PCI_I440BX

The PCI_I440BX module is a bus and DMA controller for the PCIbuses controlled by the Intel 400BX chipset.

Process Information

Prototype Name	pci
Link Order	before any PCI or ATA bus drivers
Process Name	unused ('initonly' process).

Module Options

PCI_TRACE_CONFIG	If this symbol is defined, all accesses to PCI configuration space will be traced through calls to *rome_kprintf*. This is only useful during system debugging if a device on the PCI bus appears to be incorrectly identified or configured..

Target File Definitions

PCI_BUSES	The number of PCI buses on the system. This is normally 2 (one for AGP and one for PCI).
PCI_IO_BASE	The address in IOspace of the start of the area into which device registers can be mapped.
PCI_MEM_BASE	The address of an area of main address space into which bus-device memory areas can be mapped.
PCI_NUM_SLOTS	The number of slots on each bus to be probed during startup. This usually 20 on this chipset.

Data Definitions

pci_bios.h contains the following data type definitions:

PCI_ADDRESS_MAP	The data structure representing the resource map for the device after it has been configured. The *device* and *vendor* fields contain the

	device and vendor codes for the specific device. The *int_line* and *int_pin* fields contain the interrupt line and pin values from the device's configuration. The seven-element arrays represent arrays of memory or IOspace allocated to the device. For each slot, the *io* field is *TRUE* if this is an IOspace allocation. The *base_reg* field contains the size and type information from the device, and *mem_req* is the size extracted from this field. The *mem_assigned* field gives the address in IOspace or memory to which this area has been mapped.
PCI_BIOS_LOCATION	The data structure representing the location of a device on the PCI bus. It contains the *bus_number* of the PCI bus on which the device was found, the *device_number* (or slot) on that bus, and the *function_number* for multi-function devices.

Process Operation

The initialisation routine locates the individual components of the I440BX chipset on the buses, and initialises the DMA areas. The area from c.0000h to f.000h is used for DMA control blocks and is marked uncached and the PAM registers are set to make the memory accessible as RAM. the area from 10.0000h to 20.0000h is used for DMA buffers and is also marked uncached. The buffers are represented internally as a bitmap-vector resource of 256 4k pages. All memory above 8000.0000h is available for memory-mapped devices and is uncachable.

There is no main process, and the module does not handle any messages.

Shared Library Macros and Routines

The following routines are used by drivers for devices connected through the chipset on both the PCI and ATA buses to handle device configuration and DMA requests.

The DMA interface is through a *channel* allocated to devices which can use DMA. The channels are called *PCI_DMA_PRIMARY_IDE*, *PCI_DMA_SECONDARY_IDE* and *PCI_DMA_CHANNELn* for *n = 0..7.* Channel 4 is not available as it is the cascade channel. The individual channels are allocated through the target file according to the configuration. Each channel is represented by an *area* controlling its operation. Only one operation can be active on each channel at a time. Associated with each channel is an area of uncached memory used as an input or output buffer. Data are copied into these buffers within the library, so the device always sees cached data (for optimal performance).

In most cases, the DMA channel is started when the *setup* routine is called. However the *IDE* channels require additional configuration steps in the driver and there is a separate *start* command. Calling *start* on the other channels is harmless, and probably good practice.

Devices that perform their own DMA also require access to uncached memory areas. These are allocated as *pages* and can be used as input or output buffers as required. The library contains routines to allow device drivers to obtain and release pages as needed.

pci_dma_done

int *pci_dma_done*(
    int *channel_index*)

The *pci_dma_done* routine should be called when the DMA engine signals completion to the device using *channel_index*. The routine stops the DMA channel and, if the operation was a read, copies the data into the buffer previously provided in the *setup* call. The routine returns the status flags from the DMA channel.

pci_dma_new_area

int *pci_dma_new_area*(
  int *channel*)

The *pci_dma_new_area* routine initialises the area for the DMA resource represented by *channel.* The value returned is an integer used to reference the area on subsequent calls.

pci_dma_page_free

void *pci_dma_page_free*(
  uchar **page)*

The *pci_dma_page_free* routine returns the page at *page* to the free pool. The caller should ensure that this was a page originally returned by a call to *pci_dma_page_new*, and that no device has a reference to this page (for example in a linked-list of receive buffers).

pci_dma_page_new

uchar **pci_dma_page_new*(**void**)

The *pci_dma_page_new* routine returns a pointer to a 4k page suitable for using as an input or output buffer for an external device.

pci_dma_setup

void *pci_dma_setup*(**void**)

