QNX® Neutrino® RTOS

PCI Server User's Guide

BlackBerry | QNX

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: March 06, 2020

Contents

About This Guide

The *PCI Server User's Guide* explains how you can connect to and configure Peripheral Component Interconnect devices. The following table may help you find information quickly:

For information about:	Go to:
An introduction to the PCI server	*Overview*
Details	*Design Details*
Functions, macros, and data types	*API Reference*
Using some general-purpose capability modules	*Capability Modules and APIs*
Sample code	*Example*

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	NULL
Data types	`unsigned short`
Environment variables	*PATH*
File and pathnames	**/dev/null**
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective → Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

DANGER: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
Overview

The PCI server manages the hierarchy of PCI devices attached to a system. This server uses a modular design that consists of the following main components:

`pci-server`

A resource manager that's responsible for enumerating and optionally configuring all PCI/PCIe devices, providing access control to some device information and settings and for all configuration space writes. It's located in **${ *QNX_TARGET*}/*processor*/sbin/** on your development host, and normally installed in **/sbin** on your target.

libpci.so

A linkable library of APIs is located in **${ *QNX_TARGET*}/*processor*/lib** on your development host and normally installed in **/lib** on your target. See the *API Reference* chapter.

Some of these functions require that the calling thread have I/O privileges, which you can get by successfully calling:

```
ThreadCtl( _NTO_TCTL_IO, 0);
```

For more information, see *ThreadCtl()* in the *C Library Reference*.

`pci-tool`

A utility that lets you view devices and the bus/link topology.

`rsrcdb_query`

A utility that lets you view the resource database entries for PCI-related resources, namely available address spaces and vectors for MSI/MSI-X interrupts.

For more information, see the *Utilities Reference*.

There are additional modules (some required, some optional) located in **${ *QNX_TARGET*}/*processor*/lib/dll/pci/** on your development host and in **/lib/dll/pci/** on your target. You must add the latter path to your *LD_LIBRARY_PATH* environment variable. The modules include the following:

File name	Description	In the Design Details chapter, see:
pci_cap-*	PCI capabilities	"*Capability modules*"
pci_debug.so, **pci_debug2.so**	Debug logging modules. On systems that support `slogger2`, use **pci_debug2.so**.	
pci_hw-*	Hardware-specific modules; you must use one of these	"*Hardware-dependent modules*"

File name	Description	In the Design Details chapter, see:
pci_server-*	Server configuration modules	"*The PCI server*"
pci_slog.so, **pci_slog2.so**	PCI system logger modules. On systems that support `slogger2`, use **pci_slog2.so**.	
pci_strings.so	A module that defines commonly used strings	"*Strings module*"
pcie_xcap-*	PCIe extended capabilities	"*Capability modules*"

The PCI server architecture uses optional configuration files (see **${ *QNX_TARGET*}/etc/system/config/pci/*-template.cfg** for templates) and environment variables (some optional, some not; see "*Environment variables*"). The environment variables need to be present in the environment of each executable that uses the PCI server as well as the PCI server itself.

With the exception of the hardware-dependent module (which is required by all executables), you can choose which modules to use.

For example, if you want debug logs only for a driver that you're currently working on, make sure that the *PCI_DEBUG_MODULE* environment variable of every other executable is unset. If you want to test a new capability module but don't want other executables to use it (even though they may support that capability), then you could blacklist the module for every other executable, or you could create a private capability module directory and set the *PCI_CAP_MODULE_DIR* environment variable to that directory only for the executable you're testing (remember to symlink the other capability modules to your private directory).

Environment variables

The PCI server uses various environment variables. Their names are defined in **<pci/pci.h>**, in the form `PCI_ENVVAR_`*variable*. The only mandatory environment variable is *PCI_HW_MODULE*; without a hardware-dependent module specified, there's no way to access the hardware.

Environment variables must be present in the environment of each process that uses **libpci**, including the PCI server process (`pci-server`).

PCI_HW_MODULE

(Required) The name of the hardware-dependent module for your target system; see the **pci_hw-*** files in **${*QNX_TARGET*}/*processor*/lib/dll/pci/** on your development host and in **/lib/dll/pci/** on your target. If you don't set this environment variable, you'll get a SIGSEGV.

PCI_HW_CONFIG_FILE

(Optional) The name of a hardware-dependent configuration file (see "*Configuration files*").

PCI_SLOG_MODULE

(Optional) The name of the logging module (e.g., **/lib/dll/pci/pci_slog2.so**). If unset, no logs are recorded.

PCI_DEBUG_MODULE

(Optional) The name of the debug logging module (e.g., **/lib/dll/pci/pci_debug2.so**). If unset, no debug information is logged.

PCI_CAP_MODULE_DIR

(Optional) The location of an alternate capability module directory (the default is **/lib/dll/pci**).

PCI_BASE_VERBOSITY

(Optional) Controls the amount of log detail for every module and library. Its value should be an unsigned integer. The default is 0; the higher the value, the more information is logged.

This environment variable works in conjunction with the *pci_verbosity* variable that's declared in **<pci/pci.h>**:

- For programs that don't manipulate the *pci_verbosity* variable, the value behaves the same as incrementing the verbosity parameter by the value of the *PCI_BASE_VERBOSITY* environment variable.
- For programs that do manipulate the *pci_verbosity* variable, this base level changes the default from 0 to the specified value, and any command-line options to increase *pci_verbosity* increase it from the base value set in the environment variable.

PCI_SERVER_BUSCFG_MODULE

(Optional) For the PCI server, this environment variable lets you select a bus configuration module. It's optional because systems that already have configured the PCI bus don't require the PCI server to redo this operation.

This variable is required only if the PCI server configures the PCI bus (e.g., assigning secondary/subordinate bus numbers to bridges, (re)calculating address space requirements and assigning them to devices by updating BAR registers). If the PCI server is only enumerating an existing configuration, a bus configuration module isn't required, and the environment variable is ignored.

If a configuration is performed, this environment variable is still optional, and the generic module identified by PCI_SERVER_BUSCFG_MODULE_DEFAULT is used by default. This environment variable lets you use alternative and/or board-specific configuration modules.

PCI_MODULE_BLACKLIST

(Optional) A colon-separated list of modules that you don't want to load. This is useful for testing and for controlling what capabilities a driver can use without having to recompile or use command-line controls.

The blacklist is checked when an attempt is made to load a module. Once a module has been successfully loaded, the blacklist is no longer consulted for that module, so it's important that the blacklist be created before a module is loaded for the first time.

While generally applicable to any PCI module, the blacklist's main purpose is to control the use of capability modules without modifying the driver software itself. For example, a driver written to use MSI-X and MSI (preferentially in that order) could be started with the MSI-X capability module blacklisted so that it uses the MSI module, without changing software or requiring a command-line option to the driver.

Because this is an environment variable, this control can be provided on a per-driver basis, but note that when this variable is specified for the PCI server, the effect is more global in nature. Blacklisting any capability modules in the server environment actually prevents their use for *all* driver software regardless of the setting of the variable in that driver's environment. Drivers can still read the device capabilities, but can't enable or use them.

You can specify modules as a simple name or with a leading slash. If you use a simple name, the module must exist in the **/lib/dll/pci/** directory. If the first character of the name is a slash, the name must specify the full path to the module. Only an exact pathname match prevents the loading of a module. Keep this in mind if the capability module directory has been altered with the *PCI_CAP_MODULE_DIR* environment variable.

Because a vid/did-specific capability module is always checked for before a generic capability module, if you're using a vid/did-specific capability module and wish to prevent the use of that capability, you should blacklist both the vid/did-specific and the generic modules.

PCI_SERVER_NODE_NAME

(Optional) The node name used to communicate with the PCI server (the default is **/dev/pci**).

This variable is required only if the PCI server is started with a different node name (with either the server's `-n` option or the `SERVER_NODE_NAME` parameter in a configuration file specified with the `--config` option, the `-n` option taking precedence).

You typically don't need to change this name, and we don't encourage you to do so.

PCI_STRINGS_FILE

(Optional) The strings database file. If not set, the default file, **/etc/system/config/pci/pcidatabase.com-tab_delimited.txt**, is used. See "*The Strings module*."

Configuration files

There are two primary (and optional) configuration files:

- The hardware-dependent configuration file, specified with the *PCI_HW_CONFIG_FILE* environment variable, is applicable to all driver software and the PCI server process. It contains board-specific configuration information, such as PCI Interrupt Pin assignment and/or IRQ mapping, slot and chassis numbering information, and so on.
- The server configuration file is applicable to only the PCI server process.

All support files, including configuration files and templates, are contained in **${ *QNX_TARGET*}/etc/system/config/pci/** on your development host, and are installed in **/etc/system/config/pci/** on your target. You don't have to keep your configuration files there, but it does provide a consistent location.

The files **pci_hw-template.cfg** and **pci_server-template.cfg** contain a great amount of detail about their use.

When creating a custom configuration file, you can remove any sections or parameters that don't apply to your system. Removing the comments also helps to speed up the parsing. Just make sure not to delete any section tags for which you want to set a parameter, or it won't be interpreted. Empty sections are OK.

Chapter 2
Design Details

As previously stated, the PCI server architecture is very modular. The design is analogous to the original PCI specification in the way that it allows the extension of the specification with minimal or no impact on backward compatibility. This design has the following benefits:

- It's extensible: new functionality can be delivered in a modular fashion.
- Only the required modules are loaded, reducing code size.
- There's reduced impact to the code base when bugs are fixed.

Module loading

Module loading is implicit, meaning that any code using the **libpci** APIs load the required modules automatically. This eliminates the need for drivers to perform any PCI-related module management.

Which modules are loaded is controlled by environment variables. At the time of module loading, if the environment variable associated with that module isn't set, the functionality is deemed unnecessary, and there are no further attempts to load the module. The exception to this is the hardware-dependent module; failure to set the *PCI_HW_MODULE* environment variable results in a SIGSEGV.

Capability module loading is similar in that there's no module management required of driver software, but it differs slightly in that driver software does get to explicitly trigger when a capability module is loaded. The *Capability Modules* appendix contains more details on this process.

Configuration space access

By design, each driver process has read access to the configuration space of the device it's managing, via the **libpci** and capability module APIs.

This is enabled by the hardware-dependent module when it's demand-loaded on the first hardware access. All configuration space writes are handled by the PCI server. This design allows for optimal efficiency for reads because there's no context switch as a result of a message pass when performing read operations.

Access to device configuration space registers is facilitated through a set of read and write APIs. Most read APIs require only a BDF (Bus/Device/Function—"Bus" is synonymous with a PCIe link) value of type `pci_bdf_t`. This value can be constructed using the *PCI_BDF()* macro, but more usually it's obtained from a successful call to *pci_device_find()*.

Write APIs and APIs that provide certain types of information (such as address space and interrupt information) require a handle (of type `pci_devhdl_t`) that you obtain from a successful call to *pci_device_attach()*.

Read and Write APIs for common configuration space registers are provided by specifically named APIs (see the *API Reference* chapter for details). For example, to read and write the command register, use *pci_device_read_cmd()* and *pci_device_write_cmd()*. This provides a self-documenting interface for driver code. Access to device specific registers is provided by the *pci_device_cfg_rd*()* and *pci_device_cfg_wr*()* APIs.

The PCI server

The PCI server (`pci-server`) is a resource manager process that manages the PCI hierarchy.

It's responsible for enumerating and optionally configuring all PCI/PCIe devices, providing access control to some device information and settings and for all configuration space writes. For information about the command-line options, see the entry for `pci-server` in the *Utilities Reference*.

Without any command-line arguments, the PCI server enumerates the PCI bus, but doesn't configure it. This is the mode of operation for systems that have already enumerated and configured the PCI devices.

If required (or desired), the PCI server configures a bus, using the bus configuration module specified with the *PCI_SERVER_BUSCFG_MODULE* environment variable (or **/lib/dll/pci/pci_server-buscfg-generic.so** if that variable is unset). A bus is configured if you provide the `--config=`*filename* option, and *filename* is a server configuration file that contains the parameter `DO_BUS_CONFIG=yes/true` in the `[buscfg]` section.

As mentioned in "*Configuration files*" (and as documented in **${*QNX_TARGET*}/etc/system/config/pci/pci_server-template.cfg**), the PCI server can load other optional modules, each with its own set of arguments.

Server modules

To provide maximum flexibility, the PCI server can use a number of server-specific modules. One such module is for device configuration. As previously mentioned, device configuration is optional, and so this functionality is supplied by a bus configuration module that you can specify with the *PCI_SERVER_BUSCFG_MODULE* environment variable. If not specified, and configuration is to be done, the default module **pci_server-buscfg_generic.so** is used.

The server module mechanism is a means of extending the functionality of the PCI server while minimizing impacts to the server code base. This is facilitated by a set of internal APIs between the server and server modules. It doesn't necessarily mean that a new `pci-server` binary won't be required when a new module is created, but because of the defined module interface, the impacts will be localized and minimized. It all depends on the functionality being added.

Server configuration modules are contained in **${*QNX_TARGET*}/*processor*/lib/dll/pci** and have the prefix `pci_server-`. There is "use" information in each module explaining what it's for and how to use it.

The "Known Devices" list (`pci_db`)

The PCI server creates and maintains a known devices list in **/dev/shmem/pci_db**. This database is created during the enumeration phase. If live insertion and deletion of devices (i.e., hotplug) is enabled, this database is updated by the PCI server to reflect the current hierarchy.

