Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

ACPI in Linux

Architecture, Advances, and Challenges

Len Brown Anil Keshavamurthy David Shaohua Li

Robert Moore Venkatesh Pallipadi Luming Yu

Intel Open Source Technology Center

{len.brown, anil.s.keshavamurthy, shaohua.li}@intel.com

{robert.moore, venkatesh.pallipadi, luming.yu}@intel.com

Abstract

ACPI (Advanced Configuration and Power Interface) is an open industry specification establishing industry-standard interfaces for OS-directed configuration and power management on laptops, desktops, and servers.

ACPI enables new power management technology to evolve independently in operating systems and hardware while ensuring that they continue to work together.

This paper starts with an overview of the ACPICA architecture. Next a section describes the implementation architecture in Linux.

Later sections detail recent advances and current challenges in Linux/ACPI processor power management, CPU and memory hot-plug, legacy plug-and-play configuration, and hot-keys.

1 ACPI Component Architecture

The purpose of ACPICA, 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 OS Services Layer (OSL) between the ACPI subsystem and the host operating system.

The ACPICA source code is dual-licensed such that Linux can share it with other operating systems, such as FreeBSD.

1.1 ACPICA Overview

ACPICA 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 code modifications within the CA code

itself. The components of the architecture include (from the top down):

- A user interface to the power management and configuration features.
- A power management and power policy component (OSPM). [1]
- A configuration management component.
- ACPI-related device drivers (for example, drivers for the Embedded Controller, SMBus, Smart Battery, and Control Method Battery).
- An ACPI Core Subsystem component that provides the fundamental ACPI services (such as the AML[2] interpreter and namespace[3] management).
- An OS Services Layer for each host operating system.

1.2 The ACPI Subsystem

The ACPI Subsystem implements the low level or fundamental aspects of the ACPI specification. It includes 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, the OSL translates OS requests into the native calls provided by the host operating system.

[1]OSPM, Operating System directed Power Management.

[2]AML, ACPI Machine Language exported by the BIOS in ACPI tables, interpreted by the OS.

[3]The ACPI namespace tracks devices, objects, and methods accessed by the interpreter.

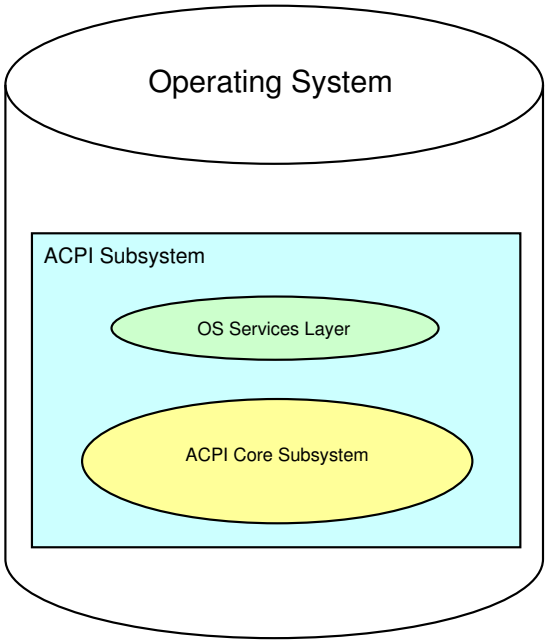

Figure 1: The ACPI Subsystem Architecture

The OS Services Layer is the only component of the ACPICA that contains code that is specific to a host operating system. Figure 1 illustrates the ACPI Subsystem is composed of the OSL and the Core.

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

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

The Core Subsystem implementation makes no assumptions about the host operating system or environment. The only way it can request

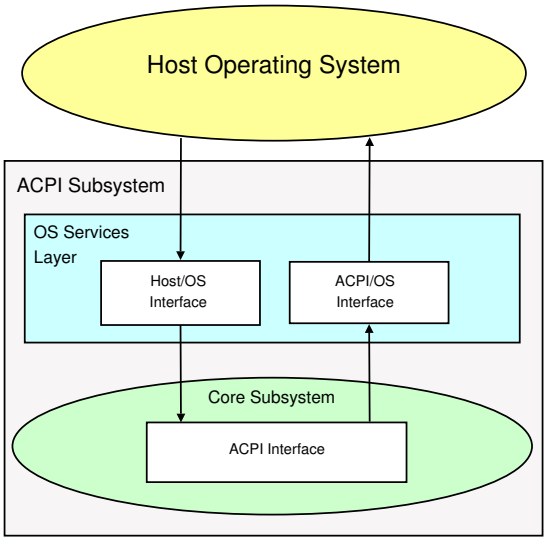

Figure 2: Interaction between the Architectural Components

operating system services is via interfaces provided by the OSL. Figure 2 shows that the OSL component "calls up" to the host operating system whenever operating system services are required, either for the OSL itself, or on behalf of the Core Subsystem component. All native calls directly to the host are confined to the OS Services Layer, allowing the core to remain OS independent.

