Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Suspend-to-RAM in Linux®

A. Leonard Brown
Intel Open Source Technology Center
len.brown@intel.com

Rafael J. Wysocki
Institute of Theoretical Physics, University of Warsaw
rjw@sisk.pl

Abstract

Mobile Linux users demand system suspend-to-RAM (STR) capability due to its its combination of low latency and high energy savings.

Here we survey the design and operation of STR in Linux, focusing on its implementation on high-volume x86 ACPI-compliant systems. We point out significant weaknesses in the current design, and propose future enhancements.

This paper will be of interest primarily to a technical audience of kernel and device driver developers, but others in the community who deploy, support, or use system sleep states may also find it useful.

1 Introduction

When a computer is powered on, it enters the *working state* and runs applications.

When a computer powered off, it consumes (almost) no energy, but it doesn't run applications.

As illustrated in Figure 1, several power-saving system *sleep states* are available between the working and power-off states. System sleep states share several properties:

- Energy is saved.
- The CPUs do not execute code.
- The I/O devices are in low-power states.
- Application state is preserved.

However, system sleep states differ from each other in two major ways:

- The size of the energy-saving benefit.

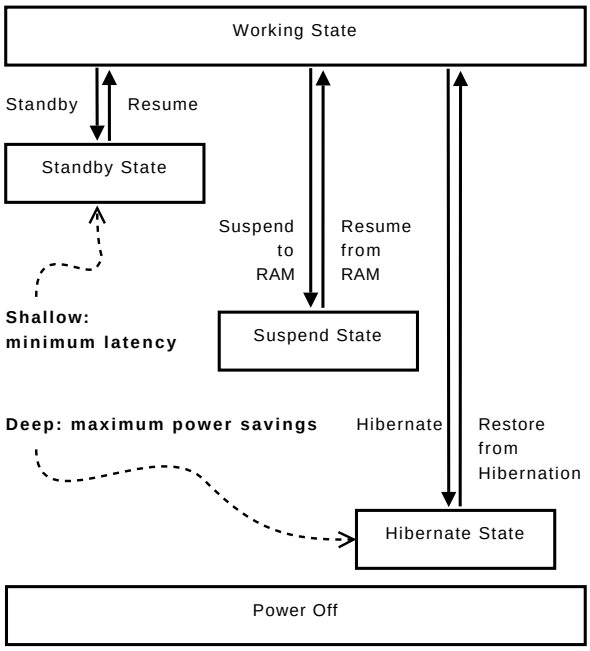

Figure 1: System States

- The *wake-up latency* cost required to return to the working state.

Linux supports three types of system-sleep states: standby, suspend-to-RAM, and hibernate-to-disk.

Standby is the most *shallow* system-sleep state. This means that it enjoys minimal wake-up latency. However, standby also delivers the least energy savings of the sleep states.

Suspend-to-RAM is a *deeper* sleep state than standby. STR saves more energy, generally by disabling all of the motherboard components except those necessary to refresh main memory and handle wake-up events.

In practice, STR generally enjoys the same latency as standby, yet saves more energy. Thus standby support is typically of little benefit, and computer systems often do not provide it.

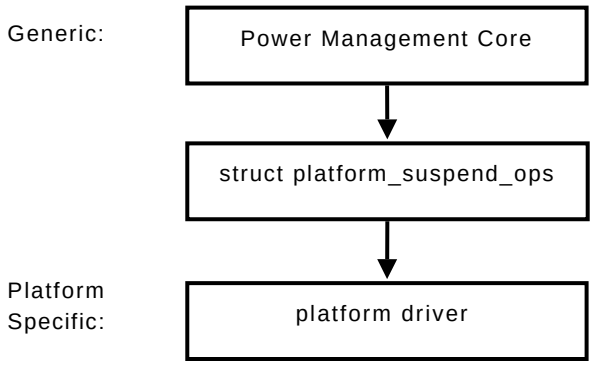

Figure 2: Linux PM infrastructure

Hibernate-to-disk is deeper than suspend-to-RAM. Indeed, it shares the same (almost zero) energy consumption as the power-off state. This makes hibernate valuable when application state needs to be preserved for a long period. Unfortunately hibernate resume latency is quite high, so it generally is not as useful as STR.

Here we focus on suspend-to-RAM, popular due to its combination of relatively low latency and relatively high energy savings. We first introduce the Linux Power Management (PM) Core, which is responsible for controlling suspend and resume operations at the kernel level. Next we describe the role of the platform firmware and ACPI in STR, and we provide an overview of the operations performed by the Linux kernel during suspend and resume. We examine some parts of the kernel's STR infrastructure in more detail, focusing on the known issues with the current implementation and planned improvements. Finally, we consider some problems related to the handling of hardware, especially graphics adapters, their current workarounds, and future solutions.

2 The Linux Power Management Core

STR and standby are available only on systems providing adequate hardware support for them: The system main memory has to be powered and refreshed as appropriate so that its contents are preserved in a sleep state. Moreover, the hardware must provide a mechanism making it possible to wake the system up from that state and pass control back to the operating system (OS) kernel. It is also necessary that power be at least partially removed from devices before putting the system into a sleep state, so that they do not drain energy

```
struct platform_suspend_ops {
  int (*valid)(suspend_state_t state);
  int (*begin)(suspend_state_t state);
  int (*prepare)(void);
  int (*enter)(suspend_state_t state);
  void (*finish)(void);
  void (*end)(void);
};
```

Figure 3: `struct platform_suspend_ops`

in vain. The part of the system providing this functionality is often referred to as *the platform*. It defines the foundation of the system—the motherboard hardware as well as the firmware that ships with it.[1]

To carry out STR and standby power transitions, the Linux kernel has to interact with the platform through a well-defined interface. For this purpose, the kernel includes *platform drivers* responsible for carrying out low-level suspend and resume operations required by particular platforms. They supply a set of callbacks via `struct platform_suspend_ops` shown in Figure 3. The `.valid()` and `.enter()` callbacks are mandatory, while the others are optional.

The platform drivers are used by the PM Core. The PM core is generic; it runs on a variety of platforms for which appropriate platform drivers are available, including ACPI-compatible personal computers (PCs) and ARM platforms.

This paper focuses primarily on ACPI-compatible platforms. The next section describes how ACPI fits into the system architecture and describes some of the specific capabilities that ACPI provides for suspend/resume. Then we combine the PM core and ACPI discussions by stepping through the suspend and resume sequences on an ACPI platform, in which all six of the `platform_suspend_ops` are invoked.

