Power Management for USB

Author: Alan Stern <stern@rowland.harvard.edu>
Date: Last-updated: February 2014

What is Power Management?

Power Management (PM) is the practice of saving energy by suspending parts of a computer system when they aren't being used. While a component is `suspended` it is in a nonfunctional low-power state; it might even be turned off completely. A suspended component can be `resumed` (returned to a functional full-power state) when the kernel needs to use it. (There also are forms of PM in which components are placed in a less functional but still usable state instead of being suspended; an example would be reducing the CPU's clock rate. This document will not discuss those other forms.)

When the parts being suspended include the CPU and most of the rest of the system, we speak of it as a "system suspend". When a particular device is turned off while the system as a whole remains running, we call it a "dynamic suspend" (also known as a "runtime suspend" or "selective suspend"). This document concentrates mostly on how dynamic PM is implemented in the USB subsystem, although system PM is covered to some extent (see `Documentation/power/*.rst` for more information about system PM).

System PM support is present only if the kernel was built with `CONFIG_SUSPEND` or `CONFIG_HIBERNATION` enabled. Dynamic PM support

for USB is present whenever the kernel was built with `CONFIG_PM` enabled.

[Historically, dynamic PM support for USB was present only if the kernel had been built with `CONFIG_USB_SUSPEND` enabled (which depended on `CONFIG_PM_RUNTIME`). Starting with the 3.10 kernel release, dynamic PM support for USB was present whenever the kernel was built with `CONFIG_PM_RUNTIME` enabled. The `CONFIG_USB_SUSPEND` option had been eliminated.]

What is Remote Wakeup?

When a device has been suspended, it generally doesn't resume until the computer tells it to. Likewise, if the entire computer has been suspended, it generally doesn't resume until the user tells it to, say by pressing a power button or opening the cover.

However some devices have the capability of resuming by themselves, or asking the kernel to resume them, or even telling the entire computer to resume. This capability goes by several names such as "Wake On LAN"; we will refer to it generically as "remote wakeup". When a device is enabled for remote wakeup and it is suspended, it may resume itself (or send a request to be resumed) in response to some external event. Examples include a suspended keyboard resuming when a key is pressed, or a suspended USB hub resuming when a device is plugged in.

When is a USB device idle?

A device is idle whenever the kernel thinks it's not busy doing anything important and thus is a candidate for being suspended. The exact definition depends on the device's driver; drivers are allowed to declare that a device isn't idle even when there's no actual communication taking place. (For example, a hub isn't considered idle unless all the devices plugged into that hub are already suspended.) In addition, a device isn't considered idle so long as a program keeps its usbfs file open, whether or not any I/O is going on.

If a USB device has no driver, its usbfs file isn't open, and it isn't being accessed through sysfs, then it definitely is idle.

Forms of dynamic PM

Dynamic suspends occur when the kernel decides to suspend an idle device. This is called `autosuspend` for short. In general, a device won't be autosuspended unless it has been idle for some minimum period of time, the so-called idle-delay time.

Of course, nothing the kernel does on its own initiative should prevent the computer or its devices from working properly. If a device has been autosuspended and a program tries to use it, the kernel will automatically resume the device (autoresume). For the same reason, an autosuspended device will usually have remote wakeup enabled, if the device supports remote wakeup.

It is worth mentioning that many USB drivers don't support autosuspend. In fact, at the time of this writing (Linux 2.6.23) the only drivers which do support it are the hub driver, kaweth, asix, usblp, usblcd, and usb-skeleton (which doesn't count). If a non-supporting driver is bound to a device, the device won't be autosuspended. In effect, the kernel pretends the device is never idle.

We can categorize power management events in two broad classes: external and internal. External events are those triggered by some agent outside the USB stack: system suspend/resume (triggered by userspace), manual dynamic resume (also triggered by userspace), and remote wakeup (triggered by the device). Internal events are those triggered within the USB stack: autosuspend and autoresume. Note that all dynamic suspend events are internal; external agents are not allowed to issue dynamic suspends.