1.3 ACPI Core Subsystem

The Core Subsystem is divided into several logical modules or sub-components. Each module implements a service or group of related services. This section describes each subcomponent and identifies the classes of external interfaces to the components, the mapping of these classes to the individual components, and the interface names. Figure 3 shows the internal modules of the ACPI Core Subsystem and their relationship to each other. The AML interpreter forms the foundation of the component, with additional services built upon this foundation.

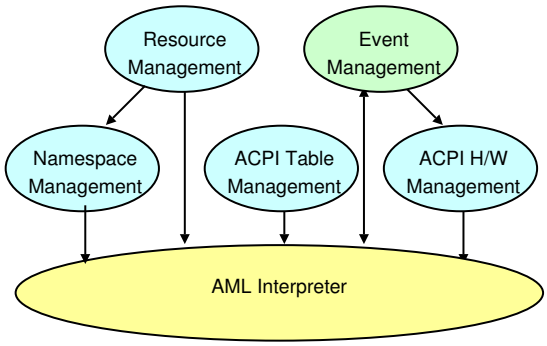

Figure 3: Internal Modules of the ACPI Core Subsystem

1.4 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. The services that the interpreter provides include:

- AML Control Method Execution
- Evaluation of Namespace Objects

1.5 ACPI Table Management

This component manages the ACPI tables. The tables may be loaded from the firmware or directly from a buffer provided by the host operating system. Services include:

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

1.6 Namespace Management

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

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

1.7 Resource Management

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

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

1.8 ACPI Hardware Management

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

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

1.9 Event Handling

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

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

2 ACPICA OS Services Layer (OSL)

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

The OS Services Layer has several roles.

1. It acts as the front-end for some OS-to-ACPI requests. It translates OS requests that are received in the native OS format (such as a system call interface, an I/O request/result segment interface, or a device driver interface) into calls to Core Subsystem interfaces.

2. It exposes a set of OS-specific application interfaces. These interfaces translate application requests to calls to the ACPI interfaces.

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

2.1 Functional Service Groups

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

Although it is the OS Services Layer that exposes these services to the rest of the operating system, it is very important to note that the OS Services Layer makes use of the services of the lower-level ACPI Core Subsystem to implement its services.

2.1.1 OS Boot-load-Time Services

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

2.1.2 Device Driver Load-Time Services

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

2.1.3 OS Run-Time Services

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

2.1.4 Asynchronous Services

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

2.2 OSL Required Functionality

There are three basic functions of the OS Services Layer:

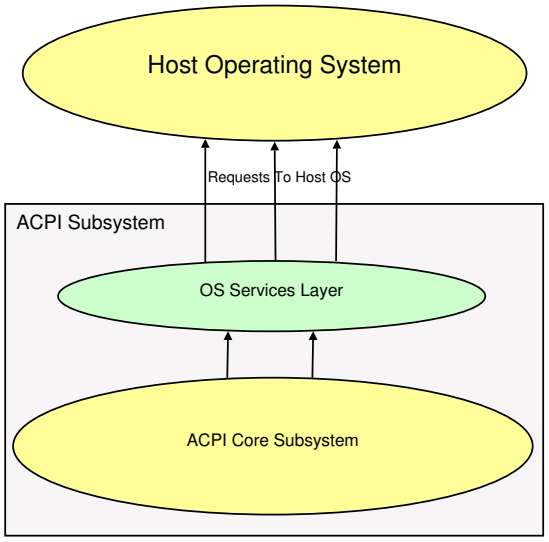

Figure 4: ACPI Subsystem to Operating System Request Flow

1. Manage the initialization of the entire ACPI subsystem, including both the OSL and ACPI Core Subsystems.

2. Translate requests for ACPI services from the host operating system (and its applications) into calls to the Core Subsystem component. This is not necessarily a one-to-one mapping. Very often, a single operating system request may be translated into many calls into the ACPI Core Subsystem.

3. Implement an interface layer that the Core Subsystem component uses to obtain operating system services. These standard interfaces (referred to in this document as the ACPI OS interfaces) include functions such as memory management and thread scheduling, and must be implemented using the available services of the host operating system.