3 ACPI and Platform System Architecture

Platform hardware defines the programming model seen by software layers above. Platforms often augment this programming model by including an embedded controller (EC) running motherboard firmware. The EC off-loads the main processors by monitoring sensors for

[1] The motherboard hardware includes not only the major processor, memory, and I/O sub-systems, but also interrupt controllers, timers, and other logic that is visible to the software layers above.

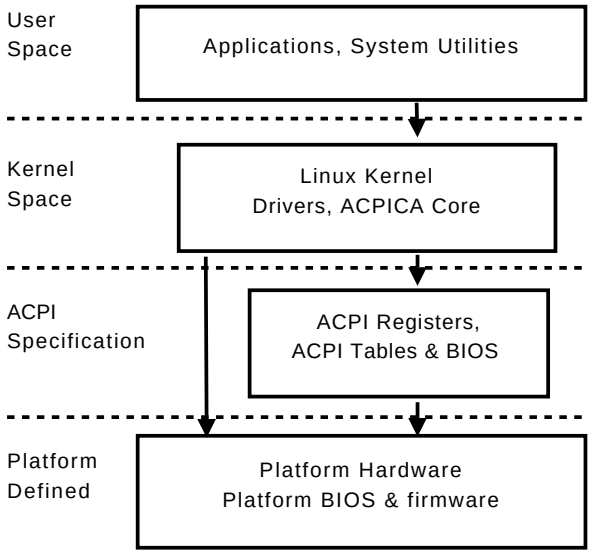

Figure 4: Platform Abstraction Layers

buttons, batteries, temperature, fans, etc. In addition, personal computer (PC) motherboards include a PC-compatible BIOS (Basic Input Output System) responsible for initializing the system and presenting some standard services, such as booting, to the OS kernel.

The ACPI specification [ACPI-SPEC] defines a layer of abstraction that sits above the platform-defined layer. ACPI specifies that the platform hardware must support certain standard registers, and that the BIOS be extended to export tables in memory to guide the OS kernel. Figure 4 shows how ACPI fits into the system architecture.

Some ACPI tables are simply static data structures that the ACPI BIOS uses to describe the machine to the OS. Other ACPI tables contain functions, known as *methods*, to be executed in the kernel, encoded in the ACPI Machine Language (AML).

AML methods are first expressed in ACPI Source Language (ASL) and then translated into the AML byte code with the help of an ASL compiler. The compiled AML tables are then burned into PROM when the motherboard is manufactured.

AML is executed by the ACPI Component Architecture [ACPICA] Core's AML interpreter residing in the next layer up—the Linux kernel. This design allows AML, which is effectively ACPI BIOS code, to run in kernel context.[2] That, in turn, allows platform designers to use many different types of hardware components and to tailor the interfaces between those components and the OS kernel to their needs.

However, since the AML code is obtained by compiling source code written in ASL, the preparation of an ACPI platform involves software development that is prone to human error. There are additional problems related to the ACPI's abstraction capability, some of which are discussed in Section 11.

4 ACPI and Suspend to RAM

Aside from its configuration and run-time power-management responsibilities, ACPI also standardizes several key hooks for suspend and resume:

1. Device power states.
2. Device wake-up control.
3. Standard mechanism to enter system sleep states.
4. Firmware wake-up vector.

When discussing ACPI and devices, it is important to realize that ACPI firmware is stored into EEPROM when a motherboard is created. Thus ACPI can be aware of all logic and devices that are permanently attached to the motherboard, including on-board legacy and PCI devices, device interrupt routing, and even PCI slot hot-plug control. But ACPI has no knowledge of the devices that may later be plugged into I/O slots or expansion buses.

ACPI defines "Device Power States" (D-states) in terms of power consumption, device context retention, device-driver restore responsibilities, and restore latency. The PCI Power Management specification [PCI-PM] similarly defines the D-states used by PCI devices; ACPI extends the notion of D-states to non-PCI motherboard devices.

D-states can be used independently of system sleep states, for run-time device power management. Note, however, that D-states are not performance states; that is, they do not describe reduced levels of performance. A device is non-functional in any D-state other than D0.

[2] Before ACPI, the choices to support run-time BIOS code were to call the BIOS directly in real mode, which required a real-mode OS, or to invisibly invoke the System Management Mode (SMM).

System sleep states, such as STR, mandate the use of D-states. Before D-states were implemented in Linux, we found several systems that would suspend and resume from RAM successfully, but they had devices which would continue to drain energy while the system was suspended—which severely shortened the the system's ability to sleep on battery power.

ACPI also defines a mechanism to enable and disable device wake-up capability. When the system is in the working state, this mechanism can be used to selectively wake up a sleeping device from a D-state. When the whole system is suspended, this capability may be used to enable automatic system resume.

Many devices export native wake-up control. In particular, modern Ethernet Network Interface Cards (NIC) support Wake-on-LAN (WOL) and their drivers export that function via `ethtool(1)`. Note that this has to be a native capability, because ACPI firmware can not provide wake-up support for add-on adapters.

Today the wake-up support in Linux is in flux. There is a legacy hook for ACPI wake-up capability in `/proc/acpi/wakeup`, but that interface is nearly unusable, as each entry refers to an arbitrary 4-letter ASL name for a device that the user (or an application) cannot reliably associate with a physical device. The device core provides its own wakeup API in `/sys/devices/.../power/wakeup`; this is not yet fully integrated with the ACPI wake-up mechanism.

The ACPI specification carefully describes the sequence of events that should take place to implement both suspend and resume. Certain platform ACPI hooks must be invoked at various stages in order for the platform firmware to correctly handle suspend and resume from RAM. These hooks are mentioned in later sections that detail the suspend and resume sequence.

Finally, ACPI provides a standard mechanism to tell the platform what address in the kernel to return to upon resume.

More information about ACPI in Linux can be found in previous Linux Symposium presentations [ACPI-OLS], as well as on the Linux/ACPI project home page [ACPI-URL].

5 Suspend Overview

There are two ways to invoke the Linux kernel's suspend capability. First, by writing `mem` into `/sys/power/state`.[3] Second, with the `SNAPSHOT_S2RAM` *ioctl* on `/dev/snapshot`, a device provided by the user-space hibernation driver. This second method is provided only as a means to implement the mixed suspend-hibernation feature[4] and will not be discussed here.

Once `mem` has been written to `/sys/power/state`, the PM core utilizes the `platform_suspend_ops` in the steps shown in Figure 5. First, it invokes the platform driver's global `.valid()` method, in order to check whether the platform supports suspend-to-RAM.

