# ACPI implementation on FreeBSD

Takanori Watanabe
*Kobe University*

## Abstract

Prior to the introduction of the Advanced Configuration and Power Management Interface (ACPI), PCs did not have a unified standard mechanism that allowed the operating system to enumerate, configure, and manage the power usage and thermal properties of built-in hardware devices. Instead, these devices were either left unmanaged, or they were managed by special BIOS-level code such as Plug-and-Play BIOS (PnP BIOS), Advanced Power Management BIOS (APM), or other vendor-specific BIOS code. These firmware-driven methods increase firmware costs, and the resulting BIOS code is difficult to alter or debug. Device management issues are becoming more important, especially in moble computing environments where fine-grain power management is often necessary. ACPI replaces PnP BIOS, APM, and a number of ad hoc methods while providing a management framework that allows increased flexibility in hardware design. Unfortunately, the increased power and flexibility of ACPI comes with a cost: it requires substantial software support from the operating system kernel. In this paper we describe ACPI, how it is implemented in FreeBSD, and the lessons we learned from working with ACPI.

## 1 Introduction

Almost all modern computer system hardware allows its power usage to be managed and its temperature to be monitored and kept at the appropriate level. This allows users to achieve the best performance from a system for a given level of power usage. This is especially important for battery-driven mobile platforms where unnecessary use of power reduces battery life and thus reduces the amount of work that can be done before a recharge is necessary. It is also useful in desktop environments where devices such as computer monitors and disk drives can be powered down during idle times to reduce energy consumption.

Unfortunately, in the typical PC most of the software that configures and manages the power and thermal environment within the computer is locked up within the BIOS. In addition, unless a power management device is attached to a PnP bus (e.g. PCI), the operating system has no easy way to detect, configure, or manage it. For example, mechanisms such as ISA PnP are usually used to enumerate add-on ISA cards rather than for on-board devices. Systems like PnP BIOS can be used to enumerate on-board devices, but it is hard to extend in a generic way. Also, PnP BIOS is written in 16-bit code, so the operating system must use 16-bit emulation in order to call PnP BIOS functions.

Prior to the introduction of the Advanced Configuration and Power Management Interface (ACPI), the Advanced Power Management (APM) BIOS was commonly used for power management. In APM, the bulk of the power management control and logic resides within the APM BIOS code itself. APM-aware operating systems communicate with the APM BIOS through a fixed BIOS API which provides basic access to BIOS functions. APM-aware operating systems must periodically poll APM for APM-related events that must be processed. The APM BIOS may also make use of special system management interrupts which are invisible to the operating system itself. APM provides four states: run, suspend, sleep and soft-off.

APM has three main limitations. First, without special vendor programs, many APM features are only available through vendor-specific BIOS menus before the operating system is loaded. For example, the amount of console idle time required before powering down the video display is usually configured this way. Also, with APM, the number of power management configurations are fixed by the BIOS vendor. For example, the APM BIOS may always slow the CPU clock or power down other devices (e.g. networking card) when powering down the monitor. Since this is under control of the BIOS, there is no way to change the policy without changing the BIOS.

Second, APM is BIOS-level code that operates outside of the scope of the operating system. This make developing and debugging APM code a challenge. It also means that users can only fix bugs in their APM BIOS by flashing a new one into ROM. Flashing a new BIOS is a dangerous operation because if the BIOS fails, the system may well become useless.

Third, as APM is vendor-specific, efforts to develop and maintain the complex APM BIOS code are duplicated across the vendors that use it. This is wasteful.

ACPI addresses the limitations of APM and other configuration mechanisms by unifying all device management within the operating system kernel rather than having BIOS code make most of the decisions. Thus, ACPI is said to allow "operating system directed" power and thermal management that is more flexible than other mechanisms. The cost of this flexibility is the extra complexity required in the kernel to support ACPI. In this paper we describe ACPI, how it is implemented in FreeBSD, and the lessons we learned from working with ACPI. In Section 2 we describe the general architecture of ACPI. Section 3 explains the architecture of the FreeBSD ACPI implementation and its impact on the rest of the kernel. Section 4 contains related work, and Section 5 has our conclusions and future work.

## 2 ACPI Architecture

The ACPI standard [1] was developed by Intel, Toshiba, and Microsoft and has been adopted by most PC manufacturers. Figure 1 shows the main components of a system that uses ACPI. At the lower level, the ACPI standard defines a set of tables that describe the hardware platform, a BIOS API for low-level management operations, and a pre-defined set of registers. In the upper level, the operating system contains some core software and drivers used to communicate with APCI in addition to the usual device framework and drivers used to manage non-ACPI devices. In this section we examine both levels of the ACPI architecture.