2.2.1 Requests from ACPI Subsystem to OS

The ACPI subsystem requests OS services via the OSL shown in Figure 4. 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.

2.3 ACPICA—more details

The ACPICA APIs are documented in detail in the *ACPICA Component Architecture Programmer Reference* available on `http://www.intel.com`.

The ACPI header files in `linux/include/acpi/` can also be used as a reference, as can the ACPICA source code in the directories under `linux/drivers/acpi/`.

3 ACPI in Linux

The ACPI specification describes platform registers, ACPI tables, and operation of the ACPI BIOS. It also specifies AML (ACPI Machine Language), which the BIOS exports via ACPI tables to abstract the hardware. AML is executed by an interpreter in the ACPI OS.[4]

In some cases the ACPI specification describes the sequence of operations required by the

[4] ACPI OS: an ACPI-enabled OS, such as Linux.

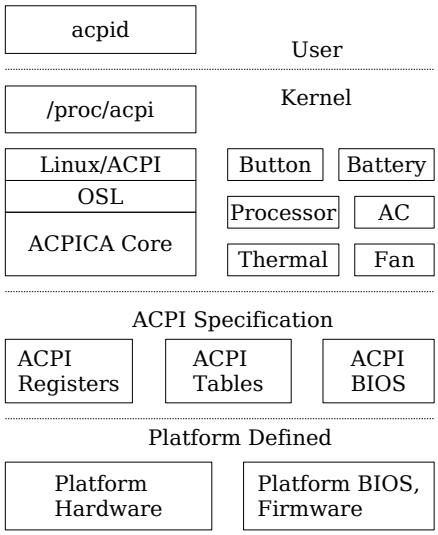

Figure 5: Implementation Architecture

ACPI OS—but generally the OS implementation is left as an exercise to the reader.

There is no platform ACPI compliance test to assure that platforms and platform BIOS' are compliant to the ACPI specification. System manufacturers assume compliance when available ACPI-enabled operating systems boot and function properly on their systems.

Figure 5 shows these ACPI components logically as a layer above the platform specific hardware and firmware.

The ACPI kernel support centers around the ACPICA core. ACPICA implements the AML interpreter as well as other OS-agnostic parts of the ACPI specification. The ACPICA code does not implement any policy, that is the realm of the Linux-specific code. A single file, `osl.c`, glues ACPICA to the Linux-specific functions it requires.

The box in Figure 5 labeled "Linux/ACPI" represents the Linux-specific ACPI code, including boot-time configuration.

Optional "ACPI drivers," such as Button, Battery, Processor, etc. are (optionally loadable) modules that implement policy related to those specific features and devices.

There are about 200 ACPI-related files in the Linux kernel source tree—about 130 of them are from ACPICA, and the rest are specific to Linux.

4 Processor Power management

Processor power management is a key ingredient in system power management. Managing processor speed and voltage based on utilization is effective in increasing battery life on laptops, reducing fan noise on desktops, and lowing power and cooling costs on servers. This section covers recent and upcoming Linux changes related to Processor Power Management.

4.1 Overview of Processor Power States

But first refer to Figure 6 for this overview of processor power management states.

1. G0—System Working State. Processor power management states have meaning only in the context of a running system—not when the system is in one of its various sleep or off-states.

2. Processor C-state: C0 is the executing CPU power state. C1–Cn are idle CPU power states used by the Linux idle loop; no instructions are executed in C1–Cn. The deeper the C-state, the more power is saved, but at the cost of higher latency to enter and exit the C-state.

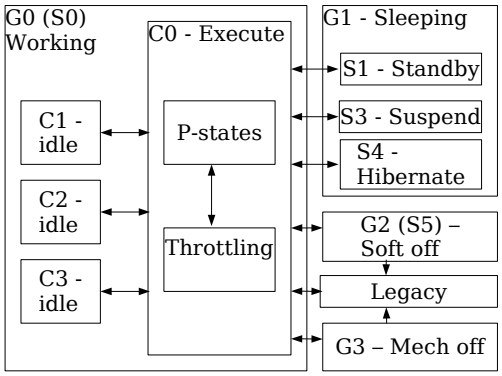

Figure 6: ACPI Global, CPU, and Sleep states

3. Processor P-state: Performance states consist of states representing different processor frequencies and voltages. This provides an opportunity to OS to dynamically change the CPU frequency to match the CPU workload.

   As power varies with the square of voltage, the voltage-lowing aspect of p-states is extremely effective at saving power.