This shared object is available to every process that uses **libpci**, and this database is consulted when you call *pci_device_find()*. This allows every driver process to search for supported devices independently of one another and without ever generating any PCI bus/link activity. You can use `rsrcdb_query` to view the resource database entries.

Hardware-dependent modules

The hardware-dependent module is a required module that's responsible for providing access to the hardware for every process that uses **libpci** and capability module APIs, including the `pci-server`.

Hardware-dependent modules reside in **${*QNX_TARGET*}/*processor*/lib/dll/pci/** on your development host, and in **/lib/dll/pci/** on your target. Their names are prefixed with `pci_hw-`. There's "use" information in each module explaining what it's for and how to use it.

Both the PCI server process and the hardware-dependent module can use a configuration file specified with the *PCI_HW_CONFIG_FILE* environment variable. The format of this configuration file is documented in **${*QNX_TARGET*}/*processor*/etc/system/config/pci/pci_hw-template.cfg** (or **/etc/system/config/pci/pci_hw-template.cfg** on your target system).

In addition to the common sections applicable to all hardware-dependent modules, hardware module developers may choose to define sections specific to a certain hardware module. This may include sections for optional parameters that provide users with the ability to configure or tune the hardware, for dealing with platform variances or for managing out-of-spec behavior or errata. The "use" information within the hardware-dependent module provides all details pertaining to specific configuration file sections and likely includes a specific hardware configuration file either because it's required, or as a representative example.

Capability modules

The original PCI specification defined a mechanism for extending itself based on the idea of adding capability fields with the configuration space of a device.

PCI Express (PCIe) is now supplanting most new PCI-based designs, but from a software perspective, it's just additional functionality that's managed by a set of defined PCIe capability registers within PCI configuration space. This mechanism has allowed the PCI specification to evolve while maintaining backward compatibility with previous software installations.

The PCI server architecture enables the use of new hardware capabilities in an analogous fashion with the ability to deliver access to these new capabilities with independent software modules and associated APIs.

Capability modules reside in **${*QNX_TARGET*}/*processor*/lib/dll/pci/** on your development host, and in **/lib/dll/pci/** on your target. Their names are prefixed with `pci_cap-` or `pcie_xcap-` for PCI or PCIe extended capabilities, respectively. There is "use" information in each module explaining what capability it's for, its supported APIs, and how to use it. See also the *Capability Modules and APIs* appendix.

Similar to the hardware-dependent and logging modules, capability modules are demand-loaded and automatically managed for users, but unlike those other modules, capability modules aren't loaded until software decides that they're required. As an example, a device might support both MSI and MSI-X capabilities for message signalled interrupts, however a driver for that device can decide which of the two it prefers and load only the selected one. In fact all capability modules are managed this way, even the PCI Express capability.

Adding new capabilities modules

The name of a capability module is in one of these forms:

- `pci_cap-0x`*nn*`.so` for PCI
- `pcie_cap-0x`*nnnn*`.so` for PCIe

where *nn* and *nnnn* are the PCISIG-assigned capability IDs. The number of digits is important; it must be two for PCI capability IDs, and four for PCIe extended capability IDs. For example:

- **pci_cap-0x10.so** is the PCIe capability module
- **pci_cap-0x05.so** is the MSI capability module
- **pcie_xcap-0x0001.so** is the PCIe extended capability module for Advanced Error Reporting (AER).

To add a capability to the PCI server, put a capability DLL into the **/lib/dll/pci/** directory (or into the directory named by the *PCI_CAP_MODULE_DIR* environment variable). A capability header file (if required) defines the APIs to the specific capability. Capability header files are typically named with the name of the capability and not the ID (e.g., **cap_msi.h**). This is to allow them to be more easily identified within the files that include them.

The goal of this mechanism is that only the users of the capability (i.e., driver software) and the capability provider (i.e., the **pci_cap-0xID.so** DLL) will need to be updated. All other modules, the PCI server library, and the PCI server itself typically remain unchanged. (Depending on the capability, it may be necessary to update the PCI server itself to also use a capability.)

How capabilities are processed

In order to allow (and in fact force) drivers to understand, select, and configure which capabilities they want to use, the PCI server architecture pushes capability processing into the driver software. This has the following advantages:

- Capability list processing isn't done at device discovery, thereby speeding up enumeration.
- If there isn't a module to process a capability, the capability isn't discovered.
- Capabilities modules aren't loaded unless at least one driver intends to use them.

In order for a capability module to be used, it must be known to both the driver wanting to use it and the PCI server process. If a capability module is blacklisted (see the *PCI_MODULE_BLACKLIST* environment variable) in the PCI server, it can't be used even if it isn't blacklisted in the driver process. Similarly, if you use the *PCI_CAP_MODULE_DIR* environment variable to specify an alternate directory for capability modules (from the default **/lib/dll/pci**), you must still make sure that any capability that's to be used in the system is visible to both the PCI server and the driver wishing to use it. You can accomplish this by making sure both have the same *PCI_CAP_MODULE_DIR* or by ensuring that the directory identified by *PCI_CAP_MODULE_DIR* and the default directory (**/lib/dll/pci**) both contain the desired capability module (via a symlink or copy).

Note that the PCI server process must have access to the capability modules required by *any* driver on the system, not just those of a specific driver, and therefore the PCI server process typically uses the default location for capability modules (i.e., it doesn't have a *PCI_CAP_MODULE_DIR* environment variable). You can then use *PCI_CAP_MODULE_DIR* to restrict the availability of capabilities to specific drivers.

You can also use this environment variable to simply modify the default location of capability modules for every process.

Driver initiation of capabilities processing

Driver software has two choices for discovering the supported capabilities of the device it's managing. It can call *pci_device_find_capid()* to determine if a specific capability exists, or it can scan for all device capabilities with repeated calls to *pci_device_read_capid()*. Once the driver determines which capabilities the device supports and which ones it intends to enable, it can call *pci_device_read_cap()* to obtain a handle (of type `pci_cap_t`) to the capability. The act of obtaining the handle is what triggers the capability module to be loaded, and therefore the driver has complete control over this process. Only after a successful call to *pci_device_read_cap()* can the driver use any capability-specific APIs, since it's the `pci_cap_t` handle that's required by all capability module APIs.

If a device supports a capability for which there's no capability module, *pci_device_read_cap()* returns an error.

Note that there's currently no way to unload a capability module. Once a module is read, it becomes part of the running process image; therefore in order to minimize the runtime image size, drivers should read in only the capability modules that they intend to use.

The other APIs related to capabilities processing that are available to driver software include:

- *pci_device_cfg_cap_enable()*
- *pci_device_cfg_cap_disable()*
- *pci_device_cfg_cap_isenabled()*

Before a capability can be used, it typically must be enabled. Before enabling a capability, the driver should use the capability APIs to configure it. Enabling the capability causes the current capability configuration to take effect. If a change is to be made to the capabilities configuration, the driver should disable the capability, reconfigure it using the capability APIs, and then re-enable it.

Some capabilities, such as PCI Express (PCIe), are always enabled and can't be disabled. For these capabilities, *pci_device_cfg_cap_isenabled()* returns true, *pci_device_cfg_cap_enable()* returns PCI_ERR_EALREADY, and *pci_device_cfg_cap_disable()* returns PCI_ERR_ENOTSUP.

Handling device errata and out-of-spec behavior

Capability modules are written to comply with specifications and necessarily use the hardware-dependent module (which itself can deal with certain types of device or platform errata), however in order to handle device errata or out-of-spec behavior related to device capabilities, the following mechanism is provided. Since a specific PCI device is uniquely identified by its vendor and device IDs, we allow for a device-specific capability module using the same naming convention with an additional tag that consists of the vendor and device IDs (*vid* and *did*): **pci_cap-***id-vid did***.so** (e.g., **pci_cap-0x10-808610d3.so**).

When a capability module is to be loaded (triggered by *pci_device_read_cap()*), an attempt is first made to load the capability module with a matching *vid did* tag as part of the filename; if found, then this module is used (discovering an already loaded module with the same *vid* and *did* means that this would be the second instance of the same device). If not found, then an attempt to find the generic capability module is made. If one is found that's already loaded, it's used; otherwise the generically named capability module is loaded. If none can be found, the capability isn't supported.

This mechanism allows the inclusion of a capability module for a specific device by simply dropping the appropriately named file into the **/lib/dll/pci/** directory (or the directory identified by *PCI_CAP_MODULE_DIR*) and restarting the device driver and PCI server process. (Vendor and device IDs are all that's used to uniquely identify a specific capability module. If multiple devices with the same vid/did exist and the named module is applicable to only one of the devices, then the specific module must handle both devices.)

The “Strings” module

One of the goals of the PCI server design is to eliminate the dependency of the code base on external factors such as vendor and device names and the addition of new devices, etc. To accomplish this, the PCI server and library don't contain any textual information related to vendors, devices, or class codes. This information is provided in the form of `#defines` (see **<pci/pci_id.h>** and **<pci/pci_ccode.h>**) based on PCISIG assignments so that they're available to driver software when calling *pci_device_find()*, but the text associated with these definitions isn't.

Instead, a strings module that contains APIs to access this information is available for linking tools and utilities against, as it's really only this software that needs to display such human-readable information. The strings module is the only module that software will link against and hence create a version dependency.

API definitions for the strings module are contained in **<pci/pci_string.h>**.

The class code strings within the strings module are fixed and as defined by the PCISIG. They correspond to the class codes defined in **<pci/pci_ccode.h>**.

The vendor and device strings are obtained by parsing a strings database file contained in **${*QNX_TARGET*}/etc/system/config/pci/** on your development host, and normally installed in **/etc/system/config/pci/** on your target. The current version of the strings module expects a tab-delimited file. The original source of this file was obtained from **pcidatabase.com**, but you can edit to add, delete, and modify the strings as required. The database file is specified with the *PCI_STRINGS_FILE* environment variable. If not set, the default file, **/etc/system/config/pci/pcidatabase.com-tab_delimited.txt**, is used. This allows users to freely edit the database file to list only known devices or to add custom ones without any PCI server changes.

The benefits include:

- It leverages an existing database of vendor and device strings.
- The information isn't hard-coded into any code.
- Customers can update or edit the file as they wish.
- Community updates can be easily provided to all through forum posts, etc.
- For systems with no slots, the file can be edited down to include only the devices known to exist on the particular hardware, thus reducing the overhead of processing a large file.

Because the strings module is a separate and independent module, the format of the database file can change in the future without impacting other PCI server subsystems.

Chapter 3
API Reference

This chapter describes the PCI functions, macros, and data types, in alphabetical order.

Some of these functions require that the calling thread have I/O privileges, which you can get by successfully calling:

```
ThreadCtl( _NTO_TCTL_IO, 0);
```

For more information, see *ThreadCtl()* in the *C Library Reference*.

pci_bdf()

Get the Bus/Device/Function associated with a device handle

Synopsis:

```
#include <pci/pci.h>

pci_bdf_t pci_bdf( pci_devhdl_t hdl );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_bdf()* function is a convenience function that allows software to keep only the `pci_devhdl_t` handle for any devices that it manages. After the device is found and successfully attached, you can use this function to obtain the Bus/Device/Function (of type `pci_bdf_t`) associated with *hdl* in order to use the read APIs. Drivers can keep the `pci_bdf_t` as well.

Returns:

If *hdl* is valid, the `pci_bdf_t` associated with the device when it was successfully attached; otherwise PCI_BDF_NONE.

Classification:

QNX Neutrino

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

pci_bdf_t

Base type that encodes a Bus/Device/Function

Synopsis:

```
#include <pci/pci.h>

typedef uint32_t                pci_bdf_t;
```

Description:

The `pci_bdf_t` data type is a base type that encodes a Bus/Device/Function, which uniquely identifies a PCI device node. The encoding is as follows:

```
xxxx xxxx xxxx xxxa bbbb bbbb dddd dfff
```

The ARI (Alternate Routing ID Interpretation) encoding is as follows:

```
xxxx xxxx xxxx xxxa bbbb bbbb ffff ffff
```

There's no device field in the ARI encoding; it's implied to be 0.

Note that bit 16 is used to determine whether the BDF encoding should be interpreted as in ARI format (1) or not (0). You can use the *PCI_IS_ARI(_bdf_)* macro to check this bit.

You can use the following macros to build a BDF:

- *PCI_BDF(_b_, _d_, _f_)*
- *PCI_BDF_ARI(_b_, _f_)*

You can use the following macros to obtain the bus, device, and function numbers from a BDF:

- *PCI_FUNC(_bdf_)*
- *PCI_DEV(_bdf_)*
- *PCI_BUS(_bdf_)*

PCI_BDF_NONE indicates a nonexistent or illegal device.

Classification:

QNX Neutrino

pci_chassis_slot_bridge()

Get the Bus/Device/Function of the bridge that's connected to the specified chassis and slot

Synopsis:

```
#include <pci/pci.h>

pci_bdf_t pci_chassis_slot_bridge( pci_cs_t cs );
```

Arguments:

cs

A `pci_cs_t` that contains the chassis and slot number; see below.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_chassis_slot_bridge()* function returns the BDF of the bridge that's connected to the specified chassis and slot. The `pci_cs_t` data type is defined as a `uint16_t`; the encoding is as follows:

```
cccs ssss ssss ssss
```

You can use the following macros to encode or decode a `pci_cs_t`:

- *PCI_CHASSIS(_cs_)*
- *PCI_SLOT(_cs_)*
- *PCI_CS(_c_, _s_)*

There's a maximum of 32 slots per chassis, as defined by the Slot Identification capability, but there are 13 bits for slot numbers to accommodate the size of the “physical slot number” field in the slot capabilities register of PCIe root ports. The 16-bit data type for `pci_cs_t` permits up to 8 chassis (7 expansion chassis).

Slot numbers start at 1.

Even if there are no chassis-based devices in the system, a chassis number of 0 exists. That is, a board with PCI devices on it (perhaps including slots) will logically reside in a chassis 0.

Returns:

The BDF of the bridge, or PCI_BDF_NONE if the chassis and slot provided were invalid or aren't associated with a bridge or Root Port.

A PCI bridge (not a PCIe Root Port) may be connected to several slots. In this case, the same BDF is returned for each slot connected to the bridge.

Classification:

QNX Neutrino

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

See also:

pci_chassis_slot_bridge(), *pci_device_chassis_slot()*

pci_chassis_slot_device()

Get the Bus/Device/Function of the device that resides in the specified chassis and slot

Synopsis:

```
#include <pci/pci.h>

pci_bdf_t pci_chassis_slot_device( pci_cs_t cs,
                                   pci_err_t *err );
```

Arguments:

cs

A `pci_cs_t` that contains the chassis and slot number; see below.

err

NULL, or a pointer to a location where the function can store an error code.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_chassis_slot_device()* function returns the BDF of the device that resides in the specified chassis and slot. The `pci_cs_t` data type is defined as a `uint16_t`; the encoding is as follows:

```
cccs ssss ssss ssss
```

You can use the following macros to encode or decode a `pci_cs_t`:

- *PCI_CHASSIS(_cs_)*
- *PCI_SLOT(_cs_)*
- *PCI_CS(_c_, _s_)*

There's a maximum of 32 slots per chassis, as defined by the Slot Identification capability, but there are 13 bits for slot numbers to accommodate the size of the “physical slot number” field in the slot capabilities register of PCIe root ports. The 16-bit data type for `pci_cs_t` permits up to 8 chassis (7 expansion chassis).

Slot numbers start at 1.

Even if there are no chassis-based bridges in the system, a chassis number of 0 exists. That is, a board with PCI devices on it (perhaps including slots) will logically reside in a chassis 0.

Returns:

The BDF of the device, or PCI_BDF_NONE. If PCI_BDF_NONE is returned and *err* isn't NULL, the object that *err* points to indicates why:

PCI_ERR_EINVAL

The chassis and slot provided were invalid.

PCI_ERR_ENODEV

The slot is empty.

Classification:

QNX Neutrino

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

See also:

pci_chassis_slot_bridge(), *pci_device_chassis_slot()*

pci_device_attach()

Attach a device

Synopsis:

```
#include <pci/pci.h>

pci_devhdl_t pci_device_attach( pci_bdf_t bdf,
                                pci_attachFlags_t flags,
                                pci_err_t *err );
```

Arguments:

bdf

A unique identifier for the device you want to attach to. You can construct this value with the *PCI_BDF()* macro, but you more typically obtain it by calling *pci_device_find()*.

flags

Flags that control the attachment; see below.

err

NULL, or a pointer to a location where the function can store an error code.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

Device attachment is required in order to perform any configuration and/or write operations on a device. It's also required in order to obtain any address space (memory and I/O BAR) or interrupt information. This is to allow the attachment flags to provide access control only to software that requires this information. Attachment isn't required to read most other configuration space information. Any APIs that require a `pci_devhdl_t` must be attached to.

The *flags* parameter controls the way in which the attachment occurs as follows (see also **<pci/pci.h>**). You must specify one of the following bitwise-exclusive values:

pci_attachFlags_e_EXCLUSIVE

If this attachment is successful, don't allow any other caller to attach to the device.

pci_attachFlags_e_SHARED

Allow other callers to attach to the device.

You can also OR in any of the following bits:

pci_attachFlags_e_OWNER

Permit access to the address space and IRQ assignments. This flag is implied for pci_attachFlags_e_EXCLUSIVE. There can be only one owner unless you set pci_attachFlags_e_MULTI on the very first attachment that includes pci_attachFlags_e_OWNER.

pci_attachFlags_e_MULTI

If this flag is set on the first attachment with pci_attachFlags_e_OWNER set, then subsequent attachments are permitted with the pci_attachFlags_e_OWNER and pci_attachFlags_e_MULTI flags set.

If you don't set this flag on the first attachment that has pci_attachFlags_e_OWNER set, then subsequent attaches with pci_attachFlags_e_OWNER set are rejected regardless of the inclusion of pci_attachFlags_e_MULTI. That is, all multiply-owned attachers *must* set both the pci_attachFlags_e_OWNER and pci_attachFlags_e_MULTI flags. It's an error to set pci_attachFlags_e_MULTI with pci_attachFlags_e_EXCLUSIVE or without pci_attachFlags_e_OWNER.

The following combinations are also defined:

```
pci_attachFlags_OWNER = (pci_attachFlags_e_SHARED | pci_attachFlags_e_OWNER)
pci_attachFlags_MULTI_OWNER = (pci_attachFlags_e_SHARED | pci_attachFlags_e_OWNER |
                               pci_attachFlags_e_MULTI)
pci_attachFlags_EXCLUSIVE_OWNER = pci_attachFlags_e_EXCLUSIVE | pci_attachFlags_e_OWNER
pci_attachFlags_DEFAULT = pci_attachFlags_OWNER
```

Returns:

A handle of type `pci_devhdl_t` that you can use in subsequent calls to access the device, or NULL if an error occurred.

If *err* is non-NULL, then the reason for the failed attachment is provided as follows:

PCI_ERR_EINVAL

Invalid flags.

PCI_ERR_ENODEV

The *bdf* argument doesn't refer to a valid device.

PCI_ERR_ATTACH_EXCLUSIVE

The device identified by *bdf* is already exclusively owned.

PCI_ERR_ATTACH_SHARED

The request was for exclusive attachment, but the device identified by *bdf* has already been successfully attached to.

PCI_ERR_ATTACH_OWNED

The request was for ownership of the device, but the device is already owned.

The following error codes are specifically related to internal resource availability and use, and hence you aren't likely to see them:

PCI_ERR_ENOMEM

Memory for internal resources couldn't be obtained. This may be a temporary condition.

PCI_ERR_LOCK_FAILURE

There was an error related to the creation, acquisition, or use of a synchronization object.

PCI_ERR_ATTACH_LIMIT

There have been too many attachments to the device identified by *bdf*.

Classification:

QNX Neutrino

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

See also:

pci_device_detach()

pci_device_cfg_cap_disable(), pci_device_cfg_cap_enable()

Enable or disable a capability

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_cfg_cap_disable( pci_devhdl_t hdl,
                                      pci_reqType_t reqType,
                                      pci_cap_t cap);

pci_err_t pci_device_cfg_cap_enable( pci_devhdl_t hdl,
                                     pci_reqType_t reqType,
                                     pci_cap_t cap);
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

reqType

The desired level of compliance to the request; one of:

- pci_reqType_e_MANDATORY — the request must be satisfied with the requested constraints, or the request fails.
- pci_reqType_e_ADVISORY — every attempt should be made to meet the request constraints, but if that isn't possible, the call doesn't fail.
- pci_reqType_e_UNSPECIFIED — the request constraints aren't specified. The PCI server is free to choose any request constraints.

cap

A pointer to a capability structure.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

These functions enable or disable the capability identified by *cap* for the device identified by *hdl*. You must have already attached to the device by successfully calling *pci_device_attach()*. You can obtain the device capability by calling *pci_device_read_cap()*.

After calling *pci_device_read_cap()* and before calling *pci_device_cfg_cap_enable()*, you might be able to modify the capability, using capability-specific APIs. See the specific capability module for details.

The *reqType* argument indicates the desired level of compliance to the request:

When you're:	Use:
Enabling a capability	pci_reqType_e_MANDATORY, pci_reqType_e_ADVISORY, or pci_reqType_e_UNSPECIFIED
Disabling a capability	pci_reqType_e_UNSPECIFIED, unless the specific capability module specifies otherwise

For example, if you're enabling the MSI or MSI-X capability, you can select the desired number of IRQs to use by calling the capability-specific APIs for MSI or MSI-X. When you're enabling the capability for the device:

- a *reqType* of pci_reqType_e_MANDATORY causes *pci_device_cfg_cap_enable()* to fail if the desired number of IRQs couldn't be allocated for the device
- a *reqType* of either pci_reqType_e_ADVISORY or pci_reqType_e_UNSPECIFIED wouldn't

For an explanation of the difference between pci_reqType_e_ADVISORY and pci_reqType_e_UNSPECIFIED, see "*Capability ID 0x5 (MSI)*" and "*Capability ID 0x11 (MSI-X)*" in the Capability Modules and APIs appendix.

If you need to change a capability's parameters, you must disable it, make the changes, and then reenable the capability.

Returns:

PCI_ERR_OK

The capability was successfully enabled or disabled.

PCI_ERR_EALREADY

The capability identified by *cap* is already enabled/disabled.

PCI_ERR_ENOTSUP

The capability can't be disabled.

PCI_ERR_EINVAL

The *hdl* doesn't refer to a device that you attached to, or some other parameter is otherwise invalid.

PCI_ERR_ENODEV

The device identified by *hdl* doesn't exist. Note that this error can also be returned if a device that supports live removal is removed after a successful call to *pci_device_find()*.

PCI_ERR_NOT_OWNER

You don't own the device associated with *hdl*.

These functions can also return any of the errors documented for the specific capability being enabled or disabled.

The following internal errors are specifically related to internal resource availability and use, and are atypical:

PCI_ERR_ENOMEM

Memory for internal resources couldn't be obtained. This may be a temporary condition.

PCI_ERR_LOCK_FAILURE

An error occurred that's related to the creation, acquisition, or use of a synchronization object.

If any error occurs, the capability for the device isn't changed.

Classification:

QNX Neutrino

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

See also:

pci_device_cfg_cap_isenabled()

pci_device_cfg_cap_isenabled()

Check to see if a capability is enabled or disabled

Synopsis:

```
#include <pci/pci.h>

bool_t pci_device_cfg_cap_isenabled( pci_devhdl_t hdl,
                                     pci_cap_t cap );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

cap

A pointer to a capability structure.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

This function determines whether the capability identified by *cap* for the device identified by *hdl* is enabled or disabled. You must have already attached to the device by successfully calling *pci_device_attach()*. You can obtain the device capability by calling *pci_device_read_cap()*.

Returns:

true or false, depending on the capability state.

Classification:

QNX Neutrino

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

See also:

pci_device_cfg_cap_disable(), *pci_device_cfg_cap_enable()*

pci_device_cfg_rd*()

Read from the device-specific PCI configuration space registers

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_cfg_rd8( pci_bdf_t bdf,
                              uint_t offset,
                              uint8_t *val);

pci_err_t pci_device_cfg_rd16( pci_bdf_t bdf,
                               uint_t offset,
                               uint16_t *val);

pci_err_t pci_device_cfg_rd32( pci_bdf_t bdf,
                               uint_t offset,
                               uint32_t *val);

pci_err_t pci_device_cfg_rd64( pci_bdf_t bdf,
                               uint_t offset,
                               uint64_t *val);
```

Arguments:

bdf

The Bus/Device/Function associated with a device.

offset

The offset, in bytes, at which you want to start reading, aligned to the size of the value being read.

val

A pointer to a location where the function can store the value read.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_cfg_rd*()* functions provide read access to the device-specific PCI configuration space registers starting at offset 0x40 (64). The register contents are returned in the memory that *val* points to. All functions return a `pci_err_t` indicating success or failure.

The *pci_device_cfg_rd*()* functions take a `pci_bdf_t` parameter as their first argument, so you don't need to attach to the device before reading the specified configuration space register. You can find the `pci_bdf_t` value for a specific device by calling *pci_device_find()*.

You should use these functions only if you can't obtain the corresponding information with an existing library or capability module API. Also check the capability module APIs defined in the **<pci/cap_*.h>** files.

Returns:

PCI_ERR_OK

Success.

PCI_ERR_EINVAL

The *offset* argument isn't within the device-specific configuration space range (between 0x40 and 0xFF for PCI devices, or between 0x40 and 0xFFF for PCIe devices), or the *offset* isn't aligned to the size of the requested read operation.

If any error occurs, you should assume that storage you provided contains invalid data.

Classification:

QNX Neutrino

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

pci_device_cfg_wr*()

Write to the device-specific PCI configuration space registers

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_cfg_wr8( pci_devhdl_t hdl,
                              uint_t offset,
                              uint8_t val,
                              uint8_t *val_p );

pci_err_t pci_device_cfg_wr16( pci_devhdl_t hdl,
                               uint_t offset,
                               uint16_t val,
                               uint16_t *val_p );

pci_err_t pci_device_cfg_wr32( pci_devhdl_t hdl,
                               uint_t offset,
                               uint32_t val,
                               uint32_t *val_p );

pci_err_t pci_device_cfg_wr64( pci_devhdl_t hdl,
                               uint_t offset,
                               uint64_t val,
                               uint64_t *val_p );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

offset

The offset, in bytes, at which you want to start writing, aligned to the size of the value being written.

val

The value that you want to write.

val_p

NULL, or a pointer to a location where the function can store the register's value after the writing is done.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_cfg_wr*()* functions provide write access to the device-specific PCI configuration space registers starting at offset 0x40 (64). The value to write is contained in *val*. If *val_p* isn't NULL, the location that it points to is set to the contents of the register after the write operation is completed (the register is read after the write). All functions return a `pci_err_t` indicating success or failure.

The *pci_device_cfg_wr*()* functions take a `pci_devhdl_t` parameter as their first argument, and therefore you must successfully attach to the device before you can write the specified configuration space register.

Returns:

PCI_ERR_OK

Success.

PCI_ERR_EINVAL

The following might have occurred:

- The *offset* argument isn't within the device-specific configuration space range (between 0x40 and 0xFF for PCI devices, or between 0x40 and 0xFFF for PCIe devices), or the *offset* isn't aligned to the size of the requested write operation.
- You didn't successfully attach to the device.

PCI_ERR_ENODEV

The *hdl* argument doesn't refer to a valid device.

If any error occurs, you should assume that the storage pointed to by a non-NULL *val_p* contains invalid data.

Classification:

QNX Neutrino

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

pci_device_chassis_slot()

Get the chassis and slot for a device

Synopsis:

```
#include <pci/pci.h>

pci_cs_t pci_device_chassis_slot( pci_bdf_t bdf );
```

Arguments:

bdf

The Bus/Device/Function for a device. You can construct this value with the *PCI_BDF()* macro, but you more typically obtain it by calling *pci_device_find()*.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_chassis_slot()* function returns the chassis and slot that device *bdf* resides in.

Returns:

A `pci_cs_t` that contains the information about the chassis and slot. For more details about this data type, see *pci_chassis_slot_device()*.

If you pass this value to the *PCI_SLOT()* macro, and the result is 0, then the device isn't in a slot.

Classification:

QNX Neutrino

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

See also:

pci_chassis_slot_bridge(), *pci_chassis_slot_device()*

pci_device_detach()

Detach a device

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_detach( pci_devhdl_t hdl );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

When the device is no longer in use, or the attaching software will no longer manage the device, you should call *pci_device_detach()* to return all resources to the system. If any capabilities have been enabled, you should first disable them and free the `pci_cap_t` that *pci_device_read_cap()* returned.

Returns:

PCI_ERR_OK

Success.

PCI_ERR_EINVAL

The *hdl* argument is invalid or doesn't refer to a device that you attached to.

PCI_ERR_ENODEV

The *hdl* argument doesn't refer to a valid device.

PCI_ERR_ENOENT

An internal error occurred; there's no recorded attachment for you.

Classification:

QNX Neutrino

Safety:

Cancellation point	No

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

pci_device_attach()

pci_device_find()

Locate a device that matches the specified criteria

Synopsis:

```
#include <pci/pci.h>

pci_bdf_t pci_device_find( const uint_t idx,
                          const pci_vid_t vid,
                          const pci_did_t did,
                          const pci_ccode_t classcode );
```

Arguments:

idx

The zero-based index of a specific device that you want to find; see below.

vid

The vendor ID you want to match, or PCI_VID_ANY.

did

The device ID you want to match, or PCI_DID_ANY.

classcode

The class or subclass that you want to match, or PCI_CCODE_ANY.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The pci_device_find() function kets you locate a specific device by specifying any combination of the vendor ID *vid*, device ID *did*, and class (and subclass) code *classcode* filter parameters. Any parameter can be wild-carded by using PCI_VID_ANY, PCI_DID_ANY, or PCI_CCODE_ANY, respectively.

The class code is of type `pci_ccode_t`, which contains 3 bytes as defined in the PCI specification:

```
MSb                                         LSb
        0000 0000 CCCC CCCC cccc cccc rrrr rrrr

C - represents the base class
c - represents the sub class
r - represents a register level programming interface
```

To encode and decode the class code, you can use the macros defined in **<pci/pci_ccode.h>**, as well as the following macros, defined in **<pci/pci.h>**:

- *PCI_CCODE_CLASS(_cc_)*
- *PCI_CCODE_SUBCLASS(_cc_)*
- *PCI_CCODE_REG_IF(_cc_)*
- *PCI_CCODE(_C_, _c_, _r_)*

Wild cards for the subclass and register interface let you widen the search for class codes. You can use the following values independently or together for the subclass and register interface respectively when constructing a classcode to search for:

- PCI_CCODE_SUBCLASS_ANY
- PCI_CCODE_REG_IF_ANY

For example:

- `PCI_CCODE(`*base_class*`, PCI_CCODE_SUBCLASS_ANY, PCI_CCODE_REG_IF_ANY)`
- `PCI_CCODE(`*base_class*`, `*subclass*`, PCI_CCODE_REG_IF_ANY)`

You can find the defined vendor IDs in **<pci/pci_id.h>**. In addition, the *idx* parameter lets you select from multiple instances of devices meeting the search criteria. By calling *pci_device_find()* repeatedly with an incrementing *idx* value, you can find all instances of a device meeting the search criteria.

The *idx* parameter has no association with a given device. It's used only as an instance token across multiple find calls when the search criteria don't change. The same device (as identified by the returned `pci_bdf_t`) could be identified for a different *idx* value if you use different search criteria. If you change the search criteria, reset the starting *idx* value to 0, or the function might return PCI_ERR_ENODEV.

The search criteria parameters allow the following filter combinations:

vid	*did*	*classcode*	Result
PCI_VID_ANY	PCI_DID_ANY	PCI_CCODE_ANY	All devices
PCI_VID_ANY	PCI_DID_ANY	A class code	All devices of a specified class
PCI_VID_ANY	A device ID	PCI_CCODE_ANY	All devices with a specific device ID
PCI_VID_ANY	A device ID	A class code	All devices of a specified class with a specific device ID
A vendor ID	PCI_DID_ANY	PCI_CCODE_ANY	All devices from a specific vendor
A vendor ID	PCI_DID_ANY	A class code	All devices of a specified class from a specific vendor
A vendor ID	A device ID	PCI_CCODE_ANY	All devices with a specific device ID from a specific vendor

vid	*did*	*classcode*	Result
A vendor ID	A device ID	A class code	All devices of a specified class with a specific device ID from a specific vendor

Returns:

A `pci_bdf_t` value that uniquely identifies a specific device and which you can use to attach to the device or to perform various read operations (see *pci_device_read_*()*), or PCI_BDF_NONE if a device couldn't be found based on the search criteria.

Classification:

QNX Neutrino

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

pci_device_find_capid()

Get the index of a capability for a device

Synopsis:

```
#include <pci/pci.h>

int_t pci_device_find_capid( pci_bdf_t bdf,
                             pci_capid_t capid );
```

Arguments:

bdf

The Bus/Device/Function associated with a device.

capid

The identifier for a capability.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

You can use *pci_device_read_capid()* and *pci_device_find_capid()* to identify which capabilities the device identified by *bdf* supports. The *pci_device_find_capid()* function returns the index of the capability *capid* for the device identified by *bdf*.

This function doesn't cause a capability module to be loaded.

The purpose of this function is to obtain the capability index for a capability that the software wishes to enable for the device. You'll use this index when you call *pci_device_read_cap()*.

Returns:

The index for the specific capability, or -1 if the capability doesn't exist.

Classification:

QNX Neutrino

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

See also:

pci_device_read_cap(), *pci_device_read_capid()*

pci_device_is_multi_func()

Determine whether or not a device is a multifunction device

Synopsis:

```
#include <pci/pci.h>

bool_t pci_device_is_multi_func( pci_bdf_t bdf );
```

Arguments:

bdf

The Bus/Device/Function associated with a device.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_is_multi_func()* function determines whether or not the device identified by *bdf* is a multifunction device. The *bdf* should be for function 0; if it isn't, *pci_device_is_multi_func()* checks for function 0.

Returns:

true

The device is a multifunction device.

false

The device isn't a multifunction device, or *bdf* is invalid.

Classification:

QNX Neutrino

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

pci_device_map_as()

Get the translation between CPU memory addresses and PCI addresses

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_map_as( pci_devhdl_t hdl,
                             const pci_ba_t * const as,
                             pci_ba_t *as_xlate );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

as

A pointer to a `pci_ba_t` structure that describes the address space.

as_xlate

A pointer to a `pci_ba_t` structure where the function can store the translation.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_map_as()* function is used by driver software to obtain the required translations between CPU memory addresses and PCI addresses. It's used (for example) when you've allocated a set of memory buffers that a PCI device should transfer in to. In this case, you need to program the PCI device with the inbound physical addresses of the memory buffers, but depending on how the hardware is configured, there may be translation hardware that maps the PCI address that must be targeted in order to resolve to the memory buffer addresses.

You should always call this function in this or any other similar scenario. If there's no translation required, *as_xlate* will effectively be the same as *as*.

The `pci_ba_t` structure is defined as follows:

```
typedef struct
{
        pci_ba_val_t addr;
        uint64_t size;
        pci_asType_e type;
        pci_asAttr_e attr;
```

```
        int_t bar_num;
} pci_ba_t;
```

The members include:

addr

The physical address of a memory buffer.

size

The size of the buffer, in bytes.

type

One of the following:

- pci_asType_e_NONE
- pci_asType_e_MEM
- pci_asType_e_IO

attr

A bitwise OR of attribute flags. You must specify one of the following:

- pci_asAttr_e_INBOUND — transfers from PCI space to CPU space.
- pci_asAttr_e_OUTBOUND — transfers from CPU space to PCI space. You generally don't need this type of translation.

You can also OR in any of the following:

- pci_asAttr_e_16BIT, pci_asAttr_e_32BIT, pci_asAttr_e_64BIT — a 2-bit field that indicates whether to encode the address space size as 16, 32, or 64 bit.
- pci_asAttr_e_PREFETCH
- pci_asAttr_e_CONTIG
- pci_asAttr_e_EXPANSION_ROM
- pci_asAttr_e_ENABLED — applicable to expansion ROM.
- pci_asAttr_e_SHARED — otherwise reserved (applicable to slot aspace reservations).

bar_num

The Base Address Register number.

The *as* attributes (defined in **<pci/pci.h>** as type `pci_asAttr_e`) should fully specify in the *as->attr* field whether the driver is requesting translation for an outbound transfer (from CPU memory to PCI device) or inbound transfer (from PCI device to CPU memory). You should also specify other attributes, such as pci_asAttr_e_CONTIG, the alignment requirements, and so on. In addition to the *as->attr* field, you should also properly initialize the *type*, *size*, and *addr* fields. Note that all addresses are specified as physical addresses.

Returns:

PCI_ERR_OK

Success.

PCI_ERR_ENODEV

The device identified by *hdl* doesn't exist. Note that this error can also be returned if a device that supports live removal is removed after a successful call to *pci_device_find()*.

PCI_ERR_EINVAL

The *hdl* doesn't refer to a valid device that you attached to, or other parameters are otherwise invalid. This error can also be returned from the hardware-dependent module.

PCI_ERR_ASPACE_INVALID

The address provided couldn't be translated into usable values.

If any error occurs, you should assume that storage you provided for *as_xlate* contains invalid data.

Classification:

QNX Neutrino

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

pci_device_read_*()

Read information from the common PCI configuration space registers

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_read_ccode( pci_bdf_t bdf, pci_ccode_t *ccode );

pci_err_t pci_device_read_clsize( pci_bdf_t bdf, pci_clsize_t *cache_linesz );

pci_err_t pci_device_read_cmd( pci_bdf_t bdf, pci_cmd_t *cmd);

pci_err_t pci_device_read_did( pci_bdf_t bdf, pci_did_t *did);

pci_err_t pci_device_read_hdrType( pci_bdf_t bdf, pci_hdrType_t *hdrType);

pci_err_t pci_device_read_latency( pci_bdf_t bdf, pci_latency_t *latency);

pci_err_t pci_device_read_revid( pci_bdf_t bdf, pci_revid_t *revid);

pci_err_t pci_device_read_ssid( pci_bdf_t bdf, pci_ssid_t *ssid);

pci_err_t pci_device_read_ssvid( pci_bdf_t bdf, pci_ssvid_t *ssvid);

pci_err_t pci_device_read_status( pci_bdf_t bdf, pci_stat_t *status);

pci_err_t pci_device_read_vid( pci_bdf_t bdf, pci_vid_t *vid);
```

Arguments:

bdf

The Bus/Device/Function associated with a device.

Additional argument

A pointer to a location where the function can store the information read.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_read_*()* functions provide read access to the common PCI configuration space registers identified by their names. The register contents are returned in the provided storage.

Register	Description	Data type
CCODE	Class and subclass code; see *pci_device_find()*	`pci_ccode_t`
CLSIZE	Cache line size	`pci_clsize_t`
CMD	Command: a bitmask of features that can be individually enabled and disabled	`pci_cmd_t`
DID	Device ID	`pci_did_t`
HDRTYPE	Header type, which indicates the layout of the remaining header data	`pci_hdrType_t`
LATENCY	Latency timer	`pci_latency_t`
REVID	Revision ID	`pci_revid_t`
SSID	Subsystem ID	`pci_ssid_t`
SSVID	Subsystem vendor ID	`pci_ssvid_t`
STATUS	Status information for PCI bus-related events	`pci_stat_t`
VID	Vendor ID	`pci_vid_t`

All functions return a `pci_err_t` indicating success or failure. If the call is successful, the storage that you provided contains the requested value. If any error occurs, you should assume that this storage contains invalid data.

You don't need to attach to the device before using a pci_device_read_*() function that takes a `pci_bdf_t` parameter as its first argument to read a configuration space register. You can find the `pci_bdf_t` value for a specific device by calling *pci_device_find()*.