The `.valid()` callback takes one argument representing the intended system-sleep state. Two values may be passed by the PM core, `PM_SUSPEND_STANDBY` and `PM_SUSPEND_MEM`, representing the standby and the suspend sleep states, respectively. The `.valid()` callback returns `true` if the platform driver can associate the state requested by the PM core with one of the system-sleep states supported by the platform. Note, however, that on non-ACPI systems the choice of the actual sleep state is up to the platform. The state should reflect the characteristics requested by the core (e.g., the STR state characteristics if `PM_SUSPEND_MEM` is used), but the platform may support more than two such sleep states. In that case, the platform driver is free to choose whichever sleep state it considers appropriate. But the choice is made later, not during `.valid()`.

If the `.valid()` platform callback returns `true`, the PM core attempts to acquire `pm_mutex` to prevent concurrent system-wide power transitions from being started. If that succeeds, it goes on to execute `sys_sync()` to help prevent unwritten file system data from being lost in case the power transition fails in an unrecoverable manner. It switches the system console to a text terminal in order to prevent the X server from interfering with device power-down. It invokes suspend notifiers to let registered kernel subsystems know about the impending suspend, as described in Section 7. It *freezes the tasks*, as detailed in Section 8. This puts all user processes into a safe, static state: They do not hold any semaphores or mutexes, and they cannot run until the PM core allows them to.

[3] The old STR user interface, based on `/proc/acpi/sleep`, is deprecated.

[4] This feature first creates a hibernation image and then suspends to RAM. If the battery lasts, then the system can resume from RAM, but if the battery fails, then the hibernation image is used. An experimental implementation is provided by the *s2both* utility included in the user-land suspend package at `http://suspend.sf.net`.

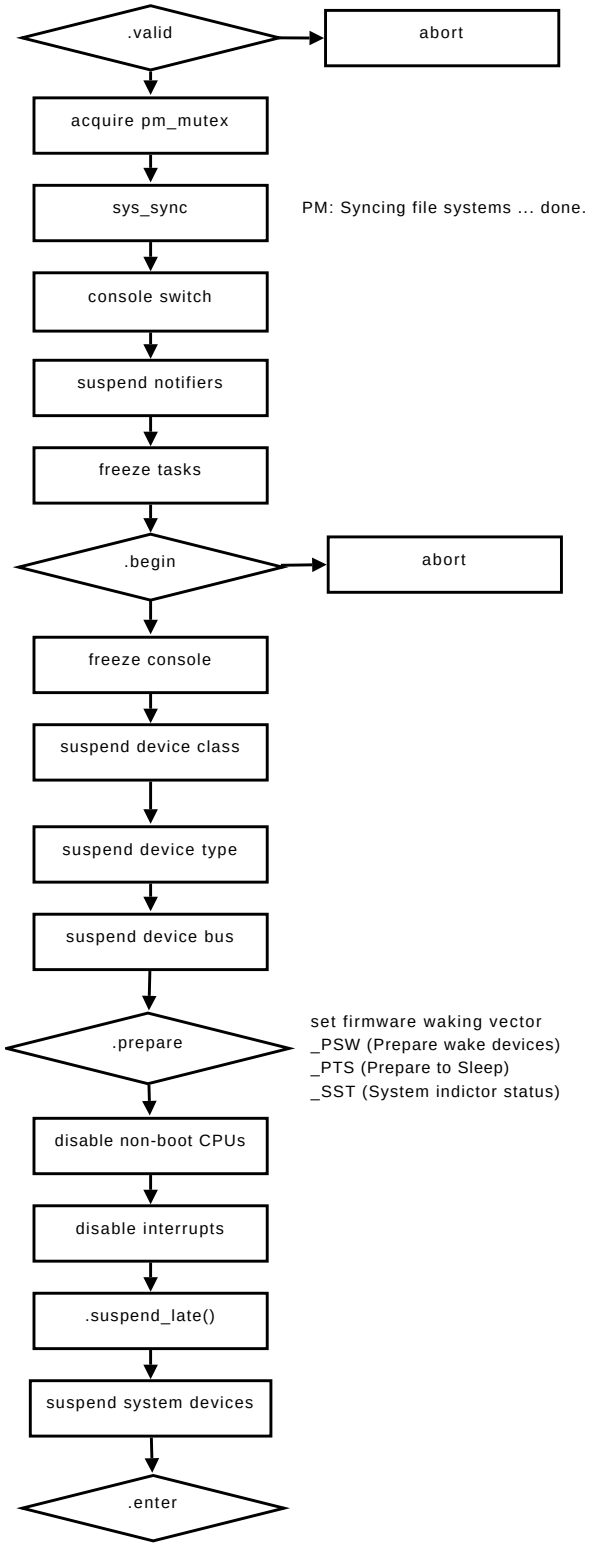

Figure 5: Suspend Sequence

Next the PM core invokes the `.begin()` platform-suspend callback to notify the platform driver of the desired power transition. The `.begin()` callback takes one argument representing the system-sleep state requested by the PM core. The interpretation as well as the possible values of its argument are the same as for the `.valid()` callback. At this point, however, the platform driver chooses the sleep state in which to place the system and stores the information for future reference. The choice made by the platform driver is not directly conveyed to the PM core, which does not have the means to represent various sleep states that may be supported by different platforms and does not really need that information. Still, the sleep state chosen by the platform driver, the *target state* of the transition, may determine the low-power states (D-states on ACPI systems) into which devices should be put. For this reason, the platform driver provides device drivers with information on the low-power states in which they are supposed to place their devices. The `.begin()` callback returns 0 on success or an error code on failure, in which case the PM core aborts the transition.

After `.begin()` succeeds, the PM core blocks system console messages in order to protect the console devices from being accessed while they are suspended,[5] and starts suspending devices (i.e., putting them in low-power states). Devices are suspended in reverse order of registration; in this way the kernel should never find itself stuck in a situation where it needs to access a suspended device or where a suspended parent has an active child. To suspend a device, the PM core invokes the suspend callbacks provided by the device-class driver, device-type driver, and bus-type driver associated with it, in that order.

Device-class, device-type, and bus-type drivers can each implement one device-suspend method, called `.suspend()`, and one corresponding device-resume method, called `.resume()`. Bus-type drivers can define extra device-suspend and -resume methods to be executed with interrupts disabled, called `.suspend_late()` and `.resume_early()`, respectively. These additional methods are invoked after the non-boot CPUs have been disabled, as described below.