4. Processor T-state: Throttle states change the processor frequency only, leaving the voltage unchanged.

   As power varies directly with frequency, T-states are less effective than P-states for saving processor power. On a system with both P-states and T-states, Linux uses T-states only for thermal (emergency) throttling.

4.2 Processor Power Saving Example

Table 1 illustrates that high-volume hardware offers dramatic power saving opportunities to the OS through these mechanisms.[5] Note that these numbers reflect processor power, and do not include other system components, such as the LCD chip-set, or disk drive. Note also that on this particular model, the savings in the C1, C2, and C3 states depend on the P-state the processor was running in when it became idle. This is because the P-states carry with them reduced voltage.

C-State	P-State	MHz	Volts	Watts
C0	P0	1600	1.484	24.5
	P1	1300	1.388	22
	P2	1100	1.180	12
	P3	600	0.956	6
C1, C2	from P0	0	1.484	7.3
	from P3	0	0.956	1.8
C3	from P0	0	1.484	5.1
	from P3	0	0.956	1.1
C4	(any)	0	0.748	0.55

Table 1: C-State and P-State Processor Power

4.3 Recent Changes

4.3.1 P-state driver

The Linux kernel cpufreq infrastructure has evolved a lot in past few years, becoming a modular interface which can connect various vendor specific CPU frequency changing drivers and CPU frequency governors which handle the policy part of CPU frequency changes. Recently different vendors have different technologies, that change the

[5] Ref: 1600MHz Pentium M processor Data-sheet.

CPU frequency and the CPU voltage, bringing with it much higher power savings than simple frequency changes used to bring before. This combined with reduced CPU frequency-changing latency (10uS–100uS) provides a opportunity for Linux to do more aggressive power savings by doing a frequent CPU frequency change and monitoring the CPU utilization closely.

The P-state feature which was common in laptops is now becoming common on servers as well. `acpi-cpufreq` and `speedstep-centrino` drivers have been changed to support SMP systems. These drivers can run with i386 and x86-64 kernel on processors supporting Enhanced Intel Speedstep Technology.

4.3.2 Ondemand governor

One of the major advantages that recent CPU frequency changing technologies (like Enhanced Intel SpeedStep Technology) brings is lower latency associated with P-state changes of the order of 10mS. In order to reap maximum benefit, Linux must perform more-frequent P-state transitions to match the current processor utilization. Doing frequent transitions with a user-level daemon will involve more kernel-to-user transitions, as well as a substantial amount of kernel-to-user data transfer. An in-kernel P-state governor, which dynamically monitors the CPU usage and makes P-state decisions based on that information, takes full advantage of low-latency P-state transitions. The ondemand policy governor is one such in-kernel P-state governor. The basic algorithm employed with the ondemand (as in Linux-2.6.11) governor is as follows:

```
Every X milliseconds
  Get the current CPU utilization
  If (utilization > UP_THRESHOLD %)
    Increase the P-state
    to the maximum frequency

Every Y milliseconds
  Get the current CPU utilization
  If (utilization < DOWN_THRESHOLD %)
    Decrease P-state
    to next available lower frequency
```

The ondemand governor, when supported by the kernel, will be listed in the `/sys` interface under `scaling_available_governors`. Users can start using the ondemand governor as the P-state policy governor by writing onto `scaling_governor`:

```
# cat scaling_available_governors
ondemand user-space performance
# echo ondemand > scaling_governor
# cat scaling_governor
ondemand
```

This sequence must be repeated on all the CPUs present in the system. Once this is done, the ondemand governor will take care of adjusting the CPU frequency automatically, based on the current CPU usage. CPU usage is based on the `idle_ticks` statistics. Note: On systems that do not support low latency P-state transitions, `scaling_governor` will not change to "ondemand" above. A single policy governor cannot satisfy all of the needs of applications in various usage scenarios, the ondemand governor supports a number of tuning parameters. More details about this can be found on Intel's web site.[6]

4.3.3 cpufreq stats

Another addition to cpufreq infrastructure is the cpufreq stats interface. This interface

[6] Enhanced Intel Speedstep Technology for the Pentium M Processor.

appears in `/sys/devices/system/cpu/cpuX/cpufreq/stats`, whenever cpufreq is active. This interface provides the statistics about frequency of a particular CPU over time. It provides

- Total number of P-state transitions on this CPU.
- Amount of time (in jiffies) spent in each P-state on this CPU.
- And a two-dimensional (n x n) matrix with value count(i,j) indicating the number of transitions from Pi to Pj.
- A `top`-like tool can be built over this interface to show the system wide P-state statistics.