Returns:

PCI_ERR_OK on success; any other value indicates that an error occurred (you can use *pci_strerror()* to get the associated description).

Classification:

QNX Neutrino

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

See also:

pci_device_write_cmd(), pci_device_write_status()

pci_device_read_ba()

Get the base addresses of a device

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_read_ba( pci_devhdl_t hdl,
                              int_t *nba,
                              pci_ba_t *ba,
                              pci_reqType_t reqType );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

nba

A pointer to a location that can hold the number of entries in the *ba* array (see below).

ba

NULL, or an array of `pci_ba_t` structures where the function can store information about the device's supported address spaces (see below). For more information about this structure, see *pci_device_map_as()*.

reqType

The desired level of compliance to the request; one of:

- pci_reqType_e_MANDATORY — the request must be satisfied with the requested constraints, or the request fails.
- pci_reqType_e_UNSPECIFIED — the request constraints aren't specified. The PCI server is free to choose any request constraints.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_read_ba()* function gets the base addresses for a device. You must have already attached to the device by successfully calling *pci_device_attach()*.

PCI devices contain multiple BARs (Base Address Registers) that may be applicable to different address space types. To simplify the retrieval of the base address information, the *pci_device_read_ba()* function

is provided. This function gets an array of `pci_ba_t` types that uniquely identify all of the supported address spaces of the device identified by *hdl*.

In general, the *ba* parameter should point to an array of `pci_ba_t` types, in which case you specify the number of entries in the array in the value pointed to by *nba*.

By modifying the parameters, you can control the returned values as follows:

- To determine how many valid base addresses the device uses, set *ba* to NULL and set the variable pointed to by *nba* to the maximum number of BARs. The number of implemented base addresses is returned in **nba*.

  

  If you set the variable pointed to by *nba* to zero, *pci_device_read_ba()* returns PCI_ERR_OK but doesn't set **nba* to the number of valid base addresses.

- If you're interested in only one base address, *ba* can be a pointer to a single `pci_ba_t` variable. In this case, *nba* can point to the value of 1, or *nba* can be NULL. Note that in this case only the first implemented base address will be returned.

Otherwise you should allocate space for an array of `pci_ba_t` items and set *nba* to point to the number of items allocated. On successful return, *nba* contains a value as follows:

- If the size of array specified by the value pointed to by *nba* was large enough to hold all implemented base addresses for the device, *nba* contains a value less than or equal to the originally assigned value of *nba* indicating the actual number of implemented base addresses. Note that this value can be zero.
- If the size of the array specified by the value pointed to by *nba* wasn't large enough to hold all implemented base addresses for the device, *nba* contains a value that's the 2's complement of the number of base addresses that wouldn't fit into the allocated array.

  For example, if *nba* contains the value of -2 upon successful return, all but two of the requested number of base addresses were successfully returned.

In all of the above cases, you should set the *reqType* argument to pci_reqType_e_UNSPECIFIED. Optionally, you can retrieve the address space information for a specific BAR or set of BARs. If you set *reqType* to pci_reqType_e_MANDATORY and initialize the *ba.bar_num* field for each of the **nba* entries, the function returns only the address space information for the specified BAR or BARs. If you specify the same BAR number more than once, multiple identical entries are returned. If you specify an invalid BAR number, the entry is of type pci_asType_e_NONE.

Valid BAR numbers are from 0 through 5, inclusive. Use a BAR number of -1 to retrieve address space information for expansion ROMs. Note also, that if you specify pci_reqType_e_MANDATORY, **nba* isn't updated to reflect the actual number of valid address spaces; it remains as you set it.

Because the base addresses for a device provide the location of memory mapped device registers, it's desirable to control access to this information. This function succeeds only if *hdl* identifies an attacher that owns the device. When you call *pci_device_attach()*, you can specify whether you want to be the owner of the device. Only the owner can successfully call *pci_device_read_ba()* and hence be able to map and control the device.

Returns:

PCI_ERR_OK

Success.

PCI_ERR_ENODEV

The device identified by *hdl* doesn't exist. Note that this error can also be returned if a device that supports live removal is removed after a successful call to *pci_device_find()*.

PCI_ERR_ENOTSUP

The device identified by *hdl* isn't supported. Note that this is different from a return of PCI_ERR_OK with *nba* pointing to the value 0.

PCI_ERR_EINVAL

The *hdl* doesn't refer to a valid device that you attached to, or other parameters are otherwise invalid. This error can also be returned from the hardware-dependent module.

PCI_ERR_NOT_OWNER

You aren't the owner of the device; see *pci_device_attach()*.

PCI_ERR_ASPACE_INVALID

The base addresses assigned to the device couldn't be translated into usable values for you. This is generally a hardware-dependent module error.

PCI_ERR_EIO

A device-specific error occurred.

If any error occurs, you should assume that the storage that you provided contains invalid data.

Classification:

QNX Neutrino

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

pci_device_read_cap()

Load a capability module

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_read_cap( pci_bdf_t bdf,
                               pci_cap_t *cap,
                               uint_t idx );
```

Arguments:

bdf

The Bus/Device/Function associated with a device.

cap

A pointer to a location where the function can store a pointer to a PCI capabilities structure (note that `pci_cap_t` itself is defined as a pointer to the structure).

idx

The index of the capability that you want to load.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

Once you've found a desired capability index (by using *pci_device_read_capid()* or *pci_device_find_capid()*), you can call *pci_device_read_cap()* using the *idx* parameter in order to initialize the *cap* argument and trigger the loading of the capabilities module.

The first time you call *pci_device_read_cap()* for a given capability *idx*, initialize the pointer referred to by *cap* to NULL (see the example below). Failure to do this will likely result in a program exception.

Although there's no need to call *pci_device_read_cap()* more than once for each capability, you can call it again using the `pci_cap_t` returned on the first successful call.

If *pci_device_read_cap()* fails on a subsequent call using the *cap* returned from an initial call (i.e., with *cap* = NULL), the `pci_cap_t` initially received is destroyed, and you must make a new call with *cap* = NULL.

After a successful call to *pci_device_read_cap()*, you can use the APIs for the capability module using the returned `pci_cap_t` argument in order to configure the capability as desired.

Finally, once the capability has been configured (as applicable), you can enable the capability by calling *pci_device_cfg_cap_enable()*. If a device supports a capability for which there's no capability module, *pci_device_read_cap()* returns an error indicating that the capability can't be used.

When you no longer need the `pci_cap_t` returned from a successful *pci_device_read_cap()* call, you can release it with *free()*.

Returns:

PCI_ERR_OK

Success; *cap* refers to a valid `pci_cap_t`, which you can use with the capability APIs.

PCI_ERR_ENOENT

The capability at *idx* doesn't exist.

PCI_ERR_EIO

A malformed capability “next” pointer was encountered.

PCI_ERR_NO_MODULE

A capability module doesn't exist (in the capability module search path) for the capability at *idx* for the device identified by *bdf*.

PCI_ERR_MODULE_BLACKLISTED

The identified capability module is contained in the *PCI_MODULE_BLACKLIST* environment variable.

PCI_ERR_MODULE_SYM

The identified capability module doesn't have a proper initialization function.

PCI_ERR_MOD_COMPAT

The identified capability module is incompatible with the version of either the `pci-server` binary or **libpci**.

The following internal errors are specifically related to internal resource availability and use, and are atypical:

PCI_ERR_ENOMEM

Memory for internal resources couldn't be obtained. This may be a temporary condition.

PCI_ERR_LOCK_FAILURE

An error occurred that's related to the creation, acquisition or use of a synchronization object.

If any error occurs, you should consider the contents of the *cap* argument to be invalid data.

Example:

```
#include <pci/pci.h>
#include <pci/cap_pcie.h>

int_t cap_pcie_idx = pci_device_find_capid(bdf, CAPID_PCIe);

if (cap_pcie_idx >= 0)
{
        pci_cap_t pcie_cap = NULL;
        pci_err_t r = pci_device_read_cap(bdf, &pcie_cap, cap_pcie_idx);
        if (r == PCI_ERR_OK)
        {
                /* enable and use the capability */

                /* done with the capability */
                free(pcie_cap);
        }
}
```

Classification:

QNX Neutrino

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

See also:

pci_device_find_capid(), *pci_device_read_capid()*

pci_device_read_capid()

Get the ID of the nth capability that a device supports

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_read_capid( pci_bdf_t bdf,
                                 pci_capid_t *capid,
                                 uint_t idx );
```

Arguments:

bdf

The Bus/Device/Function associated with a device.

capid

A pointer to a location where the function can store the capability ID.

idx

The index of the capability.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

You can use *pci_device_read_capid()* and *pci_device_find_capid()* to identify which capabilities the device identified by *bdf* supports. The *pci_device_read_capid()* function obtains the capability ID for the *n*th capability (specified by *idx* and starting at 0) that the device identified by *bdf* supports.

This function doesn't cause a capability module to be loaded.

The purpose of this function is to obtain the capability index for a capability that the software wishes to enable for the device. You'll use this index when you call *pci_device_read_cap()*.

You can use *pci_device_read_capid()* to easily loop with an incrementing *idx* value and obtain the `pci_capid_t` at that index. If **capid* is for a desired capability, you can then read the capability with *pci_device_read_cap()* and the current *idx* value.

Returns:

PCI_ERR_OK

Success; **capid* holds the ID of the capability at the specified *idx*.

PCI_ERR_ENOENT

The capability at *idx* doesn't exist. This usually means the end of the devices capability list, but if *idx* is 0, it means that the device has no capabilities.

PCI_ERR_EIO

A malformed capability "next" pointer was encountered.

If any error occurs, you should consider the contents of **capid* to be invalid data.

Classification:

QNX Neutrino

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

See also:

pci_device_find_capid(), *pci_device_read_cap()*

pci_device_read_irq()

Get the IRQs associated with a device

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_read_irq( pci_devhdl_t hdl,
                               int_t *nirq,
                               pci_irq_t *irq );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

nirq

NULL, or a pointer to a location for the number of IRQs.

irq

NULL, or an array of `pci_irq_t` entries.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_read_irq()* function gets the IRQs associated with a device. You must have already attached to the device by successfully calling *pci_device_attach()*.

There could be multiple IRQs associated with a single device. In general, the *irq* parameter should point to an array of `pci_irq_t` types, and you should specify the number of entries the array in the value pointed to by *nirq*.

By modifying the parameters, you can control returned values as follows:

- To determine how many interrupts are associated with the device identified by *hdl*, pass NULL for *irq*. The function sets the variable pointed to by *nirq* to the number of supported/configured interrupts.
- If you're interested in only one IRQ, *irq* can be a pointer to a single `pci_irq_t` variable. In this case *nirq* can point to the value of 1, or *nirq* can be NULL.

If the device uses more than one IRQ and you request only one, an arbitrary interrupt is returned.

Otherwise, you should allocate space for an array of `pci_irq_t` items and set *nirq* to point to the number of items allocated. Upon successful return, *nirq* contains a value as follows:

- If the size of array specified by the value pointed to by *nirq* was large enough to hold all IRQs associated with the device, *nirq* contains a value less than or equal to the originally assigned value of *nirq* indicating the actual number of associated interrupts. Note that it is possible for this value to be zero.
- If the size of the array specified by the value pointed to by *nirq* wasn't large enough to hold all IRQs associated with the device, *nirq* contains a value that's the 2's complement of the number of IRQs that wouldn't fit into the allocated array.

  For example, if *nirq* contains the value of -2 upon successful return, all but two of the requested number of IRQs were successfully returned.

If the device uses more than one IRQ, it is unspecified as to which device functionality is associated with which IRQ. IRQs are returned in the *irq* array in enabled interrupt source order. See also "*Capability ID 0x5 (MSI)*" and "*Capability ID 0x11 (MSI-X)*" in the Capability Modules and APIs appendix.

Because the IRQs for a device provide the ability to attach interrupt handlers, it's desirable to control access to this information. This function succeeds only if *hdl* identifies an attacher that owns the device. When you call *pci_device_attach()*, you can specify whether you want to be the owner of the device. Only the owner can successfully call *pci_device_read_irq()* and hence be able to attach interrupt service routines.

Returns:

PCI_ERR_OK

Success.

PCI_ERR_ENODEV

The device identified by *hdl* doesn't exist. Note that this error can also be returned if a device that supports live removal is removed after a successful call to *pci_device_find()*.

PCI_ERR_ENOTSUP

The device identified by *hdl* doesn't support interrupts or the type of interrupts being requested. Note that this is different from a return of PCI_ERR_OK with *nirq* pointing to the value 0.

PCI_ERR_EINVAL

The *hdl* doesn't refer to a valid device that you attached to, or other parameters are otherwise invalid.

PCI_ERR_NOT_OWNER

You aren't the owner of the device; see *pci_device_attach()*.

PCI_ERR_IRQ_CFG

An invalid configuration for interrupt sources has been established in the device.

PCI_ERR_IRQ_NOT_AVAIL

There are no interrupt vectors available.

PCI_ERR_EIO

A device-specific error occurred.

If any error occurs, you should assume that the storage that you provided contains invalid data.

Classification:

QNX Neutrino

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

pci_device_reset()

Reset a device

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_reset( pci_devhdl_t hdl,
                           pci_resetType_e resetType );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

resetType

The type of reset to initiate. The defined types are:

- pci_resetType_e_BUS
- pci_resetType_e_FUNCTION

A reset type that's less than pci_resetType_e_BUS is considered to mean “no reset” (a no-op).

