NAME

pci, **pci_alloc_msi**, **pci_alloc_msix**, **pci_disable_busmaster**, **pci_disable_io**, **pci_enable_busmaster**, **pci_enable_io**, **pci_find_bsf**, **pci_find_cap**, **pci_find_dbsf**, **pci_find_device**, **pci_find_extcap**, **pci_find_htcap**, **pci_find_next_cap**, **pci_find_next_extcap**, **pci_find_next_htcap**, **pci_find_pcie_root_port**, **pci_get_id**, **pci_get_max_payload**, **pci_get_max_read_req**, **pci_get_powerstate**, **pci_get_vpd_ident**, **pci_get_vpd_readonly**, **pci_iov_attach**, **pci_iov_attach_name**, **pci_iov_detach**, **pci_msi_count**, **pci_msix_count**, **pci_msix_pba_bar**, **pci_msix_table_bar**, **pci_pending_msix**, **pci_read_config**, **pci_release_msi**, **pci_remap_msix**, **pci_restore_state**, **pci_save_state**, **pci_set_max_read_req**, **pci_set_powerstate**, **pci_write_config**, **pcie_adjust_config**, **pcie_flr**, **pcie_get_max_completion_timeout**, **pcie_read_config**, **pcie_wait_for_pending_transactions**, **pcie_write_config** - PCI bus interface

SYNOPSIS

#include <sys/bus.h>
#include <dev/pci/pcireg.h>
#include <dev/pci/pcivar.h>

int
pci_alloc_msi(*device_t dev*, *int *count*);

int
pci_alloc_msix(*device_t dev*, *int *count*);

int
pci_disable_busmaster(*device_t dev*);

int
pci_disable_io(*device_t dev*, *int space*);

int
pci_enable_busmaster(*device_t dev*);

int
pci_enable_io(*device_t dev*, *int space*);

device_t
pci_find_bsf(*uint8_t bus*, *uint8_t slot*, *uint8_t func*);

int
pci_find_cap(*device_t dev*, *int capability*, *int *capreg*);

device_t
pci_find_dbsf(*uint32_t domain, uint8_t bus, uint8_t slot, uint8_t func*);

device_t
pci_find_device(*uint16_t vendor, uint16_t device*);

int
pci_find_extcap(*device_t dev, int capability, int *capreg*);

int
pci_find_htcap(*device_t dev, int capability, int *capreg*);

int
pci_find_next_cap(*device_t dev, int capability, int start, int *capreg*);

int
pci_find_next_extcap(*device_t dev, int capability, int start, int *capreg*);

int
pci_find_next_htcap(*device_t dev, int capability, int start, int *capreg*);

device_t
pci_find_pcie_root_port(*device_t dev*);

int
pci_get_id(*device_t dev, enum pci_id_type type, uintptr_t *id*);

int
pci_get_max_payload(*device_t dev*);

int
pci_get_max_read_req(*device_t dev*);

int
pci_get_powerstate(*device_t dev*);

int
pci_get_vpd_ident(*device_t dev, const char **identptr*);

int

pci_get_vpd_readonly(*device_t dev, const char *kw, const char **vptr*);

int
pci_msi_count(*device_t dev*);

int
pci_msix_count(*device_t dev*);

int
pci_msix_pba_bar(*device_t dev*);

int
pci_msix_table_bar(*device_t dev*);

int
pci_pending_msix(*device_t dev, u_int index*);

uint32_t
pci_read_config(*device_t dev, int reg, int width*);

int
pci_release_msi(*device_t dev*);

int
pci_remap_msix(*device_t dev, int count, const u_int *vectors*);

void
pci_restore_state(*device_t dev*);

void
pci_save_state(*device_t dev*);

int
pci_set_max_read_req(*device_t dev, int size*);

int
pci_set_powerstate(*device_t dev, int state*);

void
pci_write_config(*device_t dev, int reg, uint32_t val, int width*);

uint32_t
pcie_adjust_config(*device_t dev*, *int reg*, *uint32_t mask*, *uint32_t val*, *int width*);

bool
pcie_flr(*device_t dev*, *u_int max_delay*, *bool force*);

int
pcie_get_max_completion_timeout(*device_t dev*);

uint32_t
pcie_read_config(*device_t dev*, *int reg*, *int width*);

bool
pcie_wait_for_pending_transactions(*device_t dev*, *u_int max_delay*);

void
pcie_write_config(*device_t dev*, *int reg*, *uint32_t val*, *int width*);

void
pci_event_fn(*void *arg*, *device_t dev*);

