Sleeping Beauty – NetBSD on modern laptops

Jörg Sonnenberger <joerg@NetBSD.org>
Jared D. McNeill <jmcneill@NetBSD.org>

February 3, 2008

Abstract

This paper discusses the NetBSD Power Management Framework (PMF) and related changes to the kernel. The outlined changes allow NetBSD to support essential functions like suspend-to-RAM on most post-Y2K X86 machines. They are also the fundation for intelligent handling of device activity by enabling devices on-demand.

This work is still progressing. Many of the features will be available in the up-coming NetBSD 5.0 release.

1 Introduction

The NetBSD kernel is widely regarded to be one of the cleanest and most portable Operating System kernels available. For various reasons it is also assumed that NetBSD only runs well on older hardware. In the summer of 2006 Charles Hannum, one of the founders of NetBSD, left with a long mail mentioning as important issues the lack of proper power management and suspend-to-RAM support. One year later, Jared D. McNeill posted a plan for attacking this issue based on ideas derived from the Windows Driver Model. This plan would evolve into the new NetBSD Power Management Framework (PMF for short).

Major design goals were:

- The ability to suspend a running system, possibly including X11, and to restore the same state later.
- The ability to disable devices and restore the full state. This includes e.g. volume and current playback position for audio devices etc.
- Automatically sense which devices are not in use and place them into reduced power states.
- Provide a messaging framework for inter-driver communication for event notifications, e.g. hotkeys pressed by the user.

This paper will provide an overview of the core changes to the NetBSD kernel, the goals that have been achieved, and the challenges that have been faced along the way.

The first section will cover the PMF itself as it is the component with the greatest impact on the overall kernel. Afterwards the AMD64 and i386 specific changes to the ACPI infrastructure will be discussed. Last but not least is the special handling needed for video cards needed on many systems.

2 The NetBSD Power Management Framework

2.1 Overview

The NetBSD Power Management Framework (PMF) is a multi-layer device power management framework with inter driver asynchronous messaging support. It was originally inspired by the Windows Driver Model power management framework, but has since evolved into a model better fitting for the NetBSD kernel. The implementation is contained in:

- sys/kern/kern_pmf.c,
- sys/sys/pmf.h, and
- sys/kern/subr_autoconf.c.

2.2 Device interface

Device power management is implemented in layers:

- Device,
- Bus,
- Class (ie network, input, etc).

The basic entry points for a device driver to implement the PMF are pmf_device_register(9) and pmf_device_deregister(9). Both of these functions accept a device_t as their first argument, and in the registration case it also accepts optional driver-specific suspend and resume callbacks. These functions return true on success, and false on failure.

Device bus power management support is inherited from the parent device in the autoconfiguration tree. A device driver that attaches to the pci(4) automatically inherits PCI Power Management support. The PCI bus handlers take care of saving and restoring the common part of the PCI configuration part. They are also responsible for removing power from the device and restoring it.

Device class power management support cannot be derived as easily, so a device driver that requires class-level power management support will call the appropriate pmf_class_<type>_register and deregister functions when registering with the PMF. The currently implemented power management class types are 'network', 'input', and 'display'. Depending on the device class the register function takes additional arguments, i.e. the "struct ifnet" address for a network device.

Using this layered approach, the amount of duplicated code between device drivers is reduced and the per-driver code minimized. One example of where this is the wm(4) network driver. Since the PCI bus layer captures and restores common PCI configuration registers and the network class layer is responsible for stopping and starting the interface, no additional device specific power management code is required. Other device drivers such as bce(4) simply need a single function call in their resume callback (with no suspend callback) to restore the device to a fully operational state.

Due to its integration with the autoconfiguration subsystem, a device_t is required to register with the PMF. This differs from the former powerhook(9)

framework in previous NetBSD releases, which implemented global system suspend/resume by executing callbacks in order of registration with an opaque cookie for an argument. This interface made it impossible to control the order in which devices are suspended or resumed. With the PMF, global system suspend/resume is implemented by traversing the autoconfiguration device tree, ensuring that a device's parent is powered up before it is initialized and that a child device is suspended before its parent bus.

The most basic interfaces from kernel code to control device power are pmf_device_suspend(9) and pmf_device_resume(9). A power management device driver will typically want to suspend and resume the entire autoconfiguration tree or a subtree. In the case of a global power state transition, the power management device driver would use the pmf_system_suspend(9), pmf_system_resume(9), and pmf_system_shutdown(9) APIs. The "suspend" and "shutdown" functions are nearly the same with the exception of two points; a suspend will be aborted directly if any device in the autoconfiguration tree does not implement the PMF, and a shutdown does not invoke the bus power management layer support. Some additional support functions are available for power management drivers; pmf_device_recursive_resume(9), pmf_device_recursive_suspend(9), and pmf_device_resume_subtree(9). The first function can be used to resume a specific devices and all its parents. The other functions suspend or resume a subtree of the autoconfiguration tree.