### 2.1 ACPI Specified Components

The ACPI Specification defines three types of components:

**ACPI tables:** The ACPI tables are the central data structure of an ACPI-based system. They contain definition blocks that describe all the hardware that can be managed through ACPI. These definition

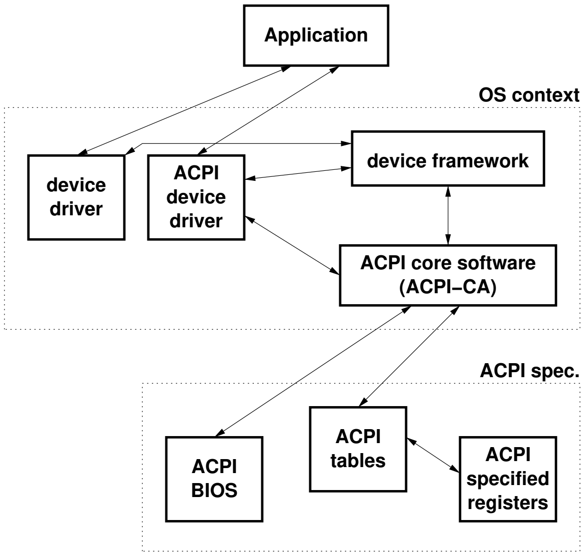


Figure 1: ACPI Architecture

blocks include both data and machine-independent byte-code that is used to perform hardware management operations.

**ACPI BIOS:** The ACPI BIOS is a small BIOS that performs basic low-level management operations on the hardware. These operations include code to help boot the system and to put the system to sleep or wake it up. Note that the ACPI BIOS is much smaller than an APM BIOS because most of the management functions have moved into the operating system and ACPI tables.

**ACPI registers:** The ACPI registers are a set of hardware management registers defined by the ACPI specification. The address of these registers is located through definition blocks in the ACPI tables. Note that hardware designers may provide additional management registers beyond the ones defined in the ACPI specification. These additional registers can be located and accessed through the byte-code stored in the device-specific part of the ACPI tables.

When an ACPI-based system is powered up, before the operating system is loaded the ACPI BIOS places the initial ACPI tables in memory. Since the ACPI tables are typically too large to put in the 128KB BIOS memory area, the ACPI BIOS obtains a physical memory map of the system in order to allocate space for the ACPI tables. When an ACPI-aware operating system kernel is started, it search for a small data structure within the BIOS memory area. If a valid structure is found (e.g. if its checksum and signature match) then the kernel uses this structure to obtain a pointer to the ACPI tables and memory map. This information is used by the kernel to preserve the ACPI tables when the virtual memory system is started.

The definition blocks within the ACPI tables are stored in a hierarchical tree-based name space. Each node in the tree is named. Node names consist of four capital alphanumeric characters and underscores (e.g. "`FOO_`," or "`_CRS`"). Namespace components are separated by periods, and the root of the namespace is denoted with a backslash ("\"). Names without a leading backslash are considered to be relative to the current scope in the name space. Node names that begin with an underscore are reserved by the ACPI specification for describing features. For example, nodes in the `\_SB` namespace refer to busses and devices attached to the main system bus, nodes in the `\_TZ` namespace relate to thermal management, and nodes in `\_GPE` are associated with general purpose ACPI events.

Except for the few operations performed by the ACPI BIOS, almost all ACPI operations are performed in the operating system context by interpreting machine-independent ACPI Machine Language (AML) byte-code stored in the ACPI tables. These blocks of AML are called methods. AML methods are stored in specially named nodes in the ACPI namespace. For example, the name `_PS0` is reserved for storing AML methods that evaluate a device's power requirements in the "D0" state (device fully on). Thus the node `\_SB.PCI0.CRD0._PS0` contains an AML `_PS0` method for the CRD0 device on the system's PCI0 bus.

AML is usually compiled from human-readable ACPI Source Language (ASL). Figure 2 shows an example block of ASL code for thermal management that defines four named data elements and two methods. The "Scope" operator defines what part of the ACPI namespace the contained block of code resides in. The "ThermalZone" operator defines a object representing a region of thermal control. The "Device" operator defines a device object, and the "PowerSource" operator defines a power switch object. The "OperationRegion" and "Field" operators are used to define blocks of registers and fields within them, respectively. The "Name" and "Method" operators define data and program elements belonging to their parent objects. For example, the first "Name" in the figure defines `\_TZ.TMZN._AC0` (the fan high-speed threshold) to be the integer 3272, which means 327.2 K. The "`_ON`" method defined in the figure contains code to turn the fan on. A graphical representation of the namespace defined in Figure 2 is shown in Figure 3.

```
Scope(\_TZ){
    ThermalZone(TMZN){
        Name(_AC0, 3272)
        Name(_AL0, Package{FAN})
        ....
    }
    Device(FAN){
        Name(_HID, 0xb00cd041)
        Name(_PR0, Package{PFAN})
    }
    OperationRegion(FANR,SystemIO,
                        0x8000, 0x10)
    Field(FANR, ByteAcc, NoLock,
                             Preserve){
        FCTL, 8
    }
    PowerSource(PFAN, 0, 0){
        Method(_ON){
            Store(0x4,FCTL)
        }
        Method(_OFF){
            Store(0x0,FCTL)
        }
    }
}
```

Figure 2: Example ASL Code

In the next three subsections we describe ACPI's configuration, power management, and thermal management subsystems.