The *pci_dma_setup* routine is called from within the main PCI initialisation process to set up the areas for DMA.

pci_dma_setup_read

void *pci_dma_setup_read*(
  int *channel_index*,
  ptr *buffer*,
  int *count*)

The *pci_dma_setup_read* routine prepares the channel area identified by *channel_index* for a read operation of up to *count* bytes.

pci_dma_setup_write

void *pci_dma_setup_write*(
  int *channel_index*,
  ptr *buffer*,
  int *count*)

The *pci_dma_setup_write* routine prepares the channel area identified by *channel_index* for a write operation. *count* bytes of data are copied from the supplied *buffer* into the channels uncached memory.

pci_dma_start_read, pci_start_dma_write

> **void** *pci_dma_start_{read|write}*(
> 　　**int** *channel_index*)

The *pci_dma_start_read* and *pci_dma_start_write* routines start the DMA engine for *channel_index*. It is assumed that the channel has been correctly prepared by a corresponding *setup* call.

pci_find_class_code

> **int** *pci_find_class_code*(
> 　　**int** *class_code*,
> 　　**int** *index*,
> 　　**PCI_DEVICE_LOCATION** **devloc*)

The *pci_find_class_code* routine locates the *index*'th device in the system with device class *class_code*, and sets *devloc* to its bus, slot and function location. The routine returns 0 if a device was found, *PCI_DEVICE_NOT_FOUND* if no such device exists, or another PCI error code.

pci_find_device

> **int** *pci_find_device*(
> 　　**int** *vendor_id*,
> 　　**int** *device_id*,
> 　　**int** *index*,
> 　　**PCI_DEVICE_LOCATION** **devloc*)

The *pci_find_device* routine locates the *index*'th device in the system with identification *vendor_id* and *device_id*, and sets *devloc* to its bus, slot and function location. The routine returns 0 if a device was found, *PCI_DEVICE_NOT_FOUND* if no such device exists, or another PCI error code.

pci_get_irqs

> **int** *pci_get_irqs*(
> 　　**PCI_DEVICE_LOCATION** **devloc*,
> 　　**uint** **where*)

The *pci_get_irqs* routine returns the interrupt vector numbers of the four PCI interrupts for tha device at *devloc* in the supplied array of integers. *where[0]* corresponds to INTA and *where[3]* to INTD. The routine returns 0 if successful and a PCI error code otherwise.

pci_read_config1, pci_read_config2, pci_read_config4

> **int** *pci_read_config{1|2|4}*(
> 　　**int** *bus*,
> 　　**int** *dev*,
> 　　**int** *func*,
> 　　**int** *reg*,
> 　　{**uchar**|**ushort**|**uint**} **data*)

The *pci_read_config* routines read one, two or four bytes from the configuration register *reg* of the device on PCI bus *bus* at slot *dev* subfunction *func*. The *reg* value should be aligned correctly for the size of the read. The value is returned through the *data* pointer and the routine returns 0 on success or a PCI error otherwise.

pci_read_controller1, pci_read_controller2, pci_read_controller4

int *pci_read_controller{1|2|4}*(
  int *controller_type*,
  int *reg*,
  {**uchar**|**ushort**|**uint**} **data*)

The *pci_read_controller* routines read one, two or four bytes from the configuration register *reg* of the PCIbus controller element specified by *controller_type*. The controller types are defined in the *pci_bios.h* header file. The *reg* value should be aligned correctly for the size of the read. The value is returned through the *data* pointer and the routine returns 0 on success or a PCI error otherwise.

pci_write_config1, pci_write_config2, pci_write_config4

int *pci_write_config{1|2|4}*(
  int *bus*,
  int *dev*,
  int *func*,
  int *reg*,
  {**uchar**|**ushort**|**uint**} *data*)

The *pci_write_config* routines write one, two or four bytes from *data* into the configuration register *reg* of the device on PCI bus *bus* at slot *dev* subfunction *func*. The *reg* value should be aligned correctly for the size of the write. The routine returns 0 on success or a PCI error otherwise

pci_write_controller1, pci_write_controller2, pci_write_controller4

int *pci_write_controller{1|2|4}*(
  int *controller_type*,
  int *reg*,
  {**uchar**|**ushort**|**uint**} *data*)

The *pci_write_controller* routines write one, two or four bytes from *data* into the configuration register *reg* of the PCIbus controller element specified by *controller_type*. The controller types are defined in the *pci_bios.h* header file. The *reg* value should be aligned correctly for the size of the write. The value is returned through the *data* pointer and the routine returns 0 on success or a PCI error otherwise.