EVENTHANDLER_REGISTER(*pci_add_device*, *pci_event_fn*);

EVENTHANDLER_DEREGISTER(*pci_delete_resource*, *pci_event_fn*);

#include <dev/pci/pci_iov.h>

int
pci_iov_attach(*device_t dev*, *nvlist_t *pf_schema*, *nvlist_t *vf_schema*);

int
pci_iov_attach_name(*device_t dev*, *nvlist_t *pf_schema*, *nvlist_t *vf_schema*, *const char *fmt*, ...);

int
pci_iov_detach(*device_t dev*);

DESCRIPTION

The **pci** set of functions are used for managing PCI devices. The functions are split into several groups: raw configuration access, locating devices, device information, device configuration, and message signaled interrupts.

Raw Configuration Access

The **pci_read_config**() function is used to read data from the PCI configuration space of the device *dev*, at offset *reg*, with *width* specifying the size of the access.

The **pci_write_config**() function is used to write the value *val* to the PCI configuration space of the device *dev*, at offset *reg*, with *width* specifying the size of the access.

The **pcie_adjust_config**() function is used to modify the value of a register in the PCI-express capability register set of device *dev*. The offset *reg* specifies a relative offset in the register set with *width* specifying the size of the access. The new value of the register is computed by modifying bits set in *mask* to the value in *val*. Any bits not specified in *mask* are preserved. The previous value of the register is returned.

The **pcie_read_config**() function is used to read the value of a register in the PCI-express capability register set of device *dev*. The offset *reg* specifies a relative offset in the register set with *width* specifying the size of the access.

The **pcie_write_config**() function is used to write the value *val* to a register in the PCI-express capability register set of device *dev*. The offset *reg* specifies a relative offset in the register set with *width* specifying the size of the access.

NOTE: Device drivers should only use these functions for functionality that is not available via another **pci**() function.

Locating Devices

The **pci_find_bsf**() function looks up the *device_t* of a PCI device, given its *bus*, *slot*, and *func*. The *slot* number actually refers to the number of the device on the bus, which does not necessarily indicate its geographic location in terms of a physical slot. Note that in case the system has multiple PCI domains, the **pci_find_bsf**() function only searches the first one. Actually, it is equivalent to:

```
pci_find_dbsf(0, bus, slot, func);
```

The **pci_find_dbsf**() function looks up the *device_t* of a PCI device, given its *domain*, *bus*, *slot*, and *func*. The *slot* number actually refers to the number of the device on the bus, which does not necessarily indicate its geographic location in terms of a physical slot.

The **pci_find_device**() function looks up the *device_t* of a PCI device, given its *vendor* and *device* IDs. Note that there can be multiple matches for this search; this function only returns the first matching device.

Device Information

The **pci_find_cap**() function is used to locate the first instance of a PCI capability register set for the device *dev*. The capability to locate is specified by ID via *capability*. Constant macros of the form PCIY_xxx for standard capability IDs are defined in <*dev/pci/pcireg.h*>. If the capability is found, then **capreg* is set to the offset in configuration space of the capability register set, and **pci_find_cap**() returns zero. If the capability is not found or the device does not support capabilities, **pci_find_cap**() returns an error. The **pci_find_next_cap**() function is used to locate the next instance of a PCI capability register set for the device *dev*. The *start* should be the **capreg* returned by a prior **pci_find_cap**() or **pci_find_next_cap**(). When no more instances are located **pci_find_next_cap**() returns an error.

