Programming PCI-Devices under Linux

by Claus Schroeter
(clausi@chemie.fu-berlin.de)

Abstract

This document is intended to be a short tutorial about PCI Programming under Linux. It describes the PCI basics and its implementation under Linux.

Introduction to PCI

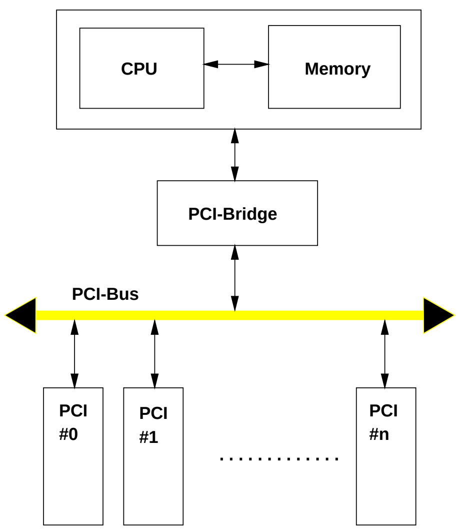

Figure 1: The Architecture of the PCI Subsystem

The Peripheral Component Interconnect - Bus (PCI) today is present in a wide variety of microcomputers ranging from Intel-based PC architectures to DEC-Alpha-based workstations. The main difference between the old fashioned ISA-Bus and PCI is the complete separation of the Bus-subsystem from the Memory Subsystem. The CPU communicates with the PCI subsystem using a special chipset known as PCI-Bridge. This is an intelligent controller, that handles all necessary tasks to transfer data from or to the CPU or the Memory Subsystem. On PCI the adresses and data are transferred as seperate chunks over the bus because all bus-lines can be used either as adress- or as data-lines. In special cases only one start address is transferred followed by a whole data block. To determine the address for each single data-word the PCI-Bridge and the PCI-Adapter increment internal counters, so the address is calculated while the data words are transmitted. This so called "burst" cycle speeds up the transfer significantly (up to 266Mbyte/sec for a 64bit data-transfer). Thanks to the intelligent handling of this tasks the programmer need not to take care about all this stuff.

A schematic view of the PCI subsystem is depicted in Figure 1.

The PCI design forces Hardware designers to use a standardized interface for PCI-Board access and control. Each bus device has its own 265byte space of Memory for configuration purposes that can be accessed through the CONFIG_ADDRESS and the CONFIG_DATA registers. On system startup the BIOS uses the device configuration block to set the different board properties as IRQ line and base address for example. No jumpers need to be placed in the right places, so IRQ or address conflicts are no longer a problem with PCI. This feature is commonly known as Plug&Play functionality.

First Contact with the PCI subsystem

To get an impression how linux sees the PCI bus in your computer try getting the PCI bus configuration from the kernel with `cat /proc/pci`. If everything works OK you will (hopefully) see:

```
PCI devices found:
  Bus  0, device  12, function  0:
    SCSI storage controller: Adaptec AIC-7881U (rev 0).
      Medium devsel.  Fast back-to-back capable.  IRQ 11.
      Master Capable.  Latency=32.  Min Gnt=8.Max Lat=8.
      I/O at 0xd800.
      Non-prefetchable 32 bit memory at 0xf7fff000.
  Bus  0, device  11, function  0:
    VGA compatible controller: S3 Inc. Vision 968 (rev 0).
      Medium devsel.  IRQ 10.
      Non-prefetchable 32 bit memory at 0xf2000000.
  Bus  0, device  10, function  0:
    Ethernet controller: 3Com 3C590 10bT (rev 0).
      Medium devsel.  IRQ 12.  Master Capable.  Latency=248.  Min Gnt=3.Max Lat=8.
      I/O at 0xe000.
  Bus  0, device   7, function  2:
    Unknown class: Intel Unknown device (rev 0).
      Vendor id=8086. Device id=7020.
      Medium devsel.  Fast back-to-back capable.  IRQ 9.  Master Capable.  Latency=32.
      I/O at 0xe400.
  Bus  0, device   7, function  1:
    IDE interface: Intel 82371SB Triton II PIIX (rev 0).
      Medium devsel.  Fast back-to-back capable.  Master Capable.  Latency=32.
      I/O at 0xe800.
  Bus  0, device   7, function  0:
    ISA bridge: Intel 82371SB Triton II PIIX (rev 0).
      Medium devsel.  Fast back-to-back capable.  Master Capable.  No bursts.
  Bus  0, device   0, function  0:
    Host bridge: Intel 82439HX Triton II (rev 1).
      Medium devsel.  Master Capable.  Latency=32.
```