4.3.4 C-states and SMP

Deeper C-states (C2 and higher) are mostly used on laptops. And in today's kernel, C-states are only supported on UP systems. But, soon laptop CPUs will be becoming Dual-Core. That means we need to support C2 and higher states on SMP systems as well. Support for C2 and above on SMP is in the base kernel now ready for future generation of processors and platforms.

4.4 Upcoming Changes

4.4.1 C4, C5, . . .

In future, one can expect more deeper C states with higher latencies. But, with Linux kernel jiffies running at 1mS, CPU may not stay long enough in a C-state to justify entering C4, C5 states. This is where we can use the existing variable HZ solution and can make use of more deeper C-states. The idea is to reduce the rate of timer interrupts (and local APIC interrupts) when the CPU is idle. That way a CPU can stay in a low power idle state longer when they are idle.

4.4.2 ACPI 3.0 based Software coordination for P and C states

ACPI 3.0 supports having P-state and C-state domains defined across CPUs. A domain will include all the CPUs that share P-state and/or C-state. Using these information from ACPI and doing the software coordination of P-states and C-states across their domains, OS can have much more control over the actual P-states and C-states and optimize the policies on systems running with different configuration.

Consider for example a 2-CPU package system, with 2 cores on each CPU. Assume the two cores on the same package share the P-states (means both cores in the same package change the frequency at the same time). If OS has this information, then if there are only 2 threads running, OS, can schedule them on different cores of same package and move the other package to lower P-state thereby saving power without loosing significant performance.

This is a work in progress, to support software coordination of P-states and C-states, whenever CPUs share the corresponding P and C states.

5 ACPI support for CPU and Memory Hot-Plug

Platforms supporting physical hot-add and hot remove of CPU/Memory devices are entering the systems market. This section covers a variety of recent changes that went into kernel

specifically to enable ACPI based platform to support the CPU and Memory hot-plug technology.

5.1 ACPI-based Hot-Plug Introduction

The hot-plug implementation can be viewed as two blocks, one implementing the ACPI specific portion of the hot-plug and the other non ACPI specific portion of the hot-plug.

The non-ACPI specific portion of CPU/Memory hot-plug, which is being actively worked by the Linux community, supports what is know as Logical hot-plug. Logical hot-plug is just removing or adding the device from the operating system perspective, but physically the device still stays in the system. In the CPU or Memory case, the device can be made to disappear or appear from the OS perspective by echoing either 0 or 1 to the respective online file. Refer to respective hot-plug paper to learn more about the logical online/off-lining support of these devices. The ACPI specific portion of the hot-plug is what bridges the gap between the platforms having the physical hot-plug capability to take advantage of the logical hot-plug in the kernel to provide true physical hot-plug. ACPI is not involved in the logical part of on-lining or off-lining the device.

5.2 ACPI Hot-Plug Architecture

At the module init time we search the ACPI device namespace. We register a system notify handler callback on each of the interested devices. In case of CPU hot-plug support we look for `ACPI_TYPE_PROCESSOR_DEVICE` and in case of Memory hot-plug support we look for PNP0C80 HID[7] and in case of container[8] we look for ACPI004 or PNP0A06 or PNP0A05 devices.

When a device is hot-plugged, the core chipset or the platform raises the SCI,[9] the SCI handler within the ACPI core clears the GPE event and runs _Lxx[10] method associated with the GPE. This _Lxx method in turn executes Notify(XXXX, 0) and notifies the ACPI core, which in turn notifies the hot-plug modules callback which was registered during the module init time.

When the module gets notified, the module notify callback handler looks for the event code and takes appropriate action based on the event. See the module Design section for more details.

5.3 ACPI Hot-Plug support Changes

The following enhancements were made to support physical Memory and/or CPU device hot-plug.

- A new `acpi_memhotplug.c` module was introduced into the drives/acpi directory for memory hot-plug.
- The existing ACPI processor driver was enhanced to support the ACPI hot-plug notification for the physical insertion/removal of the processor.
- A new container module was introduced to support hot-plug notification on a ACPI

[7] HID, Hardware ID.

[8] A container device captures hardware dependencies, such as a Processor and Memory sharing a single removable board.

[9] SCI, ACPI's System Control Interrupt, appears as "acpi" in `/proc/interrupts`.

[10] _Lxx - L stands for level-sensitive, xx is the GPE number, e.g. GPE 42 would use _L42 handler.

container device. The ACPI container device can contain multiple devices, including another container device.