Each of the suspend callbacks takes two arguments,

[5] Since this mechanism makes debugging difficult, there is a `no_console_suspend` kernel command-line parameter which prevents it from triggering.

a pointer to the appropriate device structure and a `pm_message_t` argument, representing the transition being carried out. Currently five values of this argument are recognized: `PMSG_ON`, `PMSG_FREEZE`, `PMSG_SUSPEND`, `PMSG_HIBERNATE`, and `PMSG_PRETHAW`. The first represents the transition back to the working state, while `PMSG_FREEZE`, `PMSG_HIBERNATE`, and `PMSG_PRETHAW` are specific to hibernation. Thus only `PMSG_SUSPEND` is used for standby and STR. Since the same callbacks are invoked for both suspend and hibernation, they must determine the proper actions to perform on the basis of the `pm_message_t` argument.

The device-class, device-type, and bus-type suspend callbacks are responsible for invoking the suspend callbacks implemented by individual device drivers. In principle the names of those suspend callbacks may depend on the device class, device type, or bus type the device belongs to, but traditionally drivers' suspend callbacks are called `.suspend()` (or `.suspend_late()` if they are to be executed with interrupts disabled). Also, all of them take two arguments, the first of which is a pointer to the device structure and the second of which is as described above.

The framework for suspending and resuming devices is going to be changed. The current framework has some deficiencies: It is inflexible and quite inconvenient to use from a device-driver author's point of view, and it is not adequate for suspending and resuming devices without the freezing of tasks. As stated in Section 8, the freezing of tasks is planned to be phased out in the future, so a new framework for suspending and resuming devices (described in Section 9) is being introduced.

After executing all of the device-suspend callbacks, the PM core invokes the `.prepare()` platform suspend method to prepare the platform for the upcoming transition to a sleep state. For the ACPI platform, the `_PTS` global-control method is executed at this point.

Next the PM core disables non-boot CPUs, with the help of the CPU hot-plug infrastructure. We will not discuss this infrastructure in detail, but it seems important to point out that the CPU hot-plug notifiers are called with special values of their second argument while non-boot CPUs are being disabled during suspend and enabled during resume. Specifically, the second argument is bitwise OR'ed with `CPU_TASKS_FROZEN`, so that the notifier code can avoid doing things that might lead to a deadlock or cause other problems at these times. The notifier routines should also avoid doing things that are not necessary during suspend or resume, such as un-registering device objects associated with a CPU being disabled—these objects would just have to be re-registered during the subsequent resume, an overall waste of time.[6]

After disabling the non-boot CPUs, the PM core disables hardware interrupts on the only remaining functional CPU and invokes the `.suspend_late()` methods implemented by bus-type drivers which, in turn, invoke the corresponding callbacks provided by device drivers. Then the PM core suspends the so-called system devices (also known as *sysdevs*) by executing their drivers' `.suspend()` methods. These take two arguments just like the regular `.suspend()` methods implemented by normal (i.e., not *sysdev*) device drivers, and the meaning of the arguments is the same.

To complete the suspend, the PM core invokes the `.enter()` platform-suspend method, which puts the system into the requested sleep state. If the `.begin()` method is implemented for given platform, the state chosen while it was executed is used and the argument passed to `.enter()` is ignored. Otherwise, the platform driver uses the argument passed to `.enter()` to determine the state in which to place the system.

6 Resume Overview

The resume sequence is the reverse of the suspend sequence, but some details are noteworthy.

Resume is initiated by a wake-up event—a hardware event handled by the platform firmware. This event may be opening a laptop lid, pressing the power button or a special keyboard key, or receiving a *magic* WOL network packet.

Devices must be enabled for wake-up before the suspend occurs. On ACPI platforms, the power button is always enabled as a wake-up device. The sleep button and lid switches are optional, but if present they too are enabled as wake-up devices. Any platform device may be configured as a wake-up device, but the power, sleep, and lid buttons are standard.

[6] It also would mess up the PM core's internal lists, since the objects would be re-registered while they were still suspended.

When a wake-up event occurs, the platform firmware initializes as much of the system as necessary and passes control to the Linux kernel by performing a jump to a memory location provided during the suspend. The kernel code executed at this point is responsible for switching the CPU to the appropriate mode of operation.[7] This sequence is similar to early boot, so it is generally possible to reuse some pieces of the early initialization code for performing the resume CPU-initialization operations.

Once the CPU has been successfully reinitialized, control is passed to the point it would have reached if the system had not been put into the sleep state during the suspend. Consequently the PM core sees the platform-suspend callback `.enter()` return zero. When that happens, the PM core assumes the system has just woken up from a sleep state and starts to reverse the actions of the suspend operations described in Section 5.

It resumes *sysdevs* by executing their `.resume()` callbacks, and then it invokes the device-resume callbacks to be executed with interrupts disabled. That is, it executes the `.resume_early()` callbacks provided by bus-type drivers; they are responsible for invoking the corresponding callbacks implemented by device drivers. All of these callbacks take a pointer to the device object as their only argument.

Subsequently the non-boot CPUs are enabled with the help of the CPU hot-plug code. As mentioned above, all of the CPU hot-plug notifiers executed at this time are called with their second argument OR-ed with `CPU_TASKS_FROZEN`, so that they will avoid registering new device objects or doing things that might result in a deadlock with a frozen task.

After enabling the non-boot CPUs, the PM core calls the `.finish()` platform-suspend method to prepare the platform for resuming devices. In the case of an ACPI platform, the `_WAK` global-control method is executed at this point.

Next the PM core resumes devices by executing the device-resume methods provided by bus-type, device-type, and device-class drivers, in that order. All of these methods are called `.resume()` and take a device pointer as their only argument. They are responsible for invoking the corresponding methods provided by device

