEX

Use this value if the host OS supports **mutex** objects. The various mutex interfaces must be implemented in the OSL:

- AcpiOsCreateMutex
- AcpiOsDeleteMutex
- AcpiOsAcquireMutex
- AcpiOsReleaseMutex

7.4.7 ACPI_MUTEX_DEBUG

Enables code that performs error checking on the use of mutex objects. It checks for possible deadlock conditions by enforcing a mutex ordering rule. Use of this option can impact performance considerably, so it it should only used for debugging.

7.4.8 ACPI_SIMPLE_RETURN_MACROS

Enables simplified return macros. The default implementation for the return macros has extra protection so that the macro parameter is not evaluated twice. The simplified versions of these macros are smaller, but the parameter can be evaluated twice

Protected macro:

```
#define return_ACPI_STATUS(s) \
    ACPI_DO_WHILE0 ({ \
        register ACPI_STATUS _s = (s); \
        AcpiUtStatusExit (ACPI_DEBUG_PARAMETERS, _s); \
        return (_s); })
```

Simplified macro:

```
#define return_ACPI_STATUS(s) \
    ACPI_DO_WHILE0 ({ \
        AcpiUtStatusExit (ACPI_DEBUG_PARAMETERS, (s)); \
        return((s)); })
```

7.4.9 ACPI_USE_DO_WHILE_0

Inserts a do … while(0) statement around the return macros (see examples above). Prevents some compilers from issuing warnings for these macros.

Default implementation:

```
#define ACPI_DO_WHILE0(a)               do a while(0)
```

7.5 Per-Compiler Configuration

These macros and defines allow the ACPICA subsystem to be tailored to a particular compiler.

7.5.1 COMPILER_DEPENDENT_INT64

Defines the name of a signed 64-bit integer on for this compiler. This macro is required because there is (currently) no standard method to define 64-bit integers in the C language. There is no default, this macro **must** be defined by the platform configuration file.

Examples

```
#define COMPILER_DEPENDENT_INT64   int64_t
#define COMPILER_DEPENDENT_INT64   long
#define COMPILER_DEPENDENT_INT64   __int64
#define COMPILER_DEPENDENT_INT64   long long
```

7.5.2 COMPILER_DEPENDENT_UINT64

Defines the name of an unsigned 64-bit integer on for this compiler. This macro is required because there is (currently) no standard method to define 64-bit integers in the C language. There is no default, this macro **must** be defined by the platform configuration file.

Examples

```
#define COMPILER_DEPENDENT_UINT64  uint64_t
#define COMPILER_DEPENDENT_UINT64  unsigned long
#define COMPILER_DEPENDENT_UINT64  unsigned __int64
#define COMPILER_DEPENDENT_UINT64  unsigned long long
```

7.5.3 ACPI_USE_NATIVE_DIVIDE

This switch enables native 64-bit divides. It is set by default for 64-bit machine widths. It is optional for 32-bit platforms. Only use this option on a 32-bit platform if a 64-bit double-precision math library is available for use by ACPICA. If the library is not available, then do not use this option and a local ACPICA double-precision divide function is enabled instead.

7.5.4 ACPI_DIV_64_BY_32 (Short 64-bit Divide)

This macro performs a simple 64-bit divide with a 64-bit dividend and a 32-bit divisor. The purpose of this macro is to perform a short divide on 32-bit platforms without invoking a double-precision math library. Both the quotient and remainder must be returned. There is no default, this macro **must** be defined by the platform configuration file.

Example 32-bit Implementation

```
#define ACPI_DIV_64_BY_32(n_hi, n_lo, d32, q32, r32) \
{                                   \
    __asm mov    edx, n_hi  \
    __asm mov    eax, n_lo  \
    __asm div    d32        \
    __asm mov    q32, eax   \
    __asm mov    r32, edx   \
}
```

Example 64-bit Implementation

```
#define ACPI_DIV_64_BY_32(n, n_hi, n_lo, d32, q32, r32) \
{ \
    q32 = n / d32; \
    r32 = n % d32; \
}
```

7.5.5 ACPI_SHIFT_RIGHT_64 (64-bit Shift)

This macro performs a 64-bit right shift by one bit. The purpose of this macro is to perform a shift right on 32-bit platforms without invoking a double-precision math library. There is no default, this macro **must** be defined by the platform configuration file.

Example 32-bit Implementation

```
#define ACPI_SHIFT_RIGHT_64(n_hi, n_lo) \
{                                   \
    __asm shr    n_hi, 1    \
    __asm rcr    n_lo, 1    \
}
```

Example 64-bit Implementation

```
#define ACPI_SHIFT_RIGHT_64(n, n_hi, n_lo) \
{ \
    n <<= 1; \
}
```

7.5.6 ACPI_EXPORT_SYMBOL

This macro is used to define the mechanism used to export public symbols, if applicable. Within ACPICA, it is invoked for each of the public interfaces. The default value is NULL.

Example

```
#define ACPI_EXPORT_SYMBOL(Symbol)  EXPORT_SYMBOL(Symbol);
```

7.5.7 ACPI_EXTERNAL_XFACE

This macro allows the definition of an interface type prefix (such as _cdecl, pascal, etc.) to be used in the declaration of all ACPICA external interfaces (the **Acpi*** interfaces.) The default value is NULL.

Example

```
#define ACPI_EXTERNAL_XFACE          APIENTRY
```

7.5.8 ACPI_INTERNAL_XFACE

This macro allows the definition of an interface type prefix (such as **_cdecl**, **pascal**, etc.) to be used in the declaration of all ACPICA internal interfaces. The default value is NULL.

7.5.9 ACPI_INTERNAL_VAR_XFACE

This macro allows the definition of an interface type prefix (such as **_cdecl**, **pascal**, etc.) to be used in the declaration of all ACPICA variable-argument list internal interfaces. The default value is NULL.

Example

```
#define ACPI_INTERNAL_VAR_XFACE      __cdecl
```

7.5.10 ACPI_SYSTEM_XFACE

This macro allows the definition of an interface type prefix (such as **_cdecl**, **pascal**, etc.) to be used in the declaration of all interfaces to the host OS. The default value is NULL.

Examples

```
#define ACPI_SYSTEM_XFACE    __cdecl
#define ACPI_SYSTEM_XFACE    APIENTRY
```

7.5.11 ACPI_PRINTF_LIKE

This macro defines a suffix to be used in the definitions and prototypes of internal print functions that accept a printf-like format string. Some compilers have the ability to perform additional typechecking on such functions. The default value is NULL.

Example

```
#define ACPI_PRINTF_LIKE(c) \
            __attribute__ ((__format__ (__printf__, c, c+1)))
```

7.5.12 ACPI_UNUSED_VAR

This macro defines a prefix to be used in the definition of variables that may not be used in a module (such as the **ACPI_MODULE_NAME**). This can prevent compiler warnings for such variables. The default value is NULL.

Example

```
#define ACPI_UNUSED_VAR __attribute__ ((unused))
```

7.6 Per-Machine Configuration

These macros and defines allow the ACPICA subsystem to be tailored to a particular machine or machine architecture.

7.6.1 ACPI_MACHINE_WIDTH

This macro defines the standard integer width of the target machine, either 32 or 64. There is no default, this macro **must** be defined by the platform configuration file.

Examples

```
#define ACPI_MACHINE_WIDTH          32
#define ACPI_MACHINE_WIDTH          64
```

7.6.2 ACPI_FLUSH_CPU_CACHE

Defines the instruction or instructions necessary to flush the CPU cache(s) on this machine.

Examples

```
#define ACPI_FLUSH_CPU_CACHE()  __asm {WBINVD}
#define ACPI_FLUSH_CPU_CACHE()  wbinvd()
```

7.6.3 ACPI_OS_NAME

This defines the string that is returned by the predefined "_OS_" method in the ACPI namespace.

```
#define ACPI_OS_NAME    "Microsoft Windows NT"
```

The _OS_ object is essentially obsolete, but there is a large base of ASL/AML code in existing machines that check for the string above. The use of this string usually guarantees that the ASL will execute down the most tested code path. Also, there is some code that will not execute the _OSI method unless _OS_ matches the string avove. Therefore, change this string at your own risk.

7.6.4 ACPI_ACQUIRE_GLOBAL_LOCK

This macro defines the code (in assembly or C) necessary to acquire the ACPI Global Lock on this machine.

ACPI_ACQUIRE_GLOBAL_LOCK (FacsPtr, Acquired)

Where:

FacsPtr — is a pointer to the FACS table.

Acquired — is a boolean return value. TRUE if the lock was acquired; FALSE otherwise.

Example:

```
#define ACPI_ACQUIRE_GLOBAL_LOCK(FacsPtr, Acq)  __asm \
{                                                     \
        __asm mov           eax, 0xFF                 \
        __asm mov           ecx, FacsPtr              \
        __asm or            ecx, ecx                  \
        __asm jz            exit_acq                  \
        __asm lea           ecx, [ecx].GlobalLock     \
                                                      \
        __asm acq10:                                  \
        __asm mov           eax, [ecx]                \
        __asm mov           edx, eax                  \
        __asm and           edx, 0xFFFFFFFE           \
        __asm bts           edx, 1                    \
        __asm adc           edx, 0                    \
        __asm lock cmpxchg  dword ptr [ecx], edx      \
        __asm jnz           acq10                     \
                                                      \
        __asm cmp           dl, 3                     \
        __asm sbb           eax, eax                  \
                                                      \
        __asm exit_acq:                               \
        __asm mov           Acq, al                   \
}
```

7.6.5 ACPI_RELEASE_GLOBAL_LOCK

This macro defines the code (in assembly or C) necessary to release the ACPI Global Lock on this machine.

ACPI_RELEASE_GLOBAL_LOCK (FacsPtr, Pending)

Where:

FacsPtr — is a pointer to the FACS table.

Pending — is a boolean return value. TRUE if the global lock pending bit is set; FALSE otherwise.

Example:

```
#define ACPI_RELEASE_GLOBAL_LOCK(FacsPtr, Pnd) __asm \
{                                                    \
        __asm xor           eax, eax                 \
        __asm mov           ecx, FacsPtr             \
        __asm or            ecx, ecx                 \
        __asm jz            exit_rel                 \
        __asm lea           ecx, [ecx].GlobalLock    \
                                                     \
        __asm Rel10:                                 \
        __asm mov           eax, [ecx]               \
        __asm mov           edx, eax                 \
        __asm and           edx, 0xFFFFFFFC          \
        __asm lock cmpxchg  dword ptr [ecx], edx     \
        __asm jnz           Rel10                    \
                                                     \
        __asm cmp           dl, 3                    \
        __asm and           eax, 1                   \
                                                     \
        __asm exit_rel:                              \
        __asm mov           Pnd, al                  \
}
```

7.7 Dynamic Configuration

This section describes features that may be enabled or disabled at run-time by setting various ACPICA global option variables.

The global option variables are found in the *include/acglobal.h* header.

7.7.1 Interpreter Slack Mode

Enable or disable the AML Interpreter slack mode, as decribed earlier. The default is disabled.

```
ACPI_INIT_GLOBAL (AcpiGbl_EnableInterpreterSlack, FALSE);
```

7.7.2 ACPI Register Widths

This option can be used to override the ACPI register widths that are specified in the FADT in the case where the FADT contains one or more incorrect register widths (lengths). The default value is FALSE, do not use the default register widths -- use the values as specified in the FADT.

The default register widths are as follows:

PM1A Enable,
PM1A Status,
PM1A Control,
PM1B Enable,
PM1B Status,
PM1B Control -- 16 bits each, = **ACPI_PM1_REGISTER_WIDTH**

PM2 Control -- 8 bits, = **ACPI_PM2_REGISTER_WIDTH**

PM Timer -- 32 bits, = **ACPI_PM_TIMER_WIDTH**

```
ACPI_INIT_GLOBAL (AcpiGbl_UseDefaultRegisterWidths, FALSE);
```

7.7.3 Serialized Methods

This option can be used to force all control methods to be serialized. Meaning that only one thread can enter the method at a time, similar to the **Serialized** control method option. The default is to not force serialization and let each control method dictate the serialization mode for itself. The use of this option essentially forces the AML interpreter to be single threaded.

```
ACPI_INIT_GLOBAL (AcpiGbl_AllMethodsSerialized, FALSE);
```

7.7.4 Wake GPEs

This option controls whether “wake” GPEs should be enabled at runtime or not. A wake GPE is defined as a GPE that is used only to wake the system. The default is for all wake GPEs to be disabled at runtime. They are only enabled when the system is about to sleep. The wake GPEs are determined from the _PRW methods contained in the system AML.

```
ACPI_INIT_GLOBAL (AcpiGbl_LeaveWakeGpesDisabled, TRUE);
```

7.7.5 Creation of_OSI Method

This option controls whether the predefined _OSI method is created or not. The _OSI method was defined in ACPI 2.0 and is implemented internally within the ACPICA subsystem.

```
ACPI_INIT_GLOBAL (AcpiGbl_CreateOsiMethod, TRUE);
```

7.8 Subsystem Configuration Constants

The configurable subsystem constants are specified in the *include/acconfig.h* header file. These constants may be modified at either compile time by changing the constants in *acconfig.h*, or at run-time by changing the contents of the global variables where these constants are stored.

7.8.1 ACPI_CHECKSUM_ABORT

Defines whether the table manager should abort the loading of an ACPI table if the table checksum is incorrect. Possible values are **TRUE** or **FALSE**. The default is **FALSE**.

In practice, often table checksums are found to be incorrect, not because of corruption, but because the BIOS has modified the table on the fly according to BIOS configuration options, and has inadvertently forgotten to update the checksum. Therefore, the ACPI table checksum isn’t very useful and the default is to ignore checksum errors.

7.8.2 ACPI_MAX_LOOP_INTERATIONS

This defines the number of AML While() loop executions that are permitted before the infinite loop break mechanism is invoked. The default is 64K iterations, which is a very large number of

interations for an AML loop. This mechanism prevents a catastrophic infinite loop which would block the AML interpreter forever, effectively locking up most of the ACPICA subsystem.

Infinite loops can occur in poorly written AML in a hardware polling loop. For example, if the hardware simply does not respond and the loop does not implement a timeout.

7.8.3 ACPI_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.)

7.8.4 ACPI_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.

7.8.5 ACPI_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

7.8.6 ACPI_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.

8 ACPICA Core Subsystem - External Interface Definition

This section contains documentation for the specific interfaces exported by the ACPICA 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 ACPICA Core Subsystem are prefixed by the letters "**Acpi"**.

8.1 ACPICA Subsystem Initialization and Control

8.1.1 AcpiInitializeSubsystem

Initialize all ACPICA globals and sub-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 ACPICA subsystem, including the OS Services Layer. It must be called once before any of the other **Acpi*** interfaces are called (with the exception of the Table Manager interfaces these interfaces are independent and can be called at any time.)

8.1.2 AcpiInstallInitializationHandler

Install a global handler for initialization handling.

ACPI_STATUS
AcpiInstallInitializationHandler (
ACPI_INIT_HANDLER Handler,
UINT32 Function)

PARAMETERS

Handler	A pointer to the initialization handler.
Function	Reserved.

EXCEPTIONS

AE_OK	The ACPI namespace was successfully loaded and initialized.
AE_BAD_PARAMETER	The *Handler* parameter is invalid.
AE_ALREADY_EXISTS	A global initialization handler has already been installed.

Functional Description:

This function installs a global initialization handler that is called during the subsystem initialization.

Currently, the handler is called after each Device object within the namespace has been initialized (The **_INI** and **_STA** methods have been run on the device.)