### 2.1.1 Configuration

The ACPI namespace contains a tree of devices attached to the system. ACPI-aware operating systems walk this tree to enumerate devices and gain access to the device's data and control methods. In the ACPI namespace, an ACPI device description consists of a device object node and its children. Common children of device nodes include:

**`_ADR`:** a bus-specific address. For example, this could be a PCI device and function number.

**`_HID`:** the EISA ID of an on-board device. This is used to identify the device.

**`_UID`:** the unit number of an on-board device. This is used to distinguish between same kind of device.

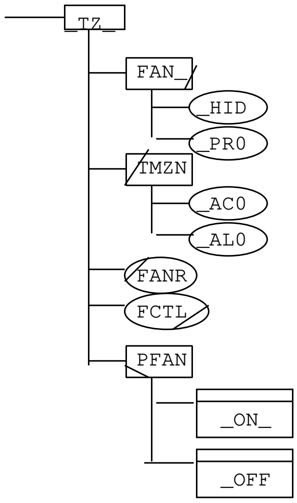


Figure 3: Example ACPI Namespace

**_CRS:** the current resource settings. This method produces a byte stream that is similar to a PnP resource encoding.

**_PRS:** the possible resource settings. This method also produces a byte stream similar to a PnP resource encoding.

**_SRS:** set resource settings. This method takes a PnP-encoded byte stream and uses it to set the device's configuration.

Note that the operating system accesses device data and methods through the ACPI-CA software layer (described in Section 2.2).

#### 2.1.2 ACPI Power Management

Hardware power management events trigger an OS-visible interrupt called a "system control interrupt" (SCI). Operating systems handle simple SCI interrupts (e.g. fixed-feature power button state change) directly. Complex SCI interrupts are are handled by the OS using AML code associated with the interrupt. For example, consider what happens when a "sleep" SCI interrupt occurs. The kernel must first save hardware state. The kernel then calls the `\_PTS` (prepare to sleep) method. Finally, it puts the system to sleep by writing the appropriate value to an ACPI register.

In ACPI there are six power states: S0, S1, S2, S3, S4, and S5. These states are defined as follows:

**S0:** the run state. In this state, the machine is fully running.

**S1:** the suspend state. In this state, the CPU will suspend activity but retain its contexts.

**S2 and S3:** sleep states. In these states, memory contexts are held but CPU contexts are lost. The differences between S2 and S3 are in CPU re-initialization done by firmware and device re-initialization.

**S4:** a sleep state in which contexts are saved to disk. The context will be restored upon the return to S0. This is identical to soft-off for hardware. This state can be implemented by either OS or firmware.

**S5:** the soft-off state. All activity will stop and all contexts are lost.

In addition to managing transitions between system power states, ACPI can also manage the power state of individual devices to a fine-grained level. For example, if two devices share the same power line, that information can be encoded in the ACPI tables in such a way that the power line is active only if one or both of the devices are in use.

#### 2.1.3 Thermal Management

ACPI supports operating system directed thermal management. Prior to ACPI there was no unified interface for thermal management (e.g. APM did not support it). On some systems the platform specific BIOS code handles thermal management, but this is invisible to the operating system and few OS developers noticed it. Thermal management is becoming more important important as system efficiency increases and the system gets hotter when working.

The ACPI thermal management subsystem provides a way to get the current temperature, and it provides hints to control thermal policy. The thermal policy information includes information on how to get the system cooler and at what point the cooling method should be invoked. There are two ways to cool the system down: active cooling, which activates cooling devices such as fans, and passive cooling, in which the CPU operation slows down to decrease heat generation.

Thermal management is controlled in the ThermalZone section of the AML namespace. Note that in ACPI, all temperatures are in tenths of degrees kelvin. Important sections of the thermal management part of the ACPI namespace include:

**_TMP:** gets the current temperature.

**_ACx:** the temperature at which the system should switch to active cooling mode "x."

**_ALx:** a list of objects that should be active when the system is in cooling mode "x."

**_CRT:** the temperature at which we should halt the entire system.

**_PSV:** the temperature at which the system should switch to passive cooling mode.

**_PSL:** a list of objects (typically the CPU) that should be slowed in passive cooling mode.

**_SCP:** a method that allows the operating system to set the cooling policy.

**_TCx:** a parameter for passive cooling mode "x."

Note that in the event of a thermal emergency (e.g. a bug in ACPI software), ACPI allows the hardware to take over thermal management in order to protect the hardware from damage.

### 2.2 ACPI Component Architecture (ACPI-CA)

An ACPI-aware operating system must include code that accesses the ACPI BIOS, registers, and tables. It must also include an AML byte-code interpreter. This upper layer of core ACPI software is shown in Figure 1. Intel has implemented an OS-independent implementation of this layer of software called ACPI Component Architecture or ACPI-CA [2]. ACPI-CA is used by many open source operating systems including FreeBSD and Linux.

ACPI-CA provides a high-level ACPI API to the operating system. The OS uses this API to implement power management, device configuration, and thermal management. All fixed features and some access to AML names are wrapped by the exported function. But some ACPI-specific devices have to access ACPI namespace. The ACPI-CA API is shown in Table 1.

Operating systems that use ACPI-CA must provide it with some basic low-level functions. Intel has implemented the low-level part for Linux. A list of these functions is shown in Table 2.

One of the main user-visible differences between FreeBSD ACPI and Linux ACPI is the user interface. FreeBSD uses sysctl to export ACPI-related kernel variables, while Linux uses the procfs `/proc` filesystem to export them.

## 3 Design of FreeBSD ACPI

In this section we describe how we made the FreeBSD kernel ACPI-aware. We first implemented our own version of the ACPI core software (we later switched to ACPI-CA). We then addressed the issues of ACPI device enumeration, supporting ACPI sleep modes, and ACPI thermal management.

### 3.1 Our ACPI Core Software Implementation

In September 1999 we started writing our own ACPI core software implementation, including an AML execution environment. The implementation was based on Doug Rabson's ACPI disassembler and our ACPI data analyzing tool.

We first wrote a ACPI memory recognition routine to detect and preserve the ACPI tables. We then wrote a process that could run AML methods manually (e.g. suspend and wakeup) based on somewhat incomplete ASL output. This allowed us to enter power state S1 and also to shutdown a machine by pushing the power button.

We also wrote an AML interpreter in user space by merging the namespace functions from our analyzing tool into the ACPI disassembler and adding a memory management module to it. After this was implemented we merged the AML interpreter module into a kernel driver and then we had a basic working version of power management.

While we were working out the bugs in our in-kernel AML interpreter, we noticed that ACPI-CA software from Intel had a suitable license to merge into FreeBSD. As we were preparing to merge our ACPI into the main branch of the FreeBSD source repository we read the ACPI-CA implementation. We then decided to switch to ACPI-CA using glue code that we wrote. The reason we switched was that ACPI-CA is an OS-independent implementation so we can share and benefit from feedback from other groups. While the ACPI-CA implementation is larger, it is also more complete and well documented.

| Function Class | Functions |
|---|---|
| ACPI subsystem management | AcpiInitializeSubsystem, AcpiEnableSubsystem, AcpiTerminate, AcpiSubsystemStatus, AcpiDisable, AcpiGetSystemInfo, AcpiFormatException, AcpiPurgeCachedObjects |
| memory management | AcpiAllocate, AcpiFree |
| ACPI table management | AcpiFindRootPointer, AcpiLoadTables, AcpiLoadTable, AcpiUnloadTable, AcpiGetTableHeader, AcpiGetTable AcpiGetFirmwareTable, |
| namespace interface | AcpiWalkNamespace, AcpiGetDevices, AcpiGetName, AcpiGetHandle, AcpiAttachData, AcpiDetachData, AcpiGetData |
| object manipulation | AcpiEvaluateObject, AcpiGetObjectInfo, AcpiGetNextObject, AcpiGetType, AcpiGetParent |
| event handler interface | AcpiInstallFixedEventHandler, AcpiRemoveFixedEventHandler, AcpiInstallNotifyHandler, AcpiRemoveNotifyHandler, AcpiInstallAddressSpaceHandler, AcpiRemoveAddressSpaceHandler, AcpiInstallGpeHandler, AcpiAcquireGlobalLock, AcpiReleaseGlobalLock, AcpiRemoveGpeHandler, AcpiEnableEvent, AcpiDisableEvent, AcpiClearEvent, AcpiGetEventStatus, |
| resource interfaces | AcpiGetCurrentResources, AcpiGetPossibleResources, AcpiSetCurrentResources, AcpiGetIrqRoutingTable, |
| hardware interface | AcpiSetFirmwareWakingVector, AcpiGetFirmwareWakingVector, AcpiEnterSleepStatePrep, AcpiEnterSleepState, AcpiLeaveSleepState |

Table 1: ACPI-CA API

So our implementation is no longer in the kernel, but it still remains in user-level tools such as `amldb(8)` and `acpidump(8)`.

### 3.2 FreeBSD ACPI Device Enumeration

FreeBSD uses a configuration system known as "newbus." [5] In this system, an opaque "device_t" type object represents each bus/device. There are functions to manipulate device_t objects. These functions (e.g. probe, attach, allocate resources) are device specific. They are invoked through a function table that acts as an associative array. The array key is called a method. This is the same technique used in the BSD virtual filesystem layer (VFS). Each device has a parent device, except for the root device. Each device_t object has two device-specific structures: "ivar" and "softc."

The softc structure is a device-specific structure used to store a device's state. Each driver determines the size of its own softc structure, and the new-bus framework allocates memory for the softc structures as devices are configured.

The "ivar" structure is used by a parent device to manage its children. This variable should not be accessed from the child device directly, but via the parent device method. The child device uses this method to obtain bus-specific values and resources. An example of this method is BUS_READ—WRITE_IVAR, which is used to access a bus-specific value. This function is usually wrapped by macro-definition in a bus-specific header file.

PnP devices are processed by a "pnp" or "pnpbios" driver that provides only the DEVICE_IDENTIFY method. The DEVICE_IDENTIFY method is usually called from a parent bus's routine, after the bus was probed and before the actual attach process, via the bus_generic_probe function. This method records the device's logical id in the ISA bus-specific ivars structure. This value is then used when the isa_get_logicalid macro is invoked. This macro calls the BUS_READ_IVAR bus method to get the ID.

In FreeBSD ACPI, all device objects, thermal zone objects, and some other fixed features are added as child devices in the acpi bus attach code. This is done with the "device_add_child" function. Once an ACPI child device is added, this function sets ACPI object handles in the ivars. The probe routine check to see whether the `_HID` of the device will match the driver. If it does, then the attach routine calls the resource parser to get the resources the driver will use.

The Host-PCI bus bridge appears in the ACPI namespace and is treated as an ACPI-specific device. The driver will call the machine-dependent PCI bridge func-

| Function Class | Functions |
|---|---|
| library initialization | AcpiOsInitialize, AcpiOsTerminate |
| semaphore control | AcpiOsCreateSemaphore, AcpiOsDeleteSemaphore, AcpiOsWaitSemaphore, AcpiOsSignalSemaphore |
| memory allocation | AcpiOsAllocate, AcpiOsCallocate, AcpiOsFree, AcpiOsMapMemory, AcpiOsUnmapMemory, AcpiOsGetPhysicalAddress |
| interrupt control | AcpiOsInstallInterruptHandler, AcpiOsRemoveInterruptHandler |
| process control | AcpiOsGetThreadId, AcpiOsQueueForExecution, AcpiOsSleep, AcpiOsStall |
| device access | AcpiOsReadPort, AcpiOsWritePort, AcpiOsReadMemory, AcpiOsWriteMemory, AcpiOsReadPciConfiguration, AcpiOsWritePciConfiguration, AcpiOsReadable, AcpiOsWritable |
| signal/timer control | AcpiOsGetTimer, AcpiOsSignal |
| diagnostic functions | AcpiOsPrintf, AcpiOsVprintf, AcpiOsGetLine, AcpiOsDbgAssert |

Table 2: OS functions provided to ACPI-CA

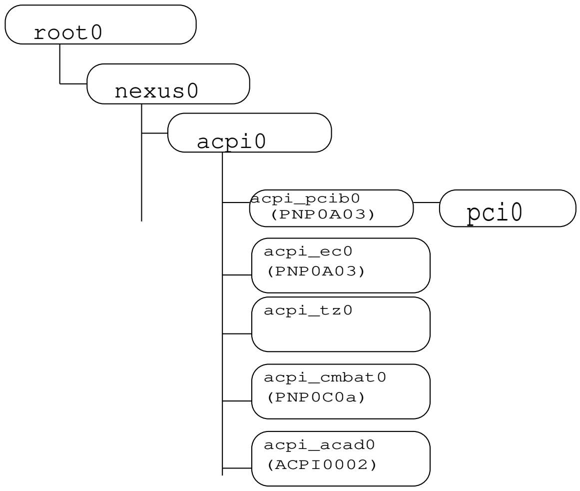


Figure 4: FreeBSD device tree corresponding to the ACPI name space

tion to implement a bus-bridge function and manage PCI interrupt routing with ACPI. In ACPI, PCI routing information is in a _PRT object. A _PRT object contains an array of structured objects, each of which has information about the slot, pin, and interrupt source. The interrupt source points to the device object of the interrupt router and assigns the interrupt resource to the device route interrupt to the pin. The interrupt router device object is not currently treated as a FreeBSD new-bus device. The ACPI device driver mimics some ISA-like behaviors. This responds to the isa_get_logicalid function and answers the device logical id value, using _HID object. Figure 4 shows device tree constructed from ACPI name space.

Prior to September 2001 a different approach was taken. The earlier approach was closer to a PnP driver. The "acpi_isa" driver was used for this. Like a PnP driver, the "acpi_isa" driver provides only the DEVICE_IDENTIFY method.

### 3.3 FreeBSD ACPI Power States

In this section we describe how FreeBSD implements ACPI power states. We start with some background of the i386 architecture before going into the details of the FreeBSD implementation.

#### 3.3.1 I386 Background

To understand ACPI power states, it is important to be familiar with i386 CPU architecture and CPU initialization. All i386 CPUs support "real mode." Real mode is a CPU state that is compatible with the old i8086 processor. In this mode each memory access pointer, including the instruction pointer, is 16 bits long. This provides a 64KB address space. Segment registers are used to extend the memory address space that can be referenced and also to separate the code, data, and stack areas. To convert a segment-based address into a physical address, take the segment register value and shift it left four bits and add it to the address. The extra four bits from the segment register allow us to reference up to 1MB of physical memory. There are six segment registers: CS, DS, SS, ES, FS and GS. The code segment register (CS) is used when accessing instructions through the instruction pointer. When accessing data memory, the data segment register (DS) is used by default. Stack operations use the stack segment register (SS).

When booting, the BIOS firmware passes program control to software in real mode, as described above. Most modern operating systems do not operate in real mode. Instead, they change the working mode from real mode to "protected mode." To switch to the protected mode, the kernel must set up the GDTR register and update control register CR0. The GDTR register points to the global descriptor table (GDT). The GDT is used for address translation in protected mode.

In protected mode, segment registers and some special registers, such as TR (Task register), point to an entry in the GDT. This entry is then used when translating addresses. The current mode of the CPU is determined by the mode-select flag in the CR0 register. Once the system is in protected mode, kernels that wish to use paging for virtual memory must enable it. To enable paging, the kernel has to set the CR3 register to point to the page table structures to use and then change the mode flag in the CR0 register.

### 3.3.2 FreeBSD ACPI Sleeping States

The S1 state is implemented simply by calling an ACPI-CA function after sending the device-sleeping request to the device driver. This ACPI-CA function uses the ACPI registers to stop the system. When the system wakes up from S1 sleep, it can immediately resume processing from where it was just before the sleep request was made.

The S2 and S3 states do not preserve the CPU context. So the FreeBSD kernel is required to do a CPU context save is required before entering the sleep state.

Since entering and exiting the S3 sleep state requires the use of real-mode, the kernel must place a "resume handler" somewhere in the first 1MB of memory so that it can be addressed from real mode. We use a perl script along with objcopy, hexdump, and nm to generate code. With this script, a real-mode executable object is turned into a header file with an array of chars and some definitions that point to the offset of the symbol in the object file. When the system boots the memory for the resume handler is allocated in the first 1MB as early as possible using a special memory allocator. The ACPI driver attach routine then copies and links in the resume handler code to the newly allocated memory area. Finally, the physical address of the resume handler is recorded in the ACPI driver software context.

When S3 is invoked, the physical address of the resume

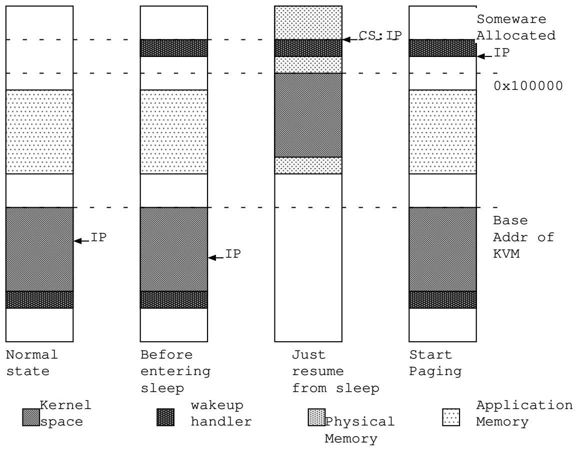


Figure 5: Memory mapping transition

handler is installed in the ACPI firmware context via an ACPI-CA function. The memory mapping table of the current process is used to create an identity mapping where a process' virtual address maps directly to the hardware's physical address. This mapping is needed to restart paging when the system resumes. Next, the sleeping procedure saves all the registers. Special registers (segment selectors, GDTR, TR, LDTR, IDTR, CR2, CR3, CR4) are put into the appropriate places so they can be restored later by the real mode wake-up code. Other registers are saved in static variables in the kernel data area.