The user interface for dynamic PM

The user interface for controlling dynamic PM is located in the `power/` subdirectory of each USB device's sysfs directory, that is, in `/sys/bus/usb/devices/.../power/` where "..." is the device's ID. The relevant attribute files are: wakeup, control, and `autosuspend_delay_ms`. (There may also be a file named `level`; this file was deprecated as of the 2.6.35 kernel and replaced by the `control` file. In 2.6.38 the `autosuspend` file will be deprecated and replaced by the `autosuspend_delay_ms` file. The only difference is that the newer file expresses the delay in milliseconds whereas the older file uses seconds. Confusingly, both files are present in 2.6.37 but only `autosuspend` works.)

`power/wakeup`

> This file is empty if the device does not support remote wakeup. Otherwise the file contains either the word `enabled` or the word `disabled`, and you can write those words to the file. The setting determines whether or not remote wakeup will be enabled when the device is next suspended. (If the setting is changed while the device is suspended, the change won't take effect until the following suspend.)

`power/control`

> This file contains one of two words: `on` or `auto`. You can write those words to the file to change the device's setting.
>
> - `on` means that the device should be resumed and autosuspend is not allowed. (Of course, system suspends are still allowed.)
> - `auto` is the normal state in which the kernel is allowed to autosuspend and autoresume the device.
>
> (In kernels up to 2.6.32, you could also specify `suspend`, meaning that the device should remain suspended and autoresume was not allowed. This setting is no longer supported.)

`power/autosuspend_delay_ms`

> This file contains an integer value, which is the number of milliseconds the device should remain idle before the kernel will autosuspend it (the idle-delay time). The default is 2000. 0 means to autosuspend as soon as the device becomes idle, and negative values mean never to autosuspend. You can write a number to the file to change the autosuspend idle-delay time.

Writing `-1` to `power/autosuspend_delay_ms` and writing `on` to `power/control` do essentially the same thing -- they both prevent the device from being autosuspended. Yes, this is a redundancy in the API.

(In 2.6.21 writing `0` to `power/autosuspend` would prevent the device from being autosuspended; the behavior was changed in 2.6.22. The `power/autosuspend` attribute did not exist prior to 2.6.21, and the `power/level` attribute did not exist prior to 2.6.22. `power/control` was added in 2.6.34, and `power/autosuspend_delay_ms` was added in 2.6.37 but did not become functional until 2.6.38.)

Changing the default idle-delay time

The default autosuspend idle-delay time (in seconds) is controlled by a module parameter in usbcore. You can specify the value when usbcore is loaded. For example, to set it to 5 seconds instead of 2 you would do:

```
modprobe usbcore autosuspend=5
```

Equivalently, you could add to a configuration file in /etc/modprobe.d a line saying:

```
options usbcore autosuspend=5
```

Some distributions load the usbcore module very early during the boot process, by means of a program or script running from an initramfs image. To alter the parameter value you would have to rebuild that image.

If usbcore is compiled into the kernel rather than built as a loadable module, you can add:

```
usbcore.autosuspend=5
```

to the kernel's boot command line.

Finally, the parameter value can be changed while the system is running. If you do:

```
echo 5 >/sys/module/usbcore/parameters/autosuspend
```

then each new USB device will have its autosuspend idle-delay initialized to 5. (The idle-delay values for already existing devices will not be affected.)

Setting the initial default idle-delay to -1 will prevent any autosuspend of any USB device. This has the benefit of allowing you then to enable autosuspend for selected devices.

Warnings

The USB specification states that all USB devices must support power management. Nevertheless, the sad fact is that many devices do not support it very well. You can suspend them all right, but when you try to resume them they disconnect themselves from the USB bus or they stop working entirely. This seems to be especially prevalent among printers and scanners, but plenty of other types of device have the same deficiency.

For this reason, by default the kernel disables autosuspend (the `power/control` attribute is initialized to `on`) for all devices other than hubs. Hubs, at least, appear to be reasonably well-behaved in this regard.

(In 2.6.21 and 2.6.22 this wasn't the case. Autosuspend was enabled by default for almost all USB devices. A number of people experienced problems as a result.)

This means that non-hub devices won't be autosuspended unless the user or a program explicitly enables it. As of this writing there aren't any widespread programs which will do this; we hope that in the near future device managers such as HAL will take on this added responsibility. In the meantime you can always carry out the necessary operations by hand or add them to a udev script. You can also change the idle-delay time; 2 seconds is not the best choice for every device.