8.1.2.1 Interface to User Callback Function

Interface to the user function that is installed via *AcpiInstallInitializationHandler*.

ACPI_STATUS (*ACPI_INIT_HANDLER) (
ACPI_HANDLE Object,
UINT32 Function)

PARAMETERS

Object	A handle for the object that is being or has just been initialized.
Function	One of the following manifest constants: **ACPI_INIT_DEVICE_INI** – the Object is a handle to a Device that has just been initialized.

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 during subsystem initialization.

8.1.3 AcpiEnableSubsystem

Complete the ACPICA 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_ACPI_ENABLE**. Do not attempt to enter ACPI mode. For hardware-independent mode only. **ACPI_NO_ADDRESS_SPACE_INIT**. Do not install the default address space handlers. For debug purposes only. **ACPI_NO_HANDLER_INIT**. Do not install the SCI and global lock handlers. For hardware-independent mode only.

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 ACPICA Subsystem.

8.1.4 AcpiInitializeObjects

Initialize objects within the ACPI namespace.

ACPI_STATUS
AcpiInitializeObjects (
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 execute the operation region _REG control methods. For debug purposes only. **ACPI_NO_OBJECT_INIT**. Do not run the final initialization pass to complete initialization of all address spaces and fields. **ACPI_NO_DEVICE_INIT**. Do not attempt to run the _STA and _INI methods on devices in the ACPI namespace. **ACPI_NO_EVENT_INIT**. Do not initialize the FADT-defined GPE blocks. For hardware independent mode only.

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 ACPICA Subsystem by initializing all ACPI Devices, Operation Regions, Buffer Fields, Buffers, and Packages. It must be called and it should only be called after a call to *AcpiEnableSubsystem.* The object cache is purged after these objects are initialized, in case an overly large number of cached objects were created during initialization (versus the size of the caches at runtime.)

8.1.5 AcpiSubsystemStatus

Obtain initialization status of the ACPICA subsystem.

ACPI_STATUS
AcpiSubsystemStatus (
void)

PARAMETERS

None

RETURN

Status	Exception code indicates success or reason for failure.

EXCEPTIONS

AE_OK	The subsystem was successfully initialized.
AE_ERROR	The subsystem has not been initialized

Functional Description:

This function allows device drivers to determine the initialization status of the ACPICA subsystem.:

8.1.6 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	The OS-dependent layer did not shutdown properly.

Functional Description:

This function performs a shutdown of the Core Subsystem portion of the ACPICA 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 ACPICA subsystem. In more detail, the terminate function performs the following:

Free all memory associated with the ACPI tables (either allocated or mapped memory).

Free all internal objects associated with the namespace.

Free all objects within the object caches.

Free all OS resources associated with mutual exclusion.

8.2 ACPI Table Management

8.2.1 AcpiInitializeTables

Initialize the ACPICA table manager.

```
ACPI_STATUS
AcpiInitializeTables (
    ACPI_TABLE_DESC         *InitialTableArray,
    UINT32                  InitialTableCount,
    BOOLEAN                 AllowResize)
```

PARAMETERS

InitialTableArray	Pointer to an array of pre-allocated **ACPI_TABLE_DES**C structures. If NULL, the array is dynamically allocated.
InitialTableCount	Requested size of InitialTableArray, in number of **ACPI_TABLE_DESC** structures.
AllowResize	Flag to tell the Table Manager if a resize of the pre-allocated array is allowed. Ignored if *InitialTableArray* is NULL.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The table manager was successfully initialized.
AE_NOT_FOUND	A valid RSDP could not be located.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.

Functional Description:

This function initializes the table manager component. A memory array is required to store information about the BIOS-provided ACPI tables. It can be pre-allocated by the caller (if dynamic memory is not available yet) or it can be allocated by this function.

Specify a static memory array for the InitialTableArray if the Table Manager is to be used early during kernel initialization, before dynamic memory is available. Otherwise, use a NULL pointer and the Table Manager will use dynamic memory to allocate the array.

8.2.2 AcpiReallocateRootTable

Copy the root ACPI information table into dynamic memory.

ACPI_STATUS
AcpiReallocateRootTable (
void)

PARAMETERS

None

RETURN

Status	Exception code indicates success or reason for failure.

EXCEPTIONS

AE_OK	The table was successfully enlarged.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.

Functional Description:

This function copies the root table into dynamic memory. The root table is used to store information about the BIOS-provided ACPI tables. This function should be called after dynamic memory is available within the kernel and if AcpiInitializeTables was called with a pre-allocated static table array.

8.2.3 AcpiFindRootPointer

Locate the RSDP via memory scan (IA-32).

ACPI_STATUS
AcpiFindRootPointer (
ACPI_SIZE ***TableAddress)**

PARAMETERS

TableAddress	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_NOT_FOUND	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 mechanism is only applicable to IA-32 systems.

This interface should only be called from the OSL function *AcpiOsGetRootPointer* if this 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.

This function is always available, regardless of the initialization state of the rest of ACPICA.

8.2.4 AcpiLoadTables

Load the BIOS-provided 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 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 ACPICA core subsystem to initialize consists of the following:

- RSDT/XSDT
- FADT
- FACS
- DSDT

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

All SSDTs found within the RSDT/XSDT are loaded.

If the operation fails an appropriate status will be returned.

8.2.5 AcpiGetTableHeader

Get the header portion of a specific installed ACPI table.

```
ACPI_STATUS
AcpiGetTableHeader (
    char                    *Signature,
    UINT32                  Instance,
    ACPI_TABLE_HEADER       *OutTableHeader)
```

PARAMETERS

Signature	A pointer to the 4-character ACPI signature for the requested table.
Instance	For table types that support multiple tables (SSDT), 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 Signature pointer is NULL.

- The OutTableHeader pointer is NULL.

AE_NOT_FOUND	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.

8.2.6 AcpiGetTable

Obtain a specific installed ACPI table.

```
ACPI_STATUS
AcpiGetTable  (
    char                    *Signature,
    UINT32                  Instance,
    ACPI_TABLE_HEADER       **Table)
```

PARAMETERS

Signature	A pointer to the 4-character ACPI signature for the requested table.
Instance	Which table instance, if multiple instances of the table are allowed (SSDT).
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_BAD_PARAMETER	At least one of the following is true:

- The Signature pointer is NULL.
- The OutTableHeader pointer is NULL.

AE_NO_ACPI_TABLES	A valid RSDP could not be located.
AE_NOT_FOUND	There is no table of this type currently loaded, or the table of the specified Instance is not loaded.
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 after the Table Manager is initialized, even before the ACPICA 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.

8.2.7 AcpiGetTableByIndex

Obtain an installed ACPI table via an index into the Root Table

```
ACPI_STATUS
AcpiGetTableByIndex (
    UINT32                  TableIndex,
    ACPI_TABLE_HEADER       **OutTable)
```

PARAMETERS

TableIndex	Index of the table within the internal Root Table list.
OutTable	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 *OutTable* pointer is `NULL`.
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. It is useful for iterating through the entire set of installed ACPI tables. To obtain a specific ACPI table, use *AcpiGetTable* or *AcpiGetTableHeader*.

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

8.2.8 AcpiInstallTableHandler

Install a handler for ACPI table load and unload events.

```
ACPI_STATUS
AcpiInstallTableHandler (
    ACPI_TABLE_HANDLER      Handler,
    void                    *Context)
```

PARAMETERS

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 *Handler* pointer is `NULL`.
AE_ALREADY_EXISTS	A global table handler is already installed.

Functional Description:

This function installs a global handler for table load/unload events.

8.2.8.1 Interface to Table Event Handlers

Definition of the handler interface for Table Events.

```
typedef
ACPI_STATUS (*ACPI_TABLE_HANDLER) (
    UINT32                  Event,
    void                    *Table,
    void                    *Context)
```

PARAMETERS

Event	The table event that occurred. One of these manifest constants:

	ACPI_TABLE_EVENT_LOAD – The table was just loaded. **ACPI_TABLE_EVENT_UNLOAD** – The table is about to be unloaded.
Table	The table that was either just loaded or is about to be unloaded.
Context	The Context value that was passed as a parameter to the AcpiInstallTableHandler function.

RETURN VALUE

None

Functional Description:

This handler is installed via *AcpiInstallTableHandler*. It is called whenever an ACPI table is either loaded or unloaded.

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

8.2.9 AcpiRemoveTableHandler

Remove a handler for ACPI table events.

ACPI_STATUS
AcpiRemoveTableHandler (
ACPI_TABLE_HANDLER Handler)

PARAMETERS

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 Handler pointer is NULL. • The Handler address is not the same as the one that is installed.
AE_NOT_EXIST	There is no handler installed for notifications on this object.

Functional Description:

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

8.3 ACPI Namespace Management

8.3.1 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:
•	A handle to the object to be evaluated.
•	A handle to a parent object that is a prefix to the pathname.
•	A NULL handle if the pathname is fully qualified.
Pathname	Pathname of namespace object to evaluate. May be either an absolute path or a path relative to the Object.
MethodParams	If the object is a control method, this is a pointer to a list of parameters to pass to the method. This pointer may be NULL if no parameters are being passed to the method or if the object is not a method.
ReturnBuffer	A pointer to a location where the return value of the object evaluation (if any) is placed. If this pointer is NULL, no value is returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The object was successfully evaluated.
AE__LIMIT	More than the maximum number of 7 arguments were passed to a method.
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 ACPICA 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: • Both the *Object* and *Pathname* parameters are `NULL`.

	• The *Object* handle is NULL, but the *Pathname* is not absolute. • The *Pathname* is relative but the *Object* is invalid. • The *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.
1. 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 Integer, 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);
```

8.3.2 AcpiEvaluateObjectTyped

Evaluate an ACPI namespace object and return the type-validated result.

ACPI_STATUS
AcpiEvaluateObjectTyped (

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

PARAMETERS

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

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The object was successfully evaluated and the correct object type was returned.
AE_NULL_OBJECT	No object was returned from the evaluation.
AE_TYPE	An object of the incorrect type was returned.
Others	See the definition of AcpiEvaluateObject.

Functional Description:

This function locates and evaluates objects in the namespace and validates that the object returned from the evaluation is of the expected type. It is a front-end to *AcpiEvaluateObject*. See the description of AcpiEvaluateObject for more information.

8.3.3 AcpiGetObjectInfo

Get information about an ACPI namespace object.

```
ACPI_STATUS
AcpiGetObjectInfo (
    ACPI_HANDLE             Object,
    ACPI_DEVICE_INFO        **OutBuffer)
```

PARAMETERS

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

RETURN

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	Device info was successfully returned. See the ACPI_DEVICE_INFO structure for valid returned fields.
AE_BAD_PARAMETER	At least one of the following is true: • The *Object* handle is invalid. • The *OutBuffer* pointer is `NULL`.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.

Functional Description:

This function obtains information about an object contained within the ACPI namespace. For all namespace objects, the following information is returned:

Type	—	The ACPI object type (**ACPI_TYPE_INTEGER**, etc.)
Name	—	The 4-character ACPI name of the object

For Control Method objects, this additional information is returned:

ParamCount	—	The required number of input parameters

For Device and Processor objects, this additional information is returned as a result of evaluating the following standard ACPI device methods and objects on behalf of the device:

_ADR	—	The address of the object (bus and device specific)
_STA	—	The current status of the object/device
_HID	—	The hardware ID of the object (string)
_UID	—	The Unique ID of the object (string)
_CID	—	The Compatibility ID list of the object (strings)
_SxW	—	Methods that return the lowest D-state values (**_S0W**, **_S1W**, **_S2W**, **_S3W**, **_S4W**)
_SxD	—	Methods that return the highest D-state values (**_S1D**, **_S2D**, **_S3D**, **_S4D**)

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

```
typedef struct
{
     UINT32                  InfoSize;
     UINT32                  Name;
     ACPI_OBJECT_TYPE        Type;
     UINT8                   ParamCount;
     UINT8                   Valid;
     UINT8                   Flags;
     UINT8                   HighestDstates[4];
     UINT8                   LowestDstates[5];
     UINT32                  CurrentStatus;
     ACPI_INTEGER            Address;
     ACPI_DEVICE_ID          HardwareId;
     ACPI_DEVICE_ID          UniqueId;
     ACPI_DEVICE_ID_LIST     CompatibleIdList;

} ACPI_DEVICE_INFO;
```

Where:

InfoSize	Entire size of the returned structure, including all ID strings that are appended to the end of the structure.
Name	The 4-character ACPI name of the object.
Type	Is the object type code.
ParamCount	If the object is a control method, this is the number of parameters defined for the method.

Valid	A bit field that indicates which of the optional fields below contain valid values. See below.
Flags	Miscellaneous information flags. The following flags are defined: **ACPI_PCI_ROOT_BRIDGE**: Indicates that either the **_HID** or one of the **_CID** values matched either **PNP0A03** (PCI root bridge) or **PNP0A08** (PCI Express root bridge)
HighestDstates	_SxD device state values. 0xFF indicates that the field is invalid.
LowestDstates	_SxW device wake state values. 0xFF indicates that the field is invalid.
CurrentStatus	The result of evaluating _STA method for this object.
Address	The result of evaluating _ADR for this object.
HardwareId	A pointer to the string obtained as a result of evaluating _HID for this object.
UniqueId	A pointer to the string obtained as a result of evaluating _UID for this object.
CompatibleIds	An array of pointers to the string(s) obtained as a result of evaluating _CID for this object (a list of _CIDs.)

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_ADR
ACPI_VALID_STA
ACPI_VALID_HID
ACPI_VALID_UID
ACPI_VALID_CID
ACPI_VALID_SXDS
ACPI_VALID_SXWS
```

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.

The sub-structures used for the variable-length device ID strings are defined as follows:

```
typedef struct
{
    UINT32              Length;   /* Length of string + null */
    char                *String;

} ACPI_DEVICE_ID;

typedef struct
{
    UINT32              Count;    /* Number of IDs in Ids array */
    UINT32              ListSize; /* Size of list, including ID strings */
    ACPI_DEVICE_ID      Ids[1];   /* ID array */

} ACPI_DEVICE_ID_LIST;
```

Within the original ACPI tables, the **_HID**, **_UID**, and **_CID** values can be of either type **ACPI_TYPE_STRING** or **ACPI_TYPE_INTEGER**. However, 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. An internal data type conversion is performed if necessary, as follows:

- 32-bit compressed **EISAID**s within **_HID** and **_CID** objects are decompressed and converted to strings.
- 64-bit integer IDs within **_UID** objects are converted to decimal string representation.

The object returned from this function should be freed via **ACPI_FREE**.

Note: The string pointers for _HID, _UID, and _CID simply point to a reserved area within the returned buffer af ter the ACPI_DEVICE_INFO structure. When the return object is freed, these pointers will become invalid.

8.3.4 AcpiGetNextObject

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

ACPI_STATUS
AcpiGetNextObject (

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

PARAMETERS

Type	The desired type of the next object.
Parent	A handle to a parent object to be searched for the next child object.
Child	A handle to a child object. The next child object of the parent object that matches the Type will be returned. Use the value of NULL to get the first child of the parent.

OutHandle	A pointer to a location where a handle to the next child object is to be returned. If this pointer is NULL, the child object handle is not returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The next object was successfully found and returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *Parent* handle is invalid. • The *Child* handle is invalid. • The *Type* parameter refers to an invalid type.
AE_NOT_FOUND	The child object parameter is the last object of the given type within the parent — a next child object was not found. If Child is NULL, this exception means that the parent object has no children.