When the machine wakes up the firmware passes control, in real mode, to the resume handler. This is done by placing the top 16 bits of the physical address of the resume handler in the code segment register and placing the lower 4 bits in the instruction pointer. The handler sets the other segment registers to the same value as code segment registers. The CPU state can then be switched to protected mode by changing CR0. At this point the special registers are restored. Note that before restoring the task register, the task selector entry in GDT should be fixed up so that it does not become marked as a busy task selector.

After the special registers have been restored, then paging can be reenabled by setting the appropriate bit in CR0. After the paging has started, the instruction pointer is still pointing to an identity-mapped page. Next, the handler passes the control back to its calling routine which resides in normal kernel virtual memory. Finally, the stack pointer and remaining preserved registers are restored. Figure 5 shows memory mapping transition.

After returning to the sleep code, hardware devices should be restored. First, the interrupt controller must be reprogrammed because some device drivers and ACPI itself depend on functioning interrupts. To do this, we split the isa_defaultirq()@intr_machdep.c function into two. Now isa_defaultirq() installs a stray interrupt handler, and then calls init_i8259 to initialize the interrupt controller. We call the resumption function icu_reinit()@intr_machdep, and then we call the routine init_i8259(). We then check the interrupt handler and enable the interrupt if the handler is not a stray interrupt handler. After this call, we use the routine call DEVICE_RESUME method of root_bus to request that the device drivers resume the device, after which normal operation resumes.

### 3.4 FreeBSD ACPI Thermal Management

The thermal zone is represented as a device in FreeBSD. The FreeBSD thermal zone device driver exports thermal information using the sysctl interface. Passive cooling is not yet implemented, though CPU throttling is implemented. The device driver checks the thermal zone when a tunable polling time expires and also when thermal zone Notify AML opcode are evaluated. Notify opcodes are typically found in the general purpose interrupt handlers or in the query handler for ACPI-capable embedded controllers. ACPI routines check the thermal zone by fetching the temperature using the thermal zone `_TMP` method and comparing its value to the `_ACx` value. If the `_ACx` value is smaller than current temperature, then we change thermal mode to the largest x value. If the new thermal state value is larger than old one, we activate the device object listed in the `_ALx` object where x is the new state number, otherwise we deactivate them.

## 4 Related Work

In this section we compare FreeBSD ACPI to Linux ACPI and we examine the relationship between Open Firmware and ACPI.

### 4.1 Comparison with Linux ACPI

As both FreeBSD and Linux use Intel ACPA-CA, the basic architecture of their ACPI subsystems is similar. The major differences are in user interface and bus enumeration. For user interface, FreeBSD uses sysctls, which are variables that kernel exports. Linux uses the "procfs" filesystem. While both system have procfs, FreeBSD uses procfs purely for query process information. The sysctl interface is interface provides a tree of for kernel tunable variable. We export some ACPI information such as temperature temperature to userspace via the sysctl interface.

The ACPI-CA code is currently distributed with a user-space ACPI interpreter which is the counterpart of FreeBSD's `amldb(8)` and ASL assembler. But ACPI-CA has no disassembler tool (like FreeBSD's `acpidump(8))` that produces ASL compatible with the ACPI-CA ASL assembler.

In Linux, there were no generalized ways to create device trees, so Intel used a "bus manager" mechanism to recognize ACPI-specific devices. The bus manager abstraction was introduced when the ACPI-CA was developed and build under WIN32. We decided not to use it early in our development process because it collides with the FreeBSD driver recognition mechanism. But the Intel Linux-ACPI team recognized the necessity of the unified mechanism to configure devices (including non-ACPI ones), so they proposed a mechanism called "Linux Driver Model." This mechanism will be introduced in next major version of the Linux kernel (2.5).

FreeBSD uses a three stage boot loader. Currently the FreeBSD boot loader detects ACPI by scanning the BIOS memory and loads the acpi kernel module automatically if it is needed. Linux use initrd mechanism to do early configuration. Initrd is special memory filesystem loaded by the boot loader. This file system is mounted as root for initial configuration such as module loading. After the configuration is finished, the file system is unmounted or moved to another mount point.

### 4.2 Comparison with Open Firmware

Open Firmware was originally developed by Sun Microsystems and is now standardized as IEEE 1275-1994 [6]. It is currently used in the SPARC architecture, the PowerPC architecture, and the ARM architecture. Open Firmware acts as monitor that can interpret the Forth language. It covers the boot process, device configuration, and power management. Operation systems running on Open Firmware based machines must implement some architecture-dependent way to access the firmware interpreter for use in auto-configuration.