Suspending drivers on shutdown using pmf_system_shutdown(9) avoids the problem of active devices trashing the system after a reboot e.g. with DMA. A number of drivers did this with ad hoc shutdown hooks before and doing it in the PMF provides it consistantly for all devices. This part the PMF will be extended at some point in the future to provide a separate optional hook for finer control.

2.3 Event interface

The PMF introduces a message passing framework for inter-device driver communication. Events can be directed (targetted to) a specific device, but in the most typical case anonymous events are used. An anonymous event is essentially a broadcast notification to all device drivers registered for that event.

One issue with making laptops "just work" was that there was no mechanism to associate a hotkey event with an appropriate action. Making decisions on how to handle events entirely in a userland script was considered too much of a kludge, so it was decided that an inter-driver event framework was necessary.

Consider brightness control hotkeys on a laptop. Every hardware vendor implements this in a different way, and the device that generates the hotkey event is not necessarily the same device that gives control of the LCD backlight. A hotkey device driver simply uses the pmf_event_inject(9) function to inject a brightness control event (PMFE_DISPLAY_BRIGHTNESS_UP, PMFE_DISPLAY_BRIGHTNESS_DOWN) into the pmf event queue. A worker thread is then woken up, and scans the list of callbacks created using the pmf_event_register(9) and pmf_event_deregister(9) functions. Since this is an anonymous event (device_t argument to pmf_event_inject(9) is NULL), all callbacks registered for the appropriate PMF_DISPLAY_BRIGHTNESS_* event are executed. The hotkey driver does not need to know or care if the console driver, generic ACPI display driver, vendor-specific ACPI device driver, or other power control driver

will handle the event. As long as one of these drivers is present and working, the brightness key press will “just work”.

For events that are better handled in userland, hotkey support was added to sysmon_power(9) and powerd(8). This is typically used for associating a "lock screen" key with xscreensaver, a "display cycle" key with xrandr, "eject" key with a DVD-ROM tray, and so on.

2.4 Vendor and model specific information

Sometimes there is no choice but to only apply a quirk in a device driver on certain hardware platforms. A very simple API was added to the PMF to act as a dictionary for obtaining information about the running system from within device drivers. Platform dependent code is responsible for filling in these tables; on AMD64 and i386 information retrieved from DMI (aka SMBIOS) tables are used.

The sony_acpi(4) driver uses this to apply workarounds in its initialization routines on certain VAIO series notebooks.

3 ACPI improvements

3.1 ACPICA

The Intel ACPI Component Architecture (ACPICA) is an OS-independent reference implementation of the ACPI specification. NetBSD 4.0 shipped with the two year old 20060217 release of the ACPICA, so it was decided that the third party code should be updated to a newer release.

At the time of the ACPICA upgrade the latest version of ACPICA available from Intel was 20061109, but it was known that other operating systems were shipping newer releases. As the APIs are constantly evolving, a significant amount of integration effort would have been been required regardless of the release selected. It was decided to go with the release of ACPICA present in FreeBSD -CURRENT at the time, 20070320.

The majority of changes required for the new APIs were mechanical, related to renamed structures and structure members. Another issue involved a change in the way that tables are accessed; previous releases of ACPICA pre-parsed some tables and stored them in global pointers. This has been changed in some cases to store a copy of the table in a global structure, and in other cases the Operating System must map the table itself using AcpiOsMapMemory.

The operating system dependent (Osd) interfaces were also changed. The following functions had minor signature changes:

- AcpiOsMapMemory,
- AcpiOsWaitSemaphore,
- AcpiOsAcquireLock,
- AcpiOsDeleteLock,
- AcpiOsGetRootPointer,
- AcpiOsGetThreadId.

In addition, new functions such as AcpiOsValidateInterface and AcpiOsValidateAddress were required, and AcpiOsQueueForExecution was renamed to AcpiOsExecute.

In the process of tracing down various interrupt issues, the ACPI initialization sequence was updated. The initialization was split into two phases. The first phase is just long enough to load the MADT. That table is required for interrupt setup, especially when using the IOAPIC. The second phase can therefore directly hook up the ACPI System Configuration Interrupt and finish the initialization sequence. This replaces the lazy interrupt setup code in the AMD64 and i386 code.

This ACPICA update exposed a fundamental design flaw in the NetBSD Embedded Controller driver, requiring it to be rewritten from scratch.

Since this work has completed a new ACPICA web site, `http://www.acpica.org`, has appeared offering a 20080123 release for download.

3.2 Embedded Controller

The Embedded Controller (EC) is one of the central hardware components of ACPI. The ACPI virtual machine is using the EC to access various devices without requiring the attention of the CPU. It is also used by the hardware to notify the ACPI VM about changes in the system configuration using the System Configuration Interrupt (SCI).

The EC has a very simple interface using two one-byte ports. The first port is used to send commands to the EC and read back the current processing status. The second port (data port) is used for operands of the commands. The EC understands the five commands "query", "read", "write", "enable burst" and "disable burst". Operands like the address of a "read" or "write" or the return value for a "query" are transfered using an internal buffer in the EC. When this buffer is empty and new data can be send, a flag is set and an interrupt is sent. The same happens for reading data from the EC.

This interface forces the driver to communicate asynchronously with the hardware. As this is often undesirable, the burst mode was added. It allows a driver to send commands and data back-to-back with the full attention of the EC. If the EC can't process a command in a timely manner, it can disable the burst mode itself.

The old EC driver as inherited from FreeBSD tried to deal with this mess by using a mixture of interrupt mode and polling mode. It was very hard to follow the flow of control and add locking without introducing dead locks. For that reason, the EC driver was rewritten from scratch as part of the jmcneill-pm branch.

The first important observation for the new driver was the symmetry between the entry points. The driver has to deal with three request types: reads, writes and SCIs. SCIs are special events raised by the EC, read and write operations are originated in the system. The access to the driver can therefore be serialised by a single mutex shared between the three entry points. For the processing of the SCIs a kernel thread is the simplest solution, but a work queue could have been used as well.

The second observation for the driver design is that most of the complications in the interface are a result of not using the state flow of the EC in the driver. The actions in the driver are much easier to formulate as finite state machine.

As soon as one of the the entry points obtains the driver lock, it writes the address for read or write access and the command to process. Afterwards it just waits for completion. The interrupt handler drives the state machine. If it finds a request for a SCI, it signals the kernel thread to wake up. Depending on the state of the machine, it writes the address or transfers the data byte. When a command is done, it wakes up the blocking originating thread.

One problem of this approach is that it depends on the hardware properly sending interrupts. Many EC implementations don't do that though. Linux and FreeBSD dealt with this by using the burst mode. A simpler alternative is to just poll the hardware after some time using a callback. In other words, if the hardware doesn't send an interrupt, simulate it.

The second problem is that during early boot, suspend and resume neither interrupts nor timeouts are processed. Polling the EC directly is similiar again to how lost interrupts are processed. Experiments have shown that spinning a bit is generally helpful as most EC commands finish in less than 5ms. Polling has to be done with care though. On a Lenovo Thinkpad R52, a busy loop without delay completely kills the EC, it doesn't even provide the emergency powerdown. Similiar issues exist with other vendors. Tests have shown that at least 100ms intervals are needed for pure polling.

The new driver has proven to be robust and much easier to adopt to new vendor bugs.

3.3 Suspend support

Proper support for suspend to RAM is one of the most often requested features in the Open Source world. The Advanced Power Management interface provided this without much complexity in the Operating System. As Microsoft has pushed ACPI for a long time, vendor support for APM started to disappear. For ACPI based suspend to RAM the Operating System is responsible for most of work. The first part of the process is saving all device state and preempting them. This is part was addressed in section 2.2. The second part is saving the CPU state and calling the firmware. NetBSD inherited the S3 support for i386 from FreeBSD. The support was working, but lacking in two importants areas. S3 was not possible on AMD64 and it only worked with a Uniprocessor kernel. With the advent of Intel's Core 2 in consumer notebooks both limitations had to be fixed.

This was addressed in two parts. First, the ACPI wake code was ported to AMD64 and later it was extended to handle Application Processors (APs). The wake code is called by the firmware almost directly after the resume. At this point, the CPU is still running in Real Mode. For switching to Protected Mode part of the address space has to be mapped to the same address in virtual and physical address space. The code inherited from FreeBSD solved this by adding the address of the wakecode to the kernel map, even though the kernel map normally doesn't cover this address range. This was fixed by using a temporary copy of the Page Directory and a special Page Table, both located in low memory. The wakecode enables paging using this temporary copy and switches to the normal version in a second step. For the AMD64 port this was crucial as the switch to Long Mode needs the equivalent of the Page Directory under the 4GB limit. The only major surprise left for the AMD64 port was the trap when a page has the NX bit set and the feature was not enabled already.

The first attempt at multiprocessor support was to migrate all threads from the APs and let them just idle. On resume the bootstrap was repeated as it is done during the normal boot. This worked somewhat as the APs came back to live, but they hit an assertion in the process scheduler pretty soon. This assertion didn't make any sense as it essentially meant that the idle thread was not scheduled.

The second try utilised the already working wakecode. The wakecode was changed to use storage in the CPU specific data instead of global variables. On suspend, the APs follow the same code path as the primary CPU and on resume, they are recovering exactly the state they were in before. After the first try on a Intel Core 2 system, the system crashed with the same assertion as during the first attempt.

Further debugging revealed that both cores disagreed on the state of the idle thread of the second core. This suggested a cache synchronisation problem and it turned out that the modifications of the second core where still in the L1 cache and not written back to main memory, when the suspend occured. The first core does an explicit invalidation before suspend, but it become obvious that the L1 cache of the second core was not affected by this. Adding the necessary flush before halting the second CPU fixed the problem.

At this point, an optimisation for the pmap module was added and the kernel changed to always use large pages to map the code segment, if the hardware supports it. This broke the i386 resume again. Just as the use of the NX bit, large pages had to be enabled earlier.

The wakecode is been improved in non-functional ways. One important change was to not use double return (like longjmp). The code flow was instead reorganised so that the suspend is entered from a leaf function. One side effect of this change is that the amount of state to save and restore has reduced as only caller-save registered are now volatile.

Further work in this area is to merge the MP bootstrap code with the ACPI wakecode. The former is almost a subset of the wakecode now. The change would allow moving the resource allocation and state setup from assembly code on the APs into the high-level code running on the AP, resulting in more simplifications.

4 Video card handling

Of all hardware in a modern PC, the most problematic part for a successful resume is the video card. One reason for this is the great variety of incompatible chips. Another reason is the complete lack of interface descriptions for many graphic chips.

The first approach to this problem is just calling the Power On Self Test (POST) code in the VGA BIOS before switching to protected mode. This has been available for a long time as option and works on a number of systems. The function depends on the firmware restoring the state of the main PCI bridges and the VGA device, which doesn't happen e.g. on Dell machines.

The second approach is a userland program called vbetool. The program either uses VM86 or a real mode software emulator to execute the POST code after the PCI code has restored the generic register set. This fixes the majority of the remaining systems. The biggest problem is that it doesn't allow you to

recover the display early enough to see and debug problems in the other drivers.

The third approach is to implement the necessary functions in chipset-specific drivers. This is actively worked on as part of the DRM code, but it is unlikely to address older, undocumented chips.

As part of the power management work a variation of the second approach was added. A size optimised version of x86emu (as used by XFree86 and vbetool) was added to NetBSD. This code allows doing the POST directly from within the kernel. Using VM86 mode would be an option for i386, but for AMD64. The CPU emulation is complemented by an emulation of the i8254 (the AT timer) to prevent the BIOS from destroying the kernel configuration.

The in-kernel VGA POST is still work-in-progress, but the goal is to completely replace vbetool and the early POST call.

5 Conclusion and future work

The jmcneill-pm branch and the related changes post-merge were a great success. Most laptops are now able to use ACPI based suspend-to-RAM. The number of systems that can't use ACPI for system configuration was greatly reduced as well. Work continues to fix any regressions left and identify remaining problems with ACPI on older hardware. Extending the ACPI S3 support is also part of the plan for NetBSD 5 to ship SMP enabled kernels by default.

The fine grained idle control of devices is still under investigation. The current implementation for audio devices has problems with uaudio(4) due to the architecture of the USB stack. Further extensions are planned though. The cardbus network drivers are currently powering down the card if it is not up. This is desirable for PCI devices as well.

The interface for device power management is focusing on making it easy to add support to an existing driver. It currently doesn't allow drive-specific logic for wake up events or multiple power states. The PCI support is currently limited to D0 and D3hot. Future work will exploit interface to support fast resume states like D1 and physically removing the power based on bridge logic (D3cold).

The event interface is used for handling many of the modern special buttons in the kernel. Future work will extend this to interact with userland components like Gnome.

The video BIOS access based on x86emu will be used for vesafb as well. Long term goal is making vesafb work on any platform with PCI devices.

To summarize the changes it is clear that NetBSD has caught up to Linux in many critical areas. The biggest remaining tasks are converting the various drivers in the kernel to support suspend/resume and to investigate the available mechanisms on other hardware architectures like ARM.