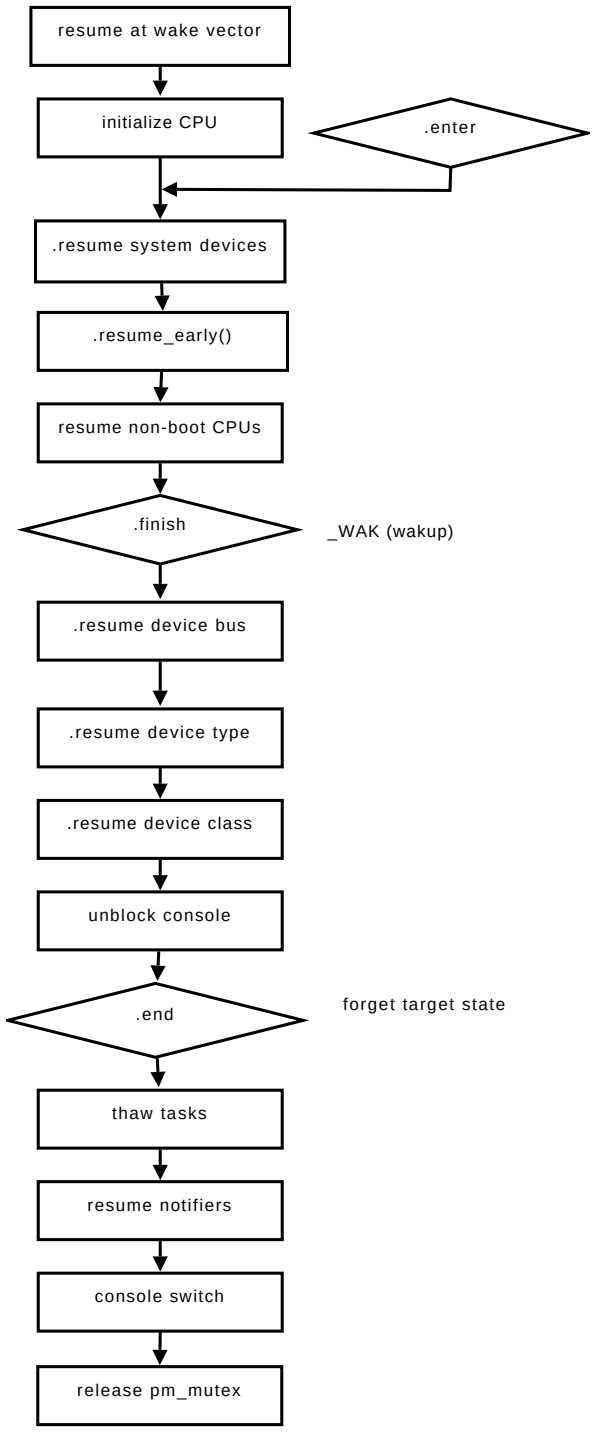

Figure 6: Resume Sequence

[7] For example, protected mode on an *i386* PC or 64-bit mode on an *x86-64* system.

drivers. Although these methods may return error codes, the PM core cannot really do anything about resume errors; the codes are used for debugging purposes only. Devices are resumed in order of registration, the reverse of the order in which they were suspended.

Once devices have been resumed, the PM core unblocks the system console so that diagnostic messages can be printed, and calls the `.end()` platform method. This method is responsible for doing the final platform-resume cleanup. In particular, it assures that the information about the target sleep state of the system stored by `.begin()` has been discarded by the platform driver.

The last three steps of resume are the thawing of tasks, invoking suspend notifiers with the appropriate argument (`PM_POST_SUSPEND`), and switching the system console back to whatever terminal it had been set to before the suspend started. Finally, the PM core releases `pm_mutex`.

7 Suspend and Hibernation Notifiers

Suspend and hibernation notifiers are available for subsystems that need to perform some preparations before tasks are frozen (see Section 8). For example, if a device driver needs to call `request_firmware()` before a suspend, that should be done from within a suspend notifier.

The notifiers are registered and un-registered using the `register_pm_notifier()` and `unregister_pm_notifier()` functions, respectively. Both these functions take one argument, a pointer to an appropriately populated `struct notifier_block`. If there is no need to un-register a suspend notifier, it can be registered with the help of the simplifying macro `pm_notifier()`, which takes only a function name and a priority as arguments.

The notifiers are called just prior to freezing tasks during both suspend and hibernation with their second argument set to `PM_SUSPEND_PREPARE` or `PM_HIBERNATION_PREPARE`, respectively,[8] as well as during resume from a sleep state or hibernation with the second argument equal to `PM_POST_SUSPEND` or `PM_POST_HIBERNATION`, respectively. In addition, they are called if suspend or hibernation fails. The PM core does not distinguish these invocations from the calls made during a successful resume; for this reason, the notifier code should be prepared to detect and handle any potential errors resulting from a suspend failure. Regardless, the rule is that if the notifiers were called with `PM_SUSPEND_PREPARE` during suspend, then they are called with `PM_POST_SUSPEND` to undo the changes introduced by the previous invocation, either during resume or in a suspend error path.

Notifiers return zero on success; otherwise they return appropriate error codes. However, while an error code returned by a notifier called during suspend causes the entire suspend to fail, error codes returned by notifiers called during resume are ignored by the PM core, since it is not able to act on them in any significant way.

8 Freezing Tasks

In both suspend and hibernation, tasks are frozen before devices are suspended. This assures that all user-space processes are in a stable state in which they do not hold any semaphores or mutexes, and they will not continue running until the PM core allows them. This mechanism was introduced with hibernation in mind, to prevent data from being written to disks after the hibernation image was created. Otherwise the on-disk data would not reflect the information preserved within the hibernation image, leading to corruption when the system resumed. Historically, Linux's support for hibernation has been much more robust than its support for STR, and drivers' suspend and resume callbacks were designed and tested with hibernation in mind. They generally expected tasks to be frozen before they were executed. Since the same callbacks were (and still are) used for both STR and hibernation, it became necessary to freeze tasks before STR as well as before hibernation.

The piece of code that freezes tasks is called the *freezer*. It is invoked by the PM core after the suspend notifiers are called (see Section 7) and just before the `.begin()` platform-suspend method is executed. It works by traversing the list of all tasks in the system and setting the `TIF_FREEZE` flag for the ones marked as freezable (i.e., those without the `PF_NOFREEZE` flag set).

It does this first for user-space tasks, calling `signal_`

[8] They also are called during resume from hibernation with `PM_RESTORE_PREPARE`, but we will not discuss that here.

wake_up() on each of them.[9] The code uses a busy loop in which the freezer checks if there still are any tasks with TIF_FREEZE set. The loop finishes when there are none left, or the only remaining ones also have the PF_FREEZER_SKIP flag set.[10]

The tasks for which TIF_FREEZE has been set are forced by the signal handler to call refrigerator(). This function unsets TIF_FREEZE, sets the PF_FROZEN flag for the current task, and puts it into the TASK_UNINTERRUPTIBLE state. The function will keep the task in this state as long as the PF_FROZEN flag is set; the PM core has to reset that flag before the task can do any more useful work. Thus, the tasks that have PF_FROZEN set and are inside the refrigerator() function are regarded as "frozen." As a result of the way in which refrigerator() is entered, the frozen tasks cannot hold any semaphores or mutexes, so it is generally safe to leave them in this state before suspending devices.