The **pci_find_extcap**() function is used to locate the first instance of a PCI-express extended capability register set for the device *dev*. The extended capability to locate is specified by ID via *capability*. Constant macros of the form PCIZ_xxx for standard extended capability IDs are defined in <*dev/pci/pcireg.h*>. If the extended capability is found, then **capreg* is set to the offset in configuration space of the extended capability register set, and **pci_find_extcap**() returns zero. If the extended capability is not found or the device is not a PCI-express device, **pci_find_extcap**() returns an error. The **pci_find_next_extcap**() function is used to locate the next instance of a PCI-express extended capability register set for the device *dev*. The *start* should be the **capreg* returned by a prior **pci_find_extcap**() or **pci_find_next_extcap**(). When no more instances are located **pci_find_next_extcap**() returns an error.

The **pci_find_htcap**() function is used to locate the first instance of a HyperTransport capability register set for the device *dev*. The capability to locate is specified by type via *capability*. Constant macros of the form PCIM_HTCAP_xxx for standard HyperTransport capability types are defined in <*dev/pci/pcireg.h*>. If the capability is found, then **capreg* is set to the offset in configuration space of the capability register set, and **pci_find_htcap**() returns zero. If the capability is not found or the device is not a HyperTransport device, **pci_find_htcap**() returns an error. The **pci_find_next_htcap**() function is used to locate the next instance of a HyperTransport capability register set for the device *dev*. The *start* should be the **capreg* returned by a prior **pci_find_htcap**() or **pci_find_next_htcap**(). When no more instances are located **pci_find_next_htcap**() returns an error.

The **pci_find_pcie_root_port**() function walks up the PCI device hierarchy to locate the PCI-express root port upstream of *dev*. If a root port is not found, **pci_find_pcie_root_port**() returns NULL.

The **pci_get_id**() function is used to read an identifier from a device. The *type* flag is used to specify which identifier to read. The following flags are supported:

PCI_ID_RID Read the routing identifier for the device.

PCI_ID_MSI Read the MSI routing ID. This is needed by some interrupt controllers to route MSI and MSI-X interrupts.

The **pci_get_vpd_ident**() function is used to fetch a device's Vital Product Data (VPD) identifier string. If the device *dev* supports VPD and provides an identifier string, then **identptr* is set to point at a read-only, null-terminated copy of the identifier string, and **pci_get_vpd_ident**() returns zero. If the device does not support VPD or does not provide an identifier string, then **pci_get_vpd_ident**() returns an error.

The **pci_get_vpd_readonly**() function is used to fetch the value of a single VPD read-only keyword for the device *dev*. The keyword to fetch is identified by the two character string *kw*. If the device supports VPD and provides a read-only value for the requested keyword, then **vptr* is set to point at a read-only, null-terminated copy of the value, and **pci_get_vpd_readonly**() returns zero. If the device does not support VPD or does not provide the requested keyword, then **pci_get_vpd_readonly**() returns an error.

The **pcie_get_max_completion_timeout**() function returns the maximum completion timeout configured for the device *dev* in microseconds. If the *dev* device is not a PCI-express device, **pcie_get_max_completion_timeout**() returns zero. When completion timeouts are disabled for *dev*, this function returns the maxmimum timeout that would be used if timeouts were enabled.

The **pcie_wait_for_pending_transactions**() function waits for any pending transactions initiated by the *dev* device to complete. The function checks for pending transactions by polling the transactions pending flag in the PCI-express device status register. It returns true once the transaction pending flag is clear. If transactions are still pending after *max_delay* milliseconds, **pcie_wait_for_pending_transactions**() returns false. If *max_delay* is set to zero, **pcie_wait_for_pending_transactions**() performs a single check; otherwise, this function may sleep while polling the transactions pending flag. **pcie_wait_for_pending_transactions** returns true if *dev* is not a PCI-express device.

Device Configuration

The **pci_enable_busmaster**() function enables PCI bus mastering for the device *dev*, by setting the PCIM_CMD_BUSMASTEREN bit in the PCIR_COMMAND register. The **pci_disable_busmaster**() function clears this bit.

The **pci_enable_io**() function enables memory or I/O port address decoding for the device *dev*, by setting the PCIM_CMD_MEMEN or PCIM_CMD_PORTEN bit in the PCIR_COMMAND register appropriately. The **pci_disable_io**() function clears the appropriate bit. The *space* argument specifies which resource is affected; this can be either SYS_RES_MEMORY or SYS_RES_IOPORT as appropriate. Device drivers should generally not use these routines directly. The PCI bus will enable decoding automatically when a SYS_RES_MEMORY or SYS_RES_IOPORT resource is activated via bus_alloc_resource(9) or bus_activate_resource(9).