If a driver knows that its device has proper suspend/resume support, it can enable autosuspend all by itself. For example, the video driver for a laptop's webcam might do this (in recent kernels they do), since these devices are rarely used and so should normally be autosuspended.

Sometimes it turns out that even when a device does work okay with autosuspend there are still problems. For example, the usbhid driver, which manages keyboards and mice, has autosuspend support. Tests with a number of keyboards show that typing on a suspended keyboard, while causing the keyboard to do a remote wakeup all right, will nonetheless frequently result in lost keystrokes. Tests with mice show that some of them will issue a remote-wakeup request in response to button presses but not to motion, and some in response to neither.

The kernel will not prevent you from enabling autosuspend on devices that can't handle it. It is even possible in theory to damage a device by suspending it at the wrong time. (Highly unlikely, but possible.) Take care.

The driver interface for Power Management

The requirements for a USB driver to support external power management are pretty modest; the driver need only define:

```
.suspend
.resume
.reset_resume
```

methods in its :c:type:`usb_driver` structure, and the `reset_resume` method is optional. The methods' jobs are quite simple:

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, line 302); *backlink***
>
> Unknown interpreted text role "c:type".

- The `suspend` method is called to warn the driver that the device is going to be suspended. If the driver returns a negative error code, the suspend will be aborted. Normally the driver will return 0, in which case it must cancel all outstanding URBs (:c:func:`usb_kill_urb`) and not submit any more.

  > **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, line 305); *backlink***
  >
  > Unknown interpreted text role "c:func".

- The `resume` method is called to tell the driver that the device has been resumed and the driver can return to normal operation. URBs may once more be submitted.
- The `reset_resume` method is called to tell the driver that the device has been resumed and it also has been reset. The driver should redo any necessary device initialization, since the device has probably lost most or all of its state (although the interfaces will be in the same altsettings as before the suspend).

If the device is disconnected or powered down while it is suspended, the `disconnect` method will be called instead of the `resume` or `reset_resume` method. This is also quite likely to happen when waking up from hibernation, as many systems do not maintain suspend current to the USB host controllers during hibernation. (It's possible to work around the hibernation-forces-disconnect problem by using the USB Persist facility.)

The `reset_resume` method is used by the USB Persist facility (see :ref:`usb-persist`) and it can also be used under certain circumstances when `CONFIG_USB_PERSIST` is not enabled. Currently, if a device is reset during a resume and the driver does not have a `reset_resume` method, the driver won't receive any notification about the resume. Later kernels will call the driver's `disconnect` method; 2.6.23 doesn't do this.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, line 330); *backlink*

Unknown interpreted text role "ref".

USB drivers are bound to interfaces, so their `suspend` and `resume` methods get called when the interfaces are suspended or resumed. In principle one might want to suspend some interfaces on a device (i.e., force the drivers for those interface to stop all activity) without suspending the other interfaces. The USB core doesn't allow this; all interfaces are suspended when the device itself is suspended and all interfaces are resumed when the device is resumed. It isn't possible to suspend or resume some but not all of a device's interfaces. The closest you can come is to unbind the interfaces' drivers.

The driver interface for autosuspend and autoresume

To support autosuspend and autoresume, a driver should implement all three of the methods listed above. In addition, a driver indicates that it supports autosuspend by setting the `.supports_autosuspend` flag in its usb_driver structure. It is then responsible for informing the USB core whenever one of its interfaces becomes busy or idle. The driver does so by calling these six functions:

```
int  usb_autopm_get_interface(struct usb_interface *intf);
void usb_autopm_put_interface(struct usb_interface *intf);
int  usb_autopm_get_interface_async(struct usb_interface *intf);
void usb_autopm_put_interface_async(struct usb_interface *intf);
void usb_autopm_get_interface_no_resume(struct usb_interface *intf);
void usb_autopm_put_interface_no_suspend(struct usb_interface *intf);
```

The functions work by maintaining a usage counter in the usb_interface's embedded device structure. When the counter is > 0 then the interface is deemed to be busy, and the kernel will not autosuspend the interface's device. When the usage counter is = 0 then the interface is considered to be idle, and the kernel may autosuspend the device.

