NAME

pci **-** generic PCI/PCIe bus driver

SYNOPSIS

To compile the PCI bus driver into the kernel, place the following line in your kernel configuration file:

```
device pci
```

To compile in support for Single Root I/O Virtualization (SR-IOV):

```
options PCI_IOV
```

To compile in support for native PCI-express HotPlug:

```
options PCI_HP
```

DESCRIPTION

The **pci** driver provides support for PCI and PCIe devices in the kernel and limited access to PCI devices for userland.

The **pci** driver provides a */dev/pci* character device that can be used by userland programs to read and write PCI configuration registers. Programs can also use this device to get a list of all PCI devices, or all PCI devices that match various patterns.

Since the **pci** driver provides a write interface for PCI configuration registers, system administrators should exercise caution when granting access to the **pci** device. If used improperly, this driver can allow userland applications to crash a machine or cause data loss. In particular, driver only allows operations on the opened */dev/pci* to modify system state if the file descriptor was opened for writing. For instance, the PCIOCREAD and PCIOCBARMMAP operations require a writeable descriptor, because reading a config register or a BAR read access could have function-specific side-effects.

The **pci** driver implements the PCI bus in the kernel. It enumerates any devices on the PCI bus and gives PCI client drivers the chance to attach to them. It assigns resources to children, when the BIOS does not. It takes care of routing interrupts when necessary. It reprobes the unattached PCI children when PCI client drivers are dynamically loaded at runtime. The **pci** driver also includes support for PCI-PCI bridges, various platform-specific Host-PCI bridges, and basic support for PCI VGA adapters.

IOCTLS

The following ioctl(2) calls are supported by the **pci** driver. They are defined in the header file <*sys/pciio.h*>.

PCIOCGETCONF

This ioctl(2) takes a *pci_conf_io* structure. It allows the user to retrieve information on all PCI devices in the system, or on PCI devices matching patterns supplied by the user. The call may set *errno* to any value specified in either copyin(9) or copyout(9). The *pci_conf_io* structure consists of a number of fields:

pat_buf_len The length, in bytes, of the buffer filled with user-supplied patterns.

num_patterns The number of user-supplied patterns.

patterns Pointer to a buffer filled with user-supplied patterns. *patterns* is a pointer to *num_patterns* *pci_match_conf* structures. The *pci_match_conf* structure consists of the following elements:

pc_sel PCI domain, bus, slot and function.

pd_name PCI device driver name.

pd_unit PCI device driver unit number.

pc_vendor PCI vendor ID.

pc_device PCI device ID.

pc_class PCI device class.

flags The flags describe which of the fields the kernel should match against. A device must match all specified fields in order to be returned. The match flags are enumerated in the *pci_getconf_flags* structure. Hopefully the flag values are obvious enough that they do not need to described in detail.

match_buf_len Length of the *matches* buffer allocated by the user to hold the results of the PCIOCGETCONF query.

num_matches Number of matches returned by the kernel.

matches Buffer containing matching devices returned by the kernel. The items in this buffer are of type *pci_conf*, which consists of the following items:

pc_sel	PCI domain, bus, slot and function.
pc_hdr	PCI header type.
pc_subvendor	PCI subvendor ID.
pc_subdevice	PCI subdevice ID.
pc_vendor	PCI vendor ID.
pc_device	PCI device ID.
pc_class	PCI device class.
pc_subclass	PCI device subclass.
pc_progif	PCI device programming interface.
pc_revid	PCI revision ID.
pd_name	Driver name.
pd_unit	Driver unit number.

offset The offset is passed in by the user to tell the kernel where it should start traversing the device list. The value passed out by the kernel points to the record immediately after the last one returned. The user may pass the value returned by the kernel in subsequent calls to the PCIOCGETCONF ioctl. If the user does not intend to use the offset, it must be set to zero.

generation PCI configuration generation. This value only needs to be set if the offset is set. The kernel will compare the current generation number of its internal device list to the generation passed in by the user to determine whether its device list has changed since the user last called the PCIOCGETCONF ioctl. If the device list has changed, a status of *PCI_GETCONF_LIST_CHANGED* will be passed back.

status — The status tells the user the disposition of his request for a device list. The possible status values are:

PCI_GETCONF_LAST_DEVICE
This means that there are no more devices in the PCI device list matching the specified criteria after the ones returned in the *matches* buffer.