When all of the user-space tasks have been frozen, the freezer sets TIF_FREEZE for the remaining freezable tasks (i.e., freezable kernel threads). They also are supposed to enter refrigerator(). But while user-space tasks are made call refrigerator() by the generic signal-handling code, kernel threads have to call it explicitly in order to be frozen. Specifically, they must call the try_to_freeze() function in suitable places. Moreover, the freezer does not call fake_signal_wake_up() on them, since we do not want to send a fake signal to a kernel thread. Instead the freezer calls wake_up_state(p, TASK_INTERRUPTIBLE) on those tasks (where p is a pointer to the task's struct task_struct object). This causes the tasks to be woken up in case they are sleeping—but it also means that kernel threads in the TASK_UNINTERRUPTIBLE state cannot be frozen.[11]

Although freezing tasks may seem to be a simple mechanism, it has several problems. First of all, the main limitation of the freezer (inability to handle uninterruptible tasks) causes it to fail in many cases where we would like it to succeed. For example, if there is a task waiting on a filesystem lock in the TASK_UNINTERRUPTIBLE state and the lock cannot be released for a relatively long time due to a network error, the freezer will fail and the entire suspend will fail as a result. Second, the freezer does not work well with device drivers having a user-space component, because they may not be able to suspend devices after their user-space parts have been frozen. Third, freezing tasks occasionally takes too much time. It usually does not take more than several milliseconds, but in extreme cases (i.e., under a heavy load) it may take up to several seconds, which is way too much for various important usage scenarios. Finally, the approach used by the freezer to distinguish user-space processes from kernel threads is not optimal. It turns out that there are kernel threads which in fact behave like user-space processes and therefore should be frozen in the same way, by sending fake signals to them with signal_wake_up(). These threads often fail to call refrigerator() in a timely manner, causing the freezer to fail.

For these reasons, the kernel developers generally agree that the freezing of tasks should not be used during suspend. Whether it should be used during hibernation is not entirely clear, but some implementations of hibernation without freezing tasks are being actively discussed. In any case, there ought to be an alternative mechanism preventing user-space processes and kernel threads from accessing devices that are in a low-power state (i.e., after they have been suspended and before they are resumed). It is generally believed that device drivers should handle this, and for this purpose it will be necessary to rework the suspend and resume framework.

9 Proposed Framework for Suspending and Resuming Devices

As stated in Section 5, the current framework for suspending and resuming devices does not seem to be adequate. It is considered inflexible and generally difficult to use in some situations. It does not include any mechanisms allowing the PM core to protect its internal data structures from damage caused by inappropriate driver implementations.[12] It does not provide enough context information to resume callbacks. Finally, it may not be

[9] The freezer distinguishes user-space tasks from kernel threads on the basis of the task's mm pointer. If this pointer is NULL or has only been set temporarily, the task is regarded as a kernel thread; otherwise it is assumed to belong to user-space.

[10] This allows the freezer to handle some corner cases, such as the vfork() system call.

[11] This also applies to user-space processes in that state.

[12] For example, if a callback or notifier routine registers a new device object below a suspended parent, the ordering of the device list used by the PM core will be incorrect and the next suspend may fail as a result.

suitable when the freezing of tasks is removed and device drivers are made responsible for preventing access to suspended devices. Consequently a new framework for suspending and resuming devices is now being introduced [PATCHES].

The first problem solved by the new framework is the lack of separation between the suspend and hibernation callbacks, especially where the resume part is concerned. Within the current framework the same device-resume callbacks are used for both hibernation and suspend, and since they take only one argument (a pointer to the device structure), it is nearly impossible for them to determine the context in which they are invoked. This is a serious limitation leading to unnecessary complications in some cases, and it is going to be fixed by introducing separate device-resume callbacks for suspend and hibernation.[13] Likewise, separate device-suspend callbacks for suspend and hibernation will be introduced, so that the `pm_message_t` argument (used for determining the type of transition being carried out, see Section 5) will not be necessary any more.

```
struct pm_ops {
  int (*prepare)(struct device *dev);
  void (*complete)(struct device *dev);
  int (*suspend)(struct device *dev);
  int (*resume)(struct device *dev);
  int (*freeze)(struct device *dev);
  int (*thaw)(struct device *dev);
  int (*poweroff)(struct device *dev);
  int (*restore)(struct device *dev);
};
```

Figure 7: Proposed `struct pm_ops`

`struct pm_ops`, representing a set of device-suspend and -resume callbacks (including hibernation-specific callbacks), is defined as shown in Figure 7. Each device-class or device-type driver implementing these callbacks will provide the PM core with a pointer to one of these structures. Since bus-type drivers generally need to define special device-suspend and -resume callbacks to be executed with interrupts disabled, the extended `struct pm_ext_ops` structure detailed in Figure 8 is provided for their benefit.

Although the implementation of suspend and resume callbacks in device drivers may generally depend on the bus type, device type, and device class their devices belong to, it is strongly recommended to use `struct pm_ops` or `struct pm_ext_ops` objects. However, the legacy callback method pointers will remain available for the time being.

```
struct pm_ext_ops {
  struct pm_ops base;
  int (*suspend_noirq)(struct device *dev);
  int (*resume_noirq)(struct device *dev);
  int (*freeze_noirq)(struct device *dev);
  int (*thaw_noirq)(struct device *dev);
  int (*poweroff_noirq)(struct device *dev);
  int (*restore_noirq)(struct device *dev);
};
```

Figure 8: Proposed `struct pm_ext_ops`

The majority of callbacks provided by `struct pm_ops` and `struct pm_ext_ops` objects are hibernation-specific and we will not discuss them. We will focus on the callbacks that are STR-specific or common to both suspend and hibernation.

The `.prepare()` callback is intended for initial preparation of the driver for a power transition, without changing the hardware state of the device. Among other things, `.prepare()` should ensure that after it returns, no new children will be registered below the device. (*Un*-registering children is allowed at any time.) It is also recommended that `.prepare()` take steps to prevent potential race conditions between the suspend thread and any other threads. The `.prepare()` callbacks will be executed by the PM core for all devices before the `.suspend()` callback is invoked for any of them, so device drivers may generally assume that the other devices are functional while `.prepare()` is being run.[14] In particular, `GFP_KERNEL` memory allocations can safely be made. The `.prepare()` callbacks will be executed during suspend as well as during hibernation.[15]

The `.suspend()` callback is suspend-specific. It will be executed before the platform `.prepare()` method is called (see Section 5) and the non-boot CPUs are disabled. In this callback the device should be put

[13] In fact, two separate device-resume callbacks are necessary for hibernation: one to be called after creating an image and one to be called during the actual resume.

[14] However, user-space tasks will already be frozen, meaning that things like `request_firmware()` cannot be used. This limitation may be lifted in the future.

[15] During hibernation they will be executed before the image is created, and during resume from hibernation they will be executed before the contents of system memory are restored from the image.

into the appropriate low-power state and the device's wake-up mechanism should be enabled if necessary. Tasks must be prevented from accessing the device after `.suspend()` has run; attempts to do so must block until `.resume()` is called.

Some drivers will need to implement the `.suspend_noirq()` callback and its resume counterpart, `.resume_noirq()`. The role of these callbacks is to switch off and on, respectively, devices that are necessary for executing the platform methods `.prepare()` and `.finish()` or for disabling and re-enabling the non-boot CPUs. They should also be used for devices that cannot be suspended with interrupts enabled, such as APICs.[16]

The `.resume()` callback is the counterpart of `.suspend()`. It should put the device back into an operational state, according to the information saved in memory by the preceding `.suspend()`. After `.resume()` has run, the device driver starts working again, responding to hardware events and software requests.

The role of `.complete()` is to undo the changes made by the preceding `.prepare()`. In particular, new child devices that were plugged in while the system was suspended and detected during `.resume()` should not be registered until `.complete()` is called. It will be executed for all kinds of resume transitions, including resume-from-hibernation, as well as in cases when a suspend or hibernation transition fails. After `.complete()` has run, the device is regarded as fully functional by the PM core and its driver should handle all requests as appropriate. The `.complete()` callbacks for all devices will be executed after the last `.resume()` callback has returned, so drivers may generally assume the other devices to be functional while `.complete()` is being executed.

All of the callbacks described above except for `.complete()` return zero on success or a nonzero error code on failure. If `.prepare()`, `.suspend()`, or `.suspend_noirq()` returns an error code, the entire transition will be aborted. However the PM core is not able to handle errors returned by `.resume()` or `.resume_noirq()` in any meaningful manner, so they will only be printed to the system logs.[17]

It is expected that the addition of the `.prepare()` and `.complete()` callbacks will improve the flexibility of the suspend and resume framework. Most importantly, these callbacks will make it possible to separate preliminary actions that may depend on the other devices being accessible from the actions needed to stop the device and put it into a low-power state. They will also help to avoid some synchronization-related problems that can arise when the freezing of tasks is removed from the suspend code path. For example, drivers may use `.prepare()` to disable their user-space interfaces, such as *ioctls* and *sysfs* attributes, or put them into a degraded mode of operation, so that processes accessing the device cannot disturb the suspend thread.

Moreover, we expect that the introduction of the hibernation-specific callbacks and the elimination of the `pm_message_t` parameter will help driver authors to write more efficient power-management code. Since all of the callbacks related to suspend and hibernation are now going to be more specialized and the context in which they are invoked is going to be clearly defined, it should be easier to decide what operations are to be performed by a given callback and to avoid doing unnecessary things (such as putting a device into a low-power state before the hibernation image is created).

10 Suspend to RAM and Graphics Adapters

One of the most visible weaknesses of Linux's current implementation of suspend-to-RAM is the handling of graphics adapters. On many systems, after resume-from-RAM, the computer's graphics adapter is not functional or does not behave correctly. In the most extreme cases this may lead to system hangs during resume and to the appearance of many unusual failure modes. It is related to the fact that the way Linux handles graphics does not meet the expectations of hardware manufacturers.[18]

For a long time graphics has been handled entirely by the X server, which from the kernel's point of view is

[16] At present, APICs are represented by *sysdev* objects and are suspended after the regular devices. It is possible, however, that they will be represented by platform device objects in the future.

[17] To change this, the resume callbacks would have to be required to return error codes *only* in case of a critical failure. This currently is not possible, since some drivers return noncritical errors from their legacy resume callbacks. In any event, drivers have a better idea of what recovery options are feasible than the PM core does.

[18] This mostly applies to the vendors of notebooks.

simply a user-space process. Usually the X server uses its own graphics driver and accesses the registers of the graphics adapter directly. In such cases the kernel does not need to provide its own driver as well, and the X server is left in control. Normally this does not lead to any problems, but unfortunately with suspend-to-RAM it does.

When the system is put into STR, power is usually removed from the graphics adapter, causing it to forget its pre-suspend settings. Hence during the subsequent resume, it is necessary to reinitialize the graphics adapter so that it can operate normally. This may be done by the computer's BIOS (which gets control over the system when a wake-up event occurs) and it often is done that way on desktop systems. However, many laptop vendors tend to simplify their BIOSes by not implementing this reinitialization, because they expect the graphics driver provided by the operating system to take care of it. Of course this is not going to work on Linux systems where the kernel does not provide a graphics driver, because the X server is activated after devices have been resumed and that may be too late for it to reinitialize the graphics adapter, let alone restore its pre-suspend state. Furthermore X may not even have been running when the system was suspended.

The ultimate solution to this problem is to implement graphics drivers in the Linux kernel. At a minimum, graphics drivers should be split into two parts, one of which will reside in the kernel and will be responsible for interacting with the other kernel subsystems and for handling events such as a system-wide power transition. The other part of the driver may still live in the X server and may communicate with the first part via a well-defined interface. Although this idea is not new, it was difficult to realize owing to the lack of documentation for the most popular graphics adapters. Recently this situation has started to change, with first Intel and then AMD making their adapters' technical documentation available to kernel and X developers. As a result, `.suspend()` and `.resume()` callbacks have been implemented in the `i915` driver for Intel adapters, and it is now supposed to correctly reinitialize the adapter during resume-from-RAM.[19] It is expected that this ability will also be added to the graphics drivers for AMD/ATI adapters in the near future.

Still, there are many systems for which the graphics adapters are not reinitialized correctly during resume-from-RAM. Fortunately it was observed that the reinitialization could often be handled by a user-space wrapper executing some special BIOS code in 16-bit emulation mode. It turned out that even more things could be done in user-space to bring graphics adapters back to life during resume, and a utility called `vbetool` was created for this purpose.[20] At the same time, a utility for manipulating backlight on systems using ATI[21] graphics adapters, called `radeontool`, was created. These two programs were merged into a single utility called `s2ram`, which is a wrapper around the Linux kernel's `/sys/power/state` interface incorporating the graphics reinitialization schemes.[22]

Although `s2ram` is a very useful tool, it has one drawback: Different systems usually require different operations to be carried out in order to reinitialize their graphics adapters, and it is necessary to instruct `s2ram` what to do by means of command-line options. Moreover, every user has to figure out which options will work on her or his system, which often is tedious and can involve several failed suspend/resume cycles. For this reason `s2ram` contains a list of systems for which a working set of options is known, and the users of these systems should be able to suspend and resume their computers successfully using `s2ram` without any additional effort.[23]

11 Problems with Platforms

The flexibility given to platform designers by ACPI can be a source of serious problems. For example, if the ACPI Machine Language (AML) routines invoked before suspend are not implemented correctly or make unreasonable assumptions, their execution may fail and leave the system in an inconsistent state. Unfortunately the kernel has no choice but to execute the AML code, trusting that it will not do any harm. Of course if given platform is known to have problems, it can be blacklisted on the basis of its DMI[24] identification. Still, be-

[19] This driver is included in the 2.6.25 kernel.

