QNX® Momentics® Development Suite

Power Management Toolkit
User's Guide

For QNX® Neutrino® 6.3

Printed under license by:

QNX Software Systems Co.
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: **info@qnx.com**
Web: **http://www.qnx.com/**

Electronic edition published 2005

Technical support options

To obtain technical support for any QNX product, visit the **Technical Support** section in the **Services** area on our website (**www.qnx.com**). You'll find a wide range of support options, including our free web-based **Developer Support Center**.

Contents

About This Guide

Purpose

The QNX Momentics power management *User's Guide* describes the power management architecture and explains how to implement a power management policy for your Neutrino-based embedded system. The power management feature is now part of the Neutrino realtime operating system (RTOS). The main purpose of this book is to help you design and implement a power managed system using the power management architecture.

The *User's Guide* is a combination "how-to" and API reference. It helps you take advantage of Neutrino's power-management technology to develop your own Neutrino-based embedded systems.

The following table may help you find information quickly in this guide:

If you want to:	Go to:
Find the introduction	What is Power Management?
Find the power management architecture	Power Management Architecture
Know the low-level power management	Low-level Power Management
Know about the power manager clients	Power Manager Clients

continued...

If you want to:	Go to:
Know implementing the device driver	Implementing Power Management in a Device Driver
Know how to implement a system power management policy	Implementing a System Power Management Policy
Know about the client and server libraries	API Reference
Know about the callbacks	Callback Reference

Who should use this guide?

This guide is intended for the following users:

- device driver writers
- BSP developers
- low-level system power management developers
- power manager developers
- application developers.

Chapter 1

What is Power Management?

In this chapter. . .

Introduction

Monitoring the power usage of peripherals and controlling its use is a unique challenge for today's electronic and embedded system. QNX neutrino provides all the necessary tools and interfaces (APIs) for controlling the power usage of an embedded system. The most salient feature of QNX Neutrino's power management feature is that users can define their own policy.

Throughout this guide, we often refer to the term *power management* and the power management *policy*. Therefore, in this chapter, we describe in brief what we mean by power management as well as its policy and goals. At the end of this chapter, we describe two example systems and their strategies for power management.

Power management explained

Power management is a dynamic computing capability that controls the power use of any device in the system, including add-on boards, such as network adapters or sound cards, as well as RF or Bluetooth devices, navigational aids, printers, keyboards, modems etc. The control procedure, i.e. what determines how and when to save energy, is called the power management policy.

In order to achieve the above purpose, power management addresses the power and performance level of each type of resources in a system. By "system" we refer to electronic systems that are heterogeneous in nature, for example:

- digital (CPU, Micro-controllers) and analog (e.g. RF) circuitry
- semiconductors (RAM, and FLASH memories)
- electronic-mechanical storage (e.g. disks) devices
- electro-optical (displays) human interfaces.

Goals of power management

The purpose for implementing a power management policy is to:

- extend operational life time
- lower power consumption
- provide graceful performance degradation
- adapt power dissipation satisfying environmental constraints.

In addition to the above, QNX Neutrino emphasizes several other policy attributes, such as demanding critical response time or maintaining operational “readiness” over a prolonged period of time.

Examples of power managed systems

There are two distinct categories of systems where power management policy can be applied effectively to control power usage. In one category, a significant class of systems has common characteristics including the following important set:

- extremely limited, diminishing power source
- timeliness constraints
- extended operational lifetime
- schedule of periodic activities.

These characteristics are perhaps best exemplified by in-car computing systems. In a vehicle, a dozen or more secondary-switched devices may continue to draw standby current from the battery when the ignition is off. Under these circumstances, it is necessary to restrict the ignition-off draw of each such device to ensure that the reserve capacity of the battery is not exceeded over a specified interval of time. This ensures that the primary purpose of the battery — to crank the engine — is maintained in the presence of any parasitic load from devices.

The importance of the other category grew due to the popularity of laptops, mobile devices, and PDAs. Here the central theme is reducing power consumption through some kind of internal or external events such as:

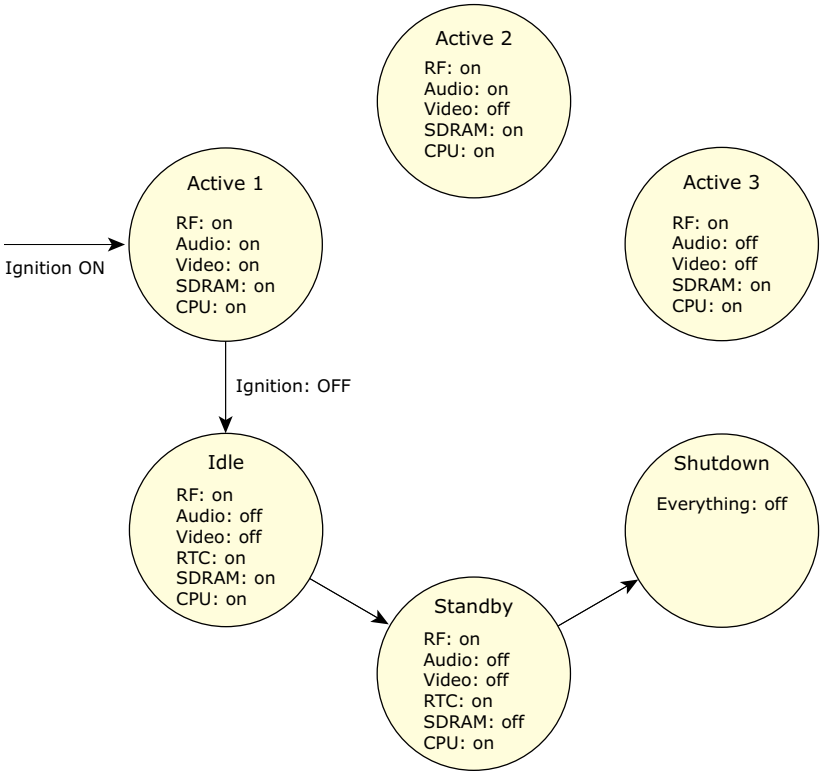
- battery level monitoring, while the system is running
- temperature variation as detected by a transducer.

In-car telematics system

In this representative example, power conservation is needed when the system is in “off” state, i.e. ignition is off. This is in sharp contrast to the general purpose system where dynamic power management is carried out for an “on” system. The resources we consider include CPU (and SDRAM memory) along with the associated hardware and other devices as follows:

- RF device (e.g. Bluetooth connectivity to the cellular phone)
- audio
- video.

Although the ignition and door locks are the actuators for this system, they are not power managed devices.



Different power modes of an in-car telematics system.

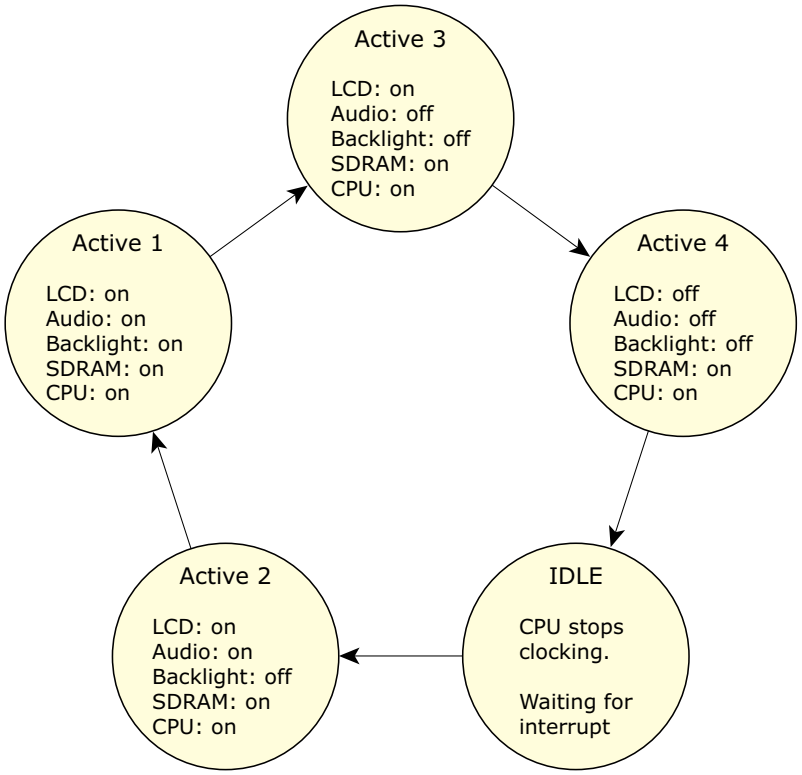
The diagram above shows that the power management policy progressively turns off the devices by transitioning through idle, standby (sleep) or shutdown states. This serves the purpose of restricting standby current drawn from the battery gradually down when the ignition is off. The system can also maintain its readiness to come alive on short notice. For example, the realtime clock (RTC) may be used for timed wakeup out of CPU sleep mode.

Hand-held computer

A hand-held computer is a representative example of a general purpose system where dynamic power management is carried out for

an “on” system. In a hand-held computer, battery life is an important issue, because data will be lost if the battery runs out of power. The following factors will shorten battery life:

- constant use of the back-light
- playing audio files through the speaker (the headphone jack consumes less power).



Different power modes of a hand-held computer.

The diagram above shows different power modes of a hand-held computer. The strategies for reducing power consumption are:

- scaling down the CPU clock to reduce power

- spinning down disks during periods of inactivity
- turning off screen if there is no user-input
- turning off power to audio chip when we don’t need it.

Chapter 2

Power Management Architecture

In this chapter...

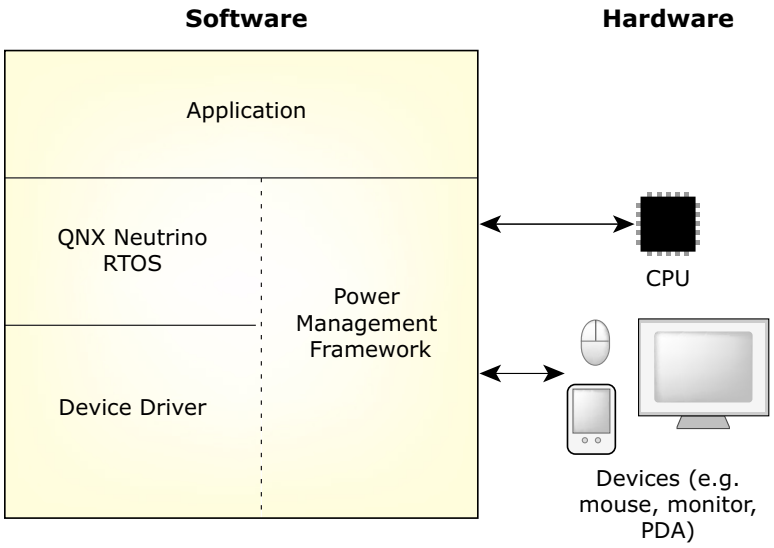
Introduction

This chapter describes the power management framework along with two different approaches for power management support. It also provides the major components of the power management architecture. In addition, it includes a discussion on power managed entities, power modes (states), and power managed objects.

QNX Neutrino power management framework

The power management framework is included as part of the QNX Neutrino RTOS. It is based on the principle of providing mechanisms — *not* policies — to control and manage power. You use these mechanisms to implement your own power management policy, i.e. what determines how to save energy or when to go to a low-power state. The framework consists of the following:

- a set of interfaces for low-level control of individual device power modes and the CPU power mode
- a more sophisticated set of services for implementing a system-wide power management policy that can manage applications or other software subsystems as well as hardware devices.



Power management framework as part of QNX Neutrino RTOS.

Power management architecture

In order to accommodate the above framework, the power management architecture can be broken into two parts:

1. A custom power management architecture using the QNX Neutrino resource manager interface. This approach provides a very simple and basic standalone power management solution that doesn't require a system-wide power manager.

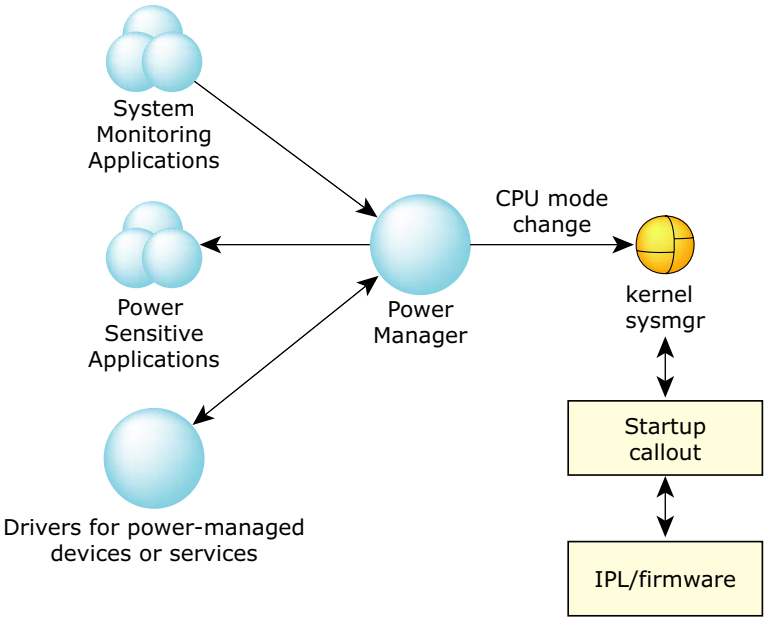
2. A sophisticated server-client architecture to build a power manager, which is the server, and implements the system-wide power management policy. The power manager interacts with the clients, such as power managed devices, to determine device power capabilities and to manipulate device power states, based on the status of the power source (e.g. battery).

Components of the power management architecture

The major components of the power management architecture are:

- a central power manager that implements the system power management policy
- device driver(s) that interact with the power manager to adjust power levels according to the system policy
- monitoring applications — may provide input events or data used by the power management policy
- applications — may receive notification of power mode changes.

Basic device and CPU power management can be implemented without requiring the power manager.



Components of the power management architecture

Power managed devices or services

These are the device drivers that manage the power consumption of the hardware devices they control. These device drivers have the most intimate knowledge of the device's usage pattern and power consumption characteristics. Therefore, they determine when the devices should change power states or which states are appropriate for current usage.

Power managed services are some arbitrary software components whose behavior can be modeled via power modes.

Power sensitive applications

These are applications whose behavior depends on the power mode of the system specific drivers or software subsystems.

Power monitoring applications

These are applications that monitor specific system parameters that are relevant to the system-wide power management policy. For example, you may consider monitoring battery levels or other data that may require changing the system power mode.

Low-level power management architecture

Low-level power management architecture provides a straight forward way of controlling the power modes of the device drivers and the CPU.

At the most basic level, the framework provides a set of low-level services to:

- set or query the device power modes of individual devices, provided via a file descriptor-based interface implemented by device drivers.
- set the CPU power mode, provided by your custom code, in co-operation with the board-specific startup and IPL software, to manage the different CPU-specific operating modes and

board-specific mechanisms for handling standby and wakeup operations.

This low-level architecture can provide a very simple form of power management, subject to the following restrictions:

- It applies only to CPU and hardware devices managed by the device drivers.
- It provides no explicit mechanism for applications to be aware of changes to either the power mode state of the system or of specific devices.
- It provides no explicit mechanism for influencing the policy that controls the system's power mode state.

Although this architecture may be suitable for some extremely simple embedded systems, a more comprehensive, system-wide power management policy would typically require much tighter integration with application-level services provided by the product. This is achieved by a higher level set of services that are based around the use of a centralized power manager, described in the next section.

You can find details about this architecture in the Low-level Power Management chapter.

Power manager architecture

The power manager architecture has two important components: the server and the client. The power manager is referred to as the server, and it implements a centralized power management policy. The power manager interacts with the clients, such as power managed devices or services to determine device power capabilities and to manipulate device power states.

The power management policy is product specific. It takes into consideration various operating modes of the product. Each mode determines the available functions of the product based on power constraints. The policy:

- determines the appropriate power mode state(s) of the product, and also includes responding to external input events, or monitoring and evaluating the system status information.
- manages the transitions between power mode states and co-ordinates changes to the power mode states of all power managed entities.

The power manager is implemented using a library that provides the basic mechanisms for:

- handling power managed entities
- managing the system power mode state machine
- interacting with device drivers and application clients.

You should use the server library to implement your own product-specific power manager. In essence, it is the specific power manager code that uses the server library and is built by linking product-specific initialization and policy code. The services provided by the library interact with the policy code to manage the overall system power mode state. See Implementing System Power Management Policy in this guide for details.

The power manager interacts with the clients to control the use of power for your hardware peripherals. See the Power Manager Clients chapter in this guide for details.

The Power Management Toolkit includes both a server and client library to provide a rich set of APIs and datatypes that enable two-way communication between the server and the clients. See the API Reference chapter in this guide for a list of APIs and datatypes.

Power managed entities

Power managed entities include:

Hardware devices

Power levels are manipulated via software control. The drivers for these devices interact with the power management policy to change the device's power mode according to policy decisions.

Software subsystems

Mode of operation is changed based on the system power mode (state). These subsystems include system services, or more general sets of applications that provide a specific service. As the power management policy changes the system power mode (state), these services are notified and change their behavior to provide only the level of service required for that system state.

In addition, the CPU is also a power managed entity; it provides active, idle and standby operational mode. Many embedded processors allow scaling of power consumption within the CPU's active operating mode using techniques such as voltage or frequency scaling.

Power modes (states)

The power management framework defines four generic power modes that apply to power managed entities:

Active Indicates that the entity is operational from a user's perspective.

Idle Indicates that the entity is partially powered; not all functions are operational. The entity is operational from a user's point of view, and transition to the *Active* mode when it is used.

Standby Indicates that the entity is partially powered; only limited functions are operational, such as implementing wakeup events. The entity is not operational from a user's perspective.

Off Indicates that the entity is powered down and is not operational.

Different power managed entities may support further *modes* — a number of different states within each of these generic power modes. For example, different *Active* states represent different operating modes that provide tradeoffs between performance and power consumption.

The framework allows each power managed entity to define these different entity-specific states and how they map to these modes.

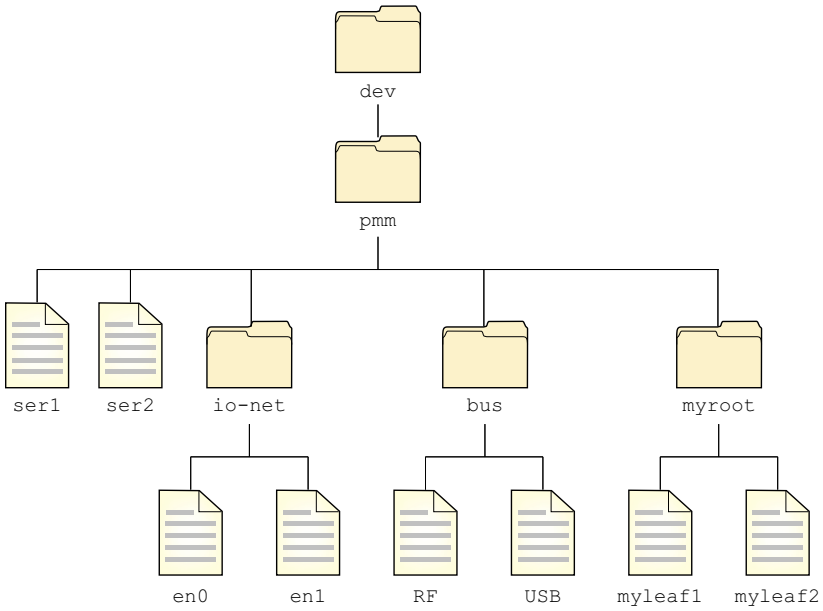
Power managed objects

Power managed objects represent the power managed entities in a hierarchical namespace. Each object within this namespace describes:

- information describing the power mode status of the object
- arbitrary properties associated with the object.

The namespace hierarchy provides a rudimentary configuration database that represents:

- physical relationships, such as the hardware topology describing the connection between buses and peripherals
- logical relationships, describing the relationship between system-level services and their constituent power managed entities.



Power managed objects as represented in a hierarchical namespace.

Chapter 3

Low-level Power Management

In this chapter. . .

Introduction

You use the resources described in this chapter to implement a low-level power managed system. Additional implementation details are found in QNX Neutrino's:

- resource manager interface (described in the Writing a Resource Manager chapter of the *Programmer's Guide*)
- *Library Reference*
- *Building Embedded Systems*.