Drivers must be careful to balance their overall changes to the usage counter. Unbalanced "get"s will remain in effect when a driver is unbound from its interface, preventing the device from going into runtime suspend should the interface be bound to a driver again. On the other hand, drivers are allowed to achieve this balance by calling the `usb_autopm_*` functions even after their `disconnect` routine has returned -- say from within a work-queue routine -- provided they retain an active reference to the interface (via `usb_get_intf` and `usb_put_intf`).

Drivers using the async routines are responsible for their own synchronization and mutual exclusion.

:c:func:`usb_autopm_get_interface` increments the usage counter and does an autoresume if the device is suspended. If the autoresume fails, the counter is decremented back.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, line 386); *backlink*

Unknown interpreted text role "c:func".

:c:func:`usb_autopm_put_interface` decrements the usage counter and attempts an autosuspend if the new value is = 0.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, line 390); *backlink*

Unknown interpreted text role "c:func".

:c:func:`usb_autopm_get_interface_async` and :c:func:`usb_autopm_put_interface_async` do almost the same things as their non-async counterparts. The big difference is that they use a workqueue to do the resume or suspend part of their jobs. As a result they can be called in an atomic context, such as an URB's completion handler, but when they return the device will generally not yet be in the desired state.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, line 393); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, line 393); *backlink*

Unknown interpreted text role "c:func".

:c:func:`usb_autopm_get_interface_no_resume` and :c:func:`usb_autopm_put_interface_no_suspend` merely increment or decrement the usage counter; they do not attempt to carry out an autoresume or an autosuspend. Hence they can be called in an atomic context.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, **line 401**); ***backlink***

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, **line 401**); ***backlink***

Unknown interpreted text role "c:func".

The simplest usage pattern is that a driver calls :c:func:`usb_autopm_get_interface` in its open routine and :c:func:`usb_autopm_put_interface` in its close or release routine. But other patterns are possible.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, **line 407**); ***backlink***

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, **line 407**); ***backlink***

Unknown interpreted text role "c:func".

The autosuspend attempts mentioned above will often fail for one reason or another. For example, the `power/control` attribute might be set to `on`, or another interface in the same device might not be idle. This is perfectly normal. If the reason for failure was that the device hasn't been idle for long enough, a timer is scheduled to carry out the operation automatically when the autosuspend idle-delay has expired.

Autoresume attempts also can fail, although failure would mean that the device is no longer present or operating properly. Unlike autosuspend, there's no idle-delay for an autoresume.

Other parts of the driver interface

Drivers can enable autosuspend for their devices by calling:

```
usb_enable_autosuspend(struct usb_device *udev);
```

in their :c:func:`probe` routine, if they know that the device is capable of suspending and resuming correctly. This is exactly equivalent to writing `auto` to the device's `power/control` attribute. Likewise, drivers can disable autosuspend by calling:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, **line 432**); ***backlink***

Unknown interpreted text role "c:func".

```
usb_disable_autosuspend(struct usb_device *udev);
```

This is exactly the same as writing `on` to the `power/control` attribute.

Sometimes a driver needs to make sure that remote wakeup is enabled during autosuspend. For example, there's not much point autosuspending a keyboard if the user can't cause the keyboard to do a remote wakeup by typing on it. If the driver sets `intf->needs_remote_wakeup` to 1, the kernel won't autosuspend the device if remote wakeup isn't available. (If the device is already autosuspended, though, setting this flag won't cause the kernel to autoresume it. Normally a driver would set this flag in its `probe` method, at which time the device is guaranteed not to be autosuspended.)

If a driver does its I/O asynchronously in interrupt context, it should call :c:func:`usb_autopm_get_interface_async` before starting output and :c:func:`usb_autopm_put_interface_async` when the output queue drains. When it receives an input event, it should call:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master] [Documentation] [driver-api] [usb]power-management.rst`, line 452); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master] [Documentation] [driver-api] [usb]power-management.rst`, line 452); *backlink*

Unknown interpreted text role "c:func".

```
usb_mark_last_busy(struct usb_device *udev);
```

in the event handler. This tells the PM core that the device was just busy and therefore the next autosuspend idle-delay expiration should be pushed back. Many of the usb_autopm_* routines also make this call, so drivers need to worry only when interrupt-driven input arrives.

Asynchronous operation is always subject to races. For example, a driver may call the :c:func:`usb_autopm_get_interface_async` routine at a time when the core has just finished deciding the device has been idle for long enough but not yet gotten around to calling the driver's `suspend` method. The `suspend` method must be responsible for synchronizing with the I/O request routine and the URB completion handler; it should cause autosuspends to fail with -EBUSY if the driver needs to use the device.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master] [Documentation] [driver-api] [usb]power-management.rst`, line 464); *backlink*

Unknown interpreted text role "c:func".

External suspend calls should never be allowed to fail in this way, only autosuspend calls. The driver can tell them apart by applying the :c:func:`PMSG_IS_AUTO` macro to the message argument to the `suspend` method; it will return True for internal PM events (autosuspend) and False for external PM events.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master] [Documentation] [driver-api] [usb]power-management.rst`, line 473); *backlink*

Unknown interpreted text role "c:func".

Mutual exclusion

For external events -- but not necessarily for autosuspend or autoresume -- the device semaphore (udev->dev.sem) will be held when a `suspend` or `resume` method is called. This implies that external suspend/resume events are mutually exclusive with calls to `probe`, `disconnect`, `pre_reset`, and `post_reset`; the USB core guarantees that this is true of autosuspend/autoresume events as well.

If a driver wants to block all suspend/resume calls during some critical section, the best way is to lock the device and call :c:func:`usb_autopm_get_interface` (and do the reverse at the end of the critical section). Holding the device semaphore will block all external PM calls, and the :c:func:`usb_autopm_get_interface` will prevent any internal PM calls, even if it fails. (Exercise: Why?)

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master] [Documentation] [driver-api] [usb]power-management.rst`, line 490); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master] [Documentation] [driver-api] [usb]power-management.rst`, line 490); *backlink*

Unknown interpreted text role "c:func".

Interaction between dynamic PM and system PM

Dynamic power management and system power management can interact in a couple of ways.

Firstly, a device may already be autosuspended when a system suspend occurs. Since system suspends are supposed to be as transparent as possible, the device should remain suspended following the system resume. But this theory may not work out well in practice; over time the kernel's behavior in this regard has changed. As of 2.6.37 the policy is to resume all devices during a system resume and let them handle their own runtime suspends afterward.

Secondly, a dynamic power-management event may occur as a system suspend is underway. The window for this is short, since system suspends don't take long (a few seconds usually), but it can happen. For example, a suspended device may send a remote-wakeup signal while the system is suspending. The remote wakeup may succeed, which would cause the system suspend to abort. If the remote wakeup doesn't succeed, it may still remain active and thus cause the system to resume as soon as the system suspend is complete. Or the remote wakeup may fail and get lost. Which outcome occurs depends on timing and on the hardware and firmware design.

xHCI hardware link PM

xHCI host controller provides hardware link power management to usb2.0 (xHCI 1.0 feature) and usb3.0 devices which support link PM. By enabling hardware LPM, the host can automatically put the device into lower power state(L1 for usb2.0 devices, or U1/U2 for usb3.0 devices), which state device can enter and resume very quickly.

The user interface for controlling hardware LPM is located in the `power/` subdirectory of each USB device's sysfs directory, that is, in `/sys/bus/usb/devices/.../power/` where "..." is the device's ID. The relevant attribute files are `usb2_hardware_lpm` and `usb3_hardware_lpm`.

`power/usb2_hardware_lpm`

> When a USB2 device which support LPM is plugged to a xHCI host root hub which support software LPM, the host will run a software LPM test for it; if the device enters L1 state and resume successfully and the host supports USB2 hardware LPM, this file will show up and driver will enable hardware LPM for the device. You can write y/Y/1 or n/N/0 to the file to enable/disable USB2 hardware LPM manually. This is for test purpose mainly.

`power/usb3_hardware_lpm_u1` `power/usb3_hardware_lpm_u2`

> When a USB 3.0 lpm-capable device is plugged in to a xHCI host which supports link PM, it will check if U1 and U2 exit latencies have been set in the BOS descriptor; if the check is passed and the host supports USB3 hardware LPM, USB3 hardware LPM will be enabled for the device and these files will be created. The files hold a string value (enable or disable) indicating whether or not USB3 hardware LPM U1 or U2 is enabled for the device.