Functional Description:

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

1. If the *Child* is non-`NULL`, it is used as the starting point (the *current object*) for the search.
2. If the *Child* is `NULL` and the *Parent* is non-`NULL`, the search is performed starting at the beginning of the scope.
3. If both the *Parent* and the *Child* parameters are `NULL`, the search begins at the start of the namespace (the search begins at the *Root Object*).

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

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

8.3.5 AcpiGetParent

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

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

PARAMETERS

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

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The parent object was successfully found and returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *Child* handle is invalid. • The *OutParent* pointer is `NULL`.
AE_NULL_ENTRY	The referenced object has no parent. (Entries at the root level do not have a parent object.)

Functional Description:

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

8.3.6 AcpiGetType

Get the type of an ACPI object.

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

PARAMETERS

Object	A handle to an object whose type is to be returned.

OutType	A pointer to a location where the object type is to be returned.

RETURN

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The object type was successfully returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *Object* handle is invalid. • The *OutType* pointer is NULL.

Functional Description:

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

8.3.7 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 Pathname. In other words, the Pathname 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.

8.3.8 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: • **ACPI_FULL_PATHNAME** – return a complete pathname (from the namespace root) to the object. • **ACPI_SINGLE_NAME** – return a single segment ACPI name for the object (4 characters, null terminated).
OutName	A pointer to a location where the fully qualified and NULL terminated name or pathname is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The full pathname associated with the handle was successfully retrieved and returned.
AE_BAD_PARAMETER	At least one of the following is true: • The Parent handle is invalid. • The Object handle is invalid. • The OutName pointer is NULL. • The *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
```

8.3.9 AcpiGetDevices

Walk the ACPI namespace to find all objects of type Device.

ACPI_STATUS
AcpiGetDevices (

char	***HID,**
ACPI_WALK_CALLBACK	**UserFunction,**
void	***UserContext,**
void	****ReturnValue)**

PARAMETERS

HID	A device Hardware ID to search for. If NULL, all objects of type Device are passed to the UserFunction.
UseFunction	A pointer to a function that is called when the namespace object is deleted:
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	The *UserFunction* address is NULL.

Functional Description:

This function performs a modified depth-first walk of the namespace tree. The *UserFunction* is invoked whenever an object of type Device with a matching HID is found. If the user function returns a non-zero value, the search is terminated immediately and this value is returned to the caller.

If the HID parameter is NULL, all objects of type Device within the namespace are passed to the User Function.

8.3.10 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 and can be retrieved at any time via *AcpiGetData*.

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:

- The Object handle is invalid.
- The *Handler* pointer is NULL.
- The *Data* pointer is NULL.

AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.
AE_NO_NAMESPACE	The namespace has not been successfully loaded.

Functional Description:

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

8.3.11 AcpiDetachData

Remove a 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_NAMESPACE	The namespace has not been successfully loaded.

Functional Description:

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

8.3.12 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: • The Object handle is invalid. • The *Handler* pointer is NULL. • The *Data* pointer is NULL.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.
AE_NO_NAMESPACE	The namespace has not been successfully loaded.

Functional Description:

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

8.3.13 AcpiInstallMethod

Install a single control method into the namespace.

ACPI_STATUS
AcpiInstallMethod (
UINT8 ***TableBuffer)**

PARAMETERS

TableBuffer	A pointer to a buffer containing a DSDT or SSDT table which in turn contains a single control method.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The method was successfully installed.
AE_BAD_HEADER	The buffer does not contain a valid ACPI table, or the table is not a DSDT or SSDT.
AE_BAD_PARAMETER	At least one of the following is true: • The *TableBuffer* pointer is NULL. • The table does not contain a valid control method as the first (and only) element of the table.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.
AE_TYPE	The name of the method already exists in the namespace, but the name is not an object of type method and cannot be overwritten.

Functional Description:

This function installs a single control method into the ACPI namespace. It is intended to override an existing method which may not work correctly or it can insert a completely new method in order to create a missing method such as **_OFF**, **_ON**, **_STA**, **_INI**, etc. It can also be used to insert a method for debugging purposes. For these cases, it is far simpler to dynamically install a single control method rather than override the entire DSDT with a modified DSDT.

AcpiInstallMethod can be used to create a new method anywhere in the namespace or to overwrite the AML for any existing control method. The name (and location) for the new method is defined within the AML contained in the ACPI table pointed to by the *TableBuffer* parameter. Either single (4 character) ACPI names may be used, or full ACPI pathnames may be used, each segment separated by periods. This function should be called only after all BIOS-defined ACPI tables have been loaded and the namespace has been created.

The method must be defined and compiled within a DSDT or SSDT. The resulting table is then passed as the parameter to *AcpiInstallMethod*. If the method needs to reference any objects that already exist within the namespace, the ASL **External** operator should be used.

Example

The example ASL code below creates a DSDT that contains one method with the name "**_SI_.ABCD**". The name dictates where the method will be created within the namespace, and can be a full pathname that references any portion of the namespace.

```
DefinitionBlock ("", "DSDT", 2, "Intel", "MTHDTEST", 0x20090512)
{
    Method (\_SI_.ABCD, 1, Serialized)
    {
        Store ("Example installed method", Debug)
        Store (Arg0, Debug)
        Return ()
    }
}
```

The example is compiled via the iASL compiler using the "-tc" option to create a C hex file:

```
> iasl –tc method.asl
```

This produces the following output, which is "C" code that can be included into a C source file:

```
/*
 * Intel ACPI Component Architecture
 * ASL Optimizing Compiler version 20090422 [April 22 2009]
 * Copyright (C) 2000 - 2009 Intel Corporation
 * Supports ACPI Specification Revision 3.0a
 *
 * Compilation of "method.asl" - Tue May 12 14:55:53 2009
 *
 * C source code output
 */
unsigned char AmlCode[] =
{
    0x44,0x53,0x44,0x54,0x53,0x00,0x00,0x00,  /* 00000000    "DSDTS..." */
    0x02,0x12,0x49,0x6E,0x74,0x65,0x6C,0x00,  /* 00000008    "..Intel." */
    0x4D,0x54,0x48,0x44,0x54,0x45,0x53,0x54,  /* 00000010    "MTHDTEST" */
    0x12,0x05,0x09,0x20,0x49,0x4E,0x54,0x4C,  /* 00000018    "... INTL" */
    0x22,0x04,0x09,0x20,0x14,0x2E,0x2E,0x5F,  /* 00000020    "".. ..._" */
    0x53,0x49,0x5F,0x41,0x42,0x43,0x44,0x09,  /* 00000028    "SI_ABCD." */
    0x70,0x0D,0x45,0x78,0x61,0x6D,0x70,0x6C,  /* 00000030    "p.Exampl" */
    0x65,0x20,0x69,0x6E,0x73,0x74,0x61,0x6C,  /* 00000038    "e instal" */
    0x6C,0x65,0x64,0x20,0x6D,0x65,0x74,0x68,  /* 00000040    "led meth" */
    0x6F,0x64,0x00,0x5B,0x31,0x70,0x68,0x5B,  /* 00000048    "od.[1ph[" */
    0x31,0xA4,0x00,
};
```

The buffer above is then used in a call to *AcpiInstallMethod*, as shown in the example code below:

```
Status = AcpiInstallMethod (AmlCode);
if (ACPI_FAILURE (Status))
{
    AcpiOsPrintf ("%s, Could not install method\n",
        AcpiFormatException (Status));
}
```

8.3.14 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      PreOrderVisit,
    ACPI_WALK_CALLBACK      PostOrderVisit,
    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.
PreOrderVisit	A pointer to a user-written function that is invoked in a pre-order manner for each matching object that is found during the walk. (See the interface specification for the user function below.)
PostOrderVisit	A pointer to a user-written function that is invoked in a post-order manner 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 User Functions (*PreOrderVisit* and/or *PostOrderVisit)* are invoked whenever an object that matches the type parameter is found during the walk. 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.

8.3.14.1 Interface to User Callback Function

Interface to the user function that is invoked from AcpiWalkNamespace.

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.*

8.4 ACPI Hardware Management

8.4.1 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.

8.4.2 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.

8.4.3 AcpiReset

Perform a system reset.

ACPI_STATUS
AcpiReset (
void)

PARAMETERS

None

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The reset register was successfully written.
AE_NOT_EXIST	The FADT flags indicate that the reset register is not supported, or the reset register address is zero.

Functional Description:

This function performs a system reset by writing the FADT-defined *Reset Value* to the FADT-defined *Reset Register* (if the register is supported, as indicated by the FADT *Flags*).

Reset registers in both memory and I/O space are supported. A reset register in PCI configuration space is not supported by this function and must be handled by the host.

8.4.4 AcpiReadBitRegister

Get the contents of an ACPI-defined Bit Register.

ACPI_STATUS
AcpiGetRegister (
UINT32 **RegisterId,**
UINT32 ***ReturnValue)**

PARAMETERS

RegisterId — The ID of the desired bit register, one of the following manifest constants:

ACPI_BITREG_TIMER_STATUS
ACPI_BITREG_BUS_MASTER_STATUS
ACPI_BITREG_GLOBAL_LOCK_STATUS
ACPI_BITREG_POWER_BUTTON_STATUS
ACPI_BITREG_SLEEP_BUTTON_STATUS
ACPI_BITREG_RT_CLOCK_STATUS
ACPI_BITREG_WAKE_STATUS
ACPI_BITREG_PCIEXP_WAKE_STATUS
ACPI_BITREG_TIMER_ENABLE
ACPI_BITREG_GLOBAL_LOCK_ENABLE
ACPI_BITREG_POWER_BUTTON_ENABLE
ACPI_BITREG_SLEEP_BUTTON_ENABLE
ACPI_BITREG_RT_CLOCK_ENABLE
ACPI_BITREG_PCIEXP_WAKE_DISABLE
ACPI_BITREG_SCI_ENABLE
ACPI_BITREG_BUS_MASTER_RLD
ACPI_BITREG_GLOBAL_LOCK_RELEASE
ACPI_BITREG_SLEEP_TYPE
ACPI_BITREG_SLEEP_ENABLE
ACPI_BITREG_ARB_DISABLE

ReturnValue — A pointer to a location where the data is to be returned.

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 interrupts enabled or disabled. The hardware is not locked during the read, as it is not necessary

8.4.5 AcpiWriteBitRegister

Set the contents of an ACPI-defined Bit Register.

ACPI_STATUS
AcpiSetRegister (
UINT32 **RegisterId,**
UINT32 **Value)**

PARAMETERS

RegisterId — The ID of the desired register, one of the following manifest constants:

ACPI_BITREG_TIMER_STATUS
ACPI_BITREG_BUS_MASTER_STATUS
ACPI_BITREG_GLOBAL_LOCK_STATUS
ACPI_BITREG_POWER_BUTTON_STATUS
ACPI_BITREG_SLEEP_BUTTON_STATUS
ACPI_BITREG_RT_CLOCK_STATUS
ACPI_BITREG_WAKE_STATUS
ACPI_BITREG_PCIEXP_WAKE_STATUS
ACPI_BITREG_TIMER_ENABLE
ACPI_BITREG_GLOBAL_LOCK_ENABLE
ACPI_BITREG_POWER_BUTTON_ENABLE
ACPI_BITREG_SLEEP_BUTTON_ENABLE
ACPI_BITREG_RT_CLOCK_ENABLE
ACPI_BITREG_PCIEXP_WAKE_DISABLE
ACPI_BITREG_SCI_ENABLE
ACPI_BITREG_BUS_MASTER_RLD
ACPI_BITREG_GLOBAL_LOCK_RELEASE
ACPI_BITREG_SLEEP_TYPE
ACPI_BITREG_SLEEP_ENABLE
ACPI_BITREG_ARB_DISABLE

Value — The data to be written.

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 writes the bit register specified in the RegisterId. The value written must be normalized to bit zero before calling. Can be used with interrupts enabled or disabled.

8.4.6 AcpiRead

Read the contents of an ACPI Register (low-level read).

```
ACPI_STATUS
AcpiRead (
    UINT64                  *ReturnValue,
    ACPI_GENERIC_ADDRESS    *Register)
```

PARAMETERS

ReturnValue	A pointer to where the data is returned. The entire 64-bit *ReturnValue* is set, regardless of the width of the register.
Register	A pointer to a valid ACPI register in generic address format.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The register was read successfully.
AE_BAD_ADDRESS	The *Address* element of the register is zero.
AE_BAD_PARAMETER	The *Register* or *ReturnValue* parameters are NULL.
AE_SUPPORT	The register width was not 8/16/32/64.
Other	The function failed at the operating system level.

Functional Description:

This function reads a register defined in the generic address format. It supports reads from memory or I/O space only. Registers must have a width of either 8, 16, 32, or 64 bits.

8.4.7 AcpiWrite

Write an ACPI Register (low-level write).

```
ACPI_STATUS
AcpiWrite (
    UINT64                  Value,
    ACPI_GENERIC_ADDRESS    *Register)
```

PARAMETERS

Value	The data to be written.
Register	A pointer to a valid ACPI register in generic address format.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The register was read successfully.
AE_BAD_ADDRESS	The Address element of the register is zero.
AE_BAD_PARAMETER	The *Register* parameter is NULL.
AE_SUPPORT	The register width was not 8/16/32/64.
Other	The function failed at the operating system level.

Functional Description:

This function writes a register defined in the generic address format. It supports writes to memory or I/O space only. Registers must have a width of either 8, 16, 32, or 64 bits.

8.4.8 AcpiAcquireGlobalLock

Acquire the ACPI Global Lock.

```
ACPI_STATUS
AcpiAcquireGlobalLock (
    UINT16                  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.

8.4.9 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*.

8.4.10 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).

8.4.11 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.

8.4.12 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 PM Timer (in ticks).

8.5 ACPI Sleep/Wake Support

8.5.1 AcpiSetFirmwareWakingVector

Set the 32-bit firmware wake vector.

ACPI_STATUS
AcpiSetFirmwareWakingVector (
UINT32 **Address32)**

PARAMETERS

Address32	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 32-bit firmware (ROM BIOS) wake vector. If a 64-bit vector exists in the current FACS, it is set to zero.

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

8.5.2 AcpiSetFirmwareWakingVector64

Set the 64-bit firmware wake vector.

ACPI_STATUS
AcpiSetFirmwareWakingVector64 (
UINT64 **Address64)**

PARAMETERS

Address64	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_NOT_EXIST	The 64-bit vector does not exist in the current FACS. Either the table is too small or the revision is less than 1.
AE_NO_ACPI_TABLES	The FACS is not loaded or could not be found.

Functional Description:

This function sets the 64-bit firmware (ROM BIOS) wake vector. The 32-bit vector is set to zero.

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

8.5.3 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.

8.5.4 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.

8.5.5 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.
AE_BAD_PARAMETER	Invalid SleepState value.
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.

8.5.6 AcpiEnterSleepStateS4Bios

Enter S4 BIOS sleep

ACPI_STATUS
AcpiEnterSleepStateS4bios (
void)

PARAMETERS

None

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 performs an S4 BIOS request.

This function must be called with interrupts disabled.

8.5.7 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 state.

8.6 ACPI Fixed Event Management

8.6.1 AcpiEnableEvent

Enable an ACPI Fixed Event.

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

PARAMETERS

Event	The fixed event to be enabled. This parameter must be one of the following manifest constants: **ACPI_EVENT_PMTIMER** **ACPI_EVENT_GLOBAL** **ACPI_EVENT_POWER_BUTTON** **ACPI_EVENT_SLEEP_BUTTON** **ACPI_EVENT_RTC**
Flags	Reserved, set to zero.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The event was successfully enabled.
AE_BAD_PARAMETER	The *Event* is invalid.
Other	Hardware access exception.

Functional Description:

This function enables a single ACPI fixed event.