The low-level power management policy is implemented using QNX Neutrino's resource manager interface. This approach provides you with a straightforward way of controlling the power modes of the devices, the CPU, and the memory. This basic power management solution is standalone, and doesn't require a system-wide power manager. This approach co-exists with the system-wide power management framework based on power manager architecture.

Low-level power management deals with the following two power modes:

- device power modes
- CPU and memory power modes.

At the most basic level, this power management framework implements a set of low-level services, such as:

- set or query the device power modes of individual devices
- set the CPU power mode.

In this approach, QNX resource manager presents interfaces to various types of devices or for the CPU and memory. In fact, you should use the resource manager's **`iofunc`** layer for your power management policy. This **`iofunc`** layer provides all the low-level power management components.

Device power modes

Device power modes let you directly access a device and control its power. The following APIs operate on a file descriptor of an individual device:

Function	Purpose
pm_get_power()	Get current device power mode
pm_set_power()	Set device power mode
pm_get_modes()	Get list of supported power modes

The following datatypes are structures that contain all the power management related information, such as power modes or power mode attributes.

Datatypes	Description
`pm_power_mode_t`	Represents a power mode
`pm_power_attr_t`	Describes the power mode attributes

Please refer to the QNX Neutrino *Library Reference* for detailed description of the above APIs and datatypes.

Examples

The following examples are provided for the benefit of low-level power managed system developers as well as for device driver writers. The **`iofunc`** interface provides a very basic, low-level mechanism for controlling the power mode of an individual device. In order to control the power, you:

- require an open file descriptor to the device
- require that the device driver implements the necessary low-level support

- get or set the power modes for individual devices.

Querying supported power modes

You use *pm_get_modes()* function to obtain a list of power modes supported by the device as shown in the following code snippet:

```
int  fd;
int  i;
int  nmodes;
pm_power_mode_t *modes;

fd = open("/dev/some_device", O_RDONLY);

// find out how many modes the device supports
nmodes = pm_get_modes(fd, 0, 0);

if (nmodes > 0) {
   // allocate an array to hold the list of modes
   modes = malloc(nmodes * sizeof(pm_power_mode_t));

   // fill the array with the modes supported by the driver
   pm_get_modes(fd, modes, nmodes);

   printf("Device supports %d modes: [");
   for (i = 0; i < nmodes; i++) {
      printf(" %d", modes[i]);
   }
   printf(" ]\\n");
}
```

Querying power mode status

You use *pm_get_power()* function to obtain the current power mode status of the driver:

```
int   fd;
int   status;
pm_power_attr_t attr;

fd = open("/dev/some_device", O_RDONLY);

// get power attributes
status = pm_get_power(fd, &attr);

if (status == EOK) {
```

```
   printf("Device supports %d power modes\\n", attr.num_modes);
   printf("Current mode is %d\\n", attr.cur_mode);

   if (attr.new_mode != attr.cur_mode) {
      printf("Mode change in progress to %d\\n", attr.new_mode);
   }

   if (attr.nxt_mode != attr.new_mode) {
      printf("Mode change is pending to %d\\n", attr.nxt_mode);
   }
}
```

The status is returned in the caller-supplied **pm_power_attr_t** structure:

- **cur_mode** is the current mode of the device
- if **new_mode** is different from **cur_mode**, it indicates that the device is in the process of changing power modes to **new_mode**
- if **nxt_mode** is different from **new_mode**, it indicates a pending power mode change that will be performed once the change to **new_mode** completes
- **num_modes** specifies the number of power modes supported by the device.

Changing the power mode

You use *pm_set_power()* function to change the power mode:

```
int fd;
int status;
pm_power_mode_t mode;

fd = open("/dev/some_device", O_RDWR);

mode = any power mode supported by the device;

// set power mode, subject to driver or Power Manager policy
status = pm_set_power(fd, mode, 0);

if (status != EOK && errno == EAGAIN) {
   // Driver or Power Manager policy refused to allow the change.
   // PM_MODE_FORCE should force the change regardless of policy
   status = pm_set_power(fd, mode, PM_MODE_FORCE);
}
```

CPU and memory power modes

This section discusses low-level power management using CPU and memory (e.g. SDRAM) power modes. QNX Neutrino follows a hardware and software cooperative approach for the CPU as well as for the memory to provide power management support.

Due to differing architectural standards, specific requirements are implied for different hardware. These lead to specific CPU related details, as described in *Building Embedded Systems* book.

Strategy for CPU and memory power management

The CPU and the memory consume a lot of power in an electronic system. While implementing a power management policy for a system, you must initiate a strategy to decrease CPU and memory's power consumption. This leads to increased battery life or system availability that users expect. For example, many system power management policies need the CPU to:

- enter low power modes to minimize power consumption
- enter standby modes.

The strategy is to put the CPU and the memory in specific power modes or states and transitioning through these different power modes. These modes decrease power consumption during the time when some or specific operations are not needed.

CPU power modes

QNX Neutrino provides a processor-independent view of the CPU power modes.

CPU power modes	Description
Active	System is actively running applications. Some peripherals or devices may be shutdown or idle.
Idle	System is not running, execution is halted. Code is all or partially resident in memory (used in conjunction with the CPU mode). Resumption time is near instantaneous.
Standby	Execution is halted, code is not resident. Resumption is long. It may require the help from realtime control (RTC) or from an Ethernet card, where an interrupt can activate the CPU
Shutdown	Minimal or zero power state. Resumption will reinitialize system

Memory power modes

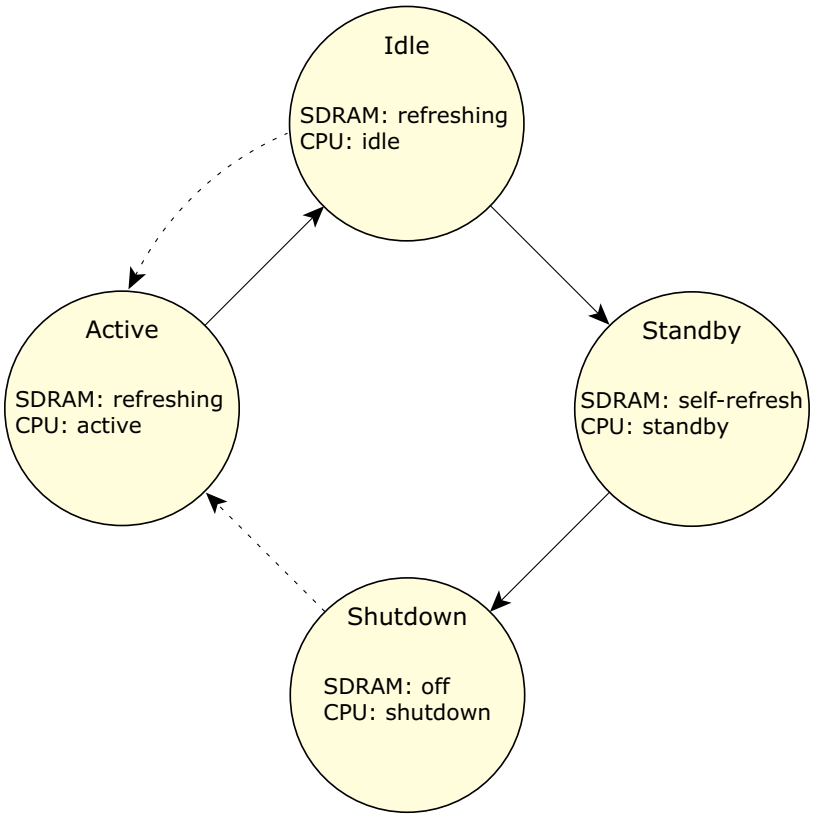
The memory has three modes that progressively need less (or no) power but at the cost of (not) retaining information. They are as follows:

Memory power modes	Description
Refreshing	This is the normal memory state. The CPU can read and write the memory contents. Refreshing is done by the CPU.

continued...

Memory power modes	Description
Self-Refresh	The power-saving mode for the SDRAM. The CPU does not refresh the SDRAM. Instead the SDRAM refreshes itself — it automatically holds its memory contents, but it is neither accessible by the CPU nor by any peripheral.
Off	No refreshing at all. The contents are undefined and unusable.

The following diagram shows different power modes of the CPU and the memory. You conserve power by transitioning through different power modes, which may or may not be implemented using a power management policy.



Power savings through power mode transitioning for CPU and memory.

OS support for CPU power management

This section describes the OS interfaces for CPU power management:

- a **`sysmgr`** interface for setting the CPU power mode
- a **`syspage pminfo_entry`** structure — used to hold platform-specific power management information
- startup *power()* callout extension.

Purpose of `sysmgr` support

The **`sysmgr`** interface provides a generic, CPU-independent interface for controlling the CPU power mode. This interface is intended to provide a standalone interface that is called by either the power manager, or by a custom power management service in systems that don't use the power manager framework. See "Changing the CPU power mode" section for further details.

The **`sysmgr`** interface *sysmgr_cpumode()* is described in detail in QNX Neutrino *Library Reference*.

The interface for controlling the CPU mode needs to be exposed to both the power manager, so that it can invoke the underlying kernel support, and custom power management applications in systems that don't use the power manager framework.

Purpose of `syspage` support

The **`syspage`** information in **`<sys/syspage.h>`** includes:

Function or structure	Description
`pminfo_entry`	Contains platform-specific power management information
init_pminfo()	Initializes the **`pminfo_entry`** section
power() callout	Validates the supplied mode and performs the specified mode change

The above functions and datatypes are described in detail in the *Building Embedded Systems* book.

Using the `pminfo_entry` structure

The information in the **`pminfo_entry`** is intended for communication between the power manager (or custom power management implementation) and the low-level BSP support provided by the IPL and startup code.

This section illustrates basic examples of how this structure can be used:

- obtain access to the **`pminfo_entry`** from a user mode program
- using the **`wakeup_pending`** flag

User mode access to `pminfo_entry`

A read-only pointer to this structure is obtained as follows:

```
struct pminfo_entry *pminfo;

if (_syspage_ptr->pminfo.entry_size != 0) {
   pminfo = SYSPAGE_ENTRY(pminfo);
}
```

You need, however, a write access — to set **`wakeup_pending`** or modify the **`managed_storage[]`** data.

On most CPUs, a writable mapping to the physical memory is created to hold the **`pminfo_entry`** structure:

```
struct pminfo_entry *pminfo;

if (_syspage_ptr->pminfo.entry_size != 0) {
   off64_t paddr;

   if (mem_offset64(SYSPAGE_ENTRY(pminfo), NOFD, sizeof(*pminfo),
      &paddr, 0) == -1) {
   error ...
}
   pminfo = mmap64(0, sizeof(*pminfo), PROT_READ|PROT_WRITE,
      MAP_PHYS|MAP_SHARED, NOFD, paddr);
   if (pminfo == MAP_FAILED) {
   error ...
   }
}
```

This alias mapping does work on certain processors (e.g. ARM) because of its MMU's virtual cache implementation. In this case, writable access can only be obtained if the process has I/O privilege, which allows writable access to the **`_syspage_ptr memory`**:

```
struct pminfo_entry *pminfo;

if (ThreadCtl(_NTO_TCL_IO, 0) == -1) {
   error ...
}
pminfo = SYSPAGE_ENTRY(pminfo);
```

Using `wakeup_pending` flag

You use the **`wakeup_pending`** flag to ensure that a wakeup interrupt is not missed when the system is being placed into a standby mode. Typically, missing this wakeup interrupt would leave the CPU in a standby mode where only a power-on-reset can restart the CPU.

The power manager (or other custom power management implementation) performs the following steps to put the system into a standby mode:

- command all device drivers to power down their devices.
- call *sysmgr_cpumode()* to put the CPU into its standby mode. See "Changing the CPU power mode" for further details.

Once in standby, the only way CPU exit that mode is via a configured wakeup interrupt or a hardware reset. Therefore, it is critical that the *power()* callout behaves correctly in the event the interrupt occurs before the standby mode is entered:

1 if the interrupt occurs during the execution of the *power()* callout, while CPU interrupts are masked, it is prevented from causing a CPU interrupt.

In this case, we assume the callout code can continue to prepare the system for standby, and that the CPU will immediately exit the standby mode because the interrupt is asserted.

2 if the interrupt occurs before the callout has masked CPU interrupts, it results in a normal CPU interrupt being handled.

On return from the interrupt handling, execution resumes in the callout, and we need to ensure that the callout does not place the CPU into standby.

The **`wakeup_pending`** flag is used to handle case (2) above:

- when the wakeup device interrupt occurs, the **`wakeup_pending`** flag is set
- the *power()* callout checks the **`wakeup_pending`** flag after it has masked CPU interrupts. If the flag is set, it re-enables CPU interrupts and returns immediately.

You use the following two approaches to implement this:

- setting **`wakeup_pending`** flag within the wakeup device driver's interrupt handler

This only works reliably if the driver is using a handler set with *InterruptAttach()*. This is due to the fact that scheduling is disabled as soon as the kernel call supporting *sysmgr_cpumode()* is entered.

- within the power manager (or other custom power management implementation)

  In this case, the power manager use *InterruptAttach()* to attach a handler to the wakeup interrupt and set the flag from that handler. This, however, requires that the power manager knows the (board-specific) interrupt vector(s) used for wakeup events.

Startup *power()* callout

The prototype for the *power()* callout in **`<sys/syspage.h>`** is changed to:

```
struct callout_entry {
   :
   _FPTR(int, power, (struct syspage_entry *, unsigned, _Uint64t *));
   :
};
```

See the *Building Embedded Systems* book for detailed information.

Changing the CPU power mode

You use *sysmgr_cpumode()*, a generic, CPU-independent interface to control different CPU power modes. This in turn invokes the CPU/platform-specific *power()* callout to perform the actual work of changing the CPU mode. The following actions are performed:

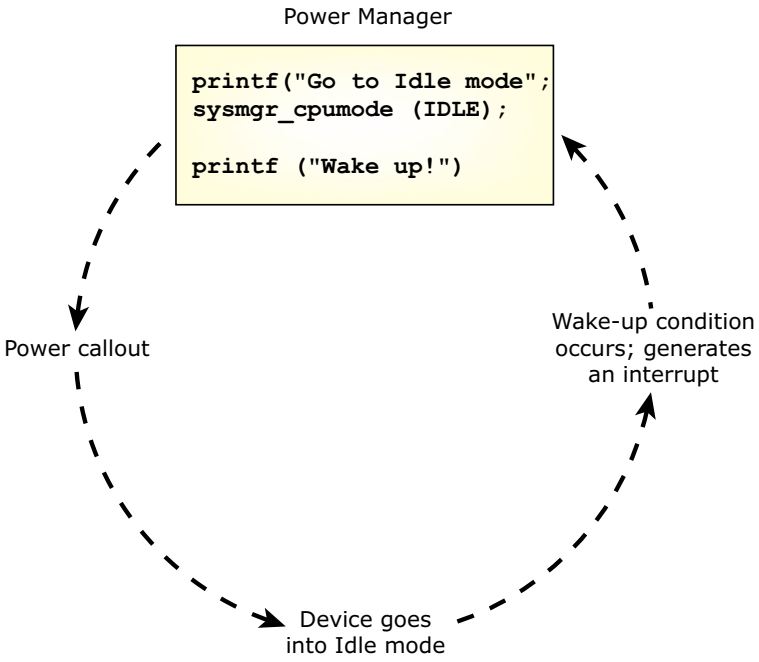
1. Mask CPU interrupts.
2. Save any data from the **`pminfo_entry`** to **`managed_storage[]`** array.
3. Place memory (i.e. SDRAM) into self-refresh or turn it off, depending on the mode.
4. Place the CPU into specific standby mode.

Design guidelines

In order to provide power management support, you need to to control different power modes of the CPU and memory you are using. You write your own custom code to use the *sysmgr_cpumode()* interface. This is done in cooperation with board-specific startup and IPL software — to manage different CPU-specific operating modes and board-specific mechanisms for handling standby and wakeup conditions.

The following diagrams depict some transitioning scenarios to provide some design insights:

Use Case 1: Transitioning to/from Active and Idle

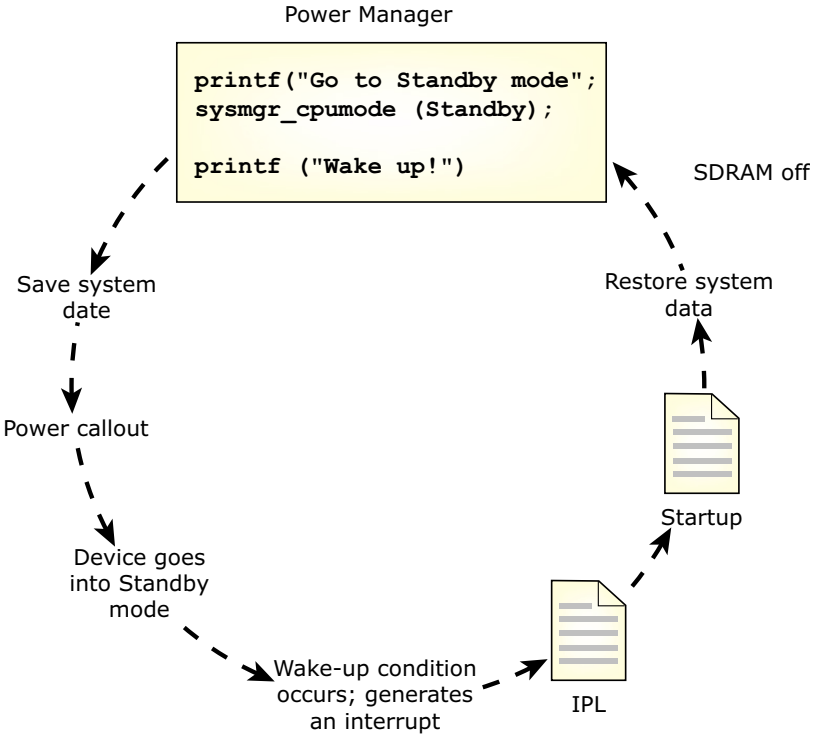


Transitioning to/from Active and Idle using power() callout and interrupt

Note the following for Use Case 1:

- System memory is preserved.
- Resumption time is near instantaneous.

Use Case 2: Transitioning from Idle to Standby, and Standby to Active



Transitioning from Idle to Standby using power() callout, and Standby to Active using interrupt.

Note the following for Use Case 2:

- System memory is not preserved.
- Resumption time is longer than that of Use case 1.

Chapter 4

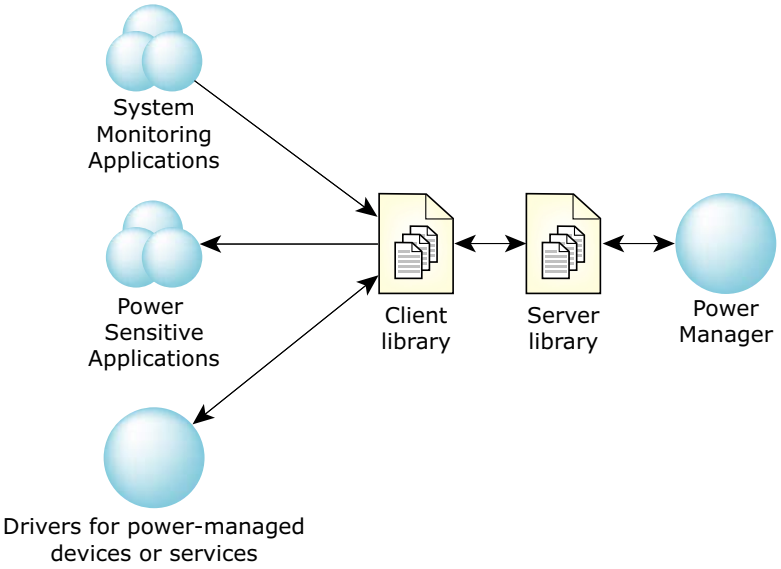
Power Manager Clients

In this chapter. . .

Overview

This chapter describes how to use the client library reference. It describes QNX Neutrino's clients for power management support, as well as the APIs (including the datatypes) available in the client library. You need these functions to interact with the power management policy in order to control the use of power for your hardware peripherals. Several client examples are provided to show how you use and apply these functions for your implementation.

At the center of QNX Neutrino's power management architecture, the most important component is the *power manager*, a server, that implements the system-wide power management policy. The power manager interacts with the *clients*, such as power managed devices (or services), to determine device power capabilities and to manipulate device power states, based on the status of the power source (e.g. battery).



The clients of the power management framework interacting with the power manager or server.