The **pci_get_max_payload**() function returns the current maximum TLP payload size in bytes for a PCI-express device. If the *dev* device is not a PCI-express device, **pci_get_max_payload**() returns zero.

The **pci_get_max_read_req**() function returns the current maximum read request size in bytes for a PCI-express device. If the *dev* device is not a PCI-express device, **pci_get_max_read_req**() returns zero.

The **pci_set_max_read_req**() sets the PCI-express maximum read request size for *dev*. The requested *size* may be adjusted, and **pci_set_max_read_req**() returns the actual size set in bytes. If the *dev* device is not a PCI-express device, **pci_set_max_read_req**() returns zero.

The **pci_get_powerstate**() function returns the current power state of the device *dev*. If the device does not support power management capabilities, then the default state of PCI_POWERSTATE_D0 is returned. The following power states are defined by PCI:

PCI_POWERSTATE_D0	State in which device is on and running. It is receiving full power from the system and delivering full functionality to the user.
PCI_POWERSTATE_D1	Class-specific low-power state in which device context may or may not be lost. Buses in this state cannot do anything to the bus, to force devices to lose context.
PCI_POWERSTATE_D2	Class-specific low-power state in which device context may or may not be lost. Attains greater power savings than PCI_POWERSTATE_D1. Buses in this state can cause devices to lose some context. Devices *must* be prepared for the bus to be in this state or higher.
PCI_POWERSTATE_D3	State in which the device is off and not running. Device context is lost, and power from the device can be removed.
PCI_POWERSTATE_UNKNOWN	State of the device is unknown.

The **pci_set_powerstate**() function is used to transition the device *dev* to the PCI power state *state*. If the device does not support power management capabilities or it does not support the specific power state *state*, then the function will fail with EOPNOTSUPP.

The **pci_iov_attach**() function is used to advertise that the given device (and associated device driver) supports PCI Single-Root I/O Virtualization (SR-IOV). A driver that supports SR-IOV must implement the PCI_IOV_INIT(9), PCI_IOV_ADD_VF(9) and PCI_IOV_UNINIT(9) methods. This function should be called during the DEVICE_ATTACH(9) method. If this function returns an error, it is recommended that the device driver still successfully attaches, but runs with SR-IOV disabled. The *pf_schema* and *vf_schema* parameters are used to define what device-specific configuration parameters the device driver accepts when SR-IOV is enabled for the Physical Function (PF) and for individual

Virtual Functions (VFs) respectively. See pci_iov_schema(9) for details on how to construct the schema. If either the *pf_schema* or *vf_schema* is invalid or specifies parameter names that conflict with parameter names that are already in use, **pci_iov_attach**() will return an error and SR-IOV will not be available on the PF device. If a driver does not accept configuration parameters for either the PF device or the VF devices, the driver must pass an empty schema for that device. The SR-IOV infrastructure takes ownership of the *pf_schema* and *vf_schema* and is responsible for freeing them. The driver must never free the schemas itself.

The **pci_iov_attach_name**() function is a variant of **pci_iov_attach**() that allows the name of the associated character device in */dev/iov* to be specified by *fmt*. The **pci_iov_attach**() function uses the name of *dev* as the device name.

The **pci_iov_detach**() function is used to advise the SR-IOV infrastructure that the driver for the given device is attempting to detach and that all SR-IOV resources for the device must be released. This function must be called during the DEVICE_DETACH(9) method if **pci_iov_attach**() was successfully called on the device and **pci_iov_detach**() has not subsequently been called on the device and returned no error. If this function returns an error, the DEVICE_DETACH(9) method must fail and return an error, as detaching the PF driver while VF devices are active would cause system instability. This function is safe to call and will always succeed if **pci_iov_attach**() previously failed with an error on the given device, or if **pci_iov_attach**() was never called on the device.