5.4 Memory module

A new `acpi_memhotplug.c` driver was introduced which adds support for the ACPI based Memory hot-plug. This driver provides support for fielding notifications on ACPI memory device (PNP0C80) which represents memory ranges that may be hot-added or hot removed during run time. This driver is enabled by enabling `CONFIG_ACPI_HOTPLUG_MEMORY` in the con fig file and is required on ACPI platforms supporting physical Memory hot plug of the Memory DIMMs (at some platform granularity).

Design: The memory hot-plug module's device notify callback gets called when the memory device is hot-plug plugged. This handler checks for the event code and for hot-add case, first checks the device for physical presence and reads the memory range reported by the _CRS method and tells the VM about the new device. The VM which resides outside of ACPI is responsible for actual addition of this range to the running kernel. The ACPI memory hotplug module does not yet implement the hot-remove case.

5.5 Processor module

The ACPI processor module can now support physical CPU hot-plug by enabling `CONFIG_ACPI_HOTPLUG_CPU` under `CONFIG_ACPI_PROCESSOR`.

Design: The processor hot-plug module's device notify callback gets called when the processor device is hot plugged. This handler checks for the event code and for the hot-add case, it first creates the ACPI device by calling `acpi_bus_add()` and `acpi_bus_scan()` and then notifies the user mode agent by invoking `kobject_hotplug()` using the kobj of the ACPI device that got hot-plugged. The user mode agent in turn on-lines the corresponding CPU devices by echoing on to the online file. The `acpi_bus_add()` would invoke the `.add` method of the processor module which in turn sets up the `apic_id` to `logical_id` required for logical online.

For the remove case, the notify callback handler in turn notifies the event to the user mode agent by invoking `kobject_hotplug()` using the kobj of the ACPI device that got hot-plugged. The user mode first off-lines the device and then echoes 1 on to the eject file under the corresponding ACPI namespace device file to remove the device physically. This action leads to call into the kernel mode routine called `acpi_bus_trim()` which in turn calls the `.remove` method of the processor driver which will tear the ACPI id to logical id mappings and releases the ACPI device.

5.6 Container module

ACPI defines a Container device with the HID being ACPI004 or PNP0A06 or PNP0A05. This device can in turn contain other devices. For example, a container device can contain multiple CPU devices and/or multiple Memory devices. On a platform which supports hotplug notify on Container device, this driver needs to be enabled in addition to the above device specific hotplug drivers. This driver is enabled by enabling `CONFIG_ACPI_CONTAINER` in the config file.

Design: The module init is pretty much the same as the other driver where in we register for the system notify callback on to every

container device with in the ACPI root namespace scope. The `container_notify_cb()` gets called when the container device is hot-plugged. For the hot-add case it first creates an ACPI device by calling `acpi_bus_add()` and `acpi_bus_scan()`. The `acpi_bus_scan()` which is a recursive call in turns calls the `.add` method of the respective hotplug devices. When the `acpi_bus_scan()` returns the container driver notifies the user mode agent by invoking `kobject_hotplug()` using kobj of the container device. The user mode agent is responsible to bring the devices to online by echoing on to the online file of each of those devices.

5.7 Future work

ACPI-based, NUMA-node hotplug support (although there are a little here and there patches to support this feature from different hardware vendors). Memory hot-remove support and handling physical hot-add of memory devices. This should be done in a manner consistent with the CPU hotplug—first kernel mode does setup and notifies user mode, then user mode brings the device on-line.

6 PNPACPI

The ACPI specification replaces the PnP BIOS specification. As of this year, on a platform that supports both specifications, the Linux PNP ACPI code supersedes the Linux PNP BIOS code. The ACPI compatible BIOS defines all PNP devices in its ACPI DSDT. [11] Every ACPI PNP device defines a PNP ID, so the OS can enumerate this kind of device through the PNP

[11] DSDT, Differentiated System Description Table, the primary ACPI table containing AML

```
pnpacpi_get_resources()
  pnpacpi_parse_allocated_resource() /* _CRS */

pnpacpi_disable_resources()
  acpi_evaluate_object (_DIS)        /* _DIS */

pnpacpi_set_resources()
  pnpacpi_build_resource_template()  /* _CRS */
  pnpacpi_encode_resources()         /* _PRS */
  acpi_set_current_resources()       /* _SRS */
```

Figure 7: ACPI PNP protocol callback routines

ID. ACPI PNP devices also define some methods for the OS to manipulate device resources. These methods include _CRS (return current resources), _PRS (return all possible resources) and _SRS (set resources).