The diagram above shows the power manager interacting with three clients, as described below:

Drivers for power managed devices or services

These are device drivers that manage the power consumption of the hardware devices they control. These device drivers have the most intimate knowledge of the device's usage pattern and power consumption characteristics. Therefore, they determine when the devices should change power states or which states are appropriate for current usage.

Power managed services are some arbitrary software components whose behavior can be modeled via power modes.

You typically start a device driver in one of two ways:

1. Explicitly by the power manager. This would be most suitable for an embedded system that has a static hardware configuration, where the power manager acts as the main configuration process.
2. Dynamically, based on user requests or dynamic hardware detection. This would be most suitable for systems with dynamic hardware configurations (e.g. a desktop system) or systems with buses that support dynamically attached devices, such as USB.

Power sensitive applications

These are applications whose behavior depends on the system power mode, or the power mode of specific drivers or software subsystems.

System monitoring applications

These are applications that monitor specific system parameters that are relevant to the system-wide power management policy. For example, monitoring battery levels or other data that may require changing the system power mode.

Client API library

This section describes the client APIs from a functional point of view. The client API library provides interactions between the server and the clients. Specifically, it provides the control and application interfaces for the clients to:

- connect (or attach) to the power manager
- request (and provide) services and information from (to) the power manager, and
- set the devices in different power states.

API for drivers for power managed devices or services

The client library provides the basic services that allow the two-way communication between drivers and the power manager:

- The power manager notifies drivers to change power mode according to the system power mode policy.
- Drivers report their power mode status to the power manager.
- Drivers may optionally request autonomous power mode changes from the power manager.

API for power sensitive applications

The client library provides the basic services for such applications to:

- receive notification of power mode changes
- query the power mode of specific services or drivers
- request power mode changes to specific services or drivers.

API for system monitoring applications

The client library provides the basic services for such applications to:

- manage properties associated with a power manager object. These properties can include arbitrary data that is processed by the

product specific system power management policy to determine the most suitable power mode

- request power mode changes to specific services or drivers based on their own evaluation criteria.

A summary of APIs and datatypes

Client library interfaces are categorized into the following functional groups. You should use these interfaces in conjunction with datatypes to build a system-wide power managed system. Consult the API Reference in this guide for full details about these APIs and datatypes.

Power manager namespace APIs

The client library provides the following interfaces for manipulating the power manager namespace:

Power manager namespace APIs:	Purpose:
pm_create()	Create a new object in the namespace
pm_unlink()	Unlink an object from the namespace

The power manager manages a hierarchical namespace for power managed objects:

- the root of this namespace represents the entire system
- components in a hierarchical name are delimited by a slash (/)
- non-leaf nodes represent subsystems that contain one or mode power managed entities
- leaf nodes represent individual power managed devices.

Power manager connection APIs

The client library provides the following interfaces for managing connections to the power manager objects:

Power manager connection APIs:	Purpose:
pm_attach()	Attach to a power manager object
pm_detach()	Detach from a power manager object

Power mode APIs

The client library provides the following interfaces for manipulating power modes of the power manager objects:

Power mode APIs:	Purpose:
pm_setmode()	Change the power mode of an object
pm_getattr()	Get an object's power mode attributes
pm_getmodes()	Get the list of power modes supported by the object
pm_notify()	Request notification when the object's power mode is changed

These APIs are called by both general purpose applications and device drivers. Each interface call uses a handle to an individual power manager object that was obtained from a *pm_attach()* call.

Device drivers are responsible for the implementation of the object's power mode, so they must obey a certain protocol to operate correctly with the power manager.

Property management APIs

The client library provides the following interfaces for managing object properties:

Property management APIs:	Purpose:
pm_get_property()	Get a property value
pm_set_property()	Set a property value
pm_add_property()	Add a new property
pm_properties()	Get a list of supported properties

This property management mechanism provides a very basic interface for allowing arbitrary data or other attributes to be associated with a power manager object. These properties are opaque to the power manager itself. For example, a flag specifies an identifier that represents an arbitrary-sized data object with user-defined semantics. This lets you use a range of user-defined property identifiers that aren't managed or interpreted in any way by the power manager itself. Similarly, a callback is used to invoke the product-specific policy code to handle the property manipulation. This mechanism is intended to allow system monitoring applications to supply arbitrary product-specific data used by the product-specific policy code to manage events or conditions that require a change to the system power mode.

Datatypes

These data types are structures that contain all the power management related information, such as power modes, properties, or list of power modes.

Datatypes	Description
pm_power_mode_t	Power mode value
pm_power_attr_t	Power mode attributes of a device or subsystem
pm_property_t	Identifies a power manager property associated with a power manager object
pm_property_attr_t	Specifies a property type and the size of the property data
pm_hdl_t	Provides a client handle to a power manager object

Examples

This section describes some examples to demonstrate how you use the client library APIs for:

- attaching to a power manager object
- receiving notification of power mode changes
- receiving notification of device driver attachment
- querying and setting power modes
- configuring the power manager namespace
- querying power manager properties
- manipulating power manager properties.

This doesn't cover device drivers acting as clients of the power manager. This is described in detail in Implementing Power Management in a Device Driver in this guide.

Attaching to a power managed object

You may want to attach an object with the power manager namespace as follows:

```
pm_hdl_t  hdl;

// attach with read-only access to object (use O_RDWR to allow modify access)
hdl = pm_attach("some/power/manager/object", O_RDONLY);

if (!pm_valid_hdl(hdl)) {
   error...
}
```

The objects we refer belong to one of the following:

- an individual device
- a hardware subsystem, for example a bus controller
- a software subsystem, for example a system service, such as the **io-net** subsystem, or a collection of related processes that are managed as a single entity by the power manager.

Receiving notification of power mode changes

This example shows you how a client receives notification of power mode changes. This could be used by a power-sensitive application to respond to changes to the power mode of specific devices or subsystems.

The client needs to obtain a connection to the relevant power manager object, with O_RDONLY access is required to receive notifications.

Once the connection is established, the client needs to:

- specify a **sigevent** that is used for the notification. This would typically be a pulse or a signal
- provide a handler for the event.

Notification via a pulse

Since a pulse can be delivered to its existing channel, this mechanism is suitable for a client that already implements some form of message handling. Also, there are some advantages of using pulses:

- they are queued
- they can specify the priority at which they are handled
- the pulse handling is done within an regular thread context, so other parts of the application can be protected using normal synchronization mechanisms.

The following code snippets demonstrate the basic steps required:

1. Define a data structure that represents the power manager object:

```
struct object {
   pm_hdl_t  hdl;
  ...
};
```

   The primary purpose of the above code is to hold the **`pm_hdl_t`** to allow the event handler to communicate with the power manager.

2. Create the **`sigevent`**:

```
int  chid;
int  coid;
struct sigevent ev;

if ((chid = ChannelCreate(0)) == -1) {
   error...
}
if ((coid = ConnectAttach(0, 0, chid, _NTO_SIDE_CHANNEL, 0)) == -1) {
   error...
}

SIGEV_PULSE_INIT(&ev, coid, prio, pulse_code, object);

if (pm_notify(object->hdl, PM_CHANGE_START|PM_CHANGE_DONE, &ev) == -1) {
   error...
}
```

If the application is already using a *dispatch_** interface for receiving messages, it can use *pulse_attach()* to register the pulse with the **dispatch** loop instead of explicitly creating its own channel.

3 Handle the pulse:

```
while (1) {
   rcvid = MsgReceive(chid, buffer, sizeof buffer, 0);
   if (rcvid == 0) {
      struct _pulse *pulse = buffer;
      struct object *object = pulse->value.sival_ptr;

      do stuff with object
   }
}
```

In the event the application explicitly creates its own channel, it needs to receive the pulses from that channel. If the application uses *pulse_attach()* to register the pulse with its existing *dispatch* handling, the function specified in *pulse_attach()* will be called directly.

Notification via a signal

This mechanism is suitable for a client that doesn't want to create a channel specifically to handle the notification. There are a number of issues that need to be taken into account:

- Signals delivered by this mechanism are not queued. Therefore, the client receives only the last notification sent by the power manager.
- Other parts of the application may need to protect themselves against code run from the signal handler.

The following code snippets demonstrate the basic steps required:

1 Define a data structure that represents the power manager object:

```
struct object {
   pm_hdl_t  hdl;
 ...
};
```

The primary purpose of the above code is to hold the **pm_hdl_t** structure to allow the event handler to communicate with the power manager:

2. Create the **sigevent**:

```
struct sigaction sa;
struct sigevent  ev;

sa.sa_sigaction = my_handler;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);

if (sigaction(signo, &sa, 0) == -1) {
   error...
}

SIGEV_SIGNAL_CODE_INIT(&ev, signo, object, SI_MINAVAIL);

if (pm_notify(object->hdl, PM_CHANGE_START|PM_CHANGE_DONE, &ev) == -1) {
   error...
}
```

3. Handle the notification:

```
void
my_handler(int signo, siginfo_t *siginfo, void *data)
{
   struct object *object = siginfo->si_value.sival_ptr;

   do stuff using object
}
```

Notification of device driver attachment

This form of notification is useful in a system where device drivers are started and stopped dynamically. For example, the power manager populates its namespace with objects that represent the various services that are required. Individual device drivers then dynamically attach to the appropriate power manager objects when they are started.

Clients are notified of these driver attachments by using the PM_DRIVER_ATTACH flag to *pm_notify()*. These notifications are used to enable specific application features that rely on the presence of that device.

Similarly, the PM_DRIVER_ATTACH flag is used by the application to notify when the driver detaches from the power manager object. For example, these notifications are used to disable specific application features that rely on the device being attached.

Manipulating power modes

The following code snippets demonstrate how the power mode can be manipulated, assuming that a connection to the relevant power manager object has already been established.

1 Query the power mode:

```
pm_power_attr_t attr;

if (pm_getattr(hdl, &attr) == -1) {
   error...
}

printf("Current mode is %d\\n", attr.cur_mode);
if (attr.new_mode != attr.cur_mode)
   printf("Device is changing modes to %d\\n", attr.new_mode);
if (attr.nxt_mode != attr.new_mode)
   printf("Pending mode change to %d\\n", attr.nxt_mode);
```

A power mode of PM_MODE_UNKNOWN indicates that there is no driver associated with the power manager. For example, this is the case when the driver has not started or has detached from the power manager.

2 Query the power modes supported by the device:

```
int  i;
int  nmodes;
pm_power_mode_t *modes;

// find out how many modes the device supports
nmodes = pm_getmodes(hdl, 0, 0);

if (nmodes > 0) {
   // allocate an array to hold the list of modes
   modes = malloc(nmodes * sizeof(pm_power_mode_t));

   // fill the array with the modes supported by the driver
   pm_get_modes(fd, modes, nmodes);
```

```
   printf("Device supports %d modes: [");
   for (i = 0; i < nmodes; i++) {
      printf(" %d", modes[i]);
   }
   printf(" ]\\n");
}
```

3 Change the power mode:

```
int status;

// attempt to change mode, subject to the power manager policy
if (pm_setmode(hdl, mode, 0) == -1) {
   if (errno == EACCES) {
      // force the power mode to change
      pm_setmode(hdl, mode, PM_MODE_FORCE);
   } else {
      error...
   }
}
```

The client must specify O_RDWR to *pm_attach()* to be able to use *pm_setmode()*.

Configuring the power manager namespace

There are applications external to the power manager that can manipulate the power manager namespace. For example, enumerators or other similar system-configuration utilities could populate the power manager namespace with objects representing the devices in the system.

The following example creates this hierarchy in the namespace:

- **`/dev/pmm/bus1`**, which represents a peripheral bus
- **`/dev/pmm/bus/device_1`**, which represents a device on the **`bus1`** bus
- **`/dev/pmm/bus/bridge1`**, which represents a bridge device on **`bus1`**

- **`/dev/pmm/bus/bridge1/dev1`**, which represents a device on **`bridge1`**

This example creates the devices with access permissions granted to everyone. You should use a real configuration utility that would specify the appropriate permission in place of 0777 and 0666.

```
if (pm_create("bus1", PM_NODE_NEXUS | 0777) == -1) {
  error...
}
if (pm_create("bus1/dev1", 0666) == -1) {
 error...
}
if (pm_create("bus1/bridge1", PM_NODE_NEXUS | 0777) == -1) {
 error...
}
if (pm_create("bus1/bridge1/dev1", 0666) == -1) {
 error...
}
```

Manipulating power manager properties

The power manager supports the association of arbitrary properties with individual power manager objects. The properties are arbitrary-sized data, named by an integer property identifier.

Applications add new properties to a power manager object, or query/modify the value of an existing property. This is used to influence the policy-specific power management policy. For example, a monitoring application may update data that causes the power manager policy to re-evaluate the system's power mode state.

Currently only user-defined, policy-specific properties are supported.

The semantics of manipulating properties must be agreed upon between product-specific applications and the power manager policy code. Later versions of the framework may define generic, policy-independent properties that are used by application or policy.

Assuming a connection is already established to a relevant power manager object, the following code snippets demonstrate how properties are manipulated:

Listing all properties

```
pm_property_attr_t  *list;
int                 nprop;

// find out how many properties are associated with object
nprop = pm_properties(hdl, 0, 0);

if (nprop > 0) {
    // allocate an array to hold the list of properties
    list = malloc(nprop * sizeof(*list));

    // read list of properties
    pm_properties(hdl, list, nprop);

    printf("Object has %d properties:\\n", nprop);
    for (i = 0; i < nprop; i++) {
        printf("\tid=0x%08x size=%d bytes\\n", list[i].id, list[i].si
    }
}
```

adding a new property

```
#define MY_PROP_ID  (PM_PROPERTY_USER | 1)   // policy specific id va
struct my_prop      value;     // property value

// initialise the value data members
   :
pm_add_property(hdl, MY_PROP_ID, &value);
```

property identifiers with CPM_PROPERTY_USER set are policy-specific and their semantics are defined by the product specific policy code.

This means that the identifier and format of the property value must be agreed between the power manager policy and applications that manipulate these properties.

Obtaining a specific property value

```
struct my_prop  value;

pm_get_property(hdl, MY_PROP_ID, &value);
```

Changing a property value

```
struct my_prop  value;

// get current property value
pm_get_property(hdl, MY_PROP_ID, &value);

// modify data members in value
   :
pm_set_property(hdl, MY_PROP_ID, &value);
```

Chapter 5

Implementing Power Management in a Device Driver

In this chapter. . .

Introduction

This chapter describes the client library support for device drivers or for other power managed subsystems. The contents of this chapter help you build a device driver. For simplicity, we use the term *device driver*, although in principle, these clients represent any arbitrary service whose functions are modeled using the notion of power modes.

Device driver support

Device drivers act as clients of the power manager. They register with the power manager as named objects, and are responsible for implementing power mode changes. Using low-level APIs, the power manager communicates with the driver (via the named objects) to coordinate power mode changes in accord with the system power management policy.

From the power manager's perspective, there are two different kinds of power managed objects:

- *directory*-like objects that act as a *nexus* for other objects that can appear below it in the namespace. These would include things such as a bus controller where the child objects represent individual devices attached to the bus.
- *leaf* objects that represent an individual power managed system.

In order to prepare a device driver for power management support, modifications are needed to handle the interaction between the device drivers and the power manager.

The device driver kit or DDK implements interfaces or mechanisms for different classes of drivers. These interfaces encapsulate low-level APIs in such a way that fits in with the DDK framework. For details, see the appropriate DDK documentation.

How to modify a device driver

You need to take several steps in order to modify a device driver for power management support. This section provides an overview of these steps:

1. During initialization, device drivers need to create a **`pmd_attr_t`** structure and register with the power manager.
2. Device drivers need to interact with the power manager to implement power mode changes.
3. Device drivers need to handle pending client requests during power mode changes.

Driver initialization

At the time of initialization, the driver needs to perform the following actions for each power managed device it manages:

1. Create and initialize a **`pmd_attr_t`** structure for the device.
2. Attach the **`pmd_attr_t`** structure to the device's **`iofunc_mount_t`** structure to support the _IO_POWER interface.
3. Create a **`sigevent`** — which the power manager uses to trigger the driver to change power modes.
4. Register the device with the power manager using *pmd_attach()*.

Initializing `pmd_attr_t`

The driver needs to allocate and initialize a **`pmd_attr_t`** structure that maintains the power mode attributes for the device:

```
pmd_attr_t *my_pmd;
pm_power_mode_t my_modes[] = { ... };

my_pmd = malloc(sizeof *my_pmd);

pmd_attr_init(my_pmd);
pmd_attr_setmodes(my_pmd, initial_mode, my_modes, sizeof(my_modes)
sizeof(pm_power_mode_t));
pmd_attr_setpower(my_pmd, my_setpower, data ptr passed to my_setpof.
```

This only shows an outline of the calls the driver needs to use:

my_modes[]	Contains all the power modes supported by the device. The driver needs to specify the initial power mode of the device. Both the initial mode and the set of supported modes are passed to the power manager to initialize the power managed object representing the device.
my_setpower()	Driver-specific function used to change power mode.
my_setpower()	Called with a driver-specific data pointer. This is typically the driver data structure that is used to control the device.

Supporting the _IO_POWER interface

The driver needs to attach the **pmd_attr_t** structure to the device's **iofunc_mount_t** structure to support the _IO_POWER interface.

This enables the **iofunc** layer to invoke *pmd_power()* to handle the necessary power manager interactions and call the driver's *setpower()* function to change the device power mode:

```
iofunc_attr_t   my_attr;
iofunc_mount_t  my_mount;

initialise  my_attr ...
intiialise  my_mount ...

my_attr.mount = &my_mount;
my_mount.power = my_pmd;
```

Creating an event for power manager communication

A driver has a **dispatch** structure that it uses for receiving its messages. The following example attaches a pulse handler to the **dispatch** structure (i.e. *my_dpp*):

```
struct sigevent  my_event;
int   code;
int   coid;

code = pulse_attach(my_dpp, MSG_FLAG_ALLOC_PULSE, 0, my_pulse_handler, 0);
if (code == -1)
   {
    error ...
   }
coid = message_connect(my_dpp, MSG_FLAG_SIDE_CHANNEL);
if (coid == -1) {
 error ...
}
SIGEV_PULSE_INIT(&my_event, coid, prio, code, my_pmd);
```

Registering with the power manager

The driver needs to call *pmd_attach()* to register with the power manager. This supplies the power manager with:

- the initial device power mode and list of supported power modes
- the event used to trigger the driver to change power modes.

```
if (pmd_attach(name, my_pmd, &my_event, mode) == -1) {
   error ...
}
```

The *mode* argument specifies:

- the type of power manager object to be created. If the object has child objects within the power manager namespace, *mode* should have PM_NODE_NEXUS set.

  These power manager objects behave like directories that contain the names of child objects. These are used for bus devices where the child objects represent individual devices attached to the bus, or a subsystem where the child objects represent individual devices or other power managed components within that subsystem.

- the access permissions (defined in **<sys/stat.h>**).

The *name* argument specifies the name within the power manager namespace for the object. This takes the form of a pathname (with no leading slash) and it is the name visible under the root of the power manager namespace (**/dev/pmm/**).

If there is no existing object, a new power manager object is created to represent this device.

In the event that the pathname contains multiple components (i.e. a pathname separated by slashes), all intermediate components of that pathname must exist, because the power manager doesn't automatically create intermediate components.

Power manager interaction

The driver interacts with the power manager in two ways:

- The driver receives requests from the power manager to implement power mode changes requested by the system power management policy.

  These requests are sent using the event registered by *pmd_attach()*.

  This is typically a pulse, attached to the driver's **dispatch** handler using *pulse_attach()*. The pulse handler function uses *pmd_handler()* to perform the necessary interaction with the power manager and invoke the driver-specific power callout:

```
int
my_pulse_handler(message_context_t *ctp, int code, unsigned flags, void *data)
{
   pmd_handler(ctp->msg->pulse.value.si_ptr);
   return 0;
}
```

We assume that the pulse is initialized with the value set to a pointer to **pmd_attr_t** structure for the device.

  The *pmd_handler()* function performs all the necessary interaction with the power manager, and calls the driver's *setpower()* function to change the device power mode.

- The driver must negotiate with the power manager to implement power mode changes requested by arbitrary clients via the _IO_POWER interface.

This is handled automatically by *pmd_power()* if the driver attaches its **`pmd_attr_t`** structure to the **`iofunc_mount_t`** structure for the device.

The driver simply has its *setpower()* function called to change the device power mode.

