# ACPICA

From OSDev Wiki

The ACPI Component Architecture **ACPICA** provides an operating system (OS)-independent reference implementation of the Advanced Configuration and Power Interface. It can be adapted to any host OS. The ACPICA code is meant to be directly integrated into the host OS, as a kernel-resident subsystem. Hosting the ACPICA requires no changes to the core ACPICA code. However, it does require a small OS-specific interface layer, which must be written specifically for each host OS.

The complexity of the ACPI specification leads to a lengthy and difficult implementation in OS 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 easily integrated into any OS.

As said before you need to implement yourself a few functions that are part of the OS interface layer (OSL). Here are those functions:

## Contents

## OS Layer

There are at least 45 functions to be implemented (luckily most of them are really simple):

### Environmental and ACPI Tables

#### AcpiOsInitialize

```
ACPI_STATUS AcpiOsInitialize()
```

This is called during ACPICA Initialization. It gives the possibility to the OSL to initialize itself. Generally it should do nothing

#### AcpiOsTerminate

```
ACPI_STATUS AcpiOsTerminate()
```

This is called during ACPICA Shutdown (which is not the computer shutdown, just the ACPI). Here you can free any memory which was allocated in AcpiOsInitialize.

#### AcpiOsGetRootPointer

```
ACPI_PHYSICAL_ADDRESS AcpiOsGetRootPointer()
```

ACPICA leaves to you the job of finding the RSDP for platform compatibility. However, on x86 you can just do:

```
ACPI_PHYSICAL_ADDRESS AcpiOsGetRootPointer()
{
	ACPI_PHYSICAL_ADDRESS  Ret;
	Ret = 0;
	AcpiFindRootPointer(&Ret);
	return Ret;
}
```

where AcpiFindRootPointer is part of ACPICA itself.

Note: The ACPI specification is highly portable specification, however, it has a static part which is generally non-portable: the location of the Root System Descriptor Pointer. This pointer may be found in many different ways depending on the chipset. On PC-compatible computers (without EFI) it is located in lower memory generally somewhere between 0x80000 and 0x100000. However, even within the PC compatible platform, an EFI-enabled board will export the RSDP to the OS on when it loads it through the EFI system tables. Other boards on server machines which are not PC-compatibles, like embedded and handheld devices which implement ACPI will again, not all be expected to position the RSDP in the same place as any other board. The RSDP is therefore located in a chipset-specific manner; From the time the OS has the RSDP, the rest of ACPI is completely portable. However, the way the RSDP is found is not. This would be the reason that the ACPICA code wouldn't try to provide routines to expressly find the RSDP in a portable manner. If your system uses EFI, locate it in the system tables or use Multiboot2 compliant loader, which provides the RSDP for you.

### AcpiOsPredefinedOverride

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

This function allows the host to override the predefined objects in the ACPI namespace. It is called when a new object is found in the ACPI namespace. However you can just put NULL in *NewValue and return.

### AcpiOsTableOverride

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

The same of AcpiOsPredefinedOverride but for entire ACPI tables. You can replace them. Just put NULL in *NewTable and return.

## Memory Management

### AcpiOsMapMemory

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

This is not really easy. ACPICA is asking you to map a physical address in the virtual address space. If you don't use paging, just return PhysicalAddress. You need:

1. To round Length up to the size of a page (Length can be 2, 1024 for example)
2. Find a range of virtual addresses where map the physical frames.
3. Map the physical frames to the virtual addresses choosen.
4. Return the virtual address plus the page offset of the physical address. (Eg. If you where asked to map 0x40E you have to return 0xF000040E and not just 0xF0000000)

### AcpiOsUnmapMemory

```
void AcpiOsUnmapMemory(void *where, ACPI_SIZE length)
```

Unmap pages mapped using AcpiOsMapMemory. Where is the Virtual address returned in AcpiOsMapMemory and length is equal to the length of the same function. Just remove the virtual address form the page directory and set that virtual address as reusable. **Note:** for the last two functions you might need a separated heap.

### AcpiOsGetPhysicalAddress

```
ACPI_STATUS AcpiOsGetPhysicalAddress(void *LogicalAddress, ACPI_PHYSICAL_ADDRESS *PhysicalAddress)
```

Get the physical address pointed by LogicalAddress and put it in *PhysicalAddress. If you do not use paging just put LogicalAddress in *PhysicalAddress

### AcpiOsAllocate

```
void *AcpiOsAllocate(ACPI_SIZE Size);
```

Dynamically allocate memory in the heap. Return NULL on error or end of memory. In other words:

```
return malloc(size)
```

### AcpiOsFree

```
void AcpiOsFree(void *Memory);
```

Free previously dynamically allocated memory.

### AcpiOsReadable

```
BOOLEAN AcpiOsReadable(void *Memory, ACPI_SIZE Length)
```

Check that the memory from Memory to (Memory + Length) is readable. This is never used (at least i did never see it used). However it should (on x86(_64)) check that a the locations of memory are present in the paging structure.

### AcpiOsWritable

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

Check that the memory from Memory to (Memory + Length) is writable. This is never used (at least i did never see it used). However it should (on x86(_64)) check that a the locations of memory are present in the paging structure and that they have the Write flag set.

### Caches

ACPICA uses caches to make things faster. You can use your kernel's cache or just let ACPICA use its internal cache. To do so just define in your platform/ac*whatever*.h file

```
#define ACPI_CACHE_T                ACPI_MEMORY_LIST
#define ACPI_USE_LOCAL_CACHE        1
```

## Multithreading and Scheduling Services

To use all the features of ACPICA you need Scheduling support too. ACPICA specifies Threads but if you have only processes, that should work too. If you don't have and don't plan to have a scheduler, you can only use the Table features of ACPICA.

### AcpiOsGetThreadId

```
ACPI_THREAD_ID AcpiOsGetThreadId()
```

Return the unique ID of the running thread. In the Linux implementation:

```
return pthread_self();
```

### AcpiOsExecute

```
ACPI_STATUS AcpiOsExecute(ACPI_EXECUTE_TYPE Type, ACPI_OSD_EXEC_CALLBACK Function, void *Context)
```

Create a new Thread (or process) with entry point at *Function* using parameter *Context*. *Type* is not really useful. When the scheduler chooses this thread it has to put Context on the stack to have something like:

```
Function(Context);
```

### AcpiOsSleep

```
void AcpiOsSleep(UINT64 Milliseconds)
```

Put the current thread to sleep for *n* milliseconds.

### AcpiOsStall

```
void AcpiOsStall(UINT32 Microseconds)
```

Stall the thread for *n* microseconds. Note: this should not put the thread in the sleep queue. The thread should keep on running. Just looping.

## Mutual Exclusion and Synchronization

Yes, you need Spinlocks, Mutexes and Semaphores too. Nobody said it was easy. :)

### AcpiOsCreateMutex

```
ACPI_STATUS AcpiOsCreateMutex(ACPI_MUTEX *OutHandle)
```

Create space for a new Mutex using malloc (or eventually new) and put the address of the Mutex in *OutHandle, return AE_NO_MEMORY if malloc or new return NULL. Else return AE_OK like in most other functions.

### AcpiOsDeleteMutex

```
void AcpiOsDeleteMutex(ACPI_MUTEX Handle)
```

This is far too complex to be explained here, so I'll leave this to your imagination.

### AcpiOsAcquireMutex

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

This would be silly too if not for the Timeout parameter. Timeout can be one of:

- 0: acquire the Mutex if it is free, but do not wait if it is not
- 1 - +inf: acquire the Mutex if it is free, but wait for *Timeout* milliseconds if it is not
- -1 (0xFFFF): acquire the Mutex if it is free, or wait until it became free, then return

### AcpiOsReleaseMutex

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

Do you need explaination?

### AcpiOsCreateSemaphore

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

Create a new Semaphore with the counter initialized to *InitialUnits* and put its address in *OutHandle. I don't know how tu use MaxUnits. The spec says: The maximum number of units this Semaphore will be required to accept. However you should be ok if you ignore this.

### AcpiOsDeleteSemaphore

```
ACPI_STATUS AcpiOsDeleteSemaphore(ACPI_SEMAPHORE Handle)
```

-_-

### AcpiOsWaitSemaphore

```
ACPI_STATUS AcpiOsWaitSemaphore(ACPI_SEMAPHORE Handle, UINT32 Units, UINT16 Timeout)
```

Just like AcpiOsAcquireMutex, same logic for *Timeout*. Units isn't used in the linux implementation. However it *should* be the number of times you have to call sem_wait. I'm not sure about this. Someone should check

### AcpiOsSignalSemaphore

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

Opposite of Wait. *Units*: number of times you *should* call sem_post.

### AcpiOsCreateLock

```
ACPI_STATUS AcpiOsCreateLock(ACPI_SPINLOCK *OutHandle)
```

Create a new spinlock and put its address in *OutHandle. Spinlock should disable interrupts on the current CPU to avoid scheduling and make sure that no other CPU will access the reserved area.

### AcpiOsDeleteLock

```
void AcpiOsDeleteLock(ACPI_HANDLE Handle)
```

### AcpiOsAcquireLock

```
ACPI_CPU_FLAGS AcpiOsAcquireLock(ACPI_SPINLOCK Handle)
```

Lock the spinlock and return a value that will be used as parameter for ReleaseLock. It is mainly used for the status of interrupts before the lock was acquired.

### AcpiOsReleaseLock

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

Release the lock. *Flags* is the return value of AcquireLock. If you used this to store the interrupt state, now is the moment to use it.

## Interrupt Handling

### AcpiOsInstallInterruptHandler

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

ACPI sometimes fires interrupt. ACPICA will take care of them. *InterruptLevel* is the IRQ number that ACPI will use. Handler is an internal function of ACPICA which handles interrupts. Context is the parameter to be past to the Handler. If you're lucky, your IRQ manager uses handlers of this form:

```
uint32_t handler(void *);
```

In this case just assign the handler to the IRQ number with that context. I wasn't as lucky so I did:

```
#include <Irq.h>
ACPI_OSD_HANDLER ServiceRout;

InterruptState *AcpiInt(InterruptState *ctx, void *Context)
{
        ServiceRout(Context);
        return ctx;
}

UINT32 AcpiOsInstallInterruptHandler(UINT32 InterruptNumber, ACPI_OSD_HANDLER ServiceRoutine, void *Context)
{
        ServiceRout = ServiceRoutine;

        RegisterIrq(InterruptNumber, AcpiInt, Contex);
        return AE_OK;
}
```

### AcpiOsRemoveInterruptHandler

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

Just UnregisterIrq (InterruptNumber). Handler is provided in case you have an IRQ manager which can have many handlers for one IRQ. This would let you know which handler on that IRQ you have to remove.

## Using ACPICA in your OS

I didn't find any good description of integrating the ACPICA source code into an operating system, and the released package is basically just a bundle of C files with little organization. This is what I ended up having to do:

1. I copied the C files from dispatcher/, events/, executer/, hardware/, parser/, namespace/, utilities/, tables/, and resources/ into a single acpi folder.
2. I copied the header files from include/
3. I created my own header file based on aclinux.h where I ripped out all of the userspace stuff, then I changed around the rest to be appropriate to my OS's definitions.
4. I edited the include/platform/acenv.h file to remove the inclusion of aclinux.h and included my header file instead.
5. I copied over acenv.h, acgcc.h, and my header file over to my include/platform/ folder.

This is in addition to writing an AcpiOs interface layer, and it is not well indicated by the reference manual that you have to actually edit header files. Many of the macros defined in the headers are documented, though.

### Visual Studio experience

From Visual Studio, although there is little organization in the files, it is relatively easy to port. In the provided/generate directory, there is a VC 9.0 solution. The only project required for integration is "AcpiSubsystem". Copy this project along with all the files listed (you can keep the old directory structure). #define's can be used to configure certain aspects of it, and perhaps changing #ifdef WIN32 to #ifdef X86 might be a good idea (Win64 -> x64). Once this is done though the base of it is in place, and actypes.h is the only header file that needs any modification (that listed above). It might be an idea to change the option "Compile as C code" to default - it's all .c anyway. This allows you to add C++ to the project without problems. Once this is done, add OSL.c or OSLPP.cpp, write your OS layer and you are done.

## Code Examples

Here are some examples of different operations that ACPICA can help with.

### Power Off

To power off the machine:

```
AcpiEnterSleepStatePrep(5);
cli(); // disable interrupts
AcpiEnterSleepState(5);
panic("power off"); // in case it didn't work!
```

## External links

- ACPICA Website (http://www.acpica.org/)
- ACPICA Programmer Reference (http://acpica.org/sites/acpica/files/acpica-reference_2.pdf) . Explains how to integrate it in your OS.