The generic Linux PNP layer abstracts PNPBIOS and ISAPNP, and some drivers use the interface. A natural thought to add ACPI PNP support is to provide a PNPBIOS-like driver to hook ACPI with PNP layer, which is what the current PNPACPI driver does. Figure 7 shows three callback routines required for PNP layer and their implementation overview. In this way, existing PNP drivers transparently support ACPI PNP. Currently there are still some systems whose PNPACPI does not work, such as the ES7000 system. Boot option `pnpacpi=off` can disable PNPACPI.

Compared with PNPBIOS, PNPACPI does not need to call into 16-bit BIOS. Rather it directly utilizes the ACPICA APIs, so it is faster and more OS friendly. Furthermore, PNPACPI works even under IA64. In the past on IA64, ACPI-specific drivers such as 8250_acpi driver were written. But since ACPI PNP works on all platforms with ACPI enabled, existing PNP drivers can work under IA64 now, and so the specific ACPI drivers can be removed. We did not remove all the drivers yet for the reason of stabilization (PNPACPI driver must be widely tested)

Another advantage of ACPI PNP is that it sup-

ports device hotplug. A PNP device can define some methods (_DIS, _STA, _EJ0) to support hotplug. The OS evaluates a device's _STA method to determine the device's status. Every time the device's status changes, the device will receive a notification. Then the OS registered device notification handler can hot add/remove the device. An example of PNP hotplug is a docking station, which generally includes some PNP devices and/or PCI devices.

In the initial implementation of ACPI PNP, we register a default ACPI driver for all PNP devices, and the driver will hook the ACPI PNP device to PNP layer. With this implementation, adding an ACPI PNP device will automatically put the PNP device into Linux PNP layer, so the driver is hot-pluggable. Unfortunately, the feature conflicted with some specific ACPI drivers (such as 8250_acpi), so we removed it. We will reintroduce the feature after the specific ACPI drivers are removed.

7 Hot-Keys

Keys and buttons on ACPI-enabled systems come in three flavors:

1. Keyboard keys, handled by the keyboard driver/Linux input sub-system/X-window system. Some platforms add additional keys to the keyboard hardware, and the input sub-system needs to be augmented to understand them through utilities to map scan-codes to characters, or though model-specific keyboard drivers.

2. Power, Sleep, and Lid buttons. These three buttons are fully described by the ACPI specification. The kernel's ACPI button.c driver sends these events to user-space via `/proc/acpi/event`. A user-space utility such as `acpid(8)` is responsible for deciding what to do with them. Typically shutdown is invoked on power button events, and suspend is invoked for sleep or lid button events.

3. The "other" keys are generally called "hot-keys," and have icons on them describing various functions such as display output switching, LCD brightness control, WiFi radio, audio volume control etc.

Hot-keys may be implemented in a variety of ways, even within the same platform.

- Full BIOS control: Here hot-keys trigger an SMI, and the SMM BIOS[12] will handle everything. Using this method, the hot-key is invisible to the kernel—to the OS they are effectively done "in hardware."

  The advantage is that the buttons will do their functions successfully, even in the presence of an ignorant or broken OS.

  The disadvantage is that the OS is completely un-aware that these functions are occurring and thus has no opportunity to optimize its policies. Also, as the SMI/SMM is shipped by the OEM in the BIOS, users are unable to either fix it when it is broken, or customize it in any way.

  Some systems include this SMI-based hot-key mechanism, but disable it when an ACPI-enabled OS boots and puts the system into ACPI-mode.

- Self-contained AML methods: from a user's—even a kernel programmer's—point of view, method is analogous to the full-BIOS control method above. The OS is is un-aware that the button is pressed and what the button does. However, the OS actually supplies the mechanics for

[12] SMI, System Management Interrupt; SMM, System Management Mode—an interrupt that sends the processor directly into BIOS firmware.

this kind of button to work, It would not work if the OS's interrupts and ACPI AML interpreter were not available.

Here a GPE[13] causes an ACPI interrupt. The ACPI sub-system responds to the interrupt, decodes which GPE caused it, and vectors to the associated BIOS-supplied GPE handler (_Lxx/_Exx/_Qxx). The handler is supplied by the BIOS in AML, and the kernel's AML interpreter make it run, but the OS is not informed about what the handler does. The handler in this scenario is hard-coded to tickle whatever hardware is necessary to to implement the button's function.