8.6.2 AcpiDisableEvent

Disable an ACPI Fixed Event.

ACPI_STATUS
AcpiDisableEvent (
UINT32 **Event,**
UINT32 **Flags)**

PARAMETERS

Event — The fixed event to be disabled. This parameter must be one of the following manifest constants:

ACPI_EVENT_PMTIMER
ACPI_EVENT_GLOBAL
ACPI_EVENT_POWER_BUTTON
ACPI_EVENT_SLEEP_BUTTON
ACPI_EVENT_RTC

Flags — Reserved, set to zero.

RETURN VALUE

Status — Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The event was successfully disabled.
AE_BAD_PARAMETER	The *Event* is invalid.
Other	Hardware access exception.

Functional Description:

This function disables a single ACPI fixed event.

8.6.3 AcpiClearEvent

Clear a pending ACPI Fixed Event.

ACPI_STATUS
AcpiClearEvent (
UINT32 **Event)**

PARAMETERS

Event — The fixed event to be cleared. This parameter must be one of the following manifest constants:

ACPI_EVENT_PMTIMER
ACPI_EVENT_GLOBAL
ACPI_EVENT_POWER_BUTTON
ACPI_EVENT_SLEEP_BUTTON
ACPI_EVENT_RTC

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The event was successfully cleared.
AE_BAD_PARAMETER	The *Event* is invalid.
Other	Hardware access exception.

Functional Description:

This function clears (zeros the status bit for) a single ACPI fixed event.

8.6.4 AcpiGetEventStatus

Obtain the status of an ACPI Fixed Event.

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

PARAMETERS

Event	The fixed event for which status will be obtained. This parameter must be one of the following manifest constants: **ACPI_EVENT_PMTIMER** **ACPI_EVENT_GLOBAL** **ACPI_EVENT_POWER_BUTTON** **ACPI_EVENT_SLEEP_BUTTON** **ACPI_EVENT_RTC**
EventStatus	Where the event status is returned. The following bits may be set: **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 *EventStatus* pointer is NULL or invalid
Other	Hardware access exception.

Functional Description:

This function obtains the current status of a single ACPI fixed event.

8.6.5 AcpiInstallFixedEventHandler

Install a handler for ACPI Fixed Events.

```
ACPI_STATUS
AcpiInstallFixedEventHandler (
    UINT32                  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_ALREADY_EXISTS	A handler for this event is already installed.

Functional Description:

This function installs a handler for a predefined fixed event.

8.6.5.1 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

Reserved	Handler should return zero.

Functional Description:

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

This function executes in the context of an interrupt handler.

8.6.6 AcpiRemoveFixedEventHandler

Remove an ACPI Fixed Event handler.

ACPI_STATUS
AcpiRemoveFixedEventHandler (
UINT32 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*.

8.7 ACPI General Purpose Event Management

8.7.1 AcpiEnableGpe

Enable an ACPI General Purpose Event.

ACPI_STATUS
AcpiEnableGpe (
ACPI_HANDLE **GpeDevice,**
UINT32 **GpeNumber,**
UINT32 **Flags)**

PARAMETERS

GpeDevice	A handle for the parent GPE Block Device of the GPE to be enabled. Specify a NULL handle to indicate that the permanent GPE blocks defined in the FADT (GPE0 and GPE1) are to be used.
GpeNumber	The GPE number to be enabled within the specified GPE Block. The GPE0 block always begins at zero. GPE1 begins at GPE1_BASE (in the FADT). Named GPE Block Devices always begin at zero.
Flags	**ACPI_NOT_ISR** – Caller is not executing from an Interrupt Service Routine (interrupt level.)

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The GPE was successfully enabled.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeDevice* is invalid or does not refer to a valid GPE Block Device. • The *GpeNumber* is out of range for the referenced *GpeDevice*.

Functional Description:

This function enables a single General Purpose Event. Both the FADT–defined GPE blocks and GPE Block Devices are supported. The GPE blocks defined in the FADT are permanent and installed during system initialization. These permanent blocks, GPE0 and GPE1, are treated as a single logical block differentiated by non-overlapping GPE numbers. GPE Block Devices are installed via *AcpiInstallGpeBlock* during bus/device enumeration.

This function may be called from an interrupt service routine (typically a GPE handler) or a device driver, depending on the setting of the *Flags* parameter.

8.7.2 AcpiDisableGpe

Disable an ACPI General Purpose Event.

```
ACPI_STATUS
AcpiDisableGpe (
    ACPI_HANDLE         GpeDevice,
    UINT32              GpeNumber,
    UINT32              Flags)
```

PARAMETERS

GpeDevice	A handle for the parent GPE Block Device of the GPE to be disabled. Specify a NULL handle to indicate that the permanent GPE blocks defined in the FADT (GPE0 and GPE1) are to be used.
GpeNumber	The GPE number to be disabled within the specified GPE Block. The GPE0 block always begins at zero. GPE1 begins at GPE1_BASE (in the FADT). Named GPE Block Devices always begin at zero.
Flags	**ACPI_NOT_ISR** – Caller is not executing from an Interrupt Service Routine (interrupt level.)

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The GPE was successfully disabled.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeDevice* is invalid or does not refer to a valid GPE Block Device. • The *GpeNumber* is out of range for the referenced *GpeDevice*.

Functional Description:

This function disables a single General Purpose Event. Both the FADT–defined GPE blocks and GPE Block Devices are supported. The GPE blocks defined in the FADT are permanent and installed during system initialization. These permanent blocks, GPE0 and GPE1, are treated as a single logical block differentiated by non-overlapping GPE numbers. GPE Block Devices are installed via *AcpiInstallGpeBlock* during bus/device enumeration.

This function may be called from an interrupt service routine (typically a GPE handler) or a device driver, depending on the setting of the *Flags* parameter.

8.7.3 AcpiClearGpe

Clear a pending ACPI General Purpose Event.

ACPI_STATUS
AcpiClearGpe (

ACPI_HANDLE	**GpeDevice,**
UINT32	**GpeNumber,**
UINT32	**Flags)**

PARAMETERS

GpeDevice	A handle for the parent GPE Block Device of the GPE to be cleared. Specify a NULL handle to indicate that the permanent GPE blocks defined in the FADT (GPE0 and GPE1) are to be used.
GpeNumber	The GPE number to be cleared within the specified GPE Block. The GPE0 block always begins at zero. GPE1 begins at GPE1_BASE (in the FADT). Named GPE Block Devices always begin at zero.
Flags	**ACPI_NOT_ISR** – Caller is not executing from an Interrupt Service Routine (interrupt level.)

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The GPE was successfully cleared.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeDevice* is invalid or does not refer to a valid GPE Block Device. • The *GpeNumber* is out of range for the referenced *GpeDevice*.

Functional Description:

This function clears a single General Purpose Event. Both the FADT–defined GPE blocks and GPE Block Devices are supported. The GPE blocks defined in the FADT are permanent and installed during system initialization. These permanent blocks, GPE0 and GPE1, are treated as a single logical block differentiated by non-overlapping GPE numbers. GPE Block Devices are installed via *AcpiInstallGpeBlock* during bus/device enumeration.

This function may be called from an interrupt service routine (typically a GPE handler) or a device driver, depending on the setting of the *Flags* parameter.

8.7.4 AcpiSetGpeType

Set the type (wake/run) of an individual ACPI General Purpose Event.

ACPI_STATUS
AcpiSetGpeType (

ACPI_HANDLE	**GpeDevice,**
UINT32	**GpeNumber,**
UINT8	**Type)**

PARAMETERS

GpeDevice	A handle for the parent GPE Block Device of the GPE. Specify a NULL handle to indicate that the permanent GPE blocks defined in the FADT (GPE0 and GPE1) are to be used.
GpeNumber	The GPE number within the specified GPE Block. The GPE0 block always begins at zero. GPE1 begins at GPE1_BASE (in the FADT). Named GPE Block Devices always begin at zero.
Type	**ACPI_GPE_TYPE_RUNTIME** – This GPE is used for runtime events only. **ACPI_GPE_TYPE_WAKE** – This GPE is used for wake events only. **ACPI_GPE_TYPE_WAKE_RUN** – This GPE is used for both runtime and wake events.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The type of the GPE was successfully set.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeDevice* is invalid or does not refer to a valid GPE Block Device. • The *GpeNumber* is out of range for the referenced *GpeDevice*. • The *Type* is invalid.

Functional Description:

This function sets the type of a single General Purpose Event. Runtime GPEs are only enabled when the system is operational. Wake GPEs are enabled only when the system is going into suspend mode. Run/wake GPEs are always enabled.

Both the FADT–defined GPE blocks and GPE Block Devices are supported. The GPE blocks defined in the FADT are permanent and installed during system initialization. These permanent blocks, GPE0 and GPE1, are treated as a single logical block differentiated by non-overlapping GPE numbers. GPE Block Devices are installed via *AcpiInstallGpeBlock* during bus/device enumeration.

8.7.5 AcpiGetGpeStatus

Obtain the status of an ACPI General Purpose Event.

```
ACPI_STATUS
AcpiGetGpeStatus (
    ACPI_HANDLE             GpeDevice,
    UINT32                  GpeNumber,
    UINT32                  Flags,
    ACPI_EVENT_STATUS       *EventStatus)
```

PARAMETERS

GpeDevice	A handle for the parent GPE Block Device of the GPE for which status is to be obtained. Specify a NULL handle to indicate that the permanent GPE blocks defined in the FADT (GPE0 and GPE1) are to be used.
GpeNumber	The GPE number to be enabled within the specified GPE Block. The GPE0 block always begins at zero. GPE1 begins at GPE1_BASE (in the FADT). Named GPE Block Devices always begin at zero.
Flags	**ACPI_NOT_ISR** – Caller is not executing from an Interrupt Service Routine (interrupt level.)
EventStatus	Where the event status is returned. The following bits may be set: **ACPI_EVENT_FLAG_SET**

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The GPE was successfully enabled.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeDevice* is invalid or does not refer to a valid GPE Block Device. • The *GpeNumber* is out of range for the referenced *GpeDevice*.

Functional Description:

This function obtains the status of a single General Purpose Event. Both the FADT–defined GPE blocks and GPE Block Devices are supported. The GPE blocks defined in the FADT are permanent and installed during system initialization. These permanent blocks, GPE0 and GPE1, are treated as a single logical block differentiated by non-overlapping GPE numbers. GPE Block Devices are installed via *AcpiInstallGpeBlock* during bus/device enumeration.

This function may be called from an interrupt service routine (typically a GPE handler) or a device driver, depending on the setting of the *Flags* parameter.

8.7.6 AcpiGetGpeDevice

Get the GPE Block Device associated with the GPE index.

ACPI_STATUS
AcpiGetGpeDevice (

UINT32	**Index,**
ACPI_HANDLE	***GpeDevice)**

PARAMETERS

Index	The system index of the GPE, defined to be from zero to the value of **AcpiCurrentGpeCount**.
GpeDevice	A pointer to where the handle of the GPE block device is returned. NULL indicates that the GPE is within one of the FADT-defined GPE blocks.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The GPE block device was successfully returned.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeDevice* pointer is invalid.
AE_NOT_EXIST	The Index refers to a non-existent GPE (it is larger than **AcpiCurrentGpeCount**).

Functional Description:

This function obtains the GPE block device associated with the Index parameter. A returned NULL GPE device indicates that the Index refers to a GPE that is contained in one of the FADT-defined GPE blocks.

The Index is a system index used to track all GPEs. First are the FADT GPE0 block GPEs, then the FADT GPE1 GPEs (if present), then any GPE block device GPEs. Valid values for the Index are

from zero to the value of the public global variable **AcpiCurrentGpeCount**. Index values are consecutive with no 'holes'.

8.7.7 AcpiDisableAllGpes

Disable all system GPEs

ACPI_STATUS
AcpiDisableAllGpes (
void)

PARAMETERS

None

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	All GPEs were successfully disabled.
Other	Hardware access exception.

Functional Description:

This function disables all GPEs currently defined in the system. This includes all runtime and wake GPEs, in both the FADT-defined GPE blocks as well as any installed GPE block devices.

8.7.8 AcpiEnableAllRuntimeGpes

Enable all runtime GPEs

ACPI_STATUS
AcpiEnableAllRuntimeGpes (
void)

PARAMETERS

None

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	All runtime GPEs were successfully enabled.

Other	Hardware access exception.

Functional Description:

This function enables all runtime GPEs currently defined in the system. This includes all runtime GPEs in both the FADT-defined GPE blocks as well as any installed GPE block devices. Runtime GPEs are defined to be any GPEs that are not Wake GPEs, as determined from the _PRW methods within the system AML.

8.7.9 AcpiInstallGpeBlock

Install a GPE Block Device.

```
ACPI_STATUS
AcpiInstallGpeBlock (
    ACPI_HANDLE               GpeDevice,
    ACPI_GENERIC_ADDRESS      *GpeBlockAddress,
    UINT32                    RegisterCount,
    UINT32                    InterruptLevel)
```

PARAMETERS

GpeDevice	A handle for the GPE Block Device to be installed.
GpeNumber	The GPE number to be enabled within the specified GPE Block. Named GPE Block Devices always begin at zero.
RegisterCount	The number of status/enable GPE register pairs in this block.
InterruptLevel	The hardware interrupt level that this GPE block is to be associated with. Can be SCI_INT or any other system interrupt level.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The GPE was successfully enabled.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeDevice* is invalid or does not refer to a valid GPE Block Device. • The *GpeNumber* is out of range for the referenced *GpeDevice*.

Functional Description:

This function installs a GPE Block Device. It is intended for use by a device driver that supports the enumeration of GPE Block Devices. The caller must identify each Block Device in the ACPI namespace (each has a _HID of **ACPI0006**) and obtain the resource requirements (_CRS, etc.) and make this call for each device found.

Gpe Block Device handling is supported in the ACPICA core subsystem because the SCI_INT is owned by the core subystem, and the FADT-defined GPE blocks are also owned by the core. Via this interface, the core also supports GPE Block Devices and the associated interrupts, detection, dispatch, and GPE control method execution — thus centralizing all GPE support to the core.

8.7.10 AcpiRemoveGpeBlock

Remove a GPE Block Device.

ACPI_STATUS
AcpiRemoveGpeBlock (
ACPI_HANDLE **GpeDevice)**

PARAMETERS

GpeDevice	A handle for the GPE Block Device to be removed.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The GPE was successfully enabled.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeDevice* is invalid or does not refer to a valid GPE Block Device. • The *GpeNumber* is out of range for the referenced *GpeDevice*.

Functional Description:

This function removed a GPE Block Device that was previously installed via *AcpiInstallGpeBlock*.

8.7.11 AcpiInstallGpeHandler

Install a handler for ACPI General Purpose Events.

```
ACPI_STATUS
AcpiInstallGpeHandler (
    ACPI_HANDLE             GpeDevice,
    UINT32                  GpeNumber,
    UINT32                  Type,
    ACPI_EVENT_HANDLER      Handler,
    void                    *Context)
```

PARAMETERS

GpeDevice	A handle for the parent GPE Block Device of the GPE for which the handler is to be installed. Specify a NULL handle to indicate that the permanent GPE blocks defined in the FADT (GPE0 and GPE1) are to be used.
GpeNumber	A zero based GPE number. GPE numbers start with GPE register bank zero, and continue sequentially through GPE bank one.
Type	Whether this GPE is edge or level triggered: **ACPI_GPE_LEVEL_TRIGGERED** **ACPI_GPE_EDGE_TRIGGERED**
Handler	Address of the handler to be installed.
Context	A context value that will be passed to the handler as a parameter.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully installed.
AE_BAD_PARAMETER	At least one of the following is true: • The *GpeNumber* is invalid. • The *Handler* pointer is `NULL`.
AE_ALREADY_EXISTS	A handler for this general-purpose event is already installed.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.

Functional Description:

This function installs a handler for a general-purpose event

8.7.11.1 Interface to General Purpose Event Handlers

Definition of the handler interface for General Purpose Events.