The **pci_save_state**() and **pci_restore_state**() functions can be used by a device driver to save and restore standard PCI config registers. The **pci_save_state**() function must be invoked while the device has valid state before **pci_restore_state**() can be used. If the device is not in the fully-powered state (PCI_POWERSTATE_D0) when **pci_restore_state**() is invoked, then the device will be transitioned to PCI_POWERSTATE_D0 before any config registers are restored.

The **pcie_flr**() function requests a Function Level Reset (FLR) of *dev*. If *dev* is not a PCI-express device or does not support Function Level Resets via the PCI-express device control register, false is returned. Pending transactions are drained by disabling busmastering and calling **pcie_wait_for_pending_transactions**() before resetting the device. The *max_delay* argument specifies the maximum timeout to wait for pending transactions as described for **pcie_wait_for_pending_transactions**(). If **pcie_wait_for_pending_transactions**() fails with a timeout and *force* is false, busmastering is re-enabled and false is returned. If **pcie_wait_for_pending_transactions**() fails with a timeout and *force* is true, the device is reset despite the timeout. After the reset has been requested, **pcie_flr** sleeps for at least 100 milliseconds before returning true. Note that **pcie_flr** does not save and restore any state around the reset. The caller should save and restore state as needed.

Message Signaled Interrupts

Message Signaled Interrupts (MSI) and Enhanced Message Signaled Interrupts (MSI-X) are PCI

capabilities that provide an alternate method for PCI devices to signal interrupts. The legacy INTx interrupt is available to PCI devices as a SYS_RES_IRQ resource with a resource ID of zero. MSI and MSI-X interrupts are available to PCI devices as one or more SYS_RES_IRQ resources with resource IDs greater than zero. A driver must ask the PCI bus to allocate MSI or MSI-X interrupts using **pci_alloc_msi**() or **pci_alloc_msix**() before it can use MSI or MSI-X SYS_RES_IRQ resources. A driver is not allowed to use the legacy INTx SYS_RES_IRQ resource if MSI or MSI-X interrupts have been allocated, and attempts to allocate MSI or MSI-X interrupts will fail if the driver is currently using the legacy INTx SYS_RES_IRQ resource. A driver is only allowed to use either MSI or MSI-X, but not both.

The **pci_msi_count**() function returns the maximum number of MSI messages supported by the device *dev*. If the device does not support MSI, then **pci_msi_count**() returns zero.

The **pci_alloc_msi**() function attempts to allocate **count* MSI messages for the device *dev*. The **pci_alloc_msi**() function may allocate fewer messages than requested for various reasons including requests for more messages than the device *dev* supports, or if the system has a shortage of available MSI messages. On success, **count* is set to the number of messages allocated and **pci_alloc_msi**() returns zero. The SYS_RES_IRQ resources for the allocated messages will be available at consecutive resource IDs beginning with one. If **pci_alloc_msi**() is not able to allocate any messages, it returns an error. Note that MSI only supports message counts that are powers of two; requests to allocate a non-power of two count of messages will fail.

The **pci_release_msi**() function is used to release any allocated MSI or MSI-X messages back to the system. If any MSI or MSI-X SYS_RES_IRQ resources are allocated by the driver or have a configured interrupt handler, this function will fail with EBUSY. The **pci_release_msi**() function returns zero on success and an error on failure.

The **pci_msix_count**() function returns the maximum number of MSI-X messages supported by the device *dev*. If the device does not support MSI-X, then **pci_msix_count**() returns zero.

The **pci_msix_pba_bar**() function returns the offset in configuration space of the Base Address Register (BAR) containing the MSI-X Pending Bit Array (PBA) for device *dev*. The returned value can be used as the resource ID with bus_alloc_resource(9) and bus_release_resource(9) to allocate the BAR. If the device does not support MSI-X, then **pci_msix_pba_bar**() returns -1.

The **pci_msix_table_bar**() function returns the offset in configuration space of the BAR containing the MSI-X vector table for device *dev*. The returned value can be used as the resource ID with bus_alloc_resource(9) and bus_release_resource(9) to allocate the BAR. If the device does not support MSI-X, then **pci_msix_table_bar**() returns -1.