[20] `vbetool` was written by Matthew Garrett.

[21] ATI was not a part of AMD at that time.

[22] The creator of `s2ram` is Pavel Machek, but many people have contributed to it since the first version was put together. `s2ram` is available from `http://suspend.sf.net`.

[23] `s2ram` matches computers against its list of known working systems based on the DMI information in the BIOS. The list is built from feedback provided by the `s2ram` users and is maintained by Stefan Seyfried.

[24] Desktop Management Interface.

fore blacklisting a system, the kernel developers have to know what kind of problems it experiences and what exactly to do to prevent them from happening. That, in turn, requires the users of those systems to report the problems and to take part in finding appropriate workarounds.

Moreover, problems may arise even if there is nothing wrong with the AML code. This happens, for example, if the kernel provides native drivers for devices that are also accessed from the AML routines, because the kernel has no means to determine which registers of a given device will be accessed by the AML code before actually executing that code. Thus, if the native driver accesses the device concurrently with an AML routine, some synchronization issues are likely to appear. It is difficult to solve those issues in a way general enough to be applicable to all systems and, again, blacklisting is necessary to make things work on the affected platforms.

Another major inconvenience related to ACPI platforms is that the requirements regarding STR changed between revisions of the ACPI specification. The suggested code ordering for suspend changed between ACPI 1.0 and ACPI 2.0, and there are systems for which only one of them is appropriate. Some of these systems fail to suspend or even crash if the code ordering implemented by the kernel is not the one they expect. Again, the kernel has no choice but to use one suspend code ordering by default and blacklist the systems that require the other one.[25]

Last but not least, testing the interactions between the kernel and a particular platform is problematic because it can be carried out on only a limited number of systems. Even if the kernel follows the ACPI specification and works on the systems available to its developers, it may very well fail to work on other systems having different implementations of the AML code in question. For this reason, it is very important that the users of STR test development kernels and immediately report any new STR-related problems, so that the developers can investigate and fix them before the new code is officially released.

12 Future Work

We've reached a level of stability where STR is useful on a large number of systems. We need to continue these stability efforts with the goal of broad, and ultimately universal, deployment.

But to make STR even more useful, we need to increase our focus on performance. We need tools to track STR performance such that performance is easily and widely measured, issues are easily identified, regressions are prevented, and benefits of tuning are permanent.

Linux needs a stable user/system API for device D-state control. D-states should be widely available to run-time device power management, which must inter-operate well with system sleep states. (Some progress in this area has already been made; a few subsystems, such as USB, can power down devices when they are not in active use.)

The wake-up API available in `sysfs` needs to be more widely used and better integrated with the platform wake-up mechanisms.

It is unclear that the existing CPU hot-plug infrastructure is ideally suited to system suspend, and alternatives should be sought.

We need to think about the work required for device drivers to properly implement suspend and make sure that the burden on driver authors is minimized. This applies to the API seen by individual device drivers as well as to the infrastructure provided by the upper-level device-class drivers.

13 How to Participate

STR in Linux is approaching a point where a critical mass of developers routinely use it, and thus test it. These developers often run the development and release-candidate kernels and thus immediately notice and report regressions. With tools such as `git-bisect` [GIT-OLS, GIT-URL], these developers are empowered to do an excellent job isolating issues, even if they never read or modify a single line of suspend-related code.

Please join them! For STR on Linux to reach the next level of success, it is critical that the Linux community assert that STR work properly on the systems that they have, and actively file bugs and help isolate regressions when STR does not meet expectations. The more active testers we have, the easier it will be for the community to successfully deploy STR on a broad range of systems.

[25] At present, the ACPI 2.0 ordering is used by default.

Also, note that there is now a dedicated kernel Bugzilla category for STR related issues: `http://bugzilla.kernel.org`, product *Power Management*, and Component *Hibernation/Suspend*.

Still another group in the community must be mobilized—driver authors. At this point, drivers are expected to include full suspend/resume support if they are used on systems that support suspend. A new driver should not be considered complete enough for inclusion in the kernel if it does not include suspend support.

14 Acknowledgments

Linux's suspend-to-RAM support is not a new idea; it has been evolving for years. We must all acknowledge that we are standing on the shoulders of the giants who came before us, and thank them for all they have done.

In particular, the authors would like to single out Pavel Machek of Novell/SuSE, co-maintainer of suspend and hibernation, who has been key to development and adoption. Also Andy Grover and Patrick Mochel, who set a lot of the foundation starting way back in Linux-2.4.

We thank Alan Stern for many valuable suggestions and for helping us to prepare the manuscript. We also thank Pavel Machek for valuable comments.

Finally, we thank the communities on the mailing lists `linux-pm@lists.linux-foundation.org` and `linux-acpi@vger.kernel.org`, where the actual work gets done.

References

[ACPI-SPEC] *Advanced Configuration and Power Interface Specification* (`http://www.acpi.info`).

[ACPICA] ACPICA project home page (`http://acpica.org`).

[PCI-PM] *PCI Bus Power Management Interface Specification* (`http://www.pcisig.com/specifications/conventional/`).

[ACPI-OLS] Brown, Keshavamurthy, Li, Moore, Pallipadi, Yu; *ACPI in Linux—Architecture, Advances, and Challenges*; In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, July 2005).

[ACPI-URL] Linux/ACPI project home page (`http://www.lesswatts.org/projects/acpi`).

[GIT-URL] GIT project home page (`http://git.or.cz`).

[GIT-OLS] J.C. Hamano, *GIT—A Stupid Content Tracker*, In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, July 2006).

[REPORT] R.J. Wysocki, *Suspend and Hibernation Status Report* (`http://lwn.net/Articles/243404`).

[PATCHES] R.J. Wysocki, *Separating Suspend and Hibernation* (`http://kerneltrap.org/Linux/Separating_Suspend_and_Hibernation`).