A reset type that's greater than pci_resetType_e_FUNCTION is considered to be a hardware-specific reset. The behavior of hardware-specific resets, if supported, is defined by the hardware-dependent module. Refer to the release notes or the usage information in the hardware-dependent module applicable to your platform for details.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_reset()* function initiates a reset of a specific device (function-level reset if the device is PCIe and supports FLR as determined by the device capabilities register or if the device is PCI and supports the Advanced Features capability and FLR as per the AF capabilities register) or of a PCI bus segment/PCIe link (secondary bus reset) if the device is a PCI-to-PCI bridge or PCIe Root Port that reports as a PCI-to-PCI bridge.

The *hdl* arguments identifies the specific device to be reset, or the bus or link on which the device resides.

Since this API can potentially impact multiple devices, you must be sure of what you're doing. During the time that the reset is occurring, configuration space accesses to the device(s) is halted, but there's currently no in-band mechanism for preventing access to device-specific registers obtained from a successful call to *pci_device_read_ba()* and *mmap()*'d by other driver software. It's your responsibility to coordinate with such software as required.

The primary types of reset are:

pci_resetType_e_BUS

Reset all devices on the bus or link on which the device identified by *hdl* resides, including any downstream buses or links for subordinate bridges.

pci_resetType_e_FUNCTION

Reset only the specific device identified by *hdl*.

In both cases, you *must* be an owner of the device identified by *hdl* and the only remaining attacher.

Affected devices

For a function reset, only the device identified by *hdl* is reset. This includes a specific function of a multifunction device. You must be an owner of the device and the only remaining attacher (all other multiowned attachers, if any, must have detached). In addition, for bus/link resets, all devices that reside on the same bus/link as the device identified by *hdl* or that reside on a bus/link that's downstream of the bus/link on which the device identified by *hdl* resides are also reset. In this case, those devices must not have any attachments (i.e., no other software must have successfully called *pci_device_attach()* on these devices without having called *pci_device_detach()* prior to the reset operation). It's the responsibility of the application software to coordinate these operations for other affected device software, or the *pci_device_reset()* call will fail.

A reset of a specific device (i.e., a BDF) is supported by the PCI specification-defined Function Level Reset (FLR) mechanism. In order to initiate this type of reset, the device must either be a PCIe endpoint with bit 28 of the device capabilities register set to 01B or have the Advanced Features (AF) capability with bit 1 of the AF capabilities register set to 0B1, otherwise this reset is unsupported.

Other reset types are allowed, but these have no defined behavior within the PCI server, and so are passed directly to the hardware-dependent module. This allows for platform-specific reset processing to be accommodated. It's the responsibility of the hardware-dependent module to document what additional reset types if any it supports and what the behaviors of those resets are.

Although the hardware-dependent module has complete control over the reset behavior, the PCI server ensures that configuration space accesses to the device being reset are halted, and after reset, that the device is configured into the D0-initialized state unless otherwise prevented from doing so by the hardware-dependent module. All of the discussions that follow regarding post-reset state and interrupts are applicable to hardware-dependent reset types unless the hardware-dependent module documents otherwise.

Post-reset driver state

On return from a successful *pci_device_reset()*, your *hdl* is still valid, as are any mappings to address spaces obtained with a successful call to *pci_device_read_ba()*, but all capabilities are disabled. In order to use the capabilities, you need to:

- reread them, using *pci_device_read_cap()*; you can reuse the same `pci_cap_t`
- reconfigure them (as required using the appropriate capability-specific APIs)
- reenable them by calling *pci_device_cfg_cap_enable()*

Device software for other affected devices (if any), must go through the entire initialization phase starting with *pci_device_attach()*, because they were required to detach in order for the reset to take place.

Regarding interrupts

If the device was configured to use the MSI or MSI-X capability, the IRQs associated with the MSI/MSI-X vectors are released when the capability is disabled. Since MSI/MSI-X vectors are allocated when the capability is enabled, and these could potentially change between the disabling and reenabling of these capabilities, driver software should call *InterruptDetach()* for previously assigned IRQs, reread them with *pci_device_read_irq()*, and then reattach them with *InterruptAttach()* or *InterruptAttachEvent()*.

If you aren't using MSI or MSI-X, you don't need to reread the pin-based IRQs originally returned because they don't change. That is, there's no need for the driver software to call *InterruptDetach()*, *pci_device_read_irq()*, and then *InterruptAttach()* or *InterruptAttachEvent()* after the reset if using pin-based (i.e., non-MSI or non-MSI-X) interrupts.

Returns:

If successful, a call to *pci_device_reset()* with a reset type of either pci_resetType_e_BUS or pci_resetType_e_FUNCTION returns PCI_ERR_OK, and all affected devices (as described above) have been reset and reconfigured to the D0-initialized state. Any driver software using those device must comprehend this condition and perform any required device-specific initialization that may have been lost as a result of the reset operation.

For reset types other than pci_resetType_e_BUS or pci_resetType_e_FUNCTION, the hardware-dependent module must document the post-reset state of affected devices.

If any error occurs, *pci_device_reset()* returns one of the following values, and you should consider all affected devices (as outlined above) to be in an unknown and unusable state. It this situation, it may be possible, and even desirable to reissue the reset command.

PCI_ERR_ENOTSUP

The reset type specified isn't supported by the device identified by *hdl*.

PCI_ERR_EINVAL

The *hdl* doesn't refer to a valid device that you attached to, or other parameters are otherwise invalid. The hardware-dependent module can also return this error.

PCI_ERR_ENODEV

The device identified by *hdl* doesn't exist. Note that this error can also be returned if a device that supports live removal is removed.

PCI_ERR_NOT_OWNER

You don't own the device identified by *hdl*.

PCI_ERR_ATTACH_SHARED

The device identified by *hdl* is currently attached to by more than just you.

PCI_ERR_ATTACH_OWNED

Devices on the same bus or link as the device identified by *hdl* or on a downstream bus or link are still attached.

Classification:

QNX Neutrino

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

pci_device_rom_disable(), pci_device_rom_enable()

Enable or disable expansion ROM

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_rom_disable( pci_devhdl_t hdl );

pci_err_t pci_device_rom_enable( pci_devhdl_t hdl );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_rom_enable()* and *pci_device_rom_disable()* functions enable and disable a device's expansion ROM, respectively. It's your responsibility to enable the Memory Access bit in the command register.

Returns:

PCI_ERR_OK

Success.

PCI_ERR_ENODEV

The device identified by *hdl* doesn't exist. Note that this error can also be returned if a device that supports live removal is removed after a successful call to *pci_device_find()*.

PCI_ERR_ENOTSUP

One of the following occurred:

- The device identified by *hdl* isn't supported.
- There are no expansion ROM entries for the device.

PCI_ERR_EINVAL

The *hdl* doesn't refer to a valid device that you attached to, or other parameters are otherwise invalid. This error can also be returned from the hardware-dependent module.

PCI_ERR_NOT_OWNER

You aren't the owner of the device; see *pci_device_attach()*.

PCI_ERR_ASPACE_INVALID

The base addresses assigned to the device couldn't be translated into usable values for you. This is generally a hardware-dependent module error.

PCI_ERR_EIO

A device-specific error occurred.

Classification:

QNX Neutrino

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

pci_device_write_cmd(), pci_device_write_status()

Write a value in the device command or status register

Synopsis:

```
#include <pci/pci.h>

pci_err_t pci_device_write_cmd( pci_devhdl_t hdl,
                                pci_cmd_t cmd,
                                pci_cmd_t *cmd_p );

pci_err_t pci_device_write_status( pci_devhdl_t hdl,
                                   pci_stat_t status,
                                   pci_stat_t *status_p );
```

Arguments:

hdl

The handle of the device, obtained by calling *pci_device_attach()*.

cmd, ***status***

The value that you want to write in the appropriate register.

cmd_p, ***status_p***

NULL, or a pointer to a location where the function can store the old value.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_device_write_cmd()* and *pci_device_write_status()* functions provide write access to the device command and status registers at offsets 0x4 and 0x6, respectively. The value to write is contained in *cmd* or *status* argument, respectively. If the optional *cmd_p* or *status_p* argument isn't NULL, the functions update it with the contents of the respective register after the write operation is completed (the register is read after the write).

Both functions return a `pci_err_t` that indicates success or failure. These functions take a `pci_devhdl_t` parameter as their first argument, and therefore you must successfully attach to the device before you can write the respective register.

Returns:

PCI_ERR_OK

Success.

PCI_ERR_EINVAL

You didn't successfully attach to the device.

PCI_ERR_ENODEV

The *hdl* argument doesn't refer to a valid device.

If any error occurs, you should assume that the storage pointed to by a non-NULL *cmd_p* or *status_p* contains invalid data.

Classification:

QNX Neutrino

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

See also:

pci_device_read()*

pci_strerror()

Get the description of an error code

Synopsis:

```
#include <pci/pci.h>

const char * const pci_strerror( pci_err_t err );
```

Arguments:

err

The error code that you want a description of. This argument is of type `pci_err_t` and can be set to any error code returned by a PCI API function. More information on this data type and the possible error codes is given below.

Library:

libpci

Use the `-l pci` option to `qcc` to link against this library.

Description:

The *pci_strerror()* function returns a string pointer for *err*. This function never returns NULL.

The *err* argument is of the enumerated type `pci_err_t`, defined in **pci/pci.h**. The meaning of specific error codes varies with the PCI API function from which they're returned; each function's reference describes the meanings of any possible errors.

Returns:

A pointer to the description of the error.

Classification:

QNX Neutrino

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

Appendix A
Capability Modules and APIs

This appendix describes some general-purpose capability modules and their APIs.

Some general notes about capability modules:

- To obtain the `pci_cap_t` structure operated on by the specific capability module API, call *pci_device_read_cap()*.
- The module-specific APIs provide a means of querying and configuring capability-specific attributes by operating on the `pci_cap_t` structure. Changes don't actually take effect until you successfully call *pci_device_cfg_cap_enable()*.
- The module-specific APIs aren't available to driver software until you successfully call *pci_device_read_cap()*. Software doesn't link against a capability module.
- In order to change capability attributes, you must disable the capability before re-enabling it with an updated `pci_cap_t`.
- When you no longer need a `pci_cap_t` obtained from a successful *pci_device_read_cap()* call, you can release it by calling *free()*.

Capability ID 0x10 (PCI Express)

The PCIe capability module provides access to the extended configuration space from 256–4095 bytes using the following APIs, which are defined in **<pci/cap_pcie.h>**:

`int_t cap_pcie_version( pci_cap_t cap )`

Return the version of PCI Express supported by the device or -1 on any error.

`pci_err_t cap_pcie_read_portType( pci_cap_t cap, cap_pcie_portType_t *portType )`

If PCI_ERR_OK is returned, *portType* contains the PCIe port type for the device.

`bool_t cap_pcie_is_slot_implemented(pci_cap_t cap )`

Returns true or false, depending on whether the Root Port supports slots.

`int_t cap_pcie_read_slot_num(pci_cap_t cap )`

Returns the assigned slot number if the Root Port supports slots, or -1 on error.

`int_t cap_pcie_read_int_msgnum(pci_cap_t cap )`

Returns the MSI interrupt number used for PCIe status register changes.

The PCIe extended capability functions (analogous to their PCI library counterparts) include:

`pci_err_t cap_pcie_read_xtnd_capid(pci_cap_t cap, pcie_capid_t *capid, uint_t idx )`

Analogous to *pci_device_read_capid()*.

`int_t cap_pcie_find_xtnd_capid(pci_cap_t cap, pcie_capid_t capid )`

Analogous to *pci_device_find_capid()*.

`int_t cap_pcie_read_xtnd_capid_version(pcie_cap_t xtnd_cap, pcie_capid_t capid )`

Returns the extended capability version, or -1 on error.

`pci_err_t cap_pcie_read_xtnd_cap(pci_cap_t cap, pcie_cap_t *xtnd_cap, uint_t idx )`

Analogous to *pci_device_read_cap()*.