PCI_GETCONF_LIST_CHANGED
This status tells the user that the PCI device list has changed since his last call to the PCIOCGETCONF ioctl and he must reset the *offset* and *generation* to zero to start over at the beginning of the list.

PCI_GETCONF_MORE_DEVS
This tells the user that his buffer was not large enough to hold all of the remaining devices in the device list that match his criteria.

PCI_GETCONF_ERROR
This indicates a general error while servicing the user's request. If the *pat_buf_len* is not equal to *num_patterns* times **sizeof**(*struct pci_match_conf*), *errno* will be set to EINVAL.

PCIOCREAD — This ioctl(2) reads the PCI configuration registers specified by the passed-in *pci_io* structure. The *pci_io* structure consists of the following fields:

pi_sel — A *pcisel* structure which specifies the domain, bus, slot and function the user would like to query. If the specific bus is not found, errno will be set to ENODEV and -1 returned from the ioctl.

pi_reg — The PCI configuration registers the user would like to access.

pi_width — The width, in bytes, of the data the user would like to read. This value may be either 1, 2, or 4. 3-byte reads and reads larger than 4 bytes are not supported. If an invalid width is passed, errno will be set to EINVAL.

pi_data — The data returned by the kernel.

PCIOCWRITE — This ioctl(2) allows users to write to the PCI configuration registers specified in the passed-in *pci_io* structure. The *pci_io* structure is described above. The limitations on data width described for reading registers, above, also apply to writing PCI configuration registers.

PCIOCATTACHED

This ioctl(2) allows users to query if a driver is attached to the PCI device specified in the passed-in *pci_io* structure. The *pci_io* structure is described above, however, the *pi_reg* and *pi_width* fields are not used. The status of the device is stored in the *pi_data* field. A value of 0 indicates no driver is attached, while a value larger than 0 indicates that a driver is attached.

PCIOCBARMMAP

This ioctl(2) command allows userspace processes to mmap(2) the memory-mapped PCI BAR into its address space. The input parameters and results are passed in the *pci_bar_mmap* structure, which has the following fields:

uint64_t *pbm_map_base*

Reports the established mapping base to the caller. If *PCIIO_BAR_MMAP_FIXED* flag was specified, then this field must be filled before the call with the desired address for the mapping.

uint64_t pbm_map_length

Reports the mapped length of the BAR, in bytes. Its .Vt uint64_t value is always multiple of machine pages.

int64_t pbm_bar_length

Reports length of the bar as exposed by the device.

int pbm_bar_off

Reports offset from the mapped base to the start of the first register in the bar.

struct pcisel pbm_sel

Should be filled before the call. Describes the device to operate on.

int pbm_reg The BAR index to mmap.

int pbm_flags Flags which augments the operation. See below.

int pbm_memattr

The caching attribute for the mapping. Typical values are

VM_MEMATTR_UNCACHEABLE for control registers BARs, and VM_MEMATTR_WRITE_COMBINING for frame buffers. Regular memory-like BAR should be mapped with VM_MEMATTR_DEFAULT attribute.

Currently defined flags are:

PCIIO_BAR_MMAP_FIXED
The resulted mappings should be established at the address specified by the *pbm_map_base* member, otherwise fail.

PCIIO_BAR_MMAP_EXCL
Must be used together with PCIIO_BAR_MMAP_FIXED If the specified base contains already established mappings, the operation fails instead of implicitly unmapping them.

PCIIO_BAR_MMAP_RW
The requested mapping allows both reading and writing. Without the flag, read-only mapping is established. Note that it is common for the device registers to have side-effects even on reads.

PCIIO_BAR_MMAP_ACTIVATE
(Unimplemented) If the BAR is not activated, activate it in the course of mapping. Currently attempt to mmap an inactive BAR results in error.