ACPI and Open Firmware are similar in that some functions of the firmware are written for an interpretor. This makes it easier to extend without breaking compatibility. But the most important difference is that ACPI byte code is interpreted by the operating system, while Open Firmware Forth code is interpreted by the firmware itself. The Open Firmware solution provides a powerful framework to describe and extend functionally, but this is not accepted in Intel architecture (IA32/IA64) for the following reasons:

- Firmware space is limited for compatibility reasons.
- Firmware cannot figure out all states in the devices, which is especially needed to implement suspend state.
- Calling firmware from 32 bit code is somewhat unstable in Intel architecture. There is no compatible way to setup without calling 16bit code. And there is no way to notify event other than polling.

# 5 Conclusion and Future Work

In this paper we have described ACPI, how it is implemented in FreeBSD, and the lessons we learned from working with ACPI. We believe that ACPI support will become more important as new devices with demanding configuration, power, and thermal management needs become more widespread. Although we have a basic working ACPI environment under FreeBSD there is still lots of work left to do.

## 5.1 Device Enumeration Enhancement

ACPI has a hot-plugging feature, and the current PCI interrupt routing code is not capable of routing interrupts for devices that are on a PCI-PCI bridge. This will require large modification to the current device enumeration scheme in ACPI. We have proposed a scheme designed so that existing drivers are not modified unless absolutely necessary. The scheme is:

1. Add a bus bridge enumerator driver (only have device_identify bus method) for each bus bridge device that can be a descendant of ACPI and appear as a namespace. In this method, add children to the bus and register acpi_name-device_t table in the acpi driver. Then evaluate `_INI` object after checking by `_STA`. Then the driver install address space handler, etc.
2. The manipulation to get ACPI_HANDLE, etc., is not done via DEVMETHOD but by a direct function call. This may require module dependency with the acpi driver, but if the driver wants to use ACPI_HANDLE, it must depend on acpi. If a device other than ACPI is used, it may not use ACPI_HANDLE to get information.
3. Add acpi attachment for devices that can be attached to acpi directly. The acpi-pcib driver is quite a different implement than the nexus-pcib driver.

## 5.2 User-land Interface

There are already some user-land interfaces in the ACPI driver. The interface is provided in two ways: /dev/acpi ioctl's and hw.acpi sysctls. Currently provided interfaces are:

- Battery charge information.
- Thermal zone information.
- CPU speed configuration.

There is a piece missing relating to the ACPI event notification system. We plan to use kqueue [7] to implement the event notification system.

## 5.3 S4 Implementation

The S4 sleep state is also known as “Suspend to Disk.” ACPI S4 implementation is achieved in two ways. The first way is called S4 BIOS: Firmware saves the running state to disk, and the operating system should do the same thing as S2/S3, then issue a suspend request to BIOS via the special port that triggers a system management interrupt. The second way to handle S4 is to have the operating system handle the saving of state to disk. In this scheme, the operating system should preserve all memory contents, device contexts, and the CPU context. FreeBSD includes a crash dump mechanism that we believe can be adapted for use when implementing OS-initiated S4 sleep.

## Acknowledgments

IWASAKI Mitsuru provided great help not only in the actual implementation but also in project management. Without his help, this project would have remained only a personal project and been forgotten without prominent achievement.

Michael Smith contributed to our experimental implementation and did great work on ACPI-CA migration. Most of the FreeBSD dependent parts of the ACPI-CA code are his work.

Andrew Grover and his Intel associates helped in the porting work. Of course, we greatly respect their work on ACPI-CA itself and appreciate their kindly allowing us to use their work.

Doug Rabson wrote the ACPI disassembler, on which our experimental ACPI implementation was based.

Yasuo Yokoyama, Munehiro Matsuda and all acpi-jp mailing list people have contributed to our experimental implementation.

The ACPI4Linux mailing list gave me useful information about ACPI and ACPI-CA.

Warner Losh and Chuck Silvers gave suggestions to improve the expression of English in this manuscript.

Chuck Cranor shepherded me for the USENIX Freenix track.

## References

[1] Advanced Configuration and Power Management Interface, `http://acpi.info/`

[2] ACPI Component Architecture, Intel Corp., `http://developer.intel.com/technology/iapc/acpi/`

[3] Takanori Watanabe, "ACPI in general and implementation of its driver," Personal Unix / FreeBSD Press, `http://www.ux.mycom.co.jp/` (in Japanese)

[4] Intel Corp., "Intel Architecture Software Developers Manual," `http://developer.intel.com/`

[5] Murray Stosky et al., "FreeBSD Developer's Handbook," `http://www.freebsd.org/`

[6] Sun Microsystems, "IEEE Std 1275-1994. Open Firmware Core Specification." `http://playground.sun.com/1275/`

[7] Jonathan Lemon, "Kqueue-A Generic and Scalable Event Notification Facility," FREENIX Proceedings, 2001.