The following APIs access the PCIe device capability, control, and status registers at offsets 0x4/0x24, 0x8/0x28 and 0xA/0x2A, respectively. For all successful writes, the parameter that contains the value to be written is updated with the current register contents after the write is completed:

```
pci_err_t cap_pcie_read_dev_cap_reg( pci_cap_t cap, cap_pcie_cap_reg_t *dev_cap_reg )
pci_err_t cap_pcie_read_dev_cap_reg2( pci_cap_t cap, cap_pcie_cap_reg_t *dev_cap_reg )
pci_err_t cap_pcie_read_dev_ctrl_reg( pci_cap_t cap, cap_pcie_ctrl_reg_t *dev_ctrl_reg )
pci_err_t cap_pcie_read_dev_ctrl_reg2( pci_cap_t cap, cap_pcie_ctrl_reg_t *dev_ctrl_reg )
pci_err_t cap_pcie_read_dev_stat_reg( pci_cap_t cap, cap_pcie_stat_reg_t *dev_stat_reg )
pci_err_t cap_pcie_read_dev_stat_reg2( pci_cap_t cap, cap_pcie_stat_reg_t *dev_stat_reg )
```

```
pci_err_t cap_pcie_write_dev_ctrl_reg( pci_devhdl_t hdl, pci_cap_t cap,
                                       cap_pcie_ctrl_reg_t *dev_ctrl_reg )
pci_err_t cap_pcie_write_dev_ctrl_reg2( pci_devhdl_t hdl, pci_cap_t cap,
                                        cap_pcie_ctrl_reg_t *dev_ctrl_reg )
pci_err_t cap_pcie_write_dev_stat_reg( pci_devhdl_t hdl, pci_cap_t cap,
                                       cap_pcie_stat_reg_t *dev_stat_reg )
pci_err_t cap_pcie_write_dev_stat_reg2( pci_devhdl_t hdl, pci_cap_t cap,
                                        cap_pcie_stat_reg_t *dev_stat_reg )
```

The following APIs access the PCIe link capabilities, control, and status registers at offsets 0xC/0x2C, 0x10/0x30 and 0x12/0x32, respectively. For all successful writes, the parameter that contains the value to be written is updated with the current register contents after the write is completed:

```
pci_err_t cap_pcie_read_link_cap_reg( pci_cap_t cap, cap_pcie_cap_reg_t *link_cap_reg )
pci_err_t cap_pcie_read_link_cap_reg2( pci_cap_t cap, cap_pcie_cap_reg_t *link_cap_reg )
pci_err_t cap_pcie_read_link_ctrl_reg( pci_cap_t cap, cap_pcie_ctrl_reg_t *link_ctrl_reg )
pci_err_t cap_pcie_read_link_ctrl_reg2( pci_cap_t cap, cap_pcie_ctrl_reg_t *link_ctrl_reg )
pci_err_t cap_pcie_read_link_stat_reg( pci_cap_t cap, cap_pcie_stat_reg_t *link_stat_reg )
pci_err_t cap_pcie_read_link_stat_reg2( pci_cap_t cap, cap_pcie_stat_reg_t *link_stat_reg )
pci_err_t cap_pcie_write_link_ctrl_reg( pci_devhdl_t hdl, pci_cap_t cap,
                                        cap_pcie_ctrl_reg_t *link_ctrl_reg )
pci_err_t cap_pcie_write_link_ctrl_reg2( pci_devhdl_t hdl, pci_cap_t cap,
                                         cap_pcie_ctrl_reg_t *link_ctrl_reg )
pci_err_t cap_pcie_write_link_stat_reg( pci_devhdl_t hdl, pci_cap_t cap,
                                        cap_pcie_stat_reg_t *link_stat_reg )
pci_err_t cap_pcie_write_link_stat_reg2( pci_devhdl_t hdl, pci_cap_t cap,
                                         cap_pcie_stat_reg_t *link_stat_reg )
```

The following APIs access the PCIe slot capabilities, control, and status registers at offsets 0x14/0x34, 0x18/0x38 and 0x1A/0x3A, respectively. For all successful writes, the parameter that contains the value to be written is updated with the current register contents after the write is completed:

```
pci_err_t cap_pcie_read_slot_cap_reg( pci_cap_t cap, cap_pcie_cap_reg_t *slot_cap_reg )
pci_err_t cap_pcie_read_slot_cap_reg2( pci_cap_t cap, cap_pcie_cap_reg_t *slot_cap_reg )
pci_err_t cap_pcie_read_slot_ctrl_reg( pci_cap_t cap, cap_pcie_ctrl_reg_t *slot_ctrl_reg )
pci_err_t cap_pcie_read_slot_ctrl_reg2( pci_cap_t cap, cap_pcie_ctrl_reg_t *slot_ctrl_reg )
pci_err_t cap_pcie_read_slot_stat_reg( pci_cap_t cap, cap_pcie_stat_reg_t *slot_stat_reg )
pci_err_t cap_pcie_read_slot_stat_reg2( pci_cap_t cap, cap_pcie_stat_reg_t *slot_stat_reg )
pci_err_t cap_pcie_write_slot_ctrl_reg( pci_devhdl_t hdl, pci_cap_t cap,
                                        cap_pcie_ctrl_reg_t *slot_ctrl_reg )
pci_err_t cap_pcie_write_slot_ctrl_reg2( pci_devhdl_t hdl, pci_cap_t cap,
                                         cap_pcie_ctrl_reg_t *slot_ctrl_reg )
pci_err_t cap_pcie_write_slot_stat_reg( pci_devhdl_t hdl, pci_cap_t cap,
                                        cap_pcie_stat_reg_t *slot_stat_reg )
pci_err_t cap_pcie_write_slot_stat_reg2( pci_devhdl_t hdl, pci_cap_t cap,
                                         cap_pcie_stat_reg_t *slot_stat_reg )
```

The following APIs access the PCIe root capabilities, control, and status registers at offsets 0x1C, 0x1E and 0x20 respectively. For all successful writes, the parameter that contains the value to be written is updated with the current register contents after the write is completed:

```
pci_err_t cap_pcie_read_root_cap_reg( pci_cap_t cap, cap_pcie_root_cap_reg_t *root_cap_reg )
pci_err_t cap_pcie_read_root_ctrl_reg( pci_cap_t cap, cap_pcie_root_ctrl_reg_t *root_ctrl_reg )
pci_err_t cap_pcie_read_root_stat_reg( pci_cap_t cap, cap_pcie_root_stat_reg_t *root_stat_reg )
pci_err_t cap_pcie_write_root_ctrl_reg( pci_devhdl_t hdl, pci_cap_t cap,
```

```
                                                cap_pcie_root_ctrl_reg_t *root_ctrl_reg )
pci_err_t cap_pcie_write_root_stat_reg( pci_devhdl_t hdl, pci_cap_t cap,
                                                cap_pcie_root_stat_reg_t *root_stat_reg )
```

Capability ID 0x5 (MSI)

The MSI capability module enables the use of Message Signalled Interrupts. Use the APIs listed below to configure the MSI capability before enabling them. Specifically, you should check the number of device-supported interrupts and then select the number that you intend to support in the driver. The number of driver supported interrupts must be less than or equal to the number of device-supported interrupts.

You must enable the MSI capability (see *pci_device_cfg_cap_enable()*) before you call *pci_device_read_irq()*. When the capability is enabled, bit 2 (Bus Master) of the Command Register is also set. This bit *isn't automatically cleared* when the capability is disabled.

The MSI and MSI-X capabilities are mutually exclusive. If you want to switch to using MSI on a device that has MSI-X enabled, you must first disable the MSI-X capability (see *pci_device_cfg_cap_disable()*). Failure to do this will result in an error of PCI_ERR_MSIX_ENABLED.

The MSI capability supports the following APIs, as defined in **<pci/cap_msi.h>**.

The following APIs allow software to obtain the number of device-supported interrupt sources and to specify the number to actually use :

`uint_t cap_msi_get_nirq(pci_cap_t cap)`

`uint_t cap_msi_get_nirq_isr(pci_cap_t cap)`

Returns the number of interrupts supported by the device, or 0 on any error. You can call the `*_isr`-suffixed version from an ISR.

`pci_err_t cap_msi_set_nirq(pci_devhdl_t hdl, pci_cap_t cap, uint_t nirq)`

Lets you set the number of interrupts to use. The value must be a power of 2 (up to a maximum of 32) and must be less than or equal to the number of device-supported interrupts.

The following APIs retrieve the interrupt pending/mask values, respectively, if Per Vector Masking (PVM) is supported. The `*_isr` versions are safe to call from an ISR. If PVM isn't supported, these functions return PCI_ERR_ENOTSUP:

- `pci_err_t cap_msi_get_irq_pend(pci_devhdl_t hdl, pci_cap_t cap, cap_msi_pend_t *pend)`
- `pci_err_t cap_msi_get_irq_pend_isr(pci_devhdl_t hdl, pci_cap_t cap, cap_msi_pend_t *pend)`
- `pci_err_t cap_msi_get_irq_mask(pci_devhdl_t hdl, pci_cap_t cap, cap_msi_mask_t *mask)`
- `pci_err_t cap_msi_get_irq_mask_isr(pci_devhdl_t hdl, pci_cap_t cap, cap_msi_mask_t *mask)`

If Per Vector Masking (PVM) is supported, the following APIs let you set or clear the interrupt mask, using a mask value or interrupt entry. The `*_isr` versions are safe to call from an ISR. If PVM isn't supported, these functions return PCI_ERR_ENOTSUP:

- `pci_err_t cap_msi_mask_irq(pci_devhdl_t hdl, pci_cap_t cap, cap_msi_mask_t mask_val)`

- `pci_err_t cap_msi_mask_irq_isr(pci_devhdl_t hdl, pci_cap_t cap, cap_msi_mask_t mask_val)`
- `pci_err_t cap_msi_unmask_irq(pci_devhdl_t hdl, pci_cap_t cap, cap_msi_mask_t unmask_val)`
- `pci_err_t cap_msi_unmask_irq_isr(pci_devhdl_t hdl, pci_cap_t cap, cap_msi_mask_t unmask_val)`
- `pci_err_t cap_msi_mask_irq_entry(pci_devhdl_t hdl, pci_cap_t cap, uint_t irq_entry)`
- `pci_err_t cap_msi_mask_irq_entry_isr(pci_devhdl_t hdl, pci_cap_t cap, uint_t irq_entry)`
- `pci_err_t cap_msi_unmask_irq_entry(pci_devhdl_t hdl, pci_cap_t cap, uint_t irq_entry)`
- `pci_err_t cap_msi_unmask_irq_entry_isr(pci_devhdl_t hdl, pci_cap_t cap, uint_t irq_entry)`

The *cap_msi_mask_irq_entry()* and *cap_msi_unmask_irq_entry()* APIs let driver software mask and unmask each of the supported device interrupt sources. These APIs operate only for devices that support the optional Per Vector Masking facility of MSI. They otherwise return PCI_ERR_ENOTSUP.

There's no direct relationship between a device's interrupt entry or the assigned MSI vector and the assigned IRQs returned from *pci_device_read_irq()*. *InterruptMask()* and *InterruptUnmask()* mask and unmask an IRQ, but *cap_msi_mask_irq_entry()* and *cap_msi_unmask_irq_entry()* (if supported) mask and unmask the interrupt sources associated with the MSI interrupt entry. You can obtain the number of interrupt sources from *cap_msi_get_nirq()*.

When you call *pci_device_read_irq()* to retrieve the list of allocated IRQs, the list itself is ordered to correspond with each of the *n* entries (starting from 0).

For example, suppose that a device supports 32 MSI interrupts, as identified by *cap_msi_get_nirq()*. Then suppose that only 8 interrupts were allocated, as identified by *pci_device_read_irq()*. This could be because of system limitations or configuration, or because the driver asked to use only this many in a call to *cap_msi_set_nirq()*. In this case, the 8 IRQs (0 through 7) returned correspond to MSI entries 0 through 7. If the number of requested and allocated IRQs is the same as the number of supported IRQs, this 1:1 relationship is exactly as you'd expect.

The *cap_msi_set_nirq()* returns PCI_ERR_EINVAL if you request more interrupts than are supported. Note that this function merely requests that *nirq* interrupts be used; it isn't until you call *pci_device_cfg_cap_enable()* that an attempt to allocate the number of requested IRQs is made. Depending on the *reqType* and the availability of IRQs, *pci_device_cfg_cap_enable()* either succeeds or fails.

When you're enabling the MSI capability with *pci_device_cfg_cap_enable()*, the *reqType* argument has the following meanings for the specific capability APIs. Any capability module APIs not listed here aren't affected by the *reqType*.

cap_msi_set_nirq()

- pci_reqType_e_MANDATORY — the requested number of IRQs must be allocated to the device, or the capability won't be enabled, and *pci_device_cfg_cap_enable()* fails with PCI_ERR_CAP_NIRQ.

- pci_reqType_e_ADVISORY — an attempt to satisfy the requested number of IRQs will be made, but a lower number may be assigned. The *pci_device_cfg_cap_enable()* call doesn't fail.
- pci_reqType_e_UNSPECIFIED — behaves the same as pci_reqType_e_ADVISORY.

Errors that may be returned by *pci_device_cfg_cap_enable()* from the MSI capability module include:

- PCI_ERR_MSIX_ENABLED — an attempt was made to enable MSI when MSI-X is already enabled.

A note about the ISR safe functions

In certain circumstances, a device being managed by driver software may become temporarily unavailable. This may be due to a bus segment reconfiguration or reset or a hot plug removal. Normally these events are conveyed to the driver prior to their occurrence, but it's possible that previously initiated transactions may result in a interrupt that results in the execution of the ISR functions. In this circumstance, the `_isr`-suffixed calls return PCI_ERR_DEV_NOT_AVAIL, indicating the temporary unavailability of the device. The ISR must detect this specific error (in addition to any others) and handle it accordingly. Access to the device's registers isn't possible in this condition.

Capability ID 0x11 (MSI-X)

The MSI-X capability module enables the use of Extended Message Signalled Interrupts. You should configure the MSI-X capability with the APIs listed below before enabling it. Specifically, you should check the number of device-supported interrupts, and then select the number that you intend to support in the driver. The number of driver-supported interrupts must be less than or equal to the number of device-supported interrupts.

You must enable the MSI-X capability (see *pci_device_cfg_cap_enable()*) before you call *pci_device_read_irq()*. When the capability is enabled, bit 2 (Bus Master) of the Command Register is also set. This bit *isn't automatically cleared* when the capability is disabled.

The MSI and MSI-X capabilities are mutually exclusive. If you want to switch to using MSI-X on a device that has MSI enabled, you must first disable the MSI capability (see *pci_device_cfg_cap_disable()*). Failure to do this results in an error of PCI_ERR_MSI_ENABLED.

The MSI-X capability supports the following APIs as defined in **<pci/cap_msix.h>**.