PCIOCBARIO
This ioctl(2) command allows users to read from and write to BARs. The I/O request parameters are passed in a *struct pci_bar_ioreq* structure, which has the following fields:

struct pcisel pbi_sel
Describes the device to operate on.

int pbi_op
The operation to perform. Currently supported values are PCIBARIO_READ and PCIBARIO_WRITE.

uint32_t pbi_bar

The index of the BAR on which to operate.

uint32_t pbi_offset
The offset into the BAR at which to operate.

uint32_t pbi_width
The size, in bytes, of the I/O operation. 1-byte, 2-byte, 4-byte and 8-byte perations are supported.

uint32_t pbi_value
For reads, the value is returned in this field. For writes, the caller specifies the value to be written in this field.

Note that this operation maps and unmaps the corresponding resource and so is relatively expensive for memory BARs. The *PCIOCBARMMAP* ioctl(2) can be used to create a persistent userspace mapping for such BARs instead.

LOADER TUNABLES

Tunables can be set at the loader(8) prompt before booting the kernel, or stored in loader.conf(5). The current value of these tunables can be examined at runtime via sysctl(8) nodes of the same name. Unless otherwise specified, each of these tunables is a boolean that can be enabled by setting the tunable to a non-zero value.

hw.pci.clear_bars (Defaults to 0)
Ignore any firmware-assigned memory and I/O port resources. This forces the PCI bus driver to allocate resource ranges for memory and I/O port resources from scratch.

hw.pci.clear_buses (Defaults to 0)
Ignore any firmware-assigned bus number registers in PCI-PCI bridges. This forces the PCI bus driver and PCI-PCI bridge driver to allocate bus numbers for secondary buses behind PCI-PCI bridges.

hw.pci.clear_pcib (Defaults to 0)
Ignore any firmware-assigned memory and I/O port resource windows in PCI-PCI bridges. This forces the PCI-PCI bridge driver to allocate memory and I/O port resources for resource windows from scratch.

By default the PCI-PCI bridge driver will allocate windows that contain the firmware-assigned resources devices behind the bridge. In addition, the PCI-PCI bridge driver will suballocate from existing window regions when possible to satisfy a resource request. As a result, both *hw.pci.clear_bars* and *hw.pci.clear_pcib* must be enabled to fully ignore firmware-supplied resource assignments.

hw.pci.default_vgapci_unit (Defaults to -1)

By default, the first PCI VGA adapter encountered by the system is assumed to be the boot display device. This tunable can be set to choose a specific VGA adapter by specifying the unit number of the associated *vgapciX* device.

hw.pci.do_power_nodriver (Defaults to 0)

Place devices into a low power state (D3) when a suitable device driver is not found. Can be set to one of the following values:

3 Powers down all PCI devices without a device driver.

2 Powers down most devices without a device driver. PCI devices with the display, memory, and base peripheral device classes are not powered down.

1 Similar to a setting of 2 except that storage controllers are also not powered down.

0 All devices are left fully powered.

A PCI device must support power management to be powered down. Placing a device into a low power state may not reduce power consumption.

hw.pci.do_power_resume (Defaults to 1)

Place PCI devices into the fully powered state when resuming either the system or an individual device. Setting this to zero is discouraged as the system will not attempt to power up non-powered PCI devices after a suspend.

hw.pci.do_power_suspend (Defaults to 1)

Place PCI devices into a low power state when suspending either the system or individual devices. Normally the D3 state is used as the low power state, but firmware may override the desired power state during a system suspend.

hw.pci.enable_ari (Defaults to 1)

Enable support for PCI-express Alternative RID Interpretation. This is often used in conjunction with SR-IOV.

hw.pci.enable_io_modes (Defaults to 1)

Enable memory or I/O port decoding in a PCI device's command register if it has firmware-assigned memory or I/O port resources. The firmware (BIOS) in some systems does not enable memory or I/O port decoding for some devices even when it has assigned resources to the device. This enables decoding for such resources during bus probe.

hw.pci.enable_msi (Defaults to 1)
Enable support for Message Signalled Interrupts (MSI). MSI interrupts can be disabled by setting this tunable to 0.