USB Port Power Control

In addition to suspending endpoint devices and enabling hardware controlled link power management, the USB subsystem also has the capability to disable power to ports under some conditions. Power is controlled through `Set/ClearPortFeature(PORT_POWER)` requests to a hub. In the case of a root or platform-internal hub the host controller driver translates `PORT_POWER` requests into platform firmware (ACPI) method calls to set the port power state. For more background see the Linux Plumbers Conference 2012 slides [1] and video [2]:

Upon receiving a `ClearPortFeature(PORT_POWER)` request a USB port is logically off, and may trigger the actual loss of VBUS to the port [3]. VBUS may be maintained in the case where a hub gangs multiple ports into a shared power well causing power to remain until all ports in the gang are turned off. VBUS may also be maintained by hub ports configured for a charging application. In any event a logically off port will lose connection with its device, not respond to hotplug events, and not respond to remote wakeup events.

> **Warning**
>
> turning off a port may result in the inability to hot add a device. Please see "User Interface for Port Power Control" for details.

As far as the effect on the device itself it is similar to what a device goes through during system suspend, i.e. the power session is lost. Any USB device or driver that misbehaves with system suspend will be similarly affected by a port power cycle event. For this reason the implementation shares the same device recovery path (and honors the same quirks) as the system resume path for the hub.

[1] http://dl.dropbox.com/u/96820575/sarah-sharp-lpt-port-power-off2-mini.pdf

[2] http://linuxplumbers.ubicast.tv/videos/usb-port-power-off-kerneluserspace-api/

[3] USB 3.1 Section 10.12

wakeup note: if a device is configured to send wakeup events the port power control implementation will block poweroff attempts on that port.

User Interface for Port Power Control

The port power control mechanism uses the PM runtime system. Poweroff is requested by clearing the `power/pm_qos_no_power_off` flag of the port device (defaults to 1). If the port is disconnected it will immediately receive a `ClearPortFeature(PORT_POWER)` request. Otherwise, it will honor the pm runtime rules and require the attached child device and all descendants to be suspended. This mechanism is dependent on the hub advertising port power switching in its hub descriptor (wHubCharacteristics logical power switching mode field).

Note, some interface devices/drivers do not support autosuspend. Userspace may need to unbind the interface drivers before the :c:type:`usb_device` will suspend. An unbound interface device is suspended by default. When unbinding, be careful to unbind interface drivers, not the driver of the parent usb device. Also, leave hub interface drivers bound. If the driver for the usb device (not interface) is unbound the kernel is no longer able to resume the device. If a hub interface driver is unbound, control of its child ports is lost and all attached child-devices will disconnect. A good rule of thumb is that if the 'driver/module' link for a device points to `/sys/module/usbcore` then unbinding it will interfere with port power control.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]power-management.rst`, line 624); *backlink***
>
> Unknown interpreted text role "c:type".

Example of the relevant files for port power control. Note, in this example these files are relative to a usb hub device (prefix):

```
prefix=/sys/devices/pci0000:00/0000:00:14.0/usb3/3-1

                 attached child device +
             hub port device +         |
hub interface device +       |         |
                     v       v         v
             $prefix/3-1:1.0/3-1-port1/device

$prefix/3-1:1.0/3-1-port1/power/pm_qos_no_power_off
$prefix/3-1:1.0/3-1-port1/device/power/control
$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intf0>/driver/unbind
$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intf1>/driver/unbind
...
$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intfN>/driver/unbind
```

In addition to these files some ports may have a 'peer' link to a port on another hub. The expectation is that all superspeed ports have a hi-speed peer:

```
$prefix/3-1:1.0/3-1-port1/peer -> ../../../../usb2/2-1/2-1:1.0/2-1-port1
../../../../usb2/2-1/2-1:1.0/2-1-port1/peer -> ../../../../usb3/3-1/3-1:1.0/3-1-port1
```

Distinct from 'companion ports', or 'ehci/xhci shared switchover ports' peer ports are simply the hi-speed and superspeed interface pins that are combined into a single usb3 connector. Peer ports share the same ancestor XHCI device.