The following APIs allow software to obtain the number of device-supported interrupt sources and to specify the number to actually use. With MSI-X, the ability to set the disposition of an interrupt source is also possible. Disposition changes take effect when the capability is enabled, and are discussed in more detail below.

`uint_t cap_msix_get_nirq(pci_cap_t cap)`

`uint_t cap_msix_get_nirq_isr(pci_cap_t cap)`

Returns the number of interrupts supported by the device, or 0 on any error. You can call the `_isr`-suffixed version from an ISR.

`pci_err_t cap_msix_set_nirq(pci_devhdl_t hdl, pci_cap_t cap, uint_t nirq)`

`pci_err_t cap_msix_set_irq_entry(pci_devhdl_t hdl, pci_cap_t cap, uint_t irq_entry, int_t disposition)`

The *cap_msix_set_nirq()* function doesn't have the same utility as it does for MSI. Instead, you should use *cap_msix_set_irq_entry()* to modify the number of independent IRQs that the device will use. See the "Interrupt disposition" discussion below.

The following API obtains a read-only pointer to the Pending Bits Array (PBA):

```
cap_msix_pba_t *cap_msix_get_pba_ptr( pci_devhdl_t hdl, pci_cap_t cap )
```

The following APIs let you set or clear interrupt masks for each device interrupt source. ISR-safe versions of these functions are provided; they function identically but are suffixed with `_isr`:

```
pci_err_t cap_msix_mask_irq_entry( pci_devhdl_t hdl, pci_cap_t cap, uint_t irq_entry)
pci_err_t cap_msix_mask_irq_entry_isr( pci_devhdl_t hdl, pci_cap_t cap, uint_t irq_entry)

pci_err_t cap_msix_unmask_irq_entry ( pci_devhdl_t hdl, pci_cap_t cap, uint_t irq_entry)
pci_err_t cap_msix_unmask_irq_entry_isr( pci_devhdl_t hdl, pci_cap_t cap, uint_t irq_entry)
```

In addition, you can mask or unmask all interrupts using the MSI-X defined "function mask" control. These functions don't affect individual entry masks, but rather set or clear the function mask bit 14 of the message control register. They return PCI_ERR_EALREADY if the requested state is already set,

so that you have an indication of the previous state. Any value other than PCI_ERR_OK indicates that the operation couldn't be performed. The ISR-safe versions of these functions are suffixed with `_isr`:

```
pci_err_t cap_msix_mask_irq_all( pci_devhdl_t hdl, pci_cap_t cap)
pci_err_t cap_msix_mask_irq_all_isr( pci_devhdl_t hdl, pci_cap_t cap)

pci_err_t cap_msix_unmask_irq_all( pci_devhdl_t hdl, pci_cap_t cap)
pci_err_t cap_msix_unmask_irq_all_isr( pci_devhdl_t hdl, pci_cap_t cap)
```

The *cap_msix_mask_irq_entry()* and *cap_msix_unmask_irq_entry()* APIs allow driver software to mask and unmask each of the supported device interrupt sources. There's no direct relationship between a device's interrupt entry or the assigned MSI-X vector and the assigned IRQs returned from *pci_device_read_irq()*. *InterruptMask()* and *InterruptUnmask()* mask and unmask an IRQ, but *cap_msix_mask_irq_entry()* and *cap_msix_unmask_irq_entry()* (if supported) mask and unmask the interrupt sources associated with the MSI-X interrupt entry. Some or all of these entries may be configured to share the same IRQ. You can obtain the number of interrupt sources from *cap_msix_get_nirq()*.

Interrupt disposition

The *cap_msix_set_irq_entry()* function allows driver software to control which MSI-X entries are unused and which are shared. By default there's no sharing. You can use this feature to:

- allow driver software to arbitrarily group device interrupt sources onto the same IRQ and hence handle multiple interrupt causes with a single Interrupt Service Routine (ISR). You could also use the same ISR when calling *InterruptAttach()*, but this mechanism allows sharing at the device level.
- disable unneeded or undesired device interrupt sources
- handle interrupt vector aliasing when the number of available MSI-X vectors is less than the number of device-supported vectors

By default, and unless unmodified by calling *cap_msix_set_irq_entry()*, each device interrupt source is assigned a unique MSI-X vector—notwithstanding the availability of MSI-X vectors when the capability is enabled—and hence a unique IRQ. Although a unique IRQ is assigned, this doesn't necessarily mean the IRQ isn't shared from a system perspective. This depends on the interrupt controller, and hence is platform-dependent.

This configuration is established when you initially read the capability with *pci_device_read_cap()*. If software modifies the disposition of one or more interrupt sources and then enables the capability, the disposition of each source will be as configured. If you subsequently disable the capability and then reenable it, the disposition is maintained. If you want to reset the disposition, you can either:

- set the disposition of each entry to itself

  or:

- disable the capability, *free()* the `pci_cap_t`, and then reread the capability. See *pci_device_read_cap()*.

You can set the disposition of an interrupt source by specifying the *irq_entry* to be operated on, along with a *disposition* parameter:

- If the disposition is -1, the interrupt source *irq_entry* is marked as unused and doesn't have a vector assigned when you call *pci_device_cfg_cap_enable()*.

- If *disposition* is greater than or equal to 0, but isn't equal to *irq_entry*, then *irq_entry* shares the same assigned MSI-X vector, and hence IRQ, as the entry identified by *disposition*.
- If *disposition* equals *irq_entry*, the interrupt source has its own MSI-X vector and IRQ assigned, and doesn't share with any other source.

Interrupt entries marked as unused (-1) are masked and can't be unmasked. Otherwise the mask state of the entry isn't altered, but the entry will have been masked when the capability was disabled in order to make disposition changes.

When you're setting the disposition of interrupt sources to shared, the *disposition* argument must always be less than or equal to *irq_entry*. That is, if you wish interrupt sources 2 and 4 to share the same MSI-X vector and hence IRQ, the calls to establish this should be as follows (error handling omitted):

```
cap_msix_set_irq_entry(hdl, msix_cap, 2, 2);
cap_msix_set_irq_entry(hdl, msix_cap, 4, 2);
```

Attempts to do the reverse are rejected.

When you call *pci_device_read_irq()* to retrieve the list of assigned IRQs, the list is ordered to correspond with each of the *n* entries (starting from 0) that aren't marked as unused. For example, suppose that a device supports 256 MSI-X interrupts as identified by *cap_msix_get_nirq()*. Then suppose that only 64 interrupts were allocated, as identified by *pci_device_read_irq()*. This could be because of system limitations or configuration, or because the driver chose to use only this many by setting the disposition of various sources to shared. In this case, each of the 64 IRQs returned (0 through 63) correspond to MSI-X entries 0 through 63. If the number of requested and allocated IRQs is the same as the number of supported IRQs, this 1:1 relationship is exactly as you'd expect.

If, prior to calling *pci_device_cfg_cap_enable()*, the driver called *cap_msix_set_irq_entry()* to mark entries 0, 5, 6 as unused, entries 13 and 14 as shared, and 22 and 23 as shared, then the following IRQ to entry assignments will exist after a successful call to *pci_device_cfg_cap_enable()*:

unused --> entry 0
irq[0] --> IRQ for entry 1
irq[1..3] --> IRQ for entries 2, 3, 4 respectively
unused --> entries 5 and 6
irq[4..9] --> IRQ for entries 7, 8, 9, 10, 11, 12 respectively
irq[10] --> IRQ for entries 13/14 (shared)
irq[11..17] --> IRQ for entries 15, 16, 17, 18, 19, 20, 21 respectively
irq[18] --> IRQ for entries 22/23 (shared)
irq[19..63] --> IRQ for entries 24 through 68 respectively

The *cap_msix_get_pba_ptr()* function returns a read-only pointer to the Pending Bits Array. The PBA allows driver software to query any pending interrupts. If an error occurred, the function returns NULL. The PBA is organized with bit 0 corresponding to interrupt source/entry 0, and bit *n* corresponding to the number of supported MSI-X interrupts the device supports minus 1.

When you disable the MSI-X capability, all interrupt sources are automatically masked. You must disable the capability before making any changes to the interrupt disposition.

When you enable the MSI-X capability with *pci_device_cfg_cap_enable()*, the *reqType* parameter has the following meanings for the specific capability APIs. Any capability module APIs not listed here aren't affected by the *reqType*.

cap_msix_set_irq_entry()

- pci_reqType_e_MANDATORY — a unique MSI-X vector and hence IRQ must be allocated for each of the interrupt sources not marked as unused, or the capability won't be enabled, and *pci_device_cfg_cap_enable()* fails with PCI_ERR_CAP_NIRQ.
- pci_reqType_e_ADVISORY — an attempt to satisfy the required number of IRQs will be made, however a lower number may be assigned down to the minimum required. The *pci_device_cfg_cap_enable()* call may fail with PCI_ERR_IRQ_NOT_AVAIL if this condition can't be met.
- pci_reqType_e_UNSPECIFIED — behaves the same as pci_reqType_e_ADVISORY.

Errors that may be returned by *pci_device_cfg_cap_enable()* from the MSI-X capability module include:

- PCI_ERR_MSI_ENABLED — an attempt was made to enable MSI-X when MSI is already enabled.

A note about the ISR-safe functions

In certain circumstances, a device being managed by driver software may become temporarily unavailable. This may be due to a bus segment reconfiguration or reset, or a hot plug removal. Normally these events are conveyed to the driver prior to their occurrence, but it is possible that previously initiated transactions may result in a interrupt that results in the execution of the ISR functions. In this circumstance, the `_isr`-suffixed calls return PCI_ERR_DEV_NOT_AVAIL, indicating the temporary unavailability of the device. The ISR must detect this specific error (in addition to any others) and handle it accordingly. Access to the device's registers isn't possible in this condition.

Appendix B
Example

This example (partly in pseudocode) looks for all the devices with a given vendor ID, and then gets the ID for each device. If the driver handles the current device, it gets the associated capabilities and loads them if they're of interest, reads the address space information and IRQs, and sets up the mappings.

You could call *pci_device_find()* with the specific VID, DID, and CCODE that your driver supports. If it supports a family of devices, loop calling *pci_device_find()* with each supported DID.

```
uint_t idx = 0;
while ((bdf = pci_device_find(idx, PCI_VID_xxx, PCI_DID_ANY, PCI_CCODE_STORAGE_ATA_ANY)) != PCI_BDF_NONE)
{
    pci_did_t did;
    pci_err_t r = pci_device_read_did(bdf, &did);

    if (r == PCI_ERR_OK)
    {
        /* does this driver handle this device ? */
        if (driver handles this DID)
        {
            pci_devhdl_t hdl = pci_device_attach(bdf, pci_attachFlags_EXCLUSIVE_OWNER, &r);

            if (hdl != NULL)
            {
                pci_ba_t ba[7];    // the maximum number of entries that can be returned
                int_t nba = NELEMENTS(ba);
                pci_irq_t irq[10];    // the maximum number of expected entries based on capabilities
                int_t nirq = NELEMENTS(irq);

                /* optionally determine capabilities of device */
                uint_t capid_idx = 0;
                pci_capid_t capid;

                /* instead of looping could use pci_device_find_capid() to select which capabilities to use */
                while ((r = pci_device_read_capid(bdf, &capid, capid_idx)) == PCI_ERR_OK)
                {
                    if (capid is supported by driver and is to be used)
                    {
                        pci_cap_t cap = NULL;
                        r = pci_device_read_cap(bdf, &cap, capid_idx);
                        if (r == PCI_ERR_OK)
                        {
                            /* use capability specific APIs to set any capability specific options */

                            /* enable the capability */
                            r = pci_device_cfg_cap_enable(hdl, pci_reqType_e_MANDATORY, cap);
                        }
                    }
                    /* get next capability ID */
```

```
                ++capid_idx;
            }

            /* read the address space information */
            r = pci_device_read_ba(hdl, &nba, ba, pci_reqType_e_UNSPECIFIED);
            if ((r == PCI_ERR_OK) && (nba > 0))
            {
                uint_t i;
                for (i=0; i<nba; i++)
                {
                    /* mmap() the address space(s) */
                }
            }
            /* read the irq information */
            r = pci_device_read_irq(hdl, &nirq, irq);
            if ((r == PCI_ERR_OK) && (nirq > 0))
            {
                uint_t i;
                for (i=0; i<nirq; i++)
                {
                    int_t iid = InterruptAttach(irq[i], my_isr, NULL, 0, 0);
                }
            }
            /*
             * do other driver specific processing. For example, if the device does DMA then translate your
             * allocated memory buffers into addresses suitable for access from PCI address space
            */
            pci_ba_t buf =
            {
                .addr = <physical address of memory buffer(s)>, .size = <size of buffer>,
                .type = pci_asType_e_MEM, .attr = pci_asAttr_e_INBOUND | pci_asAttr_e_CONTIG,
            };
            pci_ba_t xlate;
            r = pci_device_map_as(hdl, &buf, &xlate);
            if (r == PCI_ERR_OK)
            {
                /* use xlate.addr to program DMA transfers */
            }
        }
        else slog("Couldn't attach to device B%u:D%u:F%u", PCI_BUS(bdf), PCI_DEV(bdf), PCI_FUNC(bdf));
    }
    else slog("device ID 0x%x not supported", did);
}
else slog("Couldn't read device ID for B%u:D%u:F%u", PCI_BUS(bdf), PCI_DEV(bdf), PCI_FUNC(bdf));

/* get next device instance */
++idx;
}
```

Index

I

K

L

M

O

P

R

S

T

V