hw.pci.enable_msix (Defaults to 1)
Enable support for extended Message Signalled Interrupts (MSI-X). MSI-X interrupts can be disabled by setting this tunable to 0.

hw.pci.enable_pcie_ei (Defaults to 0)
Enable support for PCI-express Electromechanical Interlock.

hw.pci.enable_pcie_hp (Defaults to 1)
Enable support for native PCI-express HotPlug.

hw.pci.honor_msi_blacklist (Defaults to 1)
MSI and MSI-X interrupts are disabled for certain chipsets known to have broken MSI and MSI-X implementations when this tunable is set. It can be set to zero to permit use of MSI and MSI-X interrupts if the chipset match is a false positive.

hw.pci.iov_max_config (Defaults to 1MB)
The maximum amount of memory permitted for the configuration parameters used when creating Virtual Functions via SR-IOV. This tunable can also be changed at runtime via sysctl(8).

hw.pci.realloc_bars (Defaults to 0)
Attempt to allocate a new resource range during the initial device scan for any memory or I/O port resources with firmware-assigned ranges that conflict with another active resource.

hw.pci.usb_early_takeover (Defaults to 1 on amd64 and i386)
Disable legacy device emulation of USB devices during the initial device scan. Set this tunable to zero to use USB devices via legacy emulation when using a custom kernel without USB controller drivers.

hw.pci<D>.<B>.<S>.INT<P>.irq
These tunables can be used to override the interrupt routing for legacy PCI INTx interrupts. Unlike other tunables in this list, these do not have corresponding sysctl nodes. The tunable name includes the address of the PCI device as well as the pin of the desired INTx IRQ to

override:

<D> The domain (or segment) of the PCI device in decimal.

<B> The bus address of the PCI device in decimal.

<S> The slot of the PCI device in decimal.

<P> The interrupt pin of the PCI slot to override. One of ‘A’, ‘B’, ‘C’, or ‘D’.

The value of the tunable is the raw IRQ value to use for the INTx interrupt pin identified by the tunable name. Mapping of IRQ values to platform interrupt sources is machine dependent.

DEVICE WIRING

You can wire the device unit at a given location with device.hints. Entries of the form *hints.<name>.<unit>.at="pci<B>:<S>:<F>"* or *hints.<name>.<unit>.at="pci<D>:<B>:<S>:<F>"* will force the driver *name* to probe and attach at unit *unit* for any PCI device found to match the specification, where:

<D> The domain (or segment) of the PCI device in decimal. Defaults to 0 if unspecified

<B> The bus address of the PCI device in decimal.

<S> The slot of the PCI device in decimal.

<F> The function of the PCI device in decimal.

The code to do the matching requires an exact string match. Do not specify the angle brackets (< >) in the hints file. Wiring multiple devices to the same *name* and *unit* produces undefined results.

Examples

Given the following lines in */boot/device.hints*: **hint.nvme.3.at="pci6:0:0" hint.igb.8.at="pci14:0:0"** If there is a device that supports igb(4) at PCI bus 14 slot 0 function 0, then it will be assigned igb8 for probe and attach. Likewise, if there is an nvme(4) card at PCI bus 6 slot 0 function 0, then it will be assigned nvme3 for probe and attach. If another type of card is in either of these locations, the name and unit of that card will be the default names and will be unaffected by these hints. If other igb or nvme cards are located elsewhere, they will be assigned their unit numbers sequentially, skipping the unit numbers that have ’at’ hints.

FILES

/dev/pci Character device for the **pci** driver.

SEE ALSO

pciconf(8)

HISTORY

The **pci** driver (not the kernel's PCI support code) first appeared in FreeBSD 2.2, and was written by Stefan Esser and Garrett Wollman. Support for device listing and matching was re-implemented by Kenneth Merry, and first appeared in FreeBSD 3.0.

AUTHORS

Kenneth Merry <*ken@FreeBSD.org*>

BUGS

It is not possible for users to specify an accurate offset into the device list without calling the PCIOCGETCONF at least once, since they have no way of knowing the current generation number otherwise. This probably is not a serious problem, though, since users can easily narrow their search by specifying a pattern or patterns for the kernel to match against.