While a superspeed port is powered off a device may downgrade its connection and attempt to connect to the hi-speed pins. The implementation takes steps to prevent this:

1. Port suspend is sequenced to guarantee that hi-speed ports are powered-off before their superspeed peer is permitted to power-off. The implication is that the setting `pm_qos_no_power_off` to zero on a superspeed port may not cause the port to power-off until its highspeed peer has gone to its runtime suspend state. Userspace must take care to order the suspensions if it wants to guarantee that a superspeed port will power-off.
2. Port resume is sequenced to force a superspeed port to power-on prior to its highspeed peer.
3. Port resume always triggers an attached child device to resume. After a power session is lost the device may have been removed, or need reset. Resuming the child device when the parent port regains power resolves those states and clamps the maximum port power cycle frequency at the rate the child device can suspend (autosuspend-delay) and resume (reset-resume latency).

Sysfs files relevant for port power control:

`<hubdev-portX>/power/pm_qos_no_power_off`:

> This writable flag controls the state of an idle port. Once all children and descendants have suspended the port may suspend/poweroff provided that pm_qos_no_power_off is '0'. If pm_qos_no_power_off is '1' the port will remain active/powered regardless of the stats of descendants. Defaults to 1.

`<hubdev-portX>/power/runtime_status`:

> This file reflects whether the port is 'active' (power is on) or 'suspended' (logically off). There is no indication to userspace whether VBUS is still supplied.

`<hubdev-portX>/connect_type`:

> An advisory read-only flag to userspace indicating the location and connection type of the port. It returns one of

four values 'hotplug', 'hardwired', 'not used', and 'unknown'. All values, besides unknown, are set by platform firmware.

`hotplug` indicates an externally connectable/visible port on the platform. Typically userspace would choose to keep such a port powered to handle new device connection events.

`hardwired` refers to a port that is not visible but connectable. Examples are internal ports for USB bluetooth that can be disconnected via an external switch or a port with a hardwired USB camera. It is expected to be safe to allow these ports to suspend provided pm_qos_no_power_off is coordinated with any switch that gates connections. Userspace must arrange for the device to be connected prior to the port powering off, or to activate the port prior to enabling connection via a switch.

`not used` refers to an internal port that is expected to never have a device connected to it. These may be empty internal ports, or ports that are not physically exposed on a platform. Considered safe to be powered-off at all times.

`unknown` means platform firmware does not provide information for this port. Most commonly refers to external hub ports which should be considered 'hotplug' for policy decisions.

Note

- since we are relying on the BIOS to get this ACPI information correct, the USB port descriptions may be missing or wrong.
- Take care in clearing `pm_qos_no_power_off`. Once power is off this port will not respond to new connect events.

Once a child device is attached additional constraints are applied before the port is allowed to poweroff.

`<child>/power/control`:
: Must be `auto`, and the port will not power down until `<child>/power/runtime_status` reflects the 'suspended' state. Default value is controlled by child device driver.

`<child>/power/persist`:
: This defaults to `1` for most devices and indicates if kernel can persist the device's configuration across a power session loss (suspend / port-power event). When this value is `0` (quirky devices), port poweroff is disabled.

`<child>/driver/unbind`:
: Wakeup capable devices will block port poweroff. At this time the only mechanism to clear the usb-internal wakeup-capability for an interface device is to unbind its driver.

Summary of poweroff pre-requisite settings relative to a port device:

```
echo 0 > power/pm_qos_no_power_off
echo 0 > peer/power/pm_qos_no_power_off # if it exists
echo auto > power/control # this is the default value
echo auto > <child>/power/control
echo 1 > <child>/power/persist # this is the default value
```

Suggested Userspace Port Power Policy

As noted above userspace needs to be careful and deliberate about what ports are enabled for poweroff.

The default configuration is that all ports start with `power/pm_qos_no_power_off` set to `1` causing ports to always remain active.

Given confidence in the platform firmware's description of the ports (ACPI _PLD record for a port populates 'connect_type') userspace can clear pm_qos_no_power_off for all 'not used' ports. The same can be done for 'hardwired' ports provided poweroff is coordinated with any connection switch for the port.

A more aggressive userspace policy is to enable USB port power off for all ports (set `<hubdev-portX>/power/pm_qos_no_power_off` to `0`) when some external factor indicates the user has stopped interacting with the system. For example, a distro may want to enable power off all USB ports when the screen blanks, and re-power them when the screen becomes active. Smart phones and tablets may want to power off USB ports when the user pushes the power button.