The **pci_alloc_msix**() function attempts to allocate **count* MSI-X messages for the device *dev*. The **pci_alloc_msix**() function may allocate fewer messages than requested for various reasons including requests for more messages than the device *dev* supports, or if the system has a shortage of available MSI-X messages. On success, **count* is set to the number of messages allocated and **pci_alloc_msix**() returns zero. For MSI-X messages, the resource ID for each SYS_RES_IRQ resource identifies the index in the MSI-X table of the corresponding message. A resource ID of one maps to the first index of the MSI-X table; a resource ID two identifies the second index in the table, etc. The **pci_alloc_msix**() function assigns the **count* messages allocated to the first **count* table indices. If **pci_alloc_msix**() is not able to allocate any messages, it returns an error. Unlike MSI, MSI-X does not require message counts that are powers of two.

The BARs containing the MSI-X vector table and PBA must be allocated via bus_alloc_resource(9) before calling **pci_alloc_msix**() and must not be released until after calling **pci_release_msi**(). Note that the vector table and PBA may be stored in the same BAR or in different BARs.

The **pci_pending_msix**() function examines the *dev* device's PBA to determine the pending status of the MSI-X message at table index *index*. If the indicated message is pending, this function returns a non-zero value; otherwise, it returns zero. Passing an invalid *index* to this function will result in undefined behavior.

As mentioned in the description of **pci_alloc_msix**(), MSI-X messages are initially assigned to the first N table entries. A driver may use a different distribution of available messages to table entries via the **pci_remap_msix**() function. Note that this function must be called after a successful call to **pci_alloc_msix**() but before any of the SYS_RES_IRQ resources are allocated. The **pci_remap_msix**() function returns zero on success, or an error on failure.

The *vectors* array should contain *count* message vectors. The array maps directly to the MSI-X table in that the first entry in the array specifies the message used for the first entry in the MSI-X table, the second entry in the array corresponds to the second entry in the MSI-X table, etc. The vector value in each array index can either be zero to indicate that no message should be assigned to the corresponding MSI-X table entry, or it can be a number from one to N (where N is the count returned from the previous call to **pci_alloc_msix**()) to indicate which of the allocated messages should be assigned to the corresponding MSI-X table entry.

If **pci_remap_msix**() succeeds, each MSI-X table entry with a non-zero vector will have an associated SYS_RES_IRQ resource whose resource ID corresponds to the table index as described above for **pci_alloc_msix**(). MSI-X table entries that with a vector of zero will not have an associated SYS_RES_IRQ resource. Additionally, if any of the original messages allocated by **pci_alloc_msix**() are not used in the new distribution of messages in the MSI-X table, they will be released automatically. Note that if a driver wishes to use fewer messages than were allocated by **pci_alloc_msix**(), the driver

must use a single, contiguous range of messages beginning with one in the new distribution. The **pci_remap_msix**() function will fail if this condition is not met.

Device Events

The *pci_add_device* event handler is invoked every time a new PCI device is added to the system. This includes the creation of Virtual Functions via SR-IOV.

The *pci_delete_device* event handler is invoked every time a PCI device is removed from the system.

Both event handlers pass the *device_t* object of the relevant PCI device as *dev* to each callback function. Both event handlers are invoked while *dev* is unattached but with valid instance variables.

SEE ALSO

pci(4), pciconf(8), bus_alloc_resource(9), bus_dma(9), bus_release_resource(9), bus_setup_intr(9), bus_teardown_intr(9), devclass(9), device(9), driver(9), eventhandler(9), rman(9)

NewBus, *FreeBSD Developers' Handbook*, https://docs.freebsd.org/en/books/developers-handbook/.

Shanley and Anderson, *PCI System Architecture*, *Addison-Wesley*, 2nd Edition, ISBN 0-201-30974-2.

AUTHORS

This manual page was written by Bruce M Simpson <*bms@FreeBSD.org*> and John Baldwin <*jhb@FreeBSD.org*>.

BUGS

The kernel PCI code has a number of references to "slot numbers". These do not refer to the geographic location of PCI devices, but to the device number assigned by the combination of the PCI IDSEL mechanism and the platform firmware. This should be taken note of when working with the kernel PCI code.

The PCI bus driver should allocate the MSI-X vector table and PBA internally as necessary rather than requiring the caller to do so.