```
typedef
void (*ACPI_EVENT_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.

This function executes in the context of an interrupt handler.

8.7.12 AcpiRemoveGpeHandler

Remove an ACPI General-Purpose Event handler.

```
ACPI_STATUS
AcpiRemoveGpeHandler (
    ACPI_HANDLE                     GpeDevice,
    UINT32                          GpeNumber,
    ACPI_EVENT_HANDLER              Handler)
```

PARAMETERS

GpeDevice	A handle for the parent GPE Block Device of the GPE for which the handler is to be removed. Specify a NULL handle to indicate that the permanent GPE blocks defined in the FADT (GPE0 and GPE1) are to be used.
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*.

8.8 Miscellaneous Handler Support

8.8.1 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 (system wide) notifications of the Type specified. Otherwise, this object must be one of the following types: **ACPI_TYPE_DEVICE** **ACPI_TYPE_PROCESSOR** **ACPI_TYPE_THERMAL**
Type	Specifies the type of notifications that are to be received by this handler: **ACPI_SYSTEM_NOTIFY** – Notification values from 0x00 to 0x7F. **ACPI_DEVICE_NOTIFY** – Notification values from 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_ALREADY_EXISTS	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.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.

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)

8.8.1.1 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.

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

8.8.2 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_THERMAL**
HandlerType	Specifies the type of notify handler to be removed: **ACPI_SYSTEM_NOTIFY** – Notification values from 0x00 to 0x7F. **ACPI_DEVICE_NOTIFY** – Notification values from 0x80 to 0xFF.
Handler	Address of the previously installed handler.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

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

Functional Description:

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

8.8.3 AcpiInstallAddressSpaceHandler

Install handlers for ACPI Operation Region events.

```
ACPI_STATUS
AcpiInstallAddressSpaceHandler (
    ACPI_HANDLE                 Object,
    ACPI_ADR_SPACE_TYPE         SpaceId,
    ACPI_ADR_SPACE_HANDLER      Handler,
    ACPI_ADR_SPACE_SETUP        Setup,
    void                        *Context)
```

PARAMETERS

Object	Handle for the object for which a address space handler will be installed. This object may be specified as the ACPI_ROOT_OBJECT to request global scope. Otherwise, this object must be one of the following types: **ACPI_TYPE_DEVICE** **ACPI_TYPE_PROCESSOR** **ACPI_TYPE_THERMAL**
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 ACPICA 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:

- The object handle does not refer to an object of type Device, Processor, ThermalZone, or the root object.
- The SpaceId is invalid.
- The Handler pointer is NULL.

AE_ALREADY_EXISTS	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.

8.8.3.1 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: **ACPI_REGION_ACTIVATE** (init) **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.

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

8.8.3.2 Interface to Address Space Handlers

Definition of the handler interface for Operation Region Events.

```
typedef
void (*ACPI_ADR _SPACE_HANDLER) (
    UINT32                  Function,
    ACPI_PHYSICAL_ADDRESS   Address,
    UINT32                  BitWidth,
    ACPI_INTEGER            *Value,
    void                    *HandlerContext,
    void                    *RegionContext)
```

PARAMETERS

Function	The type of function to be performed; must be one of the following manifest constants: **ADDRESS_SPACE_READ** **ADDRESS_SPACE_WRITE**
Address	A space-specific address where the operation is to be performed.
BitWidth	The width of the operation, typically 8, 16, 32, or 64.
*Value	A pointer to the value to be written (WRITE), or where the value that was read should be returned (READ).
HandlerContext	An address space specific Context value. Typically this is the context that was passed as a parameter to the AcpiInstallAddressSpaceHandler function.
RegionContext	An operation region specific context. Created during the region setup.

RETURN VALUE

None

Functional Description:

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

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

8.8.3.3 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. The bus number is in the low 16 bits and the segment number in the high 16 bits.

8.8.4 AcpiRemoveAddressSpaceHandler

Remove an ACPI Operation Region handler.

ACPI_STATUS
AcpiRemoveAddressSpaceHandler (
ACPI_HANDLE Object,
ACPI_ADR_SPACE_TYPE SpaceId,
ACPI_ADR _SPACE_HANDLER Handler)

PARAMETERS

Object	Handle for the object for which a address space handler will be installed. This object may be specified as the ACPI_ROOT_OBJECT to request global scope. Otherwise, this object must be one of the following types: **ACPI_TYPE_DEVICE** **ACPI_TYPE_PROCESSOR** **ACPI_TYPE_THERMAL**
SpaceId	The ID of the Address Space or Operation Region whose handler is to be removed.
Handler	Address of the previously installed handler.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

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

- The object handle does not refer to an object of type Device, Processor, ThermalZone, or the root object.
- The SpaceId is invalid.
- The Handler pointer is NULL.
- 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*.

8.8.5 AcpiInstallExceptionHandler

Install a handler for ACPI interpreter run-time exceptions.

ACPI_STATUS
AcpiInstallExceptionHandler (
ACPI_EVENT_HANDLER Handler)

PARAMETERS

Handler	Address of the handler to be installed.

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 *Handler* pointer is `NULL`.
AE_ALREADY_EXISTS	A handler for this general-purpose event is already installed.

Functional Description:

This function installs a global handler for exceptions generated during the execution of control methods. Useful for error logging and debugging.

8.8.5.1 Interface to Exception Handlers

Definition of the handler interface for General Purpose Events.

```
typedef
ACPI_STATUS (*ACPI_EXCEPTION_HANDLER) (
    ACPI_STATUS             AmlStatus,
    ACPI_NAME               Name,
    UINT16                  Opcode,
    UINT32                  AmlOffset,
    void                    *Context)
```

PARAMETERS

AmlStatus	The exception code that was raised.
Name	Name of the executing control method.
Opcode	AML opcode whose execution caused the exception.
AmlOffset	Offset of the AML opcode within the control method.
Context	Reserved for future use. Currently NULL.

RETURN VALUE

None

Functional Description:

This handler is installed via *AcpiInstallExceptionHandler*. It is called whenever an exception is raised within the AML interpreter during control method execution.

The **ACPI_STATUS** that is returned by the handler is then used by the AML interpreter instead of the original exception code.

8.9 ACPI Resource Management

8.9.1 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.

8.9.2 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: • 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 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.

8.9.3 AcpiSetCurrentResources

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

```
ACPI_STATUS
AcpiSetCurrentResources (
    ACPI_HANDLE             Device,
    ACPI_BUFFER             *Buffer)
```

PARAMETERS

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

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

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

Functional Description:

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

8.9.4 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: • 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 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.

8.9.5 AcpiGetVendorResource

Find a resource of type Vendor-Defined

```
ACPI_STATUS
AcpiGetVendorResource (
    ACPI_HANDLE             DeviceHandle,
    char                    *Name,
    ACPI_VENDOR_UUID        *Uuid,
    ACPI_BUFFER             *OutBuffer)
```

PARAMETERS

DeviceHandle	A handle to the parent Device that owns the vendor resource.
Name	Name of the parent resource list (_CRS or _PRS).
Uuid	A pointer to the UUID to be matched. Includes both subtype and 16-byte UUID.
OutBuffer	Where the vendor resource is returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The vendor resource was successfully acquired.
AE_BAD_PARAMETER	At least one of the following is true: • The *DeviceHandle* is invalid. • The *Name* does not refer to a _CRS or _PRS control method. • The OutBuffer of UUID pointer is NULL. • The *Length* field of *OutBuffer* is not **ACPI_ALLOCATE_BUFFER**, but the *Pointer* field of *OutBuffer* is `NULL`.
AE_NOT_EXIST	The *Name* could not be found.

Functional Description:

This function retrieves a resource of type *vendor-defined* that matches the supplied UUID and UUID subtype.

8.9.6 AcpiResourceToAddress64

Convert an address resource descriptor to 64 bits

```
ACPI_STATUS
AcpiResourceToAddress64 (
    ACPI_RESOURCE                   *Resource,
    ACPI_RESOURCE_ADDRESS64         *OutResource)
```

PARAMETERS

Resource	The resource descriptor to be converted. This resource must be one of the following types: **ACPI_RESOURCE_TYPE_ADDRESS16** **ACPI_RESOURCE_TYPE_ADDRESS32** **ACPI_RESOURCE_TYPE_ADDRESS64**
OutResource	Where the converted resource is returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The resource was successfully converted.
AE_BAD_PARAMETER	The resource is not of the correct type.

Functional Description:

This utility function converts resources of type **ADDRESS16** and **ADDRESS32** to **ADDRESS64**. This saves the caller from having to duplicate code for different-sized address descriptors. If the input descriptor is of type **ADDRESS64**, a simple copy is performed.

8.9.7 AcpiWalkResources

Parse an ACPI Resource List.

```
ACPI_STATUS
AcpiWalkResources (
    ACPI_HANDLE                     DeviceHandle,
    char                            *Name,
    ACPI_WALK_RESOURCE_CALLBACK     UserFunction,
    void                            *UserContext)
```

PARAMETERS

DeviceHandle	A handle to the Device for which one of the resource lists will be walked:

Name	Name of a resource method (either a **_CRS** or **_PRS** method.)
UserFunction	A pointer to a user-written function that is invoked for each resource object within the resource list. (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.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The event was successfully enabled.
AE_BAD_PARAMETER	The *DeviceHandle* is invalid or the Name does not refer to a **_CRS** or **_PRS** control method.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.

Functional Description:

This function retrieves the current or possible resource list for the specified device. The User Function is called once for each resource in the list – freeing the caller from having to parse the list itself.

8.9.7.1 Interface to User Callback Function

Interface to the user function that is invoked from AcpiWalkResources.

ACPI_STATUS (*ACPI_WALK_RESOURCE_CALLBACK) (
ACPI_RESOURCE *Resource,
void *Context)

PARAMETERS

Resource	A pointer to a single resource within the resource list.
Context	The UserContext value that was passed as a parameter to the *AcpiWalkResources* function.

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 *AcpiWalkResource* for each resource object in the resource list.

8.10 Memory Management

The ACPICA 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 host OS memory management primitives.

8.10.1 ACPI_ALLOCATE

Allocate memory from the dynamic memory pool.

void *
ACPI_ALLOCATE (
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 cannot be assumed to be initialized to any particular value or values.

8.10.2 ACPI_ALLOCATE_ZEROED

Allocate and initialize memory.

void *
ACPI_ALLOCATE_ZEROED (
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 and initializes memory. The returned memory is guaranteed to be initialized to all zeros.

8.10.3 ACPI_FREE

Free previously allocated memory.

void
ACPI_FREE (
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 *ACPI_ALLOCATE* or *ACPI_ALLOCATE_ZEROED*.

8.11 Formatted Output

8.11.1 AcpiInfo and ACPI_INFO

Print a formatted information/comment string.

```
void
AcpiInfo (
    const char          *ModuleName,
    UINT32              LineNumber,
    const char          *Format,
    …)
```

PARAMETERS

ModuleName	The name of the currently executing module or filename.
LineNumber	The current line number within the currently executing module.
Format	A standard printf-style format string.

RETURN VALUE

None

EXCEPTIONS

None

Functional Description:

This function prints a formatted error message using the *AcpiOsPrintf* and *AcpiOsVprintf* OSL interfaces. The format of the output string is as follows:

ACPI: (ModuleName-LineNumber): <message> [ACPICA version number]

The ACPI_INFO macro

The front-end to this function is the ACPI_INFO macro.

Example: The following invocation of the ACPI_INFO macro:

```
ACPI_INFO ((AE_INFO, "ACPICA example info message"));
```

Produces this output:

```
ACPI: ACPICA example info message
```

The AE_INFO macro is required and automatically injects the module name and line number into the invoation of AcpiError. Note the use of double parentheses which are required in order to pass the parameters to the printf OSL functions.

8.11.2 AcpiWarning and ACPI_WARNING

Print a formatted warning string.

```
void
AcpiWarning (
    const char          *ModuleName,
    UINT32              LineNumber,
    const char          *Format,
    …)
```

PARAMETERS

ModuleName	The name of the currently executing module or filename.
LineNumber	The current line number within the currently executing module.
Format	A standard printf-style format string.

RETURN VALUE

None

EXCEPTIONS

None

Functional Description:

This function prints a formatted error message using the *AcpiOsPrintf* and *AcpiOsVprintf* OSL interfaces. The format of the output string is as follows:

ACPI Error (ModuleName-LineNumber): <message> [ACPICA version number]

The ACPI_WARNING macro

The front-end to this function is the ACPI_WARNING macro.

Example: The following invocation of the ACPI_ WARNING macro:

```
ACPI_WARNING ((AE_INFO, "ACPICA example warning message"));
```

Produces this output:

```
ACPI Warning (examples-0187): ACPICA example warn message [20080926]
```

The AE_INFO macro is required and automatically injects the module name and line number into the invoation of AcpiError. Note the use of double parentheses which are required in order to pass the parameters to the printf OSL functions.

8.11.3 AcpiError and ACPI_ERROR

Print a formatted error string.

```
void
AcpiError (
    const char          *ModuleName,
    UINT32              LineNumber,
    const char          *Format,
    …)
```

PARAMETERS

ModuleName	The name of the currently executing module or filename.
LineNumber	The current line number within the currently executing module.
Format	A standard printf-style format string.

RETURN VALUE

None

EXCEPTIONS

None

Functional Description:

This function prints a formatted error message using the *AcpiOsPrintf* and *AcpiOsVprintf* OSL interfaces. The format of the output string is as follows:

ACPI Error (ModuleName-LineNumber): <message> [ACPICA version number]

The ACPI_ERROR macro

The front-end to this function is the ACPI_ERROR macro.

Example: The following invocation of the ACPI_ERROR macro:

```
ACPI_ERROR ((AE_INFO, "ACPICA example error message"));
```

Produces this output:

```
ACPI Error (examples-0187): ACPICA example error message [20080926]
```

The AE_INFO macro is required and automatically injects the module name and line number into the invoation of AcpiError. Note the use of double parentheses which are required in order to pass the parameters to the printf OSL functions.

8.11.4 AcpiException and ACPI_EXCEPTION

Print a formatted error string with decoded ACPICA exception code

```
void
AcpiException (
    const char              *ModuleName,
    UINT32                  LineNumber,
    ACPI_STATUS             Status,
    const char              *Format,
    …)
```

PARAMETERS

ModuleName	The name of the currently executing module or filename.
LineNumber	The current line number within the currently executing module.
Status	ACPICA status to be decoded and displayed.
Format	A standard printf-style format string.

RETURN VALUE

None

EXCEPTIONS

None

Functional Description:

This function prints a formatted error message using the *AcpiOsPrintf* and *AcpiOsVprintf* OSL interfaces. The format of the output string is as follows:

ACPI Exception (ModuleName-LineNumber): <message> [ACPICA version number]

The ACPI_EXCEPTION macro

The front-end to this function is the ACPI_EXCEPTION macro.

Example: The following invocation of the ACPI_ EXCEPTION macro:

```
ACPI_EXCEPTION ((AE_INFO, Status, "ACPICA example error message"));
```

Produces this output:

```
ACPI Exception (examples-0187): AE_ERROR, ACPICA status [20080926]
```

The AE_INFO macro is required and automatically injects the module name and line number into the invoation of AcpiError. Note the use of double parentheses which are required in order to pass the parameters to the printf OSL functions.