- Event based: This is a platform-specific method. Each hot-key event triggers a corresponding hot-key event from `/proc/acpi/event` to notify user space daemon, such as `acpid(8)`. Then, acpid must execute corresponding AML methods for hot-key function.
- Polling based: Another non-standard implementation. Each hot-key pressing will trigger a polling event from `/proc/acpi/event` to notify user space daemon acpid to query the hot-key status. Then acpid should call related AML methods.

Today there are several platform specific "ACPI" drivers in the kernel tree such as `asus_acpi.c`, `ibm_acpi.c`, and `toshiba_acpi.c`, and there are even more of this group out-of-tree. The problem with these drivers is that they work only for the platforms they're designed for. If you don't have that platform, it doesn't help you. Also, the different drivers perform largely the same functions.

There are many different platform vendors, and so producing and supporting a platform-specific driver for every possible vendor is not a good strategy. So this year several efforts have been made to unify some of this code, with the goal that the kernel contain less code that works on more platforms.

[13] GPE, General Purpose Event.

7.1 ACPI Video Control Driver

The ACPI specification includes an appendix describing ACPI Extensions for Display Adapters. This year, Bruno Ducrot created the initial `acpi/video.c` driver to implement it.

This driver registers notify handlers on the ACPI video device to handle events. It also exports files in `/proc` for manual control.

The notify handlers in the video driver are sufficient on many machines to make the display control hot-keys work. This is because the AML GPE handlers associated with these buttons simply issue a Notify() event on the display device, and if the `video.c` driver is loaded and registered on that device, it receives the event and invokes the AML methods associated with the request via the ACPI interpreter.

7.2 Generic Hot-Key Driver

More recently, Luming Yu created a generic hot-key driver with the goal to factor the common code out of the platform-specific drivers. This driver is intended to support two non-standard hot-key implementations—event-based and polling-based.

The idea is that configurable interfaces can be used to register mappings between event number and GPEs associated with hot-keys, and mappings between event number and AML methods, then we don't need the platform-specific drivers.

Here the user-space daemon, acpid, needs to issue a request to an interface for the execution of those AML methods, upon receiving a specific hot-key GPE. So, the generic hot-key driver implements the following interfaces to meet the requirements of non-standard hot-key.

- Event based configure interface, `/proc/acpi/hotkey/event_config`.
  - Register mappings of event number to hot-key GPE.
  - Register ACPI handle to install notify handler for hot-key GPE.
  - Register AML methods associated with hot-key GPE.
- Polling based configure interface, `/proc/acpi/hotkey/poll_config`.
  - Register mappings of event number to hot-key polling GPE.
  - Register ACPI handle to install notify handler for hot-key polling GPE.
  - Register AML methods associated with polling GPE
  - Register AML methods associated with hot-key event.
- Action interface, `/proc/acpi/hotkey/action`.
  - Once acpid knows which event is triggered, it can issue a request to the action interface with arguments to call corresponding AML methods.
  - For polling based hot-key, once acpid knows the polling event triggered, it can issue a request to the action interface to call polling method, then it can get hot-key event number according to the results from polling methods. Then, acpid can issue another request to action interface to invoke right AML methods for that hot-key function.

The current usage model for this driver requires some hacking—okay for programmers, but not okay for distributors. Before using the generic hot-key driver for a specific platform, you need to figure out how vendor implemented hot-key for it. If it just belongs to the first two standard classes, the generic hot-key driver is useless. Because, the hot-key function can work without any hot-key driver including this generic one. Otherwise, you need to flow these steps.

- Disassemble DSDT.
- Figure out the AML method of hot-key initialization.
- Observing `/proc/acpi/event` to find out the corresponding GPE associated with each hot-key.
- Figure out the specific AML methods associated with each hot-key GPE.
- After collecting sufficient information, you can configure them through interfaces of `event_config`, `poll_config`.
- Adjust scripts for acpid to issue right command to action interface.

The hope is that this code will evolve into something that consolidates, or at least mitigates a potential explosion in platform-specific drivers. But to reach that goal, it will need to be supportable without the complicated administrator incantations that it requires today. The current thinking is that the additions of a quirks table for configuration may take this driver from prototype to something that “just works” on many platforms.

8 Acknowledgments

It has been a busy and productive year on Linux/ACPI. This progress would not have been possible without the efforts of the many developers and testers in the open source community. Thank you all for your efforts and your support—keep up the good work!

9 References

Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba *Advanced Configuration & Power Specification*, Revision 3.0, September 2, 2004. `http://www.acpi.info`

ACPICA Component Architecture Programmer Reference, Intel Corporation.

Linux/ACPI Project Home page: `http://acpi.sourceforge.net`