Handling driver client requests

When the hardware is not fully functional, but the device changes to a low-power mode — the driver needs to perform the following actions before changing the power mode:

1. Complete all pending client requests.
2. Arrange for new client requests to be blocked until the power mode is restored.

The driver must be able to respond to power mode change requests via:

- the event registered by *pmd_attach()*
- the *pmd_power()* support for _IO_POWER initiated changes.

Once the power mode is restored to a level such that the hardware becomes functional, any blocked client requests can be unblocked and processed.

A summary of power manager driver APIs

The following APIs describe the client library support for device drivers or for other power managed subsystems.

Function or structure	Purpose
pmd_attach()	Register device with power manager
pmd_attr_init()	Initialize **pmd_attr_t** structure with default values
pmd_attr_setmodes()	Initialize device power modes
pmd_attr_setpower()	Initialize driver specific power mode handling
pmd_attr_t	Power manager client information for device
pmd_detach()	Detach device from the power manager
pmd_handler()	Implement power manager initiated power mode changes
pmd_power()	Implement support for **iofunc_power** initiated changes

For details about these APIs, consult the API Reference chapter in this guide.

You can implement a power managed driver using only the *pmd_** client API; these **pmd_*** functions build on the lower level APIs to provide more convenient APIs suitable for most drivers.

Chapter 6

Implementing a System Power Management Policy

In this chapter. . .

Overview

This chapter gives you the resources to implement a system-wide power management policy. As already mentioned in the Power Management Architecture chapter, QNX Neutrino's power manager framework has two important parts: the server and the client. This chapter describes the *server* for building the power manager, including a rich set of server library routines (or APIs) that help you build the power manager itself.

The server provides the following sets of services for building a product-specific power manager:

- a multi-threaded resource manager that handles client requests
- APIs for manipulating objects that represent power managed devices
- APIs for constructing simple state machines to describe the system power states.

The server library provides a number of APIs broken down into the following functional groups:

- resource manager interface presented to power manager clients
- node configuration and management
- manipulating power modes associated with a node
- manipulating properties associated with a node
- implementing simple state machines.

Please refer to the API Reference chapter for more information.

Power manager nodes

Power manager nodes represent the power managed entities in a system. A node is a (power manager) library specific concept/data structure that contains the following:

- the current power state (maintained using a **pm_power_attr_t** structure)
- a list of the power modes supported by the entity (an array of **pm_power_mode_t** values)
- the identity of the driver or other process that is responsible for managing the entity's power mode
- a list of properties associated with the node
- a list of clients that have registered for notification when the state of the node changes.

Not all nodes have a driver process responsible for managing the power mode. These nodes are *not* power managed.

Nodes are created in a number of ways:

- By the power manager itself. For example, the power manager populates the namespace to contain nodes for all devices in a statically configured system. It then starts device drivers that attach to these pre-configured nodes.
- By device drivers. For example, in a dynamically configured system, device drivers create the power manager nodes when the drivers are started.
- By some external configuration mechanism. For example, enumerators populate the namespace as buses and devices are detected. Because the library implements a standard resource manager interface, the namespace can be built using regular file and directory operations:
  - the **mkdir** command or *mkdir()* library call are used to create directory nodes
  - the **mkfifo** or **touch** commands or *mknod()* library call are used to create leaf nodes.

The organization and meaning of the namespace is entirely defined by the product-specific power manager policy.

This implies that some sort of cooperation is needed between device drivers and the power manager in order to agree on the naming conventions that allow drivers to determine and attach to the relevant nodes.

Power manager namespace

The power manager implements a hierarchical namespace that represents the power managed entities. The namespace is built using two kinds of nodes:

directory nodes

The nodes represent power managed subsystems that contain a number of peripherals, e.g. buses. In this case, a bus driver is attached to the node to manage the bus power mode. The nodes also provide a naming context for the peripherals.

The *directory* nodes are also used for configuration or organizational purposes. In this case, there is no power management for the node itself, and it simply provides a naming context for other nodes.

leaf nodes

These nodes represent individual power managed devices. In this case, a driver is attached to the node to manage the device power mode.

The *leaf* nodes are also used for control purposes without having an associated driver. In this case, power manager properties are associated with the node. Manipulation of these properties by clients can be used to control aspects of the power manager behavior or policy.

Power mode management

The server provides the basic support for triggering and monitoring changes to the power mode on individual nodes.

The product-specific policy can change the power mode of a node. This results in a request being made to the driver responsible for the node's power mode. This request is asynchronous, and the power mode state is updated when the driver completes the request and confirms its new power mode. The library delivers notification of changes to the power mode state to registered clients.

The product-specific policy is informed when the driver changes the power mode. This is in response to a request initiated by the power manager or from the driver itself independent of the power manager policy.

The library implements control of individual node only; it doesn't manage the sequencing of power mode changes for groups of nodes.

This is the responsibility of the product-specific policy code. For example, when powering down a subsystem, the policy code is responsible for powering down individual devices within the subsystem before powering down the subsystem itself.

Product-specific policies

The server provides support for product-specific policy implementation in order to control many aspects of the life cycle for a power manager node. This is achieved by a set of *policy* functions that are invoked when:

- a node is created. This allows the policy to associate a policy-specific data structure with the node. This policy-specific data is supplied when other policy functions are invoked
- a request is made to unlink a node from the namespace. This happens when:

- the driver for a dynamically created node detaches, in order to allow the power manager node and other resources to be cleaned up
- an unlink request is made via the resource manager interface, for example via the **`mkdir`** or **`rm`** commands or *rmdir()* and **`unlink`** library calls.

This allows the policy to specify whether the node is unlinked, or should remain in the namespace.

- the last reference to an unlinked node is released. This allows the policy to free any policy specific data associated with the node when it is created
- a client attaches or detaches. This allows the policy to keep track of clients if necessary
- a driver registers and initializes the power modes for the node. This allows the policy to determine that the node is now being power managed, and to find out the initial power mode
- a client or the driver requests a power mode change. This allows the policy to determine whether the power mode is allowed within the current system power state
- a driver confirms that a power mode change has been completed. This allows the policy to determine when the asynchronous power mode change has completed, and can be used to control the sequencing of power mode changes for multiple devices based on their power resource dependencies
- a new property is added to a node, or an existing property value is changed. This allows the policy to keep track of policies associated with nodes. It also allows a mechanism for clients to manipulate data that is used to control the policy behavior.

Implementing a power manager

The power manager you implement in accord with a product-specific must:

- specify the policy-specific functions
- initialize and start the server interface
- perform additional policy-specific actions (e.g. populate the namespace, start device drivers, or perform other system services).

Once the server interface is started, the power manager becomes event driven, and the policy receives control via the policy-specific functions. These policy functions use some policy specific data to manage the overall system power mode state.

It is possible to use only default policy functions to provide a very basic power manager. This power manager simply acts as a conduit between device drivers and other clients using power manager interfaces, such that:

- drivers can dynamically create nodes
- all power mode change requests are allowed, and notifications are sent to registered clients as necessary.

This simple policy doesn't handle power dependencies between devices. Control and management of these devices must be performed by client applications using the power manager interface.

To create a functional power manager using the default policy, you may use the following code:

```
#include <sys/pmm.h>
#include <sys/procmgr.h>

int
main(int argc, char *argv[])
{
   // initialize and start the power manager interface
   if (pmm_init(0) != EOK) {
   exit(1);
   }
```

```
    pmm_start(0, 0);

    // become a daemon
    procmgr_daemon(EXIT_SUCCESS, 0);
    pthread_detach(pthread_self());
    pthread_exit(0);
    return 0;
}
```

The server library

Power manager server

The server library provides the following interfaces to handle power manager clients:

Use this function:	To:
pmm_init()	Initialize power manager resource manager interface
pmm_start()	Start a thread pool to handle client requests

The resource manger interface implements a subset of the standard POSIX **iofunc** layer calls. It also implements power manager specific operations via the custom message structures described in the client routines in the API Reference chapter:

io_open Used by *pm_attach()* and *pmd_attach()*. In addition, the standard file creation commands and library calls allow external agents to create *leaf* power manager nodes.

io_mknod Allows external agents to use standard directory creation commands and library calls to create directory power manager nodes.

io_unlink Allows external agents to use standard file or directory removal commands and library calls to unlink power manager nodes.

io_msg	Implements the power manager client messages.
io_read	Allows external agents to use standard directory reading commands and library calls to list directory entries for directory nodes. The *iofunc_read_default()* function is used for nondirectory nodes.
io_close_ocb	Implements the file close and connection detach handling.

You can manage the namespace using any combination of the following:

- static configuration by the power manager itself
- dynamic configuration in response to driver initialization
- programmatic configuration (for example, by enumerators or other configuration programs)
- shell script configuration (for example, initialization scripts)
- command line configuration (for example, manual administrator actions).

Access control and ownership of power manager nodes follow the standard filesystem semantics:

- nodes are created using the caller's identity and **`umask`** value
- ownership is changed using the **`chown`** command or library functions
- access modes are changed using the **`chmod`** command or library functions.

Power managed objects

Each power managed object is represented by a **`_pmm_node_t`** structure. This is an extension of the **`iofunc_attr_t`** that adds power manager-specific data, as follows:

- power mode attributes
- client notifications
- properties
- linkage into the power manager namespace.

This structure is opaque to the product-specific policy code, and can be manipulated only via:

- a public API for manipulating nodes
- a number of callback functions that are invoked when operations are performed on nodes.

The public interface for manipulating power manager nodes consists of the following functions:

Use this function:	To:
pmm_node_create()	Create a new node
pmm_node_lookup()	Lookup a node by name
pmm_node_unlink()	Unlink a node
pmm_mode_get()	Get the current power mode of a node
pmm_mode_list()	Get the list of modes supported by a node
pmm_mode_change()	Change the power mode of a node
pmm_mode_wait()	Wait until a mode change has completed
pmm_property_add()	Add a new property to a node

continued...

Use this function:	To:
pmm_property_set()	Set the value of a property
pmm_property_get()	Get the current value of a property
pmm_property_list()	List all properties associated with a node

Support for product-specific policies

In order to receive control during client requests on a particular node, the server library provides hooks for:

- node creation and destruction
- client attachment and detachment
- driver attachment
- power mode requests and mode change completion
- property creation and manipulation.

This support is provided by the following function and structure:

pmm_policy_funcs()

Get a pointer to the policy function table

`pmm_policy_funcs_t`

Contains a table of policy specific callbacks

Callbacks

Callbacks are available as follows:

Call this callback:	When a:
create()	New node is created
destroy()	Node is destroyed
unlink()	Node to be unlinked from the namespace
attach()	Client attaches to a power manager node
detach()	Client detaches from power manager node
mode_init()	Driver attaches to a power manager node and supplies the initial power modes
mode_request()	Request is made to change the power mode for a node
mode_confirm()	Driver confirms a mode change has been completed
property_add()	Client adds a new property to a node
property_set()	Client modifies a property value

State machine datatype and APIs

The following table lists the state machine datatype and APIs:

Datatype and APIs	Purpose
`pmm_state_t`	Represent the state
pmm_state_init()	Create a state machine
pmm_state_machine()	Execute the state machine from the calling thread
pmm_state_trigger()	Trigger the state machine thread to re-evaluate the current state
pmm_state_check()	Evaluate the current state

continued...

Datatype and APIs	Purpose
pmm_state_change()	Change state

State machine operations

This section describes how to construct state machines that are used to describe and manage the system's power mode state.

Each power mode state is represented by a **`pmm_state_t`** structure that specifies:

- a policy specific identifier for the state
- a function that evaluates whether a new system state should be entered
- a function that performs actions required when leaving this state
- a function that performs actions required when entering this state.

The *pmm_state_init()* creates a new state machine using an array of **`pmm_state_t`** structures that describe the set of allowable states.

This is used to create multiple state machines that operate independently, or interact together to implement nested state machines.

The state machine APIs support the following modes of operation:

- a single-threaded state machine, where a dedicated thread is responsible for performing all state changes. These state changes are triggered by other power manager threads, based on events that change the status of power manager nodes, or other internal events.
- a multi-threaded state machine, where any power manager thread can directly perform a state change when necessary.

Single-threaded state machines

The basic steps involved are:

1. Define the system states and data associated with the state machine:

```
pmm_state_t my_states[NSTATES] = {
   :
};
struct my_data *my_data;
```

2. Initialize the state machine:

```
hdl = pmm_state_init(NSTATES, my_states, my_data, MY_INITIAL_STATE);
```

3. Execute the state machine:

```
if (pmm_state_machine(hdl) != EOK) {
   error handling...
}
```

   Because this call never returns, it must be the last initialization action that this thread performs.

Once the state machine thread is running, other threads can trigger state changes by calling **`pmm_state_trigger(hdl)`**. This will cause the state machine thread to re-evaluate the current state and perform a state transition to a new state, if necessary.

Multi-threaded state machines

The basic steps involved are:

1. Define the system states and data associated with the state machine:

```
pmm_state_t my_states[NSTATES] = {
   :
};
struct my_data *my_data;
```

2. Initialize the state machine:

```
hdl = pmm_state_init(NSTATES, my_states, my_data, MY_INITIAL_STATE);
```

Once the state machine is initialized, any thread in the power manager can re-evaluate and change the system state as follows:

```
pmm_state_check(hdl, &cur_state, &new_state);
if (new_state != cur_state)
   pmm_state_change(hdl, new_state);
```

Alternatively, a thread may unconditionally change the state using (?)

The implementation ensures mutual exclusion between these two functions so that:

- state transitions in *pmm_state_change()* are serialized
- *pmm_state_check()* waits until a state transition completes before re-evaluating the state.

Nested state machines

You can build nested state machines where sub-state machines can affect state transitions on higher-level state machines:

- via *pmm_state_trigger()* if the higher-level state machine is single-threaded
- a combination of *pmm_state_check()* and *pmm_state_change()* if the higher-level state machine is multi-threaded.

For example, the power manager could implement separate state machines for each subsystem that implements a set of services, and the state of these individual subsystems can be used to control the system's power mode state.

An example of power manager

Chapter 7

API Reference

This chapter lists all power management APIs, including three types of functions/datatypes in alphabetical order.

Client APIs and datatypes

- *iopower_getattr()*
- *iopower_getmodes()*
- *iopower_modeattr()*
- *iopower_setmode()*
- *pm_add_property()*
- *pm_attach()*
- *pm_create()*
- *pm_detach()*
- *pm_get_property()*
- *pm_getattr()*
- *pm_getmodes()*
- *pm_modeattr()*
- *pm_notify()*
- **`pm_power_attr_t`**
- **`pm_power_mode_t`**
- *pm_properties()*
- **`pm_property_attr_t`**
- **`pm_property_t`**
- *pm_set_property()*
- *pm_setmode()*
- *pm_unlink()*
- *pm_valid_hdl()*

Device driver APIs and datatypes

- *pmd_activate()*
- *pmd_attach()*
- *pmd_attr_init()*
- *pmd_attr_setmodes()*
- *pmd_attr_setpower()*
- **pmd_attr_t**
- *pmd_confirm()*
- *pmd_detach()*
- *pmd_handler()*
- *pmd_lock_downgrade()*
- *pmd_lock_exclusive()*
- *pmd_lock_shared()*
- *pmd_lock_upgrade()*
- **pmd_mode_attr_t**
- *pmd_power()*
- *pmd_setmode()*
- *pmd_unlock_exclusive()*
- *pmd_unlock_shared()*

Server APIs and datatypes

- *pmm_init()*
- *pmm_mode_change()*
- *pmm_mode_get()*

- *pmm_mode_list()*
- *pmm_mode_wait()*
- *pmm_node_create()*
- *pmm_node_lookup()*
- *pmm_node_unlink()*
- *pmm_policy_funcs()*
- **pmm_policy_funcs_t**
- *pmm_property_add()*
- *pmm_property_get()*
- *pmm_property_list()*
- *pmm_property_set()*
- *pmm_start()*
- *pmm_state_change()*
- *pmm_state_check()*
- *pmm_state_init()*
- *pmm_state_machine()*
- **pmm_state_t**
- *pmm_state_trigger()*

iopower_getattr()

Get the power attributes of a power managed device

Synopsis:

```
#include <sys/pm.h>
int iopower_getattr(int filedes, pm_power_attr_t *attr);
```

Arguments:

filedes	A file descriptor to the device special file.
attr	Pointer to where the power attributes are returned.

Library:

libpm

Description:

The *iopower_getattr()* function replaces *pm_get_power()* that has been deprecated.

The *iopower_getattr()* gets the current power attributes of a power managed device.

The current power mode is returned in *attr*->*cur_mode*.

If the device is in the process of changing power modes, the new mode will be returned in *attr*->*new_mode*.

The number of power modes supported by the device is returned in *attr*->num_modes. This value can be supplied to *iopower_getmodes()* or *iopower_modeattr()* to retrieve the full list of supported modes.

Returns:

0	Success.
-1	An error has occurred (errno is set).

Errors:

EBADF	*filedes* is not a valid file descriptor.
EFAULT	A fault occurred accesing modes.
ENOSYS	The device specified by *filedes* doesn't implement power management.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>



int main(void)
{
   int     fd;
   pm_power_attr_t attr;

   fd = open("/dev/device", O_RDONLY);
   if (fd == -1) {
      perror("open");
      return EXIT_FAILURE;
    }
    if (iopower_getattr(fd, &attr) == -1) {
       perror("iopower_getattr");
       return EXIT_FAILURE;
    }
    printf("Device supports %d modes\n", attr.num_modes);
    printf("cur_mode = 0x%x\n", attr.cur_mode);
    printf("new_mode = 0x%x\n", attr.new_mode);
    return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_power_mode_t, *iopower_setmode()*, *iopower_getmodes()*, *iopower_modeattr()*

iopower_getmodes()

Get the power modes supported by a power managed device

Synopsis:

```
#include <sys/pm.h>
int iopower_getmodes(int filedes,
                     pm_power_mode_t *modes,
                     int nmodes);
```

Arguments:

filedes	A file descriptor to the device special file.
mode	Pointer to where the list of modes are returned.
nmodes	The number of modes to be returned.

Library:

libpm

Description:

The *iopower_getmodes()* function replaces *pm_get_modes()* that has been deprecated.

The *iopower_getmodes()* function is used to get a list of the power modes supported by a power managed device.

Returns:

The number of modes supported by the device, or -1 (errno is set).

Errors:

EBADF	*filedes* is not a valid file descriptor.
EFAULT	A fault occurred accesing modes.
ENOSYS	The device specified by *filedes* doesn't implement power management.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <alloca.h>

int
main(void)
{
  int   fd;
  int   i;
  int   nmode;
        pm_power_mode_t *modes;

  fd = open("/dev/device", O_RDONLY);
  if (fd == -1) {
  perror("open");
  return EXIT_FAILURE;
  }

  nmode = iopower_getmodes(fd, 0, 0);
  if (nmode < 0) {
  perror("iopower_getmodes");
  return EXIT_FAILURE;
  }
  printf("Device supports %d modes:", nmode);

  modes = alloca(nmode * sizeof(*modes));
  if (modes == 0) {
     perror("alloca");
     return EXIT_FAILURE;
  }

  nmode = iopower_getmodes(fd, modes, nmode);
  if (nmode < 0) {
     perror("iopower_getmodes");
     return EXIT_FAILURE;
     }

  for (i = 0; i < nmode; i++) {
  printf("%d: mode=0x%x\n", i, modes[i]);
  }

  return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_power_mode_t, *iopower_getattr()*, *iopower_setmode()*, *iopower_modeattr()*

iopower_modeattr()

Get the power modes and capabilities supported by a power managed device

Synopsis:

```
#include <sys/pm.h>
int iopower_modeattr(int filedes,
                     pmd_mode_attr_t *modes,
                     int nmodes);
```

Arguments:

filedes	A file descriptor to the device special file.
mode	Pointer to where the list of modes are returned.
nmodes	The number of modes to be returned.

Library:

libpm

Description:

The *iopower_modeattr()* is used to get a list of the power modes supported by a power managed device.

For each mode, the **pmd_mode_attr_t** contains:

mode The power mode.

flags Flags that describe the driver capabilities for that mode:

PMD_MODE_ATTR_NORAM

Support the PM_MODE_NORAM flag and can be used for system power modes where system RAM is disabled.

PMD_MODE_ATTR_HWVOL

Support the PM_MODE_HWVOL flag and can be used for system power modes where peripheral device registers are lost.

PMD_MODE_ATTR_WAKEUP

Support system wakeup functionality and can be used to configure the device to act as a wakeup source for low power system standby modes.