8.11.5 AcpiDebugPrint and ACPI_DEBUG_PRINT

Print a formatted debug string.

```
void
AcpiDebugPrint (
    UINT32          RequestedDebugLevel,
    UINT32          LineNumber,
    const char      *FunctionName,
    const char      *ModuleName,
    UINT32          ComponentId,
    const char      *Format,
    …)
```

PARAMETERS

RequestedDebugLevel — The debug level for this statement. This value is compared to the current AcpiDbgLevel mask to determine if this message will be output or not. Must be one of the following:

ACPI_DB_INIT
ACPI_DB_DEBUG_OBJECT
ACPI_DB_INFO
ACPI_DB_ALL_EXCEPTIONS
ACPI_DB_INIT_NAMES
ACPI_DB_PARSE
ACPI_DB_LOAD
ACPI_DB_DISPATCH
ACPI_DB_EXEC
ACPI_DB_NAMES
ACPI_DB_OPREGION
ACPI_DB_BFIELD
ACPI_DB_TABLES
ACPI_DB_VALUES
ACPI_DB_OBJECTS
ACPI_DB_RESOURCES
ACPI_DB_USER_REQUESTS
ACPI_DB_PACKAGE
ACPI_DB_ALLOCATIONS
ACPI_DB_FUNCTIONS
ACPI_DB_OPTIMIZATIONS
ACPI_DB_MUTEX
ACPI_DB_THREADS
ACPI_DB_IO
ACPI_DB_INTERRUPTS
ACPI_DB_EVENTS
ACPI_DB_ALL

LineNumber — The current line number within the currently executing module.

FunctionName — The name of the currently executing function.

ModuleName	The name of the currently executing module or filename.
ComponentId	The ID of the executing component. Currently defined IDs are: **ACPI_UTILITIES** **ACPI_HARDWARE** **ACPI_EVENTS** **ACPI_TABLES** **ACPI_NAMESPACE** **ACPI_PARSER** **ACPI_DISPATCHER** **ACPI_EXECUTER** **ACPI_RESOURCES** **ACPI_CA_DEBUGGER** **ACPI_OS_SERVICES** **ACPI_CA_DISASSEMBLER** **ACPI_COMPILER** **ACPI_TOOLS** **ACPI_EXAMPLE** **ACPI_DRIVER**
Format	A standard printf-style format string.

RETURN VALUE

None

EXCEPTIONS

None

Functional Description:

This function prints debug messages only if the debug level and the component ID match in the global level/layer masks. This mechanism is useful to pare down the amount of debug output that is produced. In addition to the input string, the module name, the line number, and the function name are added to the output.

The ACPI_DEBUG_PRINT macro

The front end to the AcpiDebugPrint interface

Example: The following invocation of the ACPI_ DEBUG_PRINT macro

```
ACPI_DEBUG_PRINT ((ACPI_DB_INFO, "Example Debug output"));
```

Produces this output:

```
examples-0200 [00] Examples-main         : Example Debug output
```

8.11.6 AcpiDebugPrintRaw and ACPI_DEBUG_PRINT_RAW

Print a formatted debug string, with no extra data.

```
void
AcpiDebugPrintRaw (
    UINT32                  RequestedDebugLevel,
    UINT32                  LineNumber,
    const char              *FunctionName,
    const char              *ModuleName,
    UINT32                  ComponentId,
    const char              *Format,
    …)
```

PARAMETERS

See the definition of *AcpiDebugPrint*

Functional Description:

This function prints debug messages only if the debug level and the component ID match in the global level/layer masks. This mechanism is useful to pare down the amount of debug output that is produced. The message produced by this function is not embellished with the line number, function name, and module name as is performed by ACPI_DEBUG_PRINT.

The ACPI_DEBUG_PRINT_RAW macro

The front end to the AcpiDebugPrintRaw interface.

Example: The following invocation of the ACPI_ DEBUG_PRINT_RAW macro

```
ACPI_DEBUG_PRINT_RAW ((ACPI_DB_INFO, "Example Debug output"));
```

Produces this output:

```
Example Debug output
```

8.12 Miscellaneous Utilities

8.12.1 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.

8.12.2 AcpiDebugTrace

Enable debug tracing of control method execution

```
ACPI_STATUS
AcpiDebugTrace (
    char        *Name,
    UINT32      DebugLevel,
    UINT32      DebugLayer,
    UINT32      Flags)
```

PARAMETERS

Name	Name of the control method to be traced. Currently, only a 4-character ACPI name is supported.
DebugLevel	The debug level used for the trace.
DebugLayer	The debug layer used for the trace.
Flags	Sets the type of trace: 1 – One shot trace 0 – Persistent trace

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The system information list was successfully returned.

Functional Description:

This function enables debug tracing of an individual control method.

8.12.3 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                  AcpiCaVersion;
     UINT32                  Flags;
     UINT32                  TimerResolution;
     UINT32                  Reserved1;
     UINT32                  Reserved2;
     UINT32                  DebugLevel;
     UINT32                  DebugLayer;

} ACPI_SYSTEM_INFO;
```

Where:

AcpiCaVersion	Version number of the ACPICA core subsystem, in the form 0xYYYYMMDD.
Flags	Static information about the system: **ACPI_SYS_MODE_ACPI** ACPI mode is supported on this system. **ACPI_SYS_MODE_LEGACY** Legacy mode is supported.
TimerResolution	Resolution of the ACPI Power Management Timer. Either 24 or 32 indicating the corresponding number of bits of resolution.
DebugLevel	Current value of the global variable that controls the debug output verbosity.
DebugLayer	Current value of the global variable that controls the internal layers whose debug output is enabled.

8.12.4 AcpiGetStatistics

Returns miscellaneous run-time statistics.

ACPI_STATUS
AcpiGetStatistics (
ACPI_STATISTICS ***OutStats)**

PARAMETERS

OutStats	Where the statistics are returned.

RETURN

Status	Exception code indicates success or reason for failure.

EXCEPTIONS

AE_OK	Statistics were successfully returned.

Functional Description:

This function returns execution statistics of the subsystem. Included are the number of GPEs, SCIs, and Fixed Events. Also, the number of control methods executed.

The returned **ACPI_STATISTICS** structure is shown below:

```
typedef struct acpi_statistics
{
    UINT32    SciCount;
    UINT32    GpeCount;
    UINT32    FixedEventCount[ACPI_NUM_FIXED_EVENTS];
    UINT32    MethodCount;

} ACPI_STATISTICS;
```

8.12.5 AcpiPurgeCachedObjects

Empty all internal object caches.

ACPI_STATUS
AcpiPurgeCachedObjects (
void)

PARAMETERS

None

RETURN

Status	Exception code indicates success or reason for failure.

EXCEPTIONS

AE_OK	The caches were successfully purged.

Functional Description:

This function purges all internal object caches, freeing all memory blocks: It can be used to purge the cache after particularly large operations, or the cache can be periodically flushed to ensure that no large amounts of stagnant cache objects are present. It is implemented by calling *AcpiOsPurgeCache* for each of the object caches.

8.13 Global Variables

There are several global variables that are useful for ACPICA users.

8.13.1 AcpiDbgLevel & AcpiDbgLayer

These globals control the debug output mechanism. *AcpiDbgLevel* specifies the current debug level and *AcpiDbgLayer* specifies which ACPICA components will output debug information.

See the description of **ACPI_DEBUG_PRINT** for more information.

8.13.2 AcpiGbl_FADT

This is a local copy of the system FADT, converted to a common internal format. ACPI-related device drivers often require information directly from the FADT. The table can be directly accessed via this symbol.

8.13.3 AcpiCurrentGpeCount

The current number of active (available) system GPEs. This includes the GPE blocks defined in the FADT, as well as any installed GPE block devices. This is a dynamic value that can increase or decrease as GPE block devices are installed or removed. This value also serves as the maximum index value for the *AcpiGetGpeDevice* interface.

9 OS Services Layer - External Interface Definition

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

Only the external definitions of the **AcpiOs*** interfaces are clearly defined by this document. The actual implementation of the services and interfaces is by definition OS dependent and may be very different for different operating systems.

9.1 Environmental and ACPI Tables

9.1.1 AcpiOsInitialize

Initialize the OSL subsystem.

ACPI_STATUS
AcpiOsInitialize (
void)

PARAMETERS

None

RETURN VALUE

Status	Initialization status.

Functional Description:

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

9.1.2 AcpiOsTerminate

Terminate the OSL subsystem.

ACPI_STATUS
AcpiOsTerminate (
void)

PARAMETERS

None

RETURN VALUE

Status	Termination status.

Functional Description:

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

9.1.3 AcpiOsGetRootPointer

Obtain the Root ACPI table pointer (RSDP).

ACPI_PHYSICAL_ADDRESS
AcpiOsGetRootPointer (
void)

PARAMETERS

None.

RETURN VALUE

Address	The physical address of the RSDP.

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.

9.1.4 AcpiOsPredefinedOverride

Allow the host OS to override a predefined ACPI object.

```
ACPI_STATUS
AcpiOsPredefinedOverride (
    const ACPI_PREDEFINED_NAMES    *PredefinedObject,
    ACPI_STRING                    *NewValue)
```

PARAMETERS

PredefinedObject	A pointer to a predefined object (name and initial value.)
NewValue	Where a new value for the predefined object is returned. NULL if there is no override for this object.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

Functional Description:

This function allows the host to override the predefined objects in the ACPI namespace.

9.1.5 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 header of the existing ACPI table.
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 an ACPI table that was found in the firmware. The host OS can examine the existing table header for the table signature and version number(s) and decide to replace it if desired. Note, only the table header is guaranteed to be valid and accessible, not the entire table. Further, the header is only guaranteed to be valid and accessible for the duration of the execution of this function. It may be unmapped immediately afterwards.

The full identification of an ACPI table includes the following header items:

- The 4-character ACPI signature
- The Revision
- The table Length
- The OEM ID string
- The OEM Table ID string
- The OEM Revision

ACPI Table Header Definition

```
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;
```

During initialization, ACPICA will invoke this interface once for each table defined in the RSDT/XSDT, and once for the DSDT (pointed to by the FADT). This includes all tables in the RSDT/XSDT, even tables that are not directly consumed by ACPICA such as ECDT, MADT, SRAT, SLIT, etc., and all of the OEMx tables.

Tables are installed and *AcpiOsTableOverride* is called in the order that they appear in the RSDT/XSDT. This may be important for tables that can have multiple instantiations such as the SSDT. If the host wishes to replace an individual SSDT, it can keep track of the SSDT instantiations, or it can differentiate SSDTs based upon the full ACPI table identification described above.

ACPICA will also call this interface for each table that is dynamically loaded via the **Load** AML operator. Tables that are loaded via this mechanism are typically SSDTs and OEMx tables.

The **LoadTable** AML operator is used to load the namespace from tables that appear in the RSDT/XSDT with signatures other than SSDT, typically the OEMx tables that contain executable AML code. These tables can be replaced during the initialization phase when ACPICA traverses the RSDT/XSDT as above. *AcpiOsTableOverride* is therefore not invoked when a **LoadTable** is executed.

9.2 Memory Management

These interfaces provide an OS-independent memory management interface.

9.2.1 AcpiOsCreateCache

Create a memory cache object

```
ACPI_STATUS
AcpiOsCreateCache (
    char                    *CacheName,
    UINT16                  ObjectSize,
    UINT16                  MaxDepth,
    ACPI_CACHE_T            **ReturnCache)
```

PARAMETERS

CacheName	An ASCII identifier for the cache. May or may not be used by the host.
ObjectSize	The size of each object in the cache.
MaxDepth	Maximum depth of the cache (max number of objects.) May or may not be used by the host.
ReturnCache	Where a pointer to the cache object is returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The cache was successfully created.
AE_BAD_PARAMETER	At least one of the following is true: • The ReturnCache pointer is `NULL`. • The ObjectSize is less than 16.
AE_NO_MEMORY	Insufficient dynamic memory to complete the operation.

Functional Description:

This function creates a cache object. Many host operating systems have a cache manager that can be used to implement the cache functions. The ACPICA code uses many dynamic objects of the same size (such as the ACPI_OPERAND_OBJECT), and the use of a cache can improve performance considerably.

9.2.2 AcpiOsDeleteCache

Delete a memory cache object.

ACPI_STATUS
AcpiOsDeleteCache (
ACPI_CACHE_T ***Cache)**

PARAMETERS

Cache	The cache object to be deleted.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The cache was successfully created.
AE_BAD_PARAMETER	The *Cache* pointer is NULL.

Functional Description:

This function deletes a cache object that was created via *AcpiOsCreateCache*. Any objects currently within the cache must also be deleted.

9.2.3 AcpiOsPurgeCache

Free all objects currently within a cache object.

ACPI_STATUS
AcpiOsPurgeCache (
ACPI_CACHE_T ***Cache)**

PARAMETERS

Cache	The cache object to be deleted.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The cache was successfully created.
AE_BAD_PARAMETER	The *Cache* pointer is NULL.

Functional Description:

This function deletes all objects that currently reside within a cache.

9.2.4 AcpiOsAcquireObject

Acquire an object from a cache.

```
void *
AcpiOsAcquireObject (
    ACPI_CACHE_T            *Cache)
```

PARAMETERS

Cache	The cache object from which to acquire an object.

RETURN VALUE

Object	A pointer to a cache object. NULL if the object could not be acquired.

EXCEPTIONS

NULL is returned if an object could not be acquired.

Functional Description:

This function acquires an object from the specified cache.

9.2.5 AcpiOsReleaseObject

Release an object to a cache.

```
ACPI_STATUS
AcpiOsReleaseObject (
    ACPI_CACHE_T            *Cache,
    void                    *Object)
```

PARAMETERS

Cache	The cache object to which the object will be released.
Object	The object to be released.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The cache was successfully created.
AE_BAD_PARAMETER	The Cache or Object pointer is NULL.

Functional Description:

This function releases an object back to the specified cache. It must have been previously acquired from the same cache via *AcpiOsAcquireObject*.

9.2.6 AcpiOsMapMemory

Map physical memory into the caller's address space.

```
void *
AcpiOsMapMemory (
    ACPI_PHYSICAL_ADDRESS   PhysicalAddress,
    ACPI_SIZE               Length)
```

PARAMETERS

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

RETURN VALUE

LogicalAddress	Pointer to the mapped memory. A NULL pointer indicates failure.

EXCEPTIONS

NULL is returned if there was a mapping failure.

Functional Description:

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

9.2.7 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*.

9.2.8 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.

9.2.9 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.

9.2.10 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*.

9.2.11 AcpiOsReadable

Check if a memory region is readable.

BOOLEAN
AcpiOsReadable (
void ***Memory**
ACPI_SIZE **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 ACPICA subsystem.

9.2.12 AcpiOsWritable

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

BOOLEAN
AcpiOsWritable (
void ***Memory,**
ACPI_SIZE **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 ACPICA subsystem..

9.3 Multithreading and Scheduling Services

9.3.1 AcpiOsGetThreadId

Obtain the ID of the currently executing thread.

ACPI_THREAD_ID
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.

9.3.2 AcpiOsExecute

Schedule a procedure for deferred execution.

ACPI_STATUS
AcpiOsExecute (

ACPI_EXECUTE_TYPE	**Type,**
ACPI_OSD_EXEC_CALLBACK	**Function,**
void	***Context)**

PARAMETERS

Type	Type of the callback function: **OSL_GLOBAL_LOCK_HANDLER** **OSL_NOTIFY_HANDLER** **OSL_GPE_HANDLER**

OSL_DEBUGGER_THREAD
OSL_EC_POLL_HANDLER
OSL_EC_BURST_HANDLER

Function	Address of the procedure to execute.
Context	A context value to be passed to the called procedure.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

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

Functional Description:

This function queues a procedure for later scheduling and execution.

9.3.3 AcpiOsSleep

Suspend the running task (course granularity).

void
AcpiOsSleep (
ACPI_INTEGER **Milliseconds)**

PARAMETERS

Milliseconds	The amount of time to sleep, in milliseconds.

RETURN VALUE

None

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.

9.3.4 AcpiOsStall

Wait for a short amount of time (fine granularity).

void
AcpiOsStall (
UINT32 **Microseconds)**

PARAMETERS

Microseconds	The amount of time to delay, in microseconds.

RETURN VALUE

None

Functional Description:

This function waits for the specified time. Execution of the running thread is not suspended for this time. The time granularity is one microsecond.

9.4 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.

9.4.1 AcpiOsCreateMutex

Create a mutex object.

ACPI_STATUS
AcpiOsCreateMutex (
ACPI_MUTEX ***OutHandle)**

PARAMETERS

OutHandle	A pointer to a location where a handle to the mutex is to be returned.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The mutex was successfully created.

AE_BAD_PARAMETER	The *OutHandle* pointer is `NULL`.
AE_NO_MEMORY	Insufficient memory to create the mutex.

Functional Description:

Create a mutex object. Some host operating systems have separate mutex interfaces that can be used to implement this and the other OSL mutex interfaces. If not, the the mutex interfaces can be implemented with semaphore interfaces.

9.4.2 AcpiOsDeleteMutex

Delete a mutex object.

void
AcpiOsDeleteMutex (
ACPI_MUTEX **Handle)**

PARAMETERS

Handle	The mutex to be deleted.

RETURN VALUE

None.

Functional Description:

Deletes a mutex object.

9.4.3 AcpiOsAcquireMutex

Acquire ownership of a mutex object.

ACPI_STATUS
AcpiOsAcquireMutex (
ACPI_MUTEX **Handle,**
UINT16 **Timeout)**

PARAMETERS

Handle	The mutex to be acquired.
Timeout	How long the caller is willing to wait for the requested units. The timeout is specified in milliseconds. A value of 0xFFFF (-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 mutex was successfully acquired.
AE_BAD_PARAMETER	The *Handle* pointer is `NULL`.

Functional Description:

Acquire ownership of a mutex object.

9.4.4 AcpiOsReleaseMutex

Release ownership of a mutex object.

```
void
AcpiOsReleaseMutex (
    ACPI_MUTEX              Handle)
```

PARAMETERS

Handle	The mutex to be released.

RETURN VALUE

None

Functional Description:

Release a mutex object. The mutex must have be previously acquired via *AcpiOsAcquireMutex*.

9.4.5 AcpiOsCreateSemaphore

Create a semaphore.

```
ACPI_STATUS
AcpiOsCreateSemaphore (
    UINT32                  MaxUnits,
    UINT32                  InitialUnits,
    ACPI_SEMAPHORE          *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: • The *InitialUnits* is invalid. • 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 ACPICA Core Subsystem creates semaphores of both the mutex and general-purpose variety.

9.4.6 AcpiOsDeleteSemaphore

Delete a semaphore.

ACPI_STATUS
AcpiOsDeleteSemaphore (
ACPI_SEMAPHORE **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.

9.4.7 AcpiOsWaitSemaphore

Wait for units from a semaphore.

```
ACPI_STATUS
AcpiOsWaitSemaphore (
    ACPI_SEMAPHORE          Handle,
    UINT32                  Units,
    UINT16                  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 0xFFFF (-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.

9.4.8 AcpiOsSignalSemaphore

Send units to a semaphore.

ACPI_STATUS
AcpiOsSignalSemaphore (
ACPI_SEMAPHORE 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.

9.4.9 AcpiOsCreateLock

Create a spin lock.

ACPI_STATUS
AcpiOsCreateLock (
ACPI_SPINLOCK *OutHandle)

PARAMETERS

OutHandle	A pointer to a location where a handle to the lock 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	The *OutHandle* pointer is `NULL`.
AE_NO_MEMORY	Insufficient memory to create the semaphore.

Functional Description:

Create a spin lock. Spin locks are used in the ACPICA subsystem only when there is requirement for mutual exclusion on data structures that are accessed by both interrupt handlers and normal code.

9.4.10 AcpiOsDeleteLock

Delete a spin lock.

void
AcpiOsDeleteLock (
ACPI_HANDLE **Handle)**

PARAMETERS

Handle	A handle to a lock object that was returned by a previous call to *AcpiOsCreateLock*.

RETURN VALUE

None	Exception code that indicates success or reason for failure.

EXCEPTIONS

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

Functional Description:

Delete a spin lock.

9.4.11 AcpiOsAcquireLock

Acquire a spin lock.

ACPI_CPU_FLAGS
AcpiOsAcquireLock (
ACPI_SPINLOCK **Handle)**

PARAMETERS

Handle	A handle to a lock object that was returned by a previous call to *AcpiOsCreateLock*.

RETURN VALUE

Flags	Platform-dependent CPU flags. To be used when the lock is released.

Functional Description:

Wait for and acquire a spin lock. May be called from interrupt handlers, GPE handlers, and Fixed event handlers. Single threaded OSL implementations should always return **AE_OK** for this interface.

9.4.12 AcpiOsReleaseLock

Release a spin lock.

```
void
AcpiOsReleaseLock (
    ACPI_SPINLOCK           Handle,
    ACPI_CPU_FLAGS          Flags)
```

PARAMETERS

Handle	A handle to a lock object that was returned by a previous call to *AcpiOsCreateLock*.
Flags	CPU flags that were returned from *AcpiOsAcquireLock*

RETURN VALUE

None	Exception code that indicates success or reason for failure.

Functional Description:

Release a previouslly acquired spin lock. Single threaded OSL implementations should always return **AE_OK** for this interface.

9.5 Interrupt Handling

Interrupt handler installation and removal.

9.5.1 AcpiOsInstallInterruptHandler

Install a handler for a hardware interrupt level.

```
ACPI_STATUS
AcpiOsInstallInterruptHandler (
    UINT32                  InterruptLevel,
    ACPI_OSD_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: • The *InterruptNumbe*r is invalid. • The *Handler* pointer is NULL.
AE_ALREADY_EXISTS	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.

9.5.1.1 Interface to OS-independent Interrupt Handlers

Definition of the interface for OS-independent interrupt handlers.

```
typedef
UINT32 (*ACPI_OSD_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: **ACPI_INTERRUPT_HANDLED** **ACPI_INTERRUPT_NOT_HANDLED**

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.

9.5.2 AcpiOsRemoveInterruptHandler

Remove an interrupt handler.

```
ACPI_STATUS
AcpiOsRemoveInterruptHandler (
    UINT32                          InterruptNumber,
    ACPI_OSD_HANDLER                Handler)
```

PARAMETERS

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

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The handler was successfully removed.

AE_BAD_PARAMETER	At least one of the following is true: • The *InterruptNumber* is invalid. • The *Handler* pointer is NULL. • The *Handler* address is not the same as the one that is installed.
AE_NOT_EXIST	There is no handler installed for this interrupt level.

Functional Description:

Remove a previously installed hardware interrupt handler.

9.6 Memory Access 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 ACPICA for small amounts of data transfer only, such as memory mapped I/O. For large transfers (such as reading the ACPI tables), the ACPICA code will call *AcpiOsMapMemory* instead.

Supports Operation Region access to the **ACPI_ADR_SPACE_SYSTEM_MEMORY** (SystemMemory) space.

9.6.1 AcpiOsReadMemory

Read a value from a memory location.

```
ACPI_STATUS
AcpiOsReadMemory (
    ACPI_PHYSICAL_ADDRESS   Address,
    UINT32                  *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, or 32.

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. In other words, a full 32 bits are written to the return *Value* regardless of the number of bits that were read from the memory at *Address*. The caller must ensure that no data will be overwritten by this call.

9.6.2 AcpiOsWriteMemory

Write a value to a memory location.

```
ACPI_STATUS
AcpiOsWriteMemory (
    ACPI_PHYSICAL_ADDRESS   Address,
    UINT32                  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, or 32.

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* parameter are written.

9.7 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.

Supports Operation Region access to the **ACPI_ADR_SPACE_SYSTEM_IO** (SystemIO) space.

The ACPICA subsystem checks each request against a list of protected I/O ports before calling these interfaces.

9.7.1 AcpiOsReadPort

Read a value from an input port.

```
ACPI_STATUS
AcpiOsReadPort (
    ACPI_IO_ADDRESS         Address,
    UINT32                  *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, or 32.

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.

9.7.2 AcpiOsWritePort

Write a value to an output port.

```
ACPI_STATUS
AcpiOsWritePort (
    ACPI_IO_ADDRESS         Address,
    UINT32                  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, or 32.

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* parameter are written.

9.8 PCI Configuration Space Access

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.

Supports Operation Region access to the **ACPI_ADR_SPACE_PCI_CONFIG** (Pci_Config) space.

9.8.1 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 64-bit return value even if the bit width of the location is less than 64.

9.8.2 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 64, only the lower significant bits of the Value are written.

9.8.3 AcpiOsDerivePciId

Derive and update a PCI ID for a PCI device object and PCI operation region.

```
ACPI_STATUS
AcpiOsDerivePciId (
    ACPI_HANDLE             DeviceHandle,
    ACPI_HANDLE             PciRegionHandle,
    ACPI_PCI_ID             **PciId)
```

PARAMETERS

DeviceHandle	A handle to the PCI device.
PciRegionHandle	A handle to the PCI configuration space Operation Region.
PciId	**Input:** The full PCI ID (The full PCI configuration space address, consisting of a segment number, bus number, device number, and function number) as obtained from control methods within the BIOS ACPI tables.

Output: Where the derived PCI ID is returned. Some or all of the PCI ID subfields may be updated by this function.

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

Functional Description:

This function derives a full PCI ID for a PCI device, consisting of a Segment number, a Bus number, and a Device number.

The PCI hardware dynamically configures PCI bus numbers depending on the bus topology discovered during system initialization. The *AcpiOsDerivePciId* function is invoked by the ACPICA subsystem during configuration of a *PCI_Config* Operation Region in order to (possibly) update the Bus number in the *PciId* with the actual Bus number as determined by the hardware and operating system configuration.

The *PciId* parameter is initially populated by the ACPICA subsystem during the Operation Region initialization. ACPICA then calls *AcpiOsDerivePciId*, which is expected to make any necessary modifications to the Segment, Bus, or Device number PCI ID subfields as appropriate for the current hardware and OS configuration.

9.9 Formatted Output

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

9.9.1 AcpiOsPrintf

Formatted stream output.

```
void ACPI_INTERNAL_VAR_XFACE
AcpiOsPrintf (
    const char          *Format,
    …                   <variable argument list>)
```

PARAMETERS

Format	A standard print format string.
…	Variable printf parameter list.

RETURN VALUE

None.

Functional Description:

This function provides formatted output to an open stream.

9.9.2 AcpiOsVprintf

Formatted stream output.

```
void
AcpiOsVprintf (
    const char              *Format,
    va_list                 Args)
```

PARAMETERS

Format	A standard printf format string.
Args	A variable parameter list.

RETURN VALUE

None

Functional Description:

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

9.9.3 AcpiOsRedirectOutput

Redirect the debug output.

```
void
AcpiOsRedirectOutput (
    void                    *Destination)
```

PARAMETERS

Destination	An open file handle or pointer. Debug output will be redirected to this handle/pointer. The format of this parameter is OS-specific.

RETURN VALUE

None

Functional Description:

This function redirects the output of AcpiOsPrintf and AcpiOsVprintf to the specified destination. Usually used to redirect output to a file.

9.10 Miscellaneous

9.10.1 AcpiOsValidateInterface

Check if an_OSI interface is supported by the host OS.

ACPI_STATUS
AcpiOsValidateInterface (
char ***Interface)**

PARAMETERS

Interface	Requested interface to be validated

RETURN VALUE

Status	Exception code that indicates success or reason for failure.

EXCEPTIONS

AE_OK	The interface is supported by the host OS.
AE_SUPPORT	The interface is not supported by the host.
AE_BAD_PARAMETER	The interface parameter is NULL.

Functional Description:

This function matches an interface string to the interfaces supported by the host OS. Strings originate from an AML call to the _OSI control method. See the description of _OSI in the ACPI specification for a list of currently defined strings.

9.10.2 AcpiOsGetTimer

Get current value of the system timer

UINT64
AcpiOsGetTimer (
void)

PARAMETERS

None.

RETURN VALUE

TimerValue	The current value of the system timer in 100-nanosecond units.

Functional Description:

This function returns the current value of a fine-granularity 64-bit system timer. This interface is used to implement the Timer ASL/AML function.

9.10.3 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. The following signals are supported:

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;
```

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;
```

9.10.4 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.

10 ACPICA Deployment Guide

10.1 Using the ACPICA Core Subsystem Interfaces

10.1.1 Initialization Sequence

In order to allow the most flexibility for the host operating system, there is no single interface that initializes the entire ACPICA 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 ();
   ```

10.1.2 ACPICA Initialization Examples

10.1.2.1 Full ACPICA Initialization

```
ACPI_STATUS
InitializeFullAcpi (void)
{
    ACPI_STATUS             Status;


    /* Initialize the ACPICA subsystem */

    Status = AcpiInitializeSubsystem ();
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }

    /* Initialize the ACPICA Table Manager and get all ACPI tables */

    Status = AcpiInitializeTables (NULL, 16, FALSE);
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }
```

```
    /* Create the ACPI namespace from ACPI tables */

    Status = AcpiLoadTables ();
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }

    /* Note: Local handlers should be installed here */

    /* Initialize the ACPI hardware */

    Status = AcpiEnableSubsystem (ACPI_FULL_INITIALIZATION);
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }

    /* Complete the ACPI namespace object initialization */

    Status = AcpiInitializeObjects (ACPI_FULL_INITIALIZATION);
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }

    return (AE_OK);
}
```

10.1.2.2 ACPICA Initialization With Early ACPI Table Access

```
#define ACPI_MAX_INIT_TABLES    16
static ACPI_TABLE_DESC      TableArray[ACPI_MAX_INIT_TABLES];

ACPI_STATUS
InitializeAcpiTables (void)
{
    ACPI_STATUS             Status;


    /* Initialize the ACPICA Table Manager and get all ACPI tables */

    Status = AcpiInitializeTables (TableArray, ACPI_MAX_INIT_TABLES, TRUE);
    return (Status);
}


ACPI_STATUS
InitializeAcpi (void)
{
    ACPI_STATUS             Status;
```

```
    /* Initialize the ACPICA subsystem */

    Status = AcpiInitializeSubsystem ();
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }

    /* Copy the root table list to dynamic memory */

    Status = AcpiReallocateRootTable ();
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }

    /* Create the ACPI namespace from ACPI tables */

    Status = AcpiLoadTables ();
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }

    /* Note: Local handlers should be installed here */

    /* Initialize the ACPI hardware */

    Status = AcpiEnableSubsystem (ACPI_FULL_INITIALIZATION);
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }

    /* Complete the ACPI namespace object initialization */

    Status = AcpiInitializeObjects (ACPI_FULL_INITIALIZATION);
    if (ACPI_FAILURE (Status))
    {
        return (Status);
    }

    return (AE_OK);
}
```

10.1.3 Shutdown Sequence

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

1. Unload the namespace and free all resources:

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

10.1.4 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 */
}
```

10.1.5 Traversing the ACPI Namespace (High Level)

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

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

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

```
DisplaySystemDevices (void)
{
    ACPI_HANDLE             SysBusHandle;

    AcpiNameToHandle (0, NS_SYSTEM_BUS, &SysBusHandle);

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

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

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

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


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

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

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

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

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

    return NULL;
}
```

10.2 Implementing the OS Services Layer

10.2.1 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.

10.2.2 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 ACPICA 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.

10.2.3 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 ACPICA Debugger creates a permanent thread.

10.2.4 Mutual Exclusion and Synchronization

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

10.2.5 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.

10.2.6 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.

10.2.7 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.

11 Tools and Utilities

11.1 iASL Compiler

The iASL compiler is a fully-featured translator for the ACPI Source Language (ASL). As part of the Intel ACPI Component Architecture, the Intel ASL compiler implements translation for the ACPI Source Language (ASL) to the ACPI Machine Language (AML).

iASL also includes the ACPICA disassembler, and will disassemble any ACPI table, including both tables that contain AML (DSDT, SSDT, OEMx) and tables that contain data only (all other ACPI tables such as FADT, MADT, ECDT, etc.)

The compiler is fully documented in the *iASL Compiler User Reference*.

```
Intel ACPI Component Architecture
ASL Optimizing Compiler version 20081031 [Oct 31 2008]
Copyright (C) 2000 - 2008 Intel Corporation
Supports ACPI Specification Revision 3.0a

Usage:    iasl [Options] [Files]

General Output:
  -p <prefix>    Specify path/filename prefix for all output files
  -vi            Less verbose errors and warnings for use with IDEs
  -vo            Enable optimization comments
  -vr            Disable remarks
  -vs            Disable signon
  -w<1|2|3>      Set warning reporting level

AML Output Files:
  -s<a|c>        Create AML in assembler or C source file (*.asm or *.c)
  -i<a|c>        Create assembler or C include file (*.inc or *.h)
  -t<a|c>        Create AML in assembler or C hex table (*.hex)

AML Code Generation:
  -oa            Disable all optimizations (compatibility mode)
  -of            Disable constant folding
  -oi            Disable integer optimization to Zero/One/Ones
  -on            Disable named reference string optimization
  -r<Revision>   Override table header Revision (1-255)

Listings:
  -l             Create mixed listing file (ASL source and AML) (*.lst)
  -ln            Create namespace file (*.nsp)
  -ls            Create combined source file (expanded includes) (*.src)

AML Disassembler:
  -d  [file]     Disassemble or decode binary ACPI table to file (*.dsl)
  -dc [file]     Disassemble AML and immediately compile it
                 (Obtain DSDT from current system if no input file)
  -e  [f1,f2]    Include ACPI table(s) for external symbol resolution
  -2             Emit ACPI 2.0 compatible ASL code
  -g             Get ACPI tables and write to files (*.dat)
```

```
Help:
  -h              Additional help and compiler debug options
  -hc             Display operators allowed in constant expressions
  -hr             Display ACPI reserved method names
```

11.2 AcpiExec – User Mode ACPI Execution/Simulation

This 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 ACPICA Core Subsystem including the ACPICA Debugger. All hardware access via the AML is simulated. All ACPICA debugger commands are available (See the ACPICA Debugger Reference later in this document.)

```
Intel ACPI Component Architecture
AML Execution/Debug Utility version 20081031 [Oct 31 2008]

Usage: acpiexec [Options] [InputFile]

Where:
   -?                  Display this message
   -a                  Do not abort methods on error
   -b <CommandLine>    Batch mode command execution
   -e [Method]         Batch mode method execution
   -i                  Do not run STA/INI methods during init
   -m                  Display final memory use statistics
   -o <OutputFile>     Send output to this file
   -r                  Disable OpRegion address simulation
   -s                  Enable Interpreter Slack Mode
   -t                  Enable Interpreter Serialized Mode
   -v                  Verbose init output
   -x <DebugLevel>     Specify debug output level
```

11.3 AcpiXtract – Extract ACPI Tables

This utility is used to extract binary ACPI tables from the ASCII output of the acpidump utility (acpidump is a utility that is part of the PM Tools package.)

```
Usage: acpixtract [option] <InputFile>

Extract binary ACPI tables from text acpidump output
Default invocation extracts all DSDTs and SSDTs
Version 20081031

Options:
 -a                  Extract all tables, not just DSDT/SSDT
 -l                  List table summaries, do not extract
 -s<Signature>       Extract all tables named <Signature>
```

11.4 AcpiSrc – Convert ACPICA Source Code

This utility is used to convert the ACPICA into Linux code format. It can also be used to clean the ACPICA code by removing extra trailing blanks, etc., and to generate source code statistics.

```
ACPI Source Code Conversion Utility version 20081031 [Oct 31 2008]

Usage: acpisrc [-c|l|u] [-dsvy] <SourceDir> <DestinationDir>

Where: -c          Generate cleaned version of the source
       -l          Generate Linux version of the source
       -u          Generate Custom source translation

       -d          Leave debug statements in code
       -s          Generate source statistics only
       -v          Verbose mode
       -y          Suppress file overwrite prompts
```

Example output – source code statistics for ACPICA:

```
ACPI Source Code Conversion Utility version 20081031 [Oct 31 2008]

Source code statistics only
AcpiSrc statistics:

     233 Files processed
     342 Tabs found
       0 Missing if/else braces
      22 Non-ANSI comments found
  159707 Total Lines
   82496 Lines of code
   29508 Lines of non-comment whitespace
   32210 Lines of comments
    3013 Long lines found
     2.8 Ratio of code to whitespace
     2.6 Ratio of code to comments
         51% code, 20% comments, 18% whitespace, 15% headers
```

12 ACPICA Debugger Reference

12.1 Overview

The ACPICA AML Debugger is an optional subcomponent of the ACPICA 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.

12.2 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 ACPICA 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. An example of this mode is the AcpiExec utility.

12.2.1 The AcpiExec Utility

An example of the Ring3/Ring3 model of execution is the user mode *AcpiExec* utility. This application includes the entire ACPICA 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. The AcpiExec utility simulates hardware access.

12.3 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.

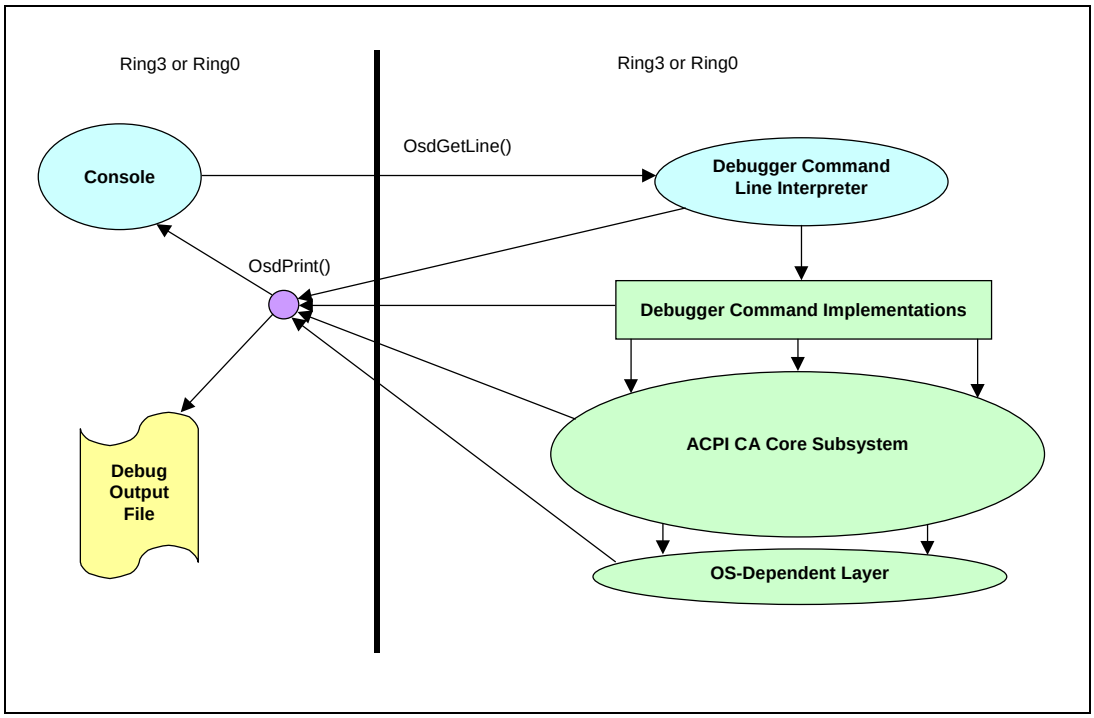
When executing in a Ring 0 environment, the debugger initialization creates a separate thread for the debugger CLI. This threads performs the following tasks until the debugger is shut down:

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 ACPICA 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 ACPICA subsystem can reside in a different protection ring than the user console support and file I/O support.

Figure 9. ACPICA Debugger Architecture



12.4 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.

12.5 Command Overview

There are four 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 **Namespace Access** commands are always available. These commands provide information about the currently loaded ACPI namespace.
3. 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
4. The **File I/O** Commands are available only if a filesystem is available to the debugger.

12.6 General Purpose Commands

12.6.1 Allocations

Memory allocation status

SYNTAX

```
- allocations
```

This command dumps the current status of the dynamic memory allocations, as maintained by the ACPICA 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.

12.6.2 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

12.6.3 Exit

Terminate

SYNTAX

```
- exit
```

Terminate the ACPICA subsystem and exit the debugger.

12.6.4 Help

Get help

SYNTAX

```
- help
```

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

12.6.5 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.

12.6.6 Level

Set debug output level

SYNTAX

```
- level [<DebugLevel>] [console]
```

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

12.6.7 Locks

Display mutex info and status

SYNTAX

```
- locks
```

This command displays information and current status of the various mutexes used for internal synchronization.

12.6.8 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 ACPICA subsystem is terminated and the debugger exits.

12.6.9 Stats

Namespace statistics

SYNTAX

```
- stats [Allocations|Memory|Misc|Objects|Sizes|Stack]
```

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.

SUBCOMMANDS

Allocations: Display a list of current dynamic memory allocations

Memory: Dump internal memory lists (If ACPICA memory cache is configured)

Misc: Namespace search and mutex use statistics

Objects: Summary of namespace objects

Sizes: Memory allocation sizes for each of the internal objects

Stack: Display CPU stack usage

12.6.10 Tables

Display ACPI table info

SYNTAX

```
- tables
```

This command displays information about each of the loaded ACPI tables. It uses the internal AcpiTbPrintTableHeader function.

12.6.11 Unload

Unload table

SYNTAX

```
- unload <TableSignature> [Instance]
```

Unload an ACPI Table <Not implemented>

12.7 Namespace Access Commands

12.7.1 BusInfo

Display system bus information

SYNTAX

```
- businfo
```

This command displays information about all device objects that have a corresponding _PRT method. Information includes the _ADR, _HID, _UID, and _CID.

12.7.2 Disassemble

Disassemble a control method

SYNTAX

```
- disassemble <Method>
```

This command will dissassemble the input method to the original ASL code.

12.7.3 Event

Generate an ACPI Event

SYNTAX

```
- event  <Value>
```

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

12.7.4 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".

12.7.5 Gpe

Generate a GPE

SYNTAX

```
- gpe <Block Address> <GPE number>
```

Generate a GPE at the GPE number within the GPE block specified at the Block Address. Use 0 for the block address to generate a GPE within the permanent FADT-defined GPE blocks (GPE0 and GPE1.).

12.7.6 Gpes

Display GPE block information

SYNTAX

```
- gpes
```

Display information on all GPE blocks, including the FADT-defined GPE blocks (GPE0 and GPE1) and all loaded GPE Block Devices.

12.7.7 Integrity

Validate namespace

SYNTAX

```
- integrity
```

This command validates the integrity of the entire loaded namespace. It walks the entire namespace and checks each namespace node for correctness.

12.7.8 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”.)

12.7.9 Namespace

Display the loaded ACPI 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.

12.7.10 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.

12.7.11 Object

Display typed objects

SYNTAX

```
- object <Object Type>
```

Display objects within the namespace of the requested type.

The ObjectType parameter must be one of the following:

ANY
INTEGERS
STRINGS
BUFFERS
PACKAGES
FIELDS
DEVICES
EVENTS
METHODS
MUTEXES
REGIONS
POWERRESOURCES
PROCESSORS
THERMALZONES
BUFFERFIELDS
DDBHANDLES
DEBUG
REGIONFIELDS
BANKFIELDS
INDEXFIELDS
REFERENCES
ALIAS

12.7.12 Owner

Display namespace by owner ID

SYNTAX

```
- owner <Owner ID> [Depth]
```

Display objects within the namespace owned by the requested Owner ID.

12.7.13 Predefined

Display and check all predefined methods/objects

SYNTAX

```
- predefined
```

This command displays and validates all predefined methods and objects (names that start with underscore and are predefined by the ACPI specification.)

The validation checks the input argument count (if object is a control method) against the count defined in the ACPI spec.

12.7.14 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.

12.7.15 References

Find all references to an object within the namespace

SYNTAX

```
- references  <Address>
```

Display all references to the object at the specified address.

12.7.16 Resources

Display device resources

SYNTAX

```
- resources  <Address>
```

Display resource lists (_PRS, _CRS, etc.) for the Device object at the specified address.

12.7.17 Set N

Set object value

SYNTAX

```
- set N <NamedObject> <Value>
```

This command sets the value of a namespace object.

12.7.18 Sleep

Simulate ACPI Sleep/Wake

SYNTAX

```
- sleep <SleepState>
```

This command simulates the sleep/wake sequence. SleepState should be an integer, 1-5. The following ACPICA interfaces are executed:

AcpiEnterSleepStatePrep
AcpiEnterSleepState
AcpiLeaveSleepState

12.7.19 Terminate

Shutdown ACPICA subsystem

SYNTAX

```
- terminate
```

Shutdown the ACPICA 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.

12.7.20 Type

Display object type

SYNTAX

```
- type <Object>
```

This command displays the type of a namespace object.

12.8 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

12.8.1 Arguments

Display Method arguments

SYNTAX

```
- arguments
- args
```

Display all arguments to the currently executing control method

12.8.2 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.

12.8.3 Call

Run to next call

SYNTAX

```
- call
```

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

12.8.4 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.

12.8.5 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.

12.8.6 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.

12.8.7 Information

Info about a control method

SYNTAX

```
- information
```

12.8.8 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.

12.8.9 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.

12.8.10 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.

12.8.11 Results

Display method result stack

SYNTAX

```
- results
```

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

12.8.12 Set

Set arguments or locals

SYNTAX

```
- set Arg|Local <ID> <Value>
```

Set the value of any of a method’s arguments or local variables. ID is 0-7 for method locals and 0-6 for method arguments.

12.8.13 Stop

Stop method

SYNTAX

```
- stop
```

Terminate the currently executing control method

12.8.14 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.

12.8.15 Trace

Set a method trace

SYNTAX

```
- trace <method name>
```

This command sets a trace command that will trace the input method if and when it is executed. Uses the AcpiDebugTrace interface.

12.8.16 Tree

Display calling tree

SYNTAX

```
- tree
```

Display the calling tree of the current method (Displays all nested control method invocations.)

12.9 File I/O Commands

12.9.1 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.

12.9.2 Load

Load ACPI table

SYNTAX

```
- load <Filename>
```

Load an ACPI table into the namespace from a file.

12.9.3 Open

Open debug output file

SYNTAX

```
- open <Filename>
```

Open a file for debug output.

This page intentionally left blank.