All devices that are known to Linux you will see at `/proc/pci`. Each device configuration block is assigned to a device and a function ID. To identify a certain device while driver writing you will at least have to know the vendor- and the device-id that is statically stored in the device configuration block (as discussed later on). Driver writers normally need to know only the base address of the device and the IRQ line that the device is using. As shown above all device id's or device class fields that cannot be resolved within the linux PCI-subsystem are printed as plain values, so driver writers have a simple chance to identify their card of choice without writing a special tool to do this task.

The Device Configuration Block

As mentioned above each PCI-device has its own assigned 256bytes PCI-configuration memory block that is accessible through PCI-BIOS routines. The first 64bytes of this reserved space are used to identify and configure the basic properties for the PCI-Device. On system startup some of the parameters in the configuration block are replaced from the Plug&Play BIOS that configures the I/O or Memory base addresses and IRQ lines for the PCI-board to operate correctly with other cards in the PCI or ISA slots. The configuration blocks of the PCI subsystem can be accessed through the `CONFIG_ADDRESS`

(0xcf8) and the CONFIG_DATA (0xcfc) registers
The Linux PCI-bios support is implemented on top of the Bios32 routines and defines some useful routines to handle the PCI configuration block and configure the PCI subsystem. Normally no changes in the configuration block are necessary because the BIOS sets up all parameters to operate correctly. Because Linux is intended to run on many hardware architectures it is recommended to use this functions instead of accessing the configuration registers directly.
Before a PCI-board can be used, the driver has to determine the board specific parameters from the configuration block and set all desired options to operate the board correctly. Normally this task is done by a autodetect routine in the kernel driver that performs the following tasks:

- Check if the PCI-Bios Support is enabled in the kernel and the BIOS is present with `pcibios_present()`
- Get the configuration block for the desired device using `pcibios_find_device()`
- Get the desired parameters from the configuration block using `pcibios_read_config_byte()`, `pcibios_read_config_word()` or `pcibios_read_config_dword()` to get 8,16 or 32bit parameter values.
- Set the desired parameters in the configuration block using `pcibios_write_config_byte()`, `pcibios_write_config_word()` or `pcibios_write_config_dword()`

The following example from the ne2000 clone driver shows how this task is performed in a real kernel network driver.

```
#if defined(CONFIG_PCI)
    if (pcibios_present()) {
        int pci_index;
        for (pci_index = 0; pci_index < 8; pci_index++) {
              unsigned char pci_bus, pci_device_fn;
            unsigned int pci_ioaddr;

            /* Currently only Realtek are making PCI ne2k clones. */
            if (pcibios_find_device (PCI_VENDOR_ID_REALTEK,
                        PCI_DEVICE_ID_REALTEK_8029, pci_index,
                        &pci_bus, &pci_device_fn) != 0)
                  break;      /* OK, now try to probe for std. ISA card */

            pcibios_read_config_byte(pci_bus, pci_device_fn,
                        PCI_INTERRUPT_LINE, &pci_irq_line);
            pcibios_read_config_dword(pci_bus, pci_device_fn,
                        PCI_BASE_ADDRESS_0, &pci_ioaddr);

            /* Strip the I/O address out of the returned value */
            pci_ioaddr &= PCI_BASE_ADDRESS_IO_MASK;

            /* Avoid already found cards from previous ne_probe() calls */
            if (check_region(pci_ioaddr, NE_IO_EXTENT))
                  continue;

            printk("ne.c: PCI BIOS reports ne2000 clone at i/o %#x, irq %d.\n",
                        pci_ioaddr, pci_irq_line);

            if (ne_probe1(dev, pci_ioaddr) != 0) {      /* Shouldn't happen. */
                  printk(KERN_ERR "ne.c: Probe of PCI card at %#x failed.\n", pci_ioaddr);
                  break;      /* Hrmm, try to probe for ISA card... */
            }
```

```
            pci_irq_line = 0;
            return 0;
        }
    }
#endif  /* defined(CONFIG_PCI) */
```

This card uses one IRQ and one base address in I/O space to communicate with the driver. The lower bits of the returned base address have different meanings depending if the address is an I/O address or a memory mapped area is returned. On an I/O address bit 0 is always 1 and bit 2 is always zero. On a memory mapped address type the lower 4 bits have the following meanings:

Bit	Description
0	always zero
1-2	adress type: 00=arbitrary 32 bit, 01=below 1M, 10=arbitrary 64-bit
3	prefetchable

To keep things simple Linux defines two macros `PCI_BASE_ADDRESS_MEM_MASK` and `PCI_BASE_ADDRESS_IO_MASK` that can be used to strip the type bits from the returned address. The prefetch bit is set if the PCI-bridge is allowed to read the memory block in a prefetch buffer without problems, this is normally the case if the memory block is a memory area on the PCI-device, if the memory area maps hardware registers into memory the prefetch bit should not be set.

There are a whole bunch of other setable and readable parameters in the configuration block, that can be accessed through the linux `pcibios_XXX()` routines. Linux defines some useful Mnemonics to gain simple access to the configuration block parameter values that are defined in `include/linux/pci.h`. The Tables 1,2 and 3 will describe the Macros in detail.

Reading and writing to a device

Reading and writing to a PCI device is as easy as reading and writing to an ISA board or reading and writing to a memory area depending on the base address type of the device.

If the device uses an I/O type base address, normal I/O can be performed using the usual `inb()/outb()`, `inw()/outw()` or the 32bit `inl()/outl()` routines. To get an impression how other linux drivers use this access modes please take a look through the network or scsi drivers.

If the device uses a memory mapped area it is recommended to use the `readb()/writeb()`, `readw()/writew()` or the longword `readl()/writel()` routines to read or write to a single location (see include/asm/io.h). `memcpy()` can also be used to transfer a whole memory block as this is the case on framegrabber devices or whatever transfers large blocks of data. In this case the PCI-Bridge handles automatically which transfer mode is optimal for the bus system. For this purpose the PCI-bridge has internal prefetch queues (one for the Bus side, one for the CPU subsystem side) that are used to store transfer data before a send or receive is initiated. If the PREF (prefetchable) bit is set in the base address bit 3, the memory area is non prefetchable, in other words data and addressed are dumped directly to the bus-subsystem. This mode is intended for devices that maps special registers in a memory area so the CPU can read or write this registers without a PCI-Bridge related delay.

Busmastering

With busmaster transfer mode a controller can transfer its data to the memory without any CPU action by driving the appropriate bus-lines on a hardware basis. This is similar to the DMA transfer on a ISA bus with the only difference that the busmaster-controller sits on the PCI-adapter card and not on the motherboard.

If a device is busmaster capable the `PCI_COMMAND_MASTER` bit should have been set in the PCI_COMMAND register. If a busmaster transfer has been aborted by the busmaster device, the PCI_STATUS_REC_MASTER_ABORT bit in the PCI_STATUS register is set, on the receiving end an abort can be notified by looking on the PCI_STATUS_REC_TARGET_ABORT bit.

Macro	Width	Description
PCI_VENDOR_ID	16bit	Unique Vendor ID (see pci.h) This ID is defined by the PCI consortium
PCI_DEVICE_ID	16bit	Unique Device ID (see pci.h) this is defined by the vendor unique for each device
PCI_COMMAND	16bit	This field is used for device specific configuration commands (see table 2)
PCI_STATUS	16bit	This is used for board specific status results (see table 3)
PCI_CLASS_REVISION	32bit	The high 24bits are used to determine the device's class type the low 8bit for revision code
PCI_CLASS_DEVICE		The device's class code (see pci.h)
PCI_BIST	8bit	If PCI_BIST_CAPABLE is set the device can perform a Build-in self test that can be started by writing PCI_BIST_START to this field. The result mask is PCI_BIST_CODE_MASK
PCI_HEADER_TYPE	8bit	Specifies the Layout type for the following 48 bytes in PCI configuration (currently only 0x0)
PCI_LATENCY_TIMER	8bit	This is the maximal time a PCI cycle may consume (time=latency+8cycles)
PCI_CACHE_LINE_SIZE	8bit	Specifies the Cache-Line Size in units of 32bytes,
PCI_BASE_ADDRESS_[0-5]	32bit	This are up to 6 memory locations the device can map its memory area(s) to. The lower 4 bits are used to