Returns:

The number of modes supported by the device, or -1 (errno is set).

Errors:

EBADF	*filedes* is not a valid file descriptor.
EFAULT	A fault occurred accesing modes.
ENOSYS	The device specified by *filedes* doesn't implement power management.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <alloca.h>

int
main(void)
{
  int  fd;
  int  i;
  int  nmode;
  pmd_mode_attr_t *modes;

  fd = open("/dev/device", O_RDONLY);
  if (fd == -1) {
     perror("open");
     return EXIT_FAILURE;
  }

  nmode = iopower_modeattr(fd, 0, 0);
  if (nmode < 0) {
     perror("iopower_modeattr");
     return EXIT_FAILURE;
  }
```

```
  printf("Device supports %d modes:", nmode);

  modes = alloca(nmode * sizeof(*mode));
  if (modes == 0) {
     perror("alloca");
     return EXIT_FAILURE;
  }

  nmode = iopower_modeattr(fd, modes, nmode);
  if (nmode < 0) {
     perror("iopower_modeattr");
     return EXIT_FAILURE;
  }

  for (i = 0; i < nmode; i++) {
      printf("%d: mode=0x%x flags=0x%x\n",
      i, modes[i].mode,
      modes[i].flags);
  }

  return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_power_mode_t, **pmd_mode_attr_t**, *iopower_getattr()*, *iopower_setmode()*, *iopower_getmodes()*

iopower_setmode()

Set the power mode of a power managed device

Synopsis:

```
#include <sys/pm.h>

int iopower_setmode(int filedes,
                    pm_power_mode_t mode,
                    unsigned flags);
```

Arguments:

filedes	A file descriptor to the device special file.
mode	The power mode to set.
flags	Flags controlling the power mode change.

Library:

libpm

Description:

The *iopower_setmode()* function replaces *pm_set_power()* that has been deprecated.

The *iopower_setmode()* is used to change the power mode of a power managed device.

The *mode* can be either a generic power mode or a device specific power mode. See **pm_power_mode_t** for details.

If the power manager is running, it determines whether the requested power mode change is allowed, subject to the current system power management policy.

If there's no power manager running, the driver may reject the mode change based on some internal policy. For example, it may refuse to power down a device if the device is in use.

The *flags* argument controls the behavior of the power mode change:

PM_MODE_FORCE

The power mode change is not refused by the power manager or device power management policy.

PM_MODE_URGENT

Used with PM_MODE_STANDBY or PM_MODE_OFF modes to indicate that driver should perform the change as quickly as possible. For example, if the driver maintains buffered data for the device, this flag will discard those buffers instead of waiting for the buffers to drain before performing the power mode change.

PM_MODE_NORAM

Used with PM_MODE_STANDBY modes to indicate that the driver may need to save any device or driver state in persistent storage. For example, if the device is being powered down in preparation for entering a system power state where system RAM is disabled.

PM_MODE_HWVOL

Used with PM_MODE_STANDBY modes to indicate that the driver may need to save any hardware state that would be lost when the system power state shuts down the CPU. For example, powering down system-on-chip processors may cause on-chip peripheral registers to lose their contents.

PM_MODE_WAKEUP

Used with PM_MODE_STANDBY modes to indicate that the driver should enable any system wakeup functionality implemented by the device. This is used for devices that can act as wake up sources when the system is placed in a low power standby mode.

If the mode change can’t be performed immediately, this call returns -1 with errno set to EINPROGRESS. This can happen for example if powering down the device requires the driver to wait for buffered data to drain.

Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EBADF *filedes* is not a valid file descriptor.

EFAULT A fault occurred accessing modes.

ENOSYS The device is in the middle of an already initiated power mode change

EINPROGRESS

the mode change has been started, but will complete later

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdlib.h>

int
main(void)
{
  int  fd;

   fd = open("/dev/device", O_RDWR);
   if (fd == -1) {
      perror("open");
      return EXIT_FAILURE;
   }

   // try to set device to active
   if (iopower_setmode(fd, PM_MODE_ACTIVE, 0) == -1 && errno != EINPROGRESS) {
               perror("iopower_setmode");

   // try to force device to active
   if (iopower_setmode(fd, PM_MODE_ACTIVE, PM_MODE_FORCE) == -1) {
      perror("iopower_setmode");
      }
   }
   return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

`pm_power_mode_t`, *iopower_getattr()*, *iopower_modeattr()*, *iopower_getmodes()*

pm_add_property()

Add a property to a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_add_property(pm_hdl_t hdl,
                    pm_property_t id,
                    void *value,
                    int size);
```

Arguments:

hdl	Handle to the power managed object, obtained via *pm_attach()*.
id	An integer identifier that specifies the property type.
value	Initial value of the property.
size	Size of the property value.

Library:

libpm

Description:

The *pm_add_property()* adds a new property to a power managed object. The power manager policy is informed whenever properties are added to an object and when property values are modified.

Each property consists of an **<***id*, *value***>** pair:

id is an integer identifier for the property:

The range of values from PM_PROPERTY_USER to UINT_MAX are available for user defined properties. These user defined properties can be used to implement system specific data that can be used by the power manager policy to evaluate the most appropriate system power mode.

The range of values from 0 to PM_PROPERTY_USER-1 are reserved for Neutrino defined properties.

The *value* variable specifies the initial value of the property. The size and format of the data pointed to by *value* depend on *id*.

Returns:

0 Success.

-1 An error has occurred (and errno has set).

Errors:

EBADF	*hdl* is not a valid handle.
EEXIST	The object already has a property with the identifier *id*.
EACCES	Caller isn't allowed to add properties to the object (for example, *hdl* wasn't opened with O_RDWR access).
EFAULT	A fault occurred accessing value.
ENOMEM	Insufficient memory to allocate power manager data structures.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdlib.h>


// define a property identifier and structure containing property data
#define PROP_ID         (PM_PROPERTY_USER + 1)
struct prop_value {
   int     data1;
   int     data2;
};

int
main()
{
```

```
  pm_hdl_t                        hdl;
  struct prop_value       value = { 1, 2 };

  hdl = pm_attach("object", O_RDWR);
  if (!pm_valid_hdl(hdl)) {
     perror("pm_attach");
     return EXIT_FAILURE;
  }

  if (pm_add_property(hdl, PROP_ID, &value, sizeof value) == -1) {
     perror("pm_add_property");
     return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_attach(), *pm_get_property()*, *pm_set_property()*, *pm_properties()*

pm_attach()

Obtain a handle to a power managed object

Synopsis:

```
#include <sys/pm.h>
#include <fcnl.h>

pm_hdl_t pm_attach(const char *name, int flags);
```

Arguments:

name Name of the object within the power manager namespace.

flags Specifies the access required to the object.

Library:

libpm

Description:

The *pm_attach()* obtains a handle allowing a client to manipulate a power managed object.

The flags argument specifies the required access:

O_RDONLY	The client isn't allowed to perform operations that modify the object.
O_RDWR	The client is allowed to perform operations that modify the object.

Returns:

An opaque handle that can be used to manipulate the object.

The *pm_valid_hdl()* function can be used to check if the handle is valid. If *pm_attach()* is unsuccessful, errno is set to indicate the error.

Errors:

ENOENT	*name* does not exist.
EACCES	Search permission is denied for a component within *name*.
EACCES	The permissions specified by flags are denied.
ENOTDIR	A component of *name* is not a directory.
EMFILE	Too many file descriptors are currently in use by the process.
ENFILE	Too many files are currently open in the system.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdlib.h>


int
main()
{
   pm_hdl_t        hdl;

   // attach to object with read-only access
   hdl = pm_attach("object", O_RDONLY);
   if (!pm_valid_hdl(hdl)) {
      perror("pm_attach");
      return EXIT_FAILURE;
   }

   if (pm_detach(hdl) == -1) {
      perror("pm_detach");
      return EXIT_FAILURE;
   }

   // attach to object with read-write access
   hdl = pm_attach("object", O_RDWR);
   if (!pm_valid_hdl(hdl)) {
```

```
        perror("pm_attach");
        return EXIT_FAILURE;
    }


    return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pm_detach(), *pm_create()*

pm_create()

Create an entry in the project manager namespace

Synopsis:

```
#include <sys/pm.h>
#include <sys/stat.h>

int pm_create(const char *name,
              mode_t mode);
```

Arguments:

name Name of the entry to be created

mode Type and access permissions associated with the entry.

Library:

libpm

Description:

The *pm_create()* creates a new entry in the power manager namespace. The *mode* specifies the file type and access permissions of the new object:

PM_NODE_NEXUS indicates the object is a non-leaf node (directory-like) object that can have further objects created below it in the name space.

If PM_NODE_NEXUS is not specified, the object is a leaf node and cannot have objects created below it in the namespace.

The access permissions are specified as for *open()* or *creat()*. See "Access Permissions" in the documentation for *stat()*.

Returns:

0 Success.

-1 An error has occurred (errno is set).

Errors:

ENOENT	There's no power manager running.
EINVAL	Name begins with a '/' character
ENOTDIR	A component of name is not a directory
EACCES	Search permission is denied in a component of *name*.
EACCES	Write permission is denied in the last directory component of *name*
EEXIST	An object with *name* already exists.

Examples:

```
#include <sys/pm.h>
#include <stat.h>
#include <stdlib.h>



int
main()
{
   // create a directory node with rwxrwxr-x permissions
   if (pm_create("dir", PM_NODE_NEXUS|S_IRWXU|S_RWXG|S_IROTH|S_IXOTH) ==
   perror("pm_create");
   return EXIT_FAILURE;
   }

   // create a node under "dir" with rw-rx-r-- permissions
   if (pm_create("dir/obj", S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH) == 
      perror("pm_create");
      return EXIT_FAILURE;
    }
   return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pm_attach(), stat()

pm_detach()

Release a handle to a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_detach(pm_hdl_t hdl);
```

Arguments:

hdl A handle to the power manager object obtained via *pm_attach()*

Library:

libpm

Description:

The *pm_detach()* closes the client connection to the power managed object.

Returns:

0 Success.

-1 An error has occurred (errno is set).

Errors:

EBADF *hdl* is not a valid handle.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdlib.h>


int
main()
{
```

```
    pm_hdl_t        hdl;

    // attach to object with read-only access
    hdl = pm_attach("object", O_RDONLY);
    if (!pm_valid_hdl(hdl)) {
       perror("pm_attach");
       return EXIT_FAILURE;
    }

    if (pm_detach(hdl) == -1) {
       perror("pm_detach");
       return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_attach()

pm_get_property()

Get the value of the property associated with a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_get_property(pm_hdl_t hdl,
                    pm_property_t id,
    void *value,
                    int size);
```

Arguments:

hdl	A handle to the power managed object obtained via *pm_attach()*.
id	An integer identifier that specifies the property type.
value	A pointer to where the property value is returned.
size	Size of the property value.

Library:

libpm

Description:

The *pm_get_property()* obtains the current value of the specified property associated with a power managed object.

Returns:

0	Success
-1	An error has occurred (errno is set)

Errors:

EBADF	*hdl* is not a valid handle.
EINVAL	No property exists with the specified identifier.

EINVAL Size of the property value is different to *size*.

EFAULT A fault occurred accessing value.

Examples:

```
#include <sys/pm.h>
#include <stat.h>
#include <stdlib.h>


// define a property identifier and structure containing property data
#define PROP_ID         (PM_PROPERTY_USER + 1)
struct prop_value {
        int     data1;
        int     data2;
};

int
main()
{
  pm_hdl_t   hdl;
  struct prop_value  value = { 1, 2 };

  hdl = pm_attach("object", O_RDWR);
  if (!pm_valid_hdl(hdl)) {
    perror("pm_attach");
    return EXIT_FAILURE;
  }

  if (pm_get_property(hdl, PROP_ID, &value, sizeof value) == -1) {
     perror("pm_add_property");
     return EXIT_FAILURE;
  }

  printf("data1 = %d\n", value.data1);
  printf("data2 = %d\n", value.data2);

  return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_add_property(), *pm_set_property()*, *pm_properties()*

pm_getattr()

Get the power attributes of a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_getattr(pm_hdl_t hdl,
               pm_power_attr_t *attr);
```

Arguments:

hdl A handle to the object obtained via *pm_attach()*.

attr Pointer to where the power attributes are returned.

Description:

The *pm_getattr()* function gets the current power attributes of a power managed object.

The current power mode is returned in *attr->cur_mode*.

If the object is in the process of changing power modes, the new mode will be returned in *attr->new_mode*.

The number of power modes supported by the object is returned in *attr->num_modes*. This value can be supplied to *pm_getmodes()* or *pm_modeattr()* to retrieve the full list of supported modes.

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Errors:

EBADF *hdl* isn't a valid handle.

EFAULT A fault occurred accessing *attr*.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>



int
main(void)
{
   pm_hdl_t                hdl;
   pm_power_attr_t attr;

   hdl = pm_attach("object", O_RDONLY);
   if (!pm_valid_hdl(hdl)) {
      perror("pm_attach");
      return EXIT_FAILURE;
   }

   if (pm_getattr(hdl, &attr) == -1) {
      perror("pm_getattr");
      return EXIT_FAILURE;
   }

   printf("Device supports %d modes\n", attr.num_modes);
   printf("cur_mode = 0x%x\n", attr.cur_mode);
   printf("new_mode = 0x%x\n", attr.new_mode);
   return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No

continued...

Safety	
Signal handler	Yes
Thread	Yes

See also:

pm_power_mode_t, *pm_attach()*, *pm_setmode()*, *pm_getmodes()*, *pm_modeattr()*

pm_getmodes()

Get the power modes supported by a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_getmodes(pm_hdl_t hdl,
                pm_power_mode_t *modes,
int nmodes);
```

Arguments:

hdl	Handle to the object obtained via *pm_attach()*.
modes	A pointer to where the list of modes are returned.
nmodes	The number of modes to be returned.

Library:

libpm

Description:

The *pm_getmodes()* function call is used to get the list of power modes supported by a power managed object.

Returns:

If successful, *pm_getmodes()* returns the number of modes supported by the object.

If an error occurs, it returns -1 and *errno* is set.

Errors:

EBADF	*hdl* is not a valid handle.
EFAULT	A fault occurred accessing to modes.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>



int
main(void)
{
   pm_hdl_t hdl;
   int      i;

   int                              nmode;
   pm_power_mode_t *modes;

   hdl = pm_attach("object", O_RDONLY);
   if (!pm_valid_hdl(hdl)) {
      perror("pm_attach");
      return EXIT_FAILURE;
   }

   nmode = pm_getmodes(fd, 0, 0);
   if (nmode < 0) {
      perror("pm_getmodes");
      return EXIT_FAILURE;
   }

   printf("Device supports %d modes:", nmode);

   modes = alloca(nmode * sizeof(*modes));

   if (modes == 0) {
      perror("alloca");
      return EXIT_FAILURE;
   }

   nmode = pm_getmodes(fd, modes, nmode);
   if (nmode < 0) {
   perror("iopower_getmodes");
   return EXIT_FAILURE;
```

```
    }

    for (i = 0; i < nmode; i++) {
        printf("%d: mode=0x%x\n", i, modes[i]);
    }

    return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_power_mode_t *pm_attach()*, *pm_getattr()*, *pm_setmode()*
pm_modeattr()

pm_modeattr()

Get the power modes and capabilities supported by a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_modeattr(pm_hdl_t hdl,
                pmd_mode_attr_t *modes,
int nmodes);
```

Arguments:

hdl	Handle to the object — obtained via *pm_attach()*.
modes	Pointer to where the list of modes are returned.
nmodes	Number of modes that are returned.

Library:

libpm

Description:

the *pm_modeattr()* is used to get a list of the power modes supported by a power managed object.

For each mode, the **pmd_mode_attr_t** describes:

mode The power mode.

flags Describe the driver capabilities for that mode:

PMD_MODE_ATTR_NORAM

Support the PM_MODE_NORAM flag and can be used for system power modes where system RAM is disabled.

PMD_MODE_ATTR_HWVOL

Support the PM_MODE_HWVOL flag and can be used for system power modes where peripheral device registers are lost.

PMD_MODE_ATTR_WAKEUP

Support system wakeup functionality and can be used to configure the device to act as a wakeup source for low power system standby modes.

Returns:

If successful, the number of modes supported by the object , or -1 when an error occurred (errno is set).

Errors:

EBADF *hdl* is not a valid handle.

EFAULT A fault occurred accessing modes.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <alloca.h>

int
main(void)
{
   pm_hdl_t    hdl;
   int         i;

   int         nmode;
   pmd_mode_attr_t *modes;

   hdl = pm_attach("object", O_RDONLY);
   if (!pm_valid_hdl(hdl)) {
       perror("pm_attach");
       return EXIT_FAILURE;
   }

   nmode = pm_modeattr(fd, 0, 0);
   if (nmode < 0) {
      perror("pm_modeattr");
      return EXIT_FAILURE;
      }
   printf("Device supports %d modes:", nmode);
```

```
   modes = alloca(nmode * sizeof(*modes));
   if (modes == 0) {
      perror("alloca");
      return EXIT_FAILURE;
      }

   nmode = pm_modeattr(fd, modes, nmode);
   if (nmode < 0) {
      perror("pm_modeattr");
      return EXIT_FAILURE;
      }

   for (i = 0; i < nmode; i++) {
   printf("%d: mode=0x%x flags=0x%x\n", i,
          modes[i].mode,
          modes[i].flags);
     }
   return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_power_mode_t, *pm_attach()*, *pm_getattr()*, *pm_setmode()*, *pm_getmodes()*

pm_notify()

Receive notification when the status pf a power managed object changes

Synopsis:

```
#include <sys/pm.h>

int pm_notify(pm_hdl_t hdl,
              unsigned flags,
              const struct sigevent *event);
```

Arguments:

hdl Handle to the power managed object obtained by *pm_attach()*.

flags Specify the changes of interest.

event Pointer to a sigevent structure delivered when the requested change occurs, or NULL to cancel notifications

Library:

libpm

Description:

The *pm_notify()* receives notification when the status of a power managed object changes.

flags specifies the changes of interest:

PM_CHANGE_START

Receive notification at the start of a power mode change.

PM_CHANGE_DONE

Receive notification on completion of a power mode change.

PM_DRIVER_ATTACH

Receive notification when a driver registers to manage the power managed object.

PM_DRIVER_DETACH

Receive notification when the driver managing the object detaches (both normal and abnormal termination of the driver).

When one of the requested changes occurs, the specified event will be delivered.

The requested notification remains in effect until it explicitly canceled. This cancellation occurs if:

- a subsequent *pm_notify()* is called with *flags* set to 0
- a subsequent *pm_notify()* is called with a NULL event.
- *pm_detach()* is called to close the connection to the power manager.

A call to *pm_notify()* replaces any notification set by a previous call and enable only the set of notifications specified by the new call.

Returns:

0	Success.
-1	An error occurred (errno is set).

Errors:

EBADF	*hdl* is not a valid handle.
EINVAL	*flags* contains an invalid combination of flags.
EFAULT	A fault occurred accessing event.
ENOMEM	Insufficient memory for allocating power manager data structures.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_attach(), *pm_detach()*

pm_power_attr_t

Describe the power mode attributes

Synopsis:

```
typedef struct pm_power_attr_t  {
   pm_power_mode_t  cur_mode;
   pm_power_mode_t  new_mode;
   pm_power_mode_t  nxt_mode;
   pm_power_mode_t  num_modes;
} pm_power_attr_t;
```

Description:

This structure describes the power mode attributes of a power managed object. The different modes are as follows:

cur_mode	Current operating mode.
new_mode	New mode if the object is in the process of a power mode change. Otherwise it is the same as *cur_mode*.
nxt_mode	Indicates a pending mode change — if a mode change request is received while the driver is still in the process of power mode change.
num_modes	Number of different modes supported by the object.

Classification:

See also:

pm_power_mode_t

Describe a power mode

Synopsis:

typedef _int32t pm_power_mode_t;

Description:

The **pm_power_mode_t** structure represents the power mode of a power managed object.

This power mode can be represented in the following ways:

1. a logical power mode that describes the object's operational mode.
2. a device power mode that describes the driver specific power mode the device is currently operating at.

Logical power modes:

PM_MODE_ACTIVE

The object is fully operational.

This is the normal operating mode of a device.

A driver may implement multiple device power modes that correspond to PM_MODE_ACTIVE that differ in their power consumption and performance characteristics.

PM_MODE_IDLE

the object may be partially powered, but is still considered to be operational from a system point of view.

This mode is typically used only by a driver-internal policy to reduce power consumption when the device is not in active use, and allows a driver to implement a low power mode where some or all device functions are disabled, while providing a short latency to return to a PM_MODE_ACTIVE mode.

A driver may implement multiple device power modes that correspond to PM_MODE_IDLE that differ in their power consumption and available device functionality.

PM_MODE_STANDBY

the object is not operational and is in a low power state.

This mode is typically used to power down devices before placing the system into a low power standby state.

A driver may implement multiple device power modes that correspond to PM_MODE_STANDBY that differ in the available functionality of the device. For example, enabling system level wakeup functionality.

PM_MODE_OFF

the object is not operational and is powered off.

PM_MODE_UNKNOWN

indicates the object is in an unknown state. This is typically set when a driver terminates abnormally before completing a power mode change, where the real device power level can no longer be determined.

The *PM_POWER_MODE()* macro can be used to convert a device specific mode to its corresponding logical power mode. Supplying an invalid mode value will return PM_MODE_INVALID.

The *PM_MODE_VALID()* macro can be used to determine if a power mode is a valid power mode value. Note that this only validates that the mode value is within the acceptable range - it doesn't validate that the mode is actually supported by any particular power managed object.

Classification:

See also:

pmd_mode_attr_t

pm_properties()

Obtain a list of properties associated with a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_properties(pm_hdl_t hdl,
                  pm_property_attr_t *list,
                  int count);
```

Arguments:

hdl	Handle to the power managed object obtained via *pm_attach()*.
list	Pointer to where the property information is returned.
count	The maximum number of property entries to return.

Library:

libpm

Description:

The *pm_properties()* is used to list the properties associated with a power managed object.

Each property is described by a **pm_property_attr_t** structure:

id specifies the property identifier of the property.

size the size of the property value.

Returns:

The number of properties associated with the object, or -1 if an error occurred (errno is set).

Errors:

EBADF	*hdl* is not a valid handle.
EFAULT	A fault occurred accessing list.

ENOMEM	Insufficient memory to allocate power manager data structures.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdlib.h>


int main()
{
   pm_hdl_t     hdl;
   int          i;
   int          count;
   pm_property_attr_t      *list;

   hdl = pm_attach("object", O_RDONLY);
   if (!pm_valid_hdl(hdl)) {
      perror("pm_attach");
      return EXIT_FAILURE;
   }

   // find out how many properties exist
   count = pm_properties(hdl, 0, 0);
   if (count == -1) {
      perror("pm_properties");
      return EXIT_FAILURE;
   }

   printf("Object has %d properties\n");

   // allocate memory for list and get all properties
   if ((list = malloc(count * sizeof(*list))) == 0) {
      perror("malloc");
      return EXIT_FAILURE;
   }

   if (pm_properties(hdl, list, count) == -1) {
      perror("pm_properties");
      return EXIT_FAILURE;
   }
```

```
   for (i = 0; i < count; i++) {
      printf("%d: id=0x%x size=%d bytes\n", i,
            list[i].id, list[i].size);
   }

   return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_add_property(), *pm_get_property()*, *pm_set_property()*

pm_property_attr_t

Identify the attributes for each property

Synopsis:

```
typedef struct {
    pm_property_t   id;
    _Uint32t        size;
}  pm_property_attr_t;
```

Description:

This structure is used by *pm_properties()* to identify the attributes for each property by an object.

id Specifies the property type.

size Specifies the size of the property data.

Classification:

See also:

pm_property_t

Provide an identifier for an arbitrary property

Synopsis:

```
typedef _Uint32t    pm_property_t;

#define PM_PROPERTY_USER    0x80000000
/* start of user defined properties */
```

Description:

This integer type provides an identifier for an arbitrary property that the power manager associates with a power managed object.

Values between 0 and PM_PROPERTY_USER-1 are reserved for power manager defined properties.

Values from PM_PROPERTY_USER and above are treated as user-defined properties and have an arbitrary, user-defined data size:

- the meaning and usage are defined entirely by the product specific policy code in the power manager.
- *pm_get_property()()* is used to determine the actual size of the data for a given property associated with a particular power managed object.

Classification:

See also:

pm_set_property()

Set the value of a property associated with a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_set_property(pm_hdl hdl,
                    pm_property_t id,
                    void *value,
                    int size);
```

Arguments:

hdl	Handle to the power managed object obtained via *pm_attach()*.
id	An integer identifier that specifies the property type.
value	A pointer to where the property value is returned.
size	Size of the property value.

Library:

libpm

Description:

The *pm_set_property()* function changes the value of the specified property associated with a power managed object.

This property change is notified to the power manager policy, and this may result in policy specific actions based on the new value. For example, user defined properties can be used to implement system specific data used by the policy to evaluate the most appropriate system power mode.

Returns:

0	Success
-1	An error occurred (errno is set)

Errors:

EBADF	*hdl* is not a valid handle.
EINVAL	No property exists with the specified identifier.
EINVAL	The size of the property value is different to size.
EACCES	Caller is not allowed to add properties to the object (for example, hdl was not opened with O_RDWR access).
EFAULT	A fault occurred accessing value.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdlib.h>


// define a property identifier and structure containing property data
#define PROP_ID         (PM_PROPERTY_USER + 1)
struct prop_value {
   int     data1;
   int     data2;
};

int
main()
{
   pm_hdl_t        hdl;
   struct prop_value       value = { 1, 2 };

   hdl = pm_attach("object", O_RDWR);
   if (!pm_valid_hdl(hdl)) {
      perror("pm_attach");
      return EXIT_FAILURE;
   }

   if (pm_set_property(hdl, PROP_ID, &value, sizeof value) == -1) {
      perror("pm_add_property");
      return EXIT_FAILURE;
   }
```

```
    return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_add_property(), *pm_get_property()*, *pm_properties()*

pm_setmode()

Set the power mode a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_setmode(pm_hdl_t hdl,
               pm_power_mode_t mode,
               unsigned flags);
```

Library:

libpm

Description:

The *pm_setmode()* is used to change the power mode of a power managed object.

The power manager determines whether the requested power mode change is allowed, subject to the current system power management policy. Even if the change is allowed by the power manager, the driver for the object may impose its own power management policy that refuses the request. For example, it may refuse to power down a device that is in use. mode can be either a generic power mode or a object specific power mode. See **pm_power_mode_t**. The *flags* variable controls the behavior of the power mode change:

PM_MODE_FORCE

The power mode change will not be refused by the Power Manager, or any driver specific power management policy.

PM_MODE_URGENT

can be used with PM_MODE_STANDBY or PM_MODE_OFF modes to indicate that the driver should perform the change as quickly as possible. For example, if the driver maintains buffered data for the device, this flag will discard those buffers instead of waiting for the buffers to drain before performing the power mode change.

PM_MODE_NORAM

used with PM_MODE_STANDBY modes to indicate that the driver may need to save any device or driver state in persistent storage. For example, if the device is being powered down in preparation for entering a system power state where system RAM is disabled.

PM_MODE_HWVOL

used with PM_MODE_STANDBY modes to indicate that the drivermay need to save any hardware state that would be lost when the system power state shuts down the CPU. For example, powering down system-on-chip processors may cause on-chip peripheral registers to lose their contents.

PM_MODE_WAKEUP

used with PM_MODE_STANDBY modes to indicate that the driver should enable any system wakeup functionality implemented by the device. This is used for devices that can act as wake up sources when the system is placed in a low power standby mode.

Returns:

0	Success.
-1	An error occurred (errno is set).

Errors:

EBADF	*hdl* is not a valid handle.
EAGAIN	The specified mode is not allowed by the power manager policy.
EINVAL	Mode is not supported by the device or flags contains invalid values.
EACCES	Caller is not allowed to add properties to the object (for example, *hdl* was not opened with O_RDWR access)

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdlib.h>


int
main(void)
{
   pm_hdl_t        hdl;

   hdl = pm_attach("object", O_RDWR);
   if (!pm_valid_hdl(hdl)) {
      perror("pm_attach");
      return EXIT_FAILURE;
   }

   // try to set device to active
   if (pm_setmode(fd, PM_MODE_ACTIVE, 0) == -1) {
       perror("pm_setmode");

        // try to force device to active
        if (pm_setmode(fd, PM_MODE_ACTIVE, PM_MODE_FORCE) == -1) {
           perror("pm_setmode");
        }
   }

   return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No

continued...

Safety	
Signal handler	Yes
Thread	Yes

See also:

pm_power_mode_t, *pm_attach()*, *pm_getattr()*, *pm_getmodes()*, *pm_modeattr()*

pm_unlink()

Remove an object from the power manager namespace

Synopsis:

```
#include <sys/pm.h>

int pm_unlink(const char *name);
```

Library:

libpm

Description:

The *pm_unlink()* removes the specified name from the power manager namespace.

Returns:

0	Success.
-1	An error has occurred (errno is set).

Errors:

ENOENT	There's no power manager running.
EINVAL	*name* begins with a "/" character.
ENOENT	A component of *name* doesn't exist.
ENOENT	*name* doesn't exist.
ENOTDIR	A component of *name* isn't a directory
EACCES	Search permission is denied in a component of *name*.
EACCES	Write permission is denied in the last directory component of *name*.
EBUSY	*name* is a non-empty directory.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pm_create()

pm_valid_hdl()

Verify handle to a power managed object

Synopsis:

```
#include <sys/pm.h>

int pm_valid_hdl(pm_hdl_t hdl);
```

Arguments:

hdl Handle to the power managed object — obtained via *pm_attach()*.

Library:

libpm

Description:

The *pm_valid_hdl()* verifies if the supplied handle is a valid handle to a power managed object.

Returns:

A non-zero value

hdl is a valid handle.

0 *hdl* is not a valid handle.

Examples:

```
#include <sys/pm.h>
#include <fcntl.h>
#include <stdlib.h>

int
main()
{
   pm_hdl_t  hdl;

   // attach to object with read-only access
   hdl = pm_attach("object", O_RDONLY);
   if (!pm_valid_hdl(hdl)) {
   perror("pm_attach");
      }
```

```
    return EXIT_SUCCESS;
}
```

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pm_attach()

pmd_activate()

Request to power up a device from a non-active power mode

Synopsis:

```
#include <sys/pm.h>

int pmd_activate(pmd_attr_t *pmd, unsigned flags);
```

Arguments:

pmd Pointer to the driver's **pmd_attr_t** structure for the device.

flags Specify the reason for requesting the power mode change.

Library:

libpm

Description:

A driver can use *pmd_activate()* to request that the device be powered up from a non-active power mode. This requests a change to the mode specified by **pmd->last_active** (the last PM_MODE_ACTIVE mode the driver was set to).

This is typically required when the driver must perform some I/O operation and determines that the device is in a low power mode.

The **pmd_attr_t** structure must be locked using *pmd_lock_shared()* to ensure the **pmd_attr_t** power mode isn't subject to modification while it's performing these checks.

flags indicate the reason why the driver wants the device to be powered up:

PM_MODE_HWEVENT

> The request is in response to some hardware generated event that can't be serviced until the device is fully powered.

If *flags* is 0, it indicates that the device must be powered up to service a driver client request.

Returns:

EOK — The device is already in a PM_MODE_ACTIVE mode. The driver can proceed with the I/O operation.

EAGAIN — The power mode change is not allowed. The driver must block the I/O operation until the power mode is later changed to an active mode. If the request is in response to a client request where the client used O_NONBLOCK to open the device, the operation should not block, and should instead fail with an EAGAIN error.

EINPROGRESS — The power mode change has been initiated and will complete later. The driver must block the I/O operation until the power mode change has completed.

The driver code responsible for performing power mode changes must arrange to unblock these blocked requests.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_attr_t, *pmd_lock_shared()*, *pmd_setmode()*

pmd_attach()

Register driver with the power manager

Synopsis:

```
#include <sys/pm.h>

int pmd_attach(const char *name,
               pmd_attr_t *attr,
               const struct sigevent *event,
               mode_t mode);
```

Arguments:

name	Name of the power managed object the driver is responsible for.
pmd	The **pmd_attr_t** structure used to manage the device power attributes.
event	The event the power manager should use to communicate with the driver.
mode	The type and access permission to use if name doesn't yet exist.

Library:

libpm

Description:

The *pmd_attach()* is used by a device driver to register with the power manager.

Before calling *pmd_attach()*, the driver must initialize the following:

- use *pmd_attr_init()* to initialize the *pmd* with default values
- use *pmd_attr_setmodes()* to specify the device's supported power modes
- use *pmd_attr_setpower()* to specify the driver's power mode function

- initialize event (typically a pulse).

On return from *pmd_attach()*, *pmd* is updated with initialization information from the power manager:

- pmd->new_attr specifies the initial power mode the device should be set to.
- pmd->pmm_flags specifies what persistent storage actions the driver is expected to perform:

  PMD_NO_PSTORE

  if this is set, the driver does not need to use any persistent storage services. If this flag is NOT set, the driver must create persistent storage objects for any driver or device state that must be preserved when the device is set to a PM_MODE_STANDBY mode.

  PMD_NO_PSTORE_INIT

  if this is set, the driver should not initialize itself from its persistent storage object. If this flags is NOT set, the driver should restore the driver or device state from the state it saved in its persistent storage object.

The power manager delivers the specified event to initiate power mode changes. This is typically a pulse, and the driver's pulse handler should use *pmd_handler()* to perform the necessary interaction with the Power Manager to co-ordinate the power mode change. pmd_handler() will call the driver's *setpower()* function to perform the requested power mode change.

pmd_attach() doesn't return an error if there's no power manager running, in order to allow drivers to run as normal (except that there will be no system wide power management policy applied to them). In this case:

- *pmd->hdl* is an invalid handle
- *pmd->new_attr* is set to the device's default PM_MODE_ACTIVE mode

- *pmd*->*pmm_flags* have both PMD_NO_PSTORE and PMD_NO_PSTORE_INIT set

Returns:

0	Success.
-1	An error occurred (errno is set)

Errors:

EINVAL	*pmd* hasn't been correctly initialized.
EINVAL	*event* is not a valid sigevent.
ENOMEM	insufficient memory to allocate power manager resources.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

`pmd_attr_t`, *pmd_attr_init()*, *pmd_attr_setmodes()*, *pmd_attr_setpower()*

pmd_attr_init()

Initialize **pmd_attr_t** *structure with default values*

Synopsis:

```
#include <sys/pm.h>

void pmd_attr_init(pmd_attr_t *pmd);
```

Arguments:

pmd Pointer to the **pmd_attr_t** structure.

Library:

libpm

Description:

The *pmd_attr_init()* function initializes a **pmd_attr_t** with default values:

- *pmd->hdl* to set an invalid handle
- *pmd->cur_attr* and *pmd->new_attr* are set to NULL
- *pmd->setpower* is set to a default function that returns an EINVAL error.

All other fields in *pmd* are set to 0.

After calling *pmd_attr_init()*, a driver should call the following to fully initialize the structure:

- Call *pmd_attr_setmodes()* to specify the device's supported power modes
- Call *pmd_attr_setpower()* to specify the driver's *setpower()* function.

Returns:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_attr_t, *pmd_attr_setmodes()*, *pmd_attr_setpower()*

pmd_attr_setmodes()

Specify supported power modes for a device

Synopsis:

```
#include <sys/pm.h>

void pmd_attr_setmodes(pmd_attr_t *pmd,
                       pm_power_mode_t mode,
                       const pmd_power_mode_t *modes,
                       int nmodes);
```

Arguments:

pmd	Pointer to the **pmd_attr_t** structure to be initialized.
mode	The current device power mode.
modes	An array containing all power modes supported by the device NOTE: *pmd->modes* is simply assigned to modes, so the modes array cannot be an automatic variable.
nmodes	The number of **pmd_mode_attr_t** entries in the modes array.

Library:

libpm

Description:

The *pmd_attr_setmodes()* is used to specify the supported power modes for a device.

The mode argument indicates the current power mode of the device. This mode must be one of the modes listed in the modes array.

If mode corresponds to a PM_MODE_ACTIVE mode, *pmd->last_active* is set.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_attr_t, *pmd_attr_init()*

pmd_attr_setpower()

Specify driver's setpower() function

Synopsis:

```
#include <sys/pm.h>

void pmd_attr_setpower(pmd_attr_t *pmd,
                       pmd_setpower_t func,
                       void *data);
```

Arguments:

pmd	Pointer to the **pmd_attr_t** structure to be initialized.
func	Pointer to the driver specific function to set the device power mode.
data	Pointer to driver specific data passed to *func*.

Library:

libpm

Description:

The *pmd_attr_setpower()* initializes the **pmd_attr_t**'s setpower and data fields to the specified driver specific values.

The *func()* is the driver specific function to set the device power mode:

```
int (*func)(pmd_attr_t *pmd, unsigned flags);
```

Where:

pmd->*cur_attr* points to the **pmd_mode_attr_t** for the current device mode.

pmd->*new_attr* points to the **pmd_mode_attr_t** for the mode to be set.

pmd->*new_flags* contains the mode flags that apply to the new power mode.

This function is responsible for performing all necessary actions to set the new mode:

- if the device is being powered down, it may have to flush driver buffers to the device before powering it down. The PM_MODE_URGENT flag set in *pmd->new_flags* indicates the driver should discard these buffers so that the power down occurs immediately.

  If this draining operation will complete some time later, the function should return at this point with EINPROGRESS to indicate that the mode change has been started, but will complete later. The driver is responsible for calling *pmd->setpower()* itself when this occurs to complete processing of the power mode change.

- if PM_MODE_HWVOL is set in *pmd->new_flags*, the driver may need to save information required to reinitialize the device hardware registers.

- if PM_MODE_NORAM is set in *pmd->new_flags*, the driver may need to save information in its persistent storage object so that driver and device state can be restored when the driver is restarted after a system wakeup.

- once buffers have flushed and any necessary state has been saved, the hardware can be set to the appropriate power level.

- once the power level has been changed, *pmd_confirm()* should be used to inform the Power Manager that the mode change has completed.

- if the mode has changed from a non-active to a PM_MODE_ACTIVE mode, the driver must perform whatever driver specific actions are required to unblock or restart requests that were blocked because they could not proceed in the previous power mode.

pmd->setpower() is called by the following library helper functions:

- by *pmd_handler()* to set the power mode requested by the power manager

- by *pmd_power()* to set the power mode requested by an *iopower_setmode()* call to the driver.

These functions call *setpower()* with the pmd exclusive lock held to serialize power mode changes. If the driver needs to call *pmd->setpower()* itself, for example, when a draining operation completes, it should acquire this lock using *pmd_lock_exclusive()* first.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

`pmd_attr_t`, *pmd_attr_init()*, *pmd_confirm()*, *pmd_activate()*, *pmd_handler()*, *pmd_power()*, *pmd_lock_exclusive()*

pmd_attr_t

Power management support structure for device drivers

Synopsis:

```
typedef struct pmd_attr  pmd_attr_t;

struct pmd_attr {
   pm_hdl_t hdl;
   unsigned pmm_flags; /* flags set by pmd_attach() */

   const pmd_mode_attr_t  *cur_attr;  /* current device power attr  */
   unsigned               cur_flags;  /* flags used to set cur_mode */
   const pmd_mode_attr_t  *new_attr;  /* new attr if in transition  */
   unsigned               new_flags;  /* flags used to set new_mode */
   pm_power_mode_t      last_active;  /* last active mode device was in*/

   const pmd_mode_attr_t  *modes;     /* device power modes   */

   int nmodes;

   void  *data;                       /* driver specific data pointer */
   pmd_setpower_t  setpower;          /* driver mode change function  */
   short
   setpower_flags;                    /* driver flags for setpower() use

   short lock_flag;
   short want_ex;
   short lock_sh;

   struct sigevent  event;

  _Uint32t  seqno;
   Uint32t  reserved[7];
};
```

Description:

The **pmd_attr_t** structure contains driver level power management information used by the **libpm** library functions.

hdl	Handle to the power manager object for the device.
pmm_flags	Contain various flags set by *pmd_attach()*.

cur_attr	Contain the mode flags used to set the current mode.
new_attr	Pointer to the **pmd_mode_attr_t** for the new mode to be set once mode change has been requested. If no mode change is in progress, it is same as *cur_attr*.
new_flags	contains the mode flags used to set the new mode.
setpower	Pointer to a driver specific function invoked to perform a power mode change.
data	pointer to a driver specific data structure passed to the *setpower()* function when it's invoked.
setpower_flags	Available for driver use to set driver specific flags during and across calls to *setpower()*. It's not used by any **libpm** library functions.
last_active	The most recent PM_MODE_ACTIVE that was set. This is the mode used by *pmd_active()* to restore the device to its previous PM_MODE_ACTIVE mode.

Other fields within the **pmd_attr_t** structure are intended only for internal use by the **libpm** library.

Classification:

See also:

pmd_mode_attr_t, *pmd_attr_init()*, *pmd_attr_setmodes()*, *pmd_attr_setpower()*, *pmd_attach()*

pmd_confirm()

Confirm completion of a power mode change to the power manager

Synopsis:

```
#include <sys/pm.h>

int pmd_confirm(pmd_attr_t *pmd, int status);
```

Arguments:

pmd	Pointer to the **pmd_attr_t** structure for the device.
status	The status of the power mode change request.

Library:

libpm

Description:

The *pmd_confirm()* confirms the completion of a power mode change to the power manager and updates the internal fields of the **pmd_attr_t** to reflect the status. If status is EOK, the power mode change was successful and the following are set to indicate the completion of the power mode change:

pmd->cur_mode is set to *pmd->new_mode*

pmd->cur_flags is set to *pmd->new_flags*

If status is not EOK, the power mode change was unsuccessful and the following are set to indicate the power mode change did not occur:

pmd->new_mode is set to *pmd->cur_mode*

pmd->new_flags is set to *pmd->cur_flags*

The *pmd_confirm()* must be called with the **pmd_attr_t** exclusive lock held to serialize power mode changes.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_attr_setpower(), *pmd_lock_exclusive()*

pmd_detach()

Detach a driver from the power manager

Synopsis:

```
#include <sys/pm.h>

int pmd_detach(pmd_attr_t *pmd);
```

Arguments:

pmd Pointer to the device's **pmd_attr_t** structure.

Library:

libpm

Description:

The *pmd_detach()* detaches the device from the power manager and invalidates the handle in *pmd*-**>** *hdl*.

The *pmd_detach()* allows a driver that manages multiple devices to cleanly power down and detach specific devices from the system power management policy.

Driver connections to the power manager are automatically detached during abnormal termination (in which case the device may be left in an unknown state that cannot be recovered from).

Returns:

0 Success.

-1 An error occurred (errno is set).

This function, currently, always succeeds.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_attr_t, *pmd_attach()*

pmd_handler()

Support function for handling power manager requests to change power mode

Synopsis:

```
#include <sys/pm.h>

void pmd_handler(pmd_attr_t *pmd);
```

Arguments:

pmd Pointer to the **pmd_attr_t** for the device.

Library:

libpm

Description:

The *pmd_handler()* function is a support function that the driver calls to perform a power mode change requested by the power manager.

The driver receives these requests via the sigevent supplied to *pmd_attach()*. If the driver has multiple devices, it must ensure that the this sigevent contains sufficient information to allow the driver's event handler to determine which device is being requested.

The *pmd_handler()* essentially performs the following steps:

1. call *pmd_lock_exclusive()* to prevent access to the **pmd_attr_t** structure
2. communicate with the power manager to find out the new mode and flags
3. set pmd->new_mode and pmd->new_flags as required
4. call pmd->setpower() to initiate the mode change
5. call *pmd_unlock_exclusive()* to unlock the **pmd_attr_t**.

The *setpower()* function is expected to call *pmd_confirm()* when the mode change is completed.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_attr_t, *pmd_attach()*, *pmd_attr_setpower()*, *pmd_confirm()*, *pmd_lock_exclusive()*, *pmd_unlock_exclusive()*

pmd_lock_downgrade()

Convert a **pmd_attr_t** *exclusive lock to a shared lock*

Synopsis:

```
#include <sys/pm.h>

int pmd_lock_downgrade(pmd_attr_t *pmd);
```

Arguments:

pmd Pointer to the **pmd_attr_t** structure.

Library:

libpm

Description:

The *pmd_lock_downgrade()* is used to atomically convert an exclusive lock to a shared lock.

Any threads blocked in *pmd_lock_exclusive()* remain blocked until the shared lock is released.

This downgrade preserves the “exclusive preference” for the lock, and unblocks only threads blocked in *pmd_lock_shared()* if there are no threads waiting to acquire the exclusive lock.

Returns:

EOK Success.

Error code from *pthread_sleepon_lock()*, *pthread_sleepon_broadcast()*, and *pthread_sleepon_unlock()*.
An error has occurred.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_lock_upgrade(), pmd_lock_exclusive(), pmd_lock_shared(), pmd_unlock_shared(), pthread_sleepon_lock(), pthread_sleepon_unlock(), pthread_sleepon_broadcast()

pmd_lock_exclusive()

Acquire an exclusive lock on a **pmd_attr_t** *structure*

Synopsis:

```
#include <sys/pm.h>

int pmd_lock_exclusive(pmd_attr_t *pmd);
```

Arguments:

pmd Pointer to the **pmd_attr_t** that is to be locked.

Library:

libpm

Description:

The *pmd_lock_exclusive()* function acquires an exclusive lock on the **pmd_attr_t** structure. This lock is required when calling *pmd->setpower()* or *pmd_confirm()* to protect access to the power mode status.

If the lock is already held either exclusively or shared, *pmd_lock_exclusive()* blocks until all locks are released.

The lock implements an “exclusive preference” so that any thread waiting for an exclusive lock prevents new shared locks being acquired if the lock is currently held shared. This allows the exclusive lock to be acquired as soon as all current shared locks are released.

Returns:

EOK Success.

Error code from *pthread_sleepon_lock()*, *pthread_sleepon_unlock()*, *pthread_sleepon_wait()*
Otherwise.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_unlock_exclusive(), *pmd_lock_downgrade()*, *pmd_lock_shared()*, *pmd_attr_setpower()*, *pmd_confirm()*, *pthread_sleepon_lock()*, *pthread_sleepon_unlock()*, *pthread_sleepon_wait()*

pmd_lock_shared()

Acquire a shared lock on a **pmd_attr_t** *structure*

Synopsis:

```
#include <sys/pm.h>

int pmd_lock_shared(pmd_attr_t *pmd);
```

Arguments:

pmd Pointer to the **pmd_attr_t** structure to be locked.

Library:

libpm

Description:

The *pmd_lock_shared()* function acquires a shared lock on the **pmd_attr_t** structure. Multiple threads can hold this lock and blocks any thread attempting a *pmd_lock_exclusive()* until all shared locks are released.

If the lock is currently held exclusively, *pmd_lock_shared()* blocks until the exclusive lock is released.

The lock implements an “exclusive preference” so that any thread waiting for an exclusive lock prevents new shared locks being acquired if the lock is currently held shared. This allows the exclusive lock to be acquired as soon as all current shared locks are released.

Returns:

EOK Success.

Error code from *pthread_sleepon_lock()*, *pthread_sleepon_unlock()* or *pthread_sleepon_wait()*.
Otherwise.

Examples:

For more information, see the Device Driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_unlock_shared(), *pmd_lock_exclusive()*, *pthread_lock_upgrade()*, *pthread_sleepon_lock()*, *pthread_sleepon_unlock()*, *pthread_sleepon_wait()*

pmd_lock_upgrade()

Upgrade a shared lock to an exclusive lock on a **pmd_attr_t** *structure*

Synopsis:

```
#include <sys/pm.h>

int pmd_lock_upgrade(pmd_attr_t *pmd);
```

Arguments:

pmd Pointer to the **pmd_attr_t** that's currently locked.

Library:

libpm

Description:

The *pmd_lock_upgrade()* converts a shared lock to an exclusive lock.

If there are other shared locks currently held, *pmd_lock_upgrade()* block until those locks are released. Any new attempts to acquire the shared lock is prevented and blocked in *pmd_lock_shared()*.

Returns:

EOK Success

Error code from *pthread_sleepon_lock()*, *pthread_sleepon_wait()*, and *pthread_sleepon_unlock()*.
An error has occurred.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_lock_shared(), *pthread_sleepon_lock()*, *pthread_sleepon_unlock()*, *pthread_sleepon_wait()*

pmd_mode_attr_t

Describe the attributes and capabilities for a device power mode

Synopsis:

```
typedef struct pmd_mode_attr  pmd_mode_attr_t;

struct pmd_mode_attr {
   pm_power_mode_t  mode;
   unsigned flags;
   union
   {
     _Uint32t flags;
     void B  *ptr;
    }
   drvprivate;
   _Uint32t  rsv1;
}
```

Description:

The **pmd_mode_attr_t** structure describes the capabilities for a device power mode. It contains the following:

mode The power mode.

flags Describe the driver capabilities for that mode:

PMD_MODE_ATTR_NORAM

Support the PM_MODE_NORAM flag and can be used for system power modes where system RAM is disabled.

PMD_MODE_ATTR_HWVOL

Support the PM_MODE_HWVOL flag and can be used for system power modes where peripheral device registers are lost.

PMD_MODE_ATTR_WAKEUP

Support system wakeup functionality and can be used to configure the device to act as a wakeup source for low power system standby modes.

drvprivate Contain driver specific information related to the mode. This can be used internally within the driver, for example to indicate which hardware components are functional in that mode. The contents of drvprivate have no defined meaning outside the driver that defines it, although they are returned to external programs using *iopower_getattr()* or *pm_getattr()*, allowing them to examine these attributes if they have driver specific knowledge.

A device driver defines its supported modes using *pmd_attr_setmodes()*. In order to support system standby modes that allow system RAM to be disabled the driver must define at least one PM_MODE_STANDBY mode that specifies the PMD_MODE_NORAM flag. If no modes specify this flag, returning from such a system standby state will cause all driver and device state to be lost.

The mode values should be defined as device specific power modes using the *PM_DEVICE_MODE()* macro.

Examples:

```
/*
 * Define 5 device modes that correspond to the 4 logical power modes
 * The device supports two standby modes:
 * - both support PM_MODE_NORAM and PM_MODE_HWVOL
 * - one mode additionally implements system wakeup functionality
 */
pmd_mode_attr_t device_modes[] = {
   {
   PM_DEVICE_MODE(PM_MODE_ACTIVE, 0),
   0,
   },
   {
   PM_DEVICE_MODE(PM_MODE_IDLE, 0),
   0,
   },
   {
   PM_DEVICE_MODE(PM_MODE_STANDBY, 0),
   PMD_MODE_ATTR_NORAM|PMD_MODE_ATTR_HWVOL,
```

```
    },
    {
    PM_DEVICE_MODE(PM_MODE_STANDBY, 0),
    PMD_MODE_ATTR_NORAM|PMD_MODE_ATTR_HWVOL|PMD_MODE_ATTR_WAKEUP,
    },
    {
    PM_DEVICE_MODE(PM_MODE_OFF, 0),
    0,
    }
};
```

Classification:

See also:

pm_power_mode_t, *iopower_getattr()*, *pm_getattr()*, *pmd_attr_setmodes()*

pmd_power()

Support function for implementing _IO_POWER messages

Synopsis:

```
#include <sys/pm.h>

int pmd_power(void *ctp,
              void *msg,
              pmd_attr_t *attr);
```

Arguments:

ctp Pointer to the **resmgr_context_t** structure for the message request.

msg Pointer to the **io_power_t** message.

pmd Pointer to the **pmd_attr_t** structure for the device.

Library:

libpm

Description:

The *pmd_power()* is a support function that implements the actions required to handle the _IO_POWER message types.

This is called by the *iofunc_power()* in the iofunc layer to handle these messages.

If the driver overrides this function, it should call *pmd_power()* directly. The driver's mount structure should be initialized to point to the **pmd_attr_t** in the mount-**>** power field.

Returns:

Various status values passed back to the resmgr layer to reply to the client message.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

iofunc_power()

pmd_setmode()

Change device power mode

Synopsis:

```
#include <sys/pm.h>

int pmd_setmode(pmd_attr_t *pmd,
                pm_power_mode_t mode,
                unsigned flags);
```

Arguments:

pmd	Pointer to the driver **pmd_attr_t**'s structure for the device.
mode	Power mode to set.
flags	Control the behavior of power mode change.

Library:

libpm

Description:

The *pmd_setmode()* function is used to synchronously change the power mode within device driver code.

The **pmd_attr_t** must be locked using *pmd_lock_shared()* before calling *pmd_setmode()*.

The *pmd_setmode()* essentially performs the following steps:

1. Convert the shared lock to an exclusive lock using *pmd_lock_upgrade()*. This blocks until all other existing shared locks are released.
2. Request permission from the power manager to change to the new mode. If this request is denied, the mode change is aborted with an error.
3. Call *pmd->setpower()* to change the device power mode.
4. Convert the exclusive lock to a shared lock using *pmd_lock_downgrade()*.

Returns:

EOK — Power mode has been changed.

EINPROGRESS — Power mode change was initiated, but will complete later.

EAGAIN — the Power mode was denied by the power manager policy.

EINVAL — *mode* is not a valid power mode or *flags* contains invalid flags.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_attr_setmodes(), *pmd_lock_shared()*, *pmd_lock_upgrade()*, *pmd_lock_downgrade()*

pmd_unlock_exclusive()

Release an exclusive lock on a **pmd_attr_t** *structure*

Synopsis:

```
#include <sys/pm.h>

int pmd_unlock_exclusive(pmd_attr_t *pmd);
```

Arguments:

pmd Pointer to the **pmd_attr_t** structure that is locked.

Library:

libpm

Description:

The *pmd_unlock_exclusive()* release an exclusive lock held on a **pmd_attr_t** structure.

The lock implements an "exclusive preference" so that a thread blocked in *pmd_lock_exclusive()* is unblocked in preference to threads blocked in *pmd_lock_shared()*.

Returns:

EOK Success.

Error code from *pthread_sleepon_lock()*, *pthread_sleepon_unlock()*, *pthread_sleepon_signal()*, *pthread_sleepon_broadcast()*.
An error has occurred.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmd_lock_shared(), *pmd_lock_exclusive()*, *pthread_sleepon_lock()*, *pthread_sleepon_unlock()*, *pthread_sleepon_signal()*, *pthread_sleepon_broadcast()*,

pmd_unlock_shared()

Release a shared lock on a **pmd_attr_t** *structure*

Synopsis:

```
#include <sys/pm.h>

int pmd_unlock_shared(pmd_attr_t *pmd);
```

Library:

libpm

Description:

The *pmd_unlock_shared()* releases a shared lock on a **pmd_attr_t** structure.

If this is the last shared lock and threads are blocked in *pmd_lock_shared()*, one of those threads will be unblocked.

Returns:

EOK Success.

Error code from *pthread_sleepon_lock()*, *pthread_sleepon_unlock()* or *pthread_sleepon_signal()*.
An error has occurred.

Examples:

For more information, see the device driver chapter.

Classification:

Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No

continued...

Safety	
Signal handler	Yes
Thread	Yes

See also:

pmd_lock_shared(), *pmd_lock_exclusive()*, *pthread_sleepon_lock()*, *pthread_sleepon_unlock()*, *pthread_sleepon_signal()*

pmm_init()

Initialise and start the power manager interface

Synopsis:

```
#include <sys/pmm.h>

int pmm_init(dispatch_t *dpp);
```

Library:

libc

Description:

This function call is used to initialise the library and its internal data. These are the operations the power manager performs first.

dpp Pointer to a **dispatch** structure used by the server resource manager.

If *dpp* is NULL, *pmm_init()* allocates its own **dispatch**.

This performs the following:

- allocates the root power manager object.

This results in the policy specific *create()* function being called. If the policy needs to override any of the default policy functions, it must call *pmm_policy_funcs()* and set up the policy specific functions before calling *pmm_init()*.

- attaches the power manager to the pathname space.

  This allows the product specific initialisation code to launch drivers or other power manager clients if necessary. Any subsequent client *pm_attach()* or *pmd_attach()* messages locates the power manager, but is blocked until it has called *pmm_start()* to begin receiving and processing client messages.

If succesful, this call returns 0.

If an error occurs, it returns -1 and *errno* is set one of the following:

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pmm_mode_change()

Trigger change of power mode on a node

Synopsis:

```
#include <sys/pmm.h>

int pmm_mode_change(pmm_node_t *node, pm_power_mode_t mode,
                                                unsigned flags);
```

Library:

libc

Description:

This call is used to change the power mode of a node:

node Pointer to the node.

mode Power mode to be set.

flags Either 0, or PM_MODE_FORCE.

If there is a policy specific *mode_request()* function, it calls to check whether the mode change can be allowed.

If the requested change is allowed, the driver responsible for managing the node's power mode is notified, and this call returns EOK.

The power mode change is asynchronous. *pmm_mode_wait()* is used to wait for the driver to confirm when then mode change is complete.

If an error occurs, it returns -1 and *errno* is set one of the following:

Returns:

EPERM Flags contained PM_MODE_REQUEST or PM_MODE_CONFIRM. These flags are reserved for driver use only.

EACCES	The policy *mode_request()* function did not allow a change to *mode*.
EINVAL	*mode* is not a valid power mode.
ESRCH	No driver has registered to manage the power modes for the power manager object, so the power mode can not be changed.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_mode_get()

Get the current power mode of the node

Synopsis:

```
#include <sys/pmm.h>

int pmm_mode_get(pmm_node_t *node, pm_power_attr_t *attr);
```

Library:

libc

Description:

This call is used to get the current power mode of the node:

node Pointer to the node.

attr Pointer to where the node's power attributes will be copied.

This call always returns EOK.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_mode_list()

Get the list of supported power modes

Synopsis:

```
#include <sys/pmm.h>

int pmm_mode_list(pmm_node_t *node, pm_power_mode_t *modes,
                  int count);
```

Library:

libc

Description:

This call is used to get the list of supported power modes:

node	Pointer to the node.
modes	Pointer to where the list of modes is copied.
count	Size of the *modes* array.

This function returns the actual number of modes supported by the node.

If modes is non-NULL, the minimum of *count* and the actual number of supported modes will be copied to *modes*.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_mode_wait()

Wait until a mode change has been completed

Synopsis:

```
#include <sys/pmm.h>

pm_power_mode_t pmm_mode_wait(pmm_node_t *node);
```

Library:

libc

Description:

This call is used to wait until a mode change has been completed:

node Pointer to the node.

If no mode change is in progress, this call returns immediately. Otherwise, it blocks until either:

- the driver confirms the mode change is complete
- the driver terminates or detaches from the power manager before the mode change is confirmed. In this case, the mode is forced to PM_MODE_UNKNOWN.

On return, this function call returns the current power mode of the node.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No

continued...

Safety	
Signal handler	Yes
Thread	Yes

See also:

pmm_node_create()

Create a new node

Synopsis:

```
#include <sys/pmm.h>

void * pmm_node_create(pmm_node_t *parent, const char *name,
                       mode_t mode);
```

Library:

libc

Description:

This call is used to create a new node in the power manager namespace:

parent Pointer to the node which contains the new node.

If this is NULL, the new node is created at the root of namespace.

name Name of the new node within the *parent* node.

The name must be a single component (i.e cannot contain any '/').

mode Type and access permission of the new node.

If PM_NODE_NEXUS is specified, the new node is a *directory*, otherwise, it is a *leaf* node.

The access permissions follow the standard permissions defined in **< sys/stat.h>**.

If successful, this call returns a pointer to the policy specific data associated with the node:

- if the policy specifies a *create()* function, this is the return valuefrom that function.

  It is assumed that this policy specific data contains a pointer to the newly created **pmm_node_t** to allow subsequent manipulation of the node.

- if the policy does not supply a *create()* function, the return value isa pointer to the new **pmm_node_t** itself, to allow subsequent manipulation of the node.

If this function fails, it returns NULL, and *errno* is set to one of the following error codes:

Returns:

EINVAL	Type encoded in *mode* is non-zero and not PM_NODE_NEXUS.
ENOMEM	Insufficient memory to allocate the necessary data structures.
EEXIST	*parent* already contains a node with *name*.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pmm_node_lookup()

Look up a node by name

Synopsis:

```
#include <sys/pmm.h>

void * pmm_node_lookup(pmm_node_t *parent,
                       const char *name);
```

Library:

libc

Description:

This call is used to look up a node by name:

parent Pointer to the node at which the name resolution begins.

If it is NULL, the name resolution begins at the root of the namespace.

name Name to be resolved. This can contain multiple '/'-separated components.

If successful, this returns the policy specific data assocaietd with the node when it isinitially created.

If the name resolution fails, it returns NULL, and *errno* is set one of the following:

Returns:

ENOENT	Final component of *name* does not exist.
ENOTDIR	one of the intermediate components in *name* is not a directory
EACCES	one of the intermediate directory components in *name* does not have search permission.
ENOMEM	Insufficient memory to allocate required resources.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_node_unlink()

Unlink the node

Synopsis:

```
#include <sys/pmm.h>

int pmm_node_unlink(pmm_node_t *node);
```

Library:

libc

Description:

This call unlinks the node specified by *node*.

If the policy provides a *unlink()* function, it is called to determine whether the policy allows that node to be unlinked. If the *unlink()* function refuses to allow the unlink operation, this call returns an error code (typically EBUSY) that was returned by the *unlink()* function.

The default *unlink()* policy is to allow the unlink to occur.

If the unlink is allowed, this call returns 0:

- if there are no client references, the node is removed from the name space and it will be deleted. The policy specific *destroy()* function is called to free any policy specific data.
- if there are client references, the node is simply removed from the namespace. It will be deleted when the last client reference is removed.
- if *node* is a non-empty directory, any nodes below it are also (recursively) unlinked.

  If any of these nodes have no client references, they are deleted. Otherwise, they are deleted when the last client reference is removed.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pmm_policy_funcs()

Return a pointer to the policy function table

Synopsis:

```
#include <sys/pmm.h>

pmm_policy_funcs_t * pmm_policy_funcs();
```

Library:

libc

Description:

This function returns a pointer to the policy function table:

if the policy is going to override any functions, it should do this before calling *pmm_init()*

pmm_init() creates the root power manager object, which results in a call to the *create()* function.

The caller can then set individual members of this structure to override the default policy actions. For example:

```
pmm_policy_funcs_t *funcs;

funcs = pmm_policy_funcs();
funcs->create = my_create;
funcs->destroy = my_destroy;
```

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_policy_funcs_t

Contain function pointers

Synopsis:

```
typedef struct {
   void * (*create)(pmm_node_t *node, void *parent,
                    const char *name, unsigned flags);
   void  (*destroy)(void *data);
   int (*unlink)(void *data, int nlink, unsigned flags);
   void  (*attach)(void *data, resmgr_context_t *ctp,
                                          int count);
   void  (*detach)(void *data, resmgr_context_t *ctp,
                             int count, unsigned flags);
   void  (*mode_init)(void *data, resmgr_context_t *ctp,
                              const  pm_power_attr_t *attr,
   const pm_power_mode_t *modes);
   int  (*mode_request)(void *data, pm_power_mode_t mode,
                                          unsigned flags,
   const pm_power_attr_t *attr);
   void  (*mode_confirm)(void *data, pm_power_mode_t mode,
                               const  pm_power_attr_t *attr);
   int  (*property_add)(void *data, pm_property_attr_t *prop,
                                                  void *val);
   int (*property_set)(void *data, pm_property_attr_t *prop,
                                                 void *val);
} pmm_policy_funcs_t;
```

Description:

The structure contains function pointers for each of the policy actions.

All these policy functions are called with the node *locked* to prevent concurrent access to the node itself.

Classification:

See also:

pmm_property_add()

Add a new property to a node

Synopsis:

```
#include <sys/pmm.h>

int pmm_property_add(pmm_node_t *node, pm_property_t id,
                                       void *data, int size);
```

Library:

libc

Description:

This calls adds a new property to a node:

node Pointer to the node.

id Property identifier.

Currently only user-defined properties above PM_PROPERTY_USER are supported.

data Pointer to the initial data that represents the property value.

size Size in bytes of the data.

If successful, this function call allocates internal storage to hold a copy of data and returns 0.

If the policy specifies *properety_add()* function, it is called to inform the policy of the new property and its initial value.

If an error occurs, the *errno* is set to one of the following:

Returns:

ENOMEM Insufficient memory to allocate required data structures.

EEXIT Property with the same identifier as *id* already exists.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pmm_property_get()

Get the current value of a property associated with a node

Synopsis:

```
#include <sys/pmm.h>

int pmm_property_get(pmm_node_t *node, pm_property_t id,
                                       void *data, int size);
```

Library:

libc

Description:

This call is used to get the current value of a property associated with a node:

node Pointer to the node.

id Property identifier.

data Pointer to a buffer where the property value is copied to.

size Size in bytes of the *data* buffer.

If no property with an identifier of *id* is present, this call returns EINVAL.

Otherwise, if data is non-NULL, it copies the minimum of *size* bytes and the actual size of the property data to the *data* buffer.

If *id* is valid, this function returns the actual size of the property's data.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_property_list()

Get a list of all properties associated with a node

Synopsis:

```
#include <sys/pmm.h>

int pmm_property_list(pmm_node_t *node, pm_property_attr_t *list,
                                                        int size);
```

Library:

libc

Description:

This call is used to get a list of all properties associated with a node:

node Pointer to the node.

list Pointer to where the property list is copied.

size Number of entries to be returned in *list*.

The return value of this call is the actual number of properties.

If *list* is non-NULL, the minimum of *size* and the actual number of properties are copied to *list*.

Each entry in the returned list specifies:

- the property identifier for the property
- the property size.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_property_set()

Modify the value of a property

Synopsis:

```
#include <sys/pmm.h>

int pmm_property_set(pmm_node_t *node, pm_property_t id,
                                       void *data, int size);
```

Library:

libc

Description:

This call is used to modify the value of a property:

node Pointer to the node.

id Property identifier.

data Pointer to the data that contains the new property value.

size Size in bytes of the *data*.

If successful, this copies the property value from *data* into the internal buffer that holds the property value. If the policy specifies a *property_set()* function, it is called to inform the policy that the property value has changed.

If an error occurs, the *errno* is set one of the following:

Returns:

EINVAL No property with an identifier of *id* exists.

EINVAL ?? Property size does not match *size*.

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_start()

Start a thread pool to handle client requests

Synopsis:

```
#include <sys/pmm.h>

int pmm_start(thread_pool_attr_t *tatttr);
```

Library:

libc

Description:

This starts a thread pool to begin servicing client requests, and is the last action the product specific initialisation code performs:

attr Pointer to the power manager server attributes.

tattr Specifies the attributes of the thread pool used to service client requests.

If *tattr* is NULL, the thread pool is created with internal defaults.

The return value from this function depends on the thread pool flags:

- if *tattr* is NULL, an initial thread pool thread is created and the calling thread returns 0.
- if *tattr*->*flags* specifies POOL_FLAG_USE_SELF, the calling thread becomes the initial thread pool thread and does not return unless an error has occurred.
- if *tattr*->*flags* specifies POOL_FLAG_EXIT_SELF, an initial thread pool thread is created and the calling thread exits without returning.

  In this case, the caller should call *procmgr_daemon()* before calling *>pmm_start()* in order to set the process as a daemon.

If an error occurs, it returns -1 and *errno* is set one of the following:

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pmm_state_change()

Perform a state transition to a new state

Synopsis:

```
#include <sys/pmm.h>

int pmm_state_change(void *hdl, unsigned new_state);
```

Library:

libc

Description:

This function is used to perform a state transtion to a new state.

hdl	Handle to the state machine, returned by *pmm_state_init()*.
new_state	State to be entered. If this is the same as the current state, this call returns without invoking any of the state functions.

This function ensures that only one thread at a time performs a state transition:

- this is serialised with any transition being performed via *pmm_state_machine()*
- this blocks any *pmm_state_check()* function until the transition is completed.

If successful, this call returns EOK.

If an error occurs, it returns -1 and *errno* is set one of the following:

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_state_check()

Obtain the current state

Synopsis:

```
#include <sys/pmm.h>

int pmm_state_check(void *hdl, unsigned *cur_state,
                                unsigned *new_state);
```

Library:

libc

Description:

This function is used to obtain the current state or to re-evaluate the state machine's state by calling the current state's *check()* function:

hdl	Handle to the state machine, returned by *pmm_state_init()*.
cur_state	Pointer to where the current state will be returned.
new_state	Causes the current state's *check()* function to be invoked to re-evaluate the current state and the resulting state is stored in *new_state()*.

This function ensures that:

- only one thread at a time invokes the *check()* function
- it is serialised with any state transition being performed by *pmm_state_machine()* or *pmm_state_change()*.

  This ensures that *cur_state* and *new_state* are consistent.

This function normally returns EOK, and updates *cur_state* or *new_state* if they are non-NULL.

If an error occurs, it returns -1 and *errno* is set to one of the following:

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_state_init()

Create a new state machine

Synopsis:

```
#include <sys/pmm.h>

void * pmm_state_init(int nstates, pmm_state_t *states,
                      void *data, unsigned init_state);
```

Library:

libc

Description:

This function creates a new state machine based on the set of supplied states:

nstates Specifies the number of states in the state machine.

states An array of structures that describe each state

The internal data structure for the state machine directly uses this pointer, so the **states[]** array must be global/static.

data An opaque value passed to each of the state functions. is an opaque value passed to each the state functions

init_state Initial state that the state machine will be set to.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	No

continued...

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pmm_state_machine()

Implement a single threded state machine

Synopsis:

```
#include <sys/pmm.h>

int pmm_state_machine(void *hdl);
```

Library:

libc

Description:

This function executes a loop that implements a single threaded state machine:

hdl Handle to the state machine, returned by *pmm_state_init()*.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

pmm_state_t

Represent a particular state machine

Synopsis:

```
typedef struct {
   unsigned state;
   int (*enter)(void *data, unsigned old_state, unsigned
                                                  new_state);
   int  (*leave)(void *data, unsigned old_state, unsigned
                                                   new_state);
   unsigned (*check)(void *data, unsigned cur_state);
} pmm_state_t;
```

Description:

The **pmm_state_t** structure represents a particular state machine:

state Implementation specific value that identifies this state.

The library attaches no meaning to this value, and only requires that the *state* value for all states within a single state machine are unique.

enter Function that will be called when entering this state.

leave Function that will be called when leaving this state.

check Evaluate whether a state change is necessary.

A *check()* function must be defined for each **pmm_state_t** in a state machine.

Classification:

See also:

pmm_state_trigger()

Trigger state machine thread to re-evaluate current state

Synopsis:

```
#include <sys/pmm.h>

int pmm_start_thread(void *hdl);
```

Library:

libc

Description:

This function is used to trigger a thread executing *pmm_state_machine()* to re-evaluate its current state and perform a state change if necessary:

hdl Handle to the state machine, returned by *pmm_state_init()*.

This function normally returns EOK.

If an error occurs, it returns -1 and *errno* is set to one of the following:

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	Yes
Thread	Yes

See also:

Chapter 8

Callback Reference

This chapter list all power management callbacks in alphabetical order.

- *attach()* callback
- *create()* callback
- *destroy()* callback
- *detach()* callback
- *mode_confirm()* callback
- *mode_init()* callback
- *mode_request()* callback
- *property_add()* callback
- *property_set()* callback
- *unlink()* callback

attach() callback

Called to attach a client to a power manager node

Synopsis:

```
#include <sys/pmm.h>

void attach(void *data, resmgr_context_t *ctp, int count);
```

Library:

libc

Description:

This function is called when a client attaches to a power manager node:

data Policy specific data returned by the *create()* function when the node was created.

ctp Obtain information about the client process.

count Current count of clients attached to the node.

This is called for all clients:

- device drivers, when their *pmd_attach()* call attaches to the node
- other clients, when their *pm_attach()* call attaches to the node

It is provided to allow the policy to keep track of client references if necessary.

There is no default action for this call.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also:

create() callback

Called to get a list of all properties associated with a node

Synopsis:

```
#include <sys/pmm.h>

void * create(pmm_node_t *node, void *parent,
              const char *name, unsigned flags);
```

Library:

libc

Description:

This function is called when a new node is created:

node Pointer to the newly created node.

parent Policy specific data associated with parent of this node in the name space.

name Name of the node within the parent directory.

flags Currently contains either 0 or PMM_NODE_CLIENT:

- if PMM_NODE_CLIENTis set, this node is created in response to a client request
- otherwise, it is created by the power manager itself.

The *create()* function is called from the following operations:

- *pmm_init()* when the root node (in the namespace) is created.

  In this case, since there is no parent node, the parent will be NULL.
- *pmm_node_create()* when code within power manager creates a node.
- a client create request (from the **open** or **mknod** interface).

  In this case, flags will be set to PMM_NODE_CLIENT.

The return value from this function is the policy specific data to be associated with the node. This value is passed to all other policy functions in order to allow the policy to operate on its policy specific data.

The policy would typically use this function to allocate a policy specific data structure.

It should store the *node* value in this structure as it needs to be supplied to the API functions.

The default action (if the policy does not override this function) is to use *node* itself as the policy data. This ensures that whenever any of the other policy functions are called, they will be provided with the *node* pointer to allow them to perform operations on the node.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also:

destroy() callback

Called when a node is destroyed

Synopsis:

```
#include <sys/pmm.h>

void destroy(void *data)
```

Library:

libc

Description:

This function is called when a node is destroyed:

data Policy specific data returned by the *create()* function when the node is created.

The *destroy()* function is called when the last client reference to an unlinked node is removed:

- during an unlink operation — if there are no client references when the node is unlinked
- during the close operation when the last client detaches and the node has already been unlinked.

This function cleans up any policy specific state for this node, and free any data allocated for it.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown

continued...

Safety	
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also:

detach() callback

Called to detach a client from a power manager node

Synopsis:

```
#include <sys/pmm.h>

void detach(void *data, resmgr_context_t *ctp,
                  int count, unsigned flags);
```

Library:

libc

Description:

This function is called when a client detaches from a power manager node:

data Policy specific data returned by the *create()* function when the node was created

ctp Obtain information about the client process

count Current count of clients attached to the node

flags Contain the following:

PMM_NODE_DETACH

The driver responsible for managing the node's power mode has just been detached.

This call is provided to allow the policy to keep track of client references if necessary.

There is no default action for this call.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also:

mode_confirm() callback

Called to confirm the mode change completion

Synopsis:

```
#include <sys/pmm.h>

void mode_confirm(void *data, pm_power_mode_t mode,
                  const pm_power_attr_t *attr);
```

Library:

libc

Description:

This function is called when a driver confirms that a mode change has been completed:

data Policy specific data returned by the *create()* function when the node is created.

mode Power that the driver has set the device to.

attr Contains the current power mode status of the node:

- *attr*->*cur_mode* is the original mode before this mode change has started
- if *attr*->*new_mode* is the mode that the driver is requested to change to
  If this is different to *mode*, it indicates that the driver can not honor the request.
- *attr*->*nxt_mode* is different to *new_mode*, if another mode change is outstanding.
  In this case, that mode change triggers on return from this function and the driver is instructed to change to this mode.

This call is intended to allow the policy to keep track of outstanding mode change requests, and to detect whether the mode change has been completed correctly.

There is no default action for this call.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also:

mode_init() callback

Called when the driver supplies the initial power modes

Synopsis:

```
#include <sys/pmm.h>

void mode_init(void *data, resmgr_context_t *ctp,
                     const pm_power_attr_t *attr,
                   const pm_power_mode_t *modes);
```

Library:

libc

Description:

This function is called when a driver attaches to a power manager node and supplies the initial power modes:

data	Policy specific data returned by the *create()* function when the node was created.
ctp	Obtain information about the driver process.
attr	Describes initial power mode of the device.
modes	Contains a list of the supported power modes. It is specified by *attr* **->** *num_modes*.

This call is provided to allow the policy to determine when a node's power modes become valid. Nodes cannot be power managed until a driver attaches to register. Driver is responsible for implementing the power mode changes for the node.

There is no default action for this call.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also:

mode_request() callback

Called to request power mode change for a node

Synopsis:

```
#include <sys/pmm.h>

int mode_request(void *data, pm_power_mode_t mode,
     unsigned flags, const pm_power_attr_t *attr);
```

Library:

libc

Description:

This function is called when a request is made to change the power mode for a node:

data Policy specific data returned by the *create()* function when the node is created.

mode New mode that is being requested.

flags Contain the following:

PM_MODE_FORCE

Force the object to change to *mode*.

If this is set, the policy should allow the change.

PM_MODE_REQUEST

Use this flag to request permission from the power manager to implement a power mode change.

This is typically be used by a driver to indicate to the power manager that *mode* is a more appropriate mode for the driver in its current state of operation.

attr Contains the current power mode attributes of the node:

- *attr*->*cur_mode* is the current mode of the device

- if *attr->new_mode* is different from *cur_mode*, it indicates that the driver is in the process of changing to *new_mode*, but it hasn't confirmed that that mode change is complete
- if *attr->nxt_mode* is different from *new_mode*, it indicates that a mode change request was received before the driver has changed to *new_mode*.

The return value from this function determines whether the mode change is allowed by the policy:

- if the policy agrees, it should return EOK

  The policy must allow the change if PM_MODE_FORCE is set.

  if PM_MODE_REQUEST is set, the driver changes the mode on return from the power manager request. Otherwise, the library will deliver an event to the driver. This is in order to request a mode change.

- if the policy does not agree and PM_MODE_REQUEST is set, this function should return EAGAIN to inform the driver that it can not change modes.

  If PM_MODE_REQUEST is not set, and the policy does not agree to the mode change, it should return an appropriate error code.

The default action (if the policy does not override this function) is to allow all requests.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown

continued...

Safety	
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also:

property_add() callback

Called when a client adds a new property to a node

Synopsis:

```
#include <sys/pmm.h>

int property_add(void *data, pm_property_attr_t *prop,
                                                 void *val);
```

Library:

libc

Description:

This function is called when a client adds a new property to a node:

data Policy specific data returned by the *create()* function when the node was created.

prop Describes the property:

- *prop*->*id* is the property identifier
- *prop*->*size* is the size of the property data

val Initial property data set by the client.

The return value from this call indicates whether the property can be attached:

- if this returns EOK, the property is allowed, and this call allows the policy to detect when new properties are added.
- if this returns an error, that error code is passed back to the client's *pm_add_property()* call, and the property is not attached to the node.

The default action (if the policy does not override this function) is to allow any property to be added.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also:

property_set() callback

Called when a client modifies a property value

Synopsis:

```
#include <sys/pmm.h>

int property_set(void *data, pm_property_attr_t *prop,
                                                 void *val);
```

Library:

libc

Description:

This function is called when a client modifies a property value:

data Policy specific data returned by the *create()* function when the node was created.

prop Describes the property:

- *prop*->*id* is the property identifier
- *prop*->*size* is the size of the property data.

val Property data set by the client.

The return value from this function is passed back to the client's *pm_set_property()* call.

The property value is set, even if this call returns an error.

It is expected that the policy always return EOK and this call simply serves to notify the policy of the new value. This is used by the policy to trigger actions that change the system state.

The default action (if the policy does not override this function) is to return EOK to the client.

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also:

unlink() callback

Called to check if a node is unlinked from the namespace

Synopsis:

```
#include <sys/pmm.h>

int unlink(void *data, int nlink, unsigned flags);
```

Library:

libc

Description:

This function is called to check whether the policy will allow a node to be unlinked from the namespace:

data Policy specific data returned by the *create()* function when the node was created.

nlink Link count for the namespace:

- non-empty directories will have a count **>** 1.
- empty directories and leaf nodes will have a link count of 1.

flags 0, or a combination of the following:

PMM_NODE_CLIENT

Node was created by a client request.

If this flag is not set, it is created by the power manager.

PMM_NODE_DETACH

Driver responsible for the node's power mode has just detached.

If this flag is not set, the unlink is being performed by an explicit unlink request (via the **resmgr** interface or by *pmm_node_unlink()*)

The *unlink()* function is called from the following:

- *pmm_node_unlink()*. In this case, *flags* is 0 or PMM_NODE_CLIENT.
- the **`resmgr`** unlink handler. In this case, *flags* is 0 or PMM_NODE_CLIENT.
- the resmgr close handler when the driver responsible for the node's power modes detaches.

  In this case, *flags* is PMM_NODE_DETACH set, and may also have PMM_NODE_CLIENT set if the node was created by a client request

If the policy will allow the unlink, it should return EOK:

- on return from this function, the specified node will be unlinked, and if there are no client references, it will be automatically deleted and the *destroy()* function called to clean up policy specific data for the node.
- if the node is a non-empty directory, all nodes below it will also be unlinked.

  Any nodes that have no client references will automatically be deleted, and the *destroy()* function called to clean up policy specific data for the node.
- any nodes that have client references remain in existence until the last client detaches. They will not be visible from the name space, so no new client references can be created.

Otherwise, it should return EBUSY, and the unlink operation will fail.

The default action (if the *unlink()* function is not overriden) is to allow:

Returns:

Classification:

QNX Neutrino

Safety	
Cancellation point	Unknown
Interrupt handler	Unknown
Signal handler	Unknown
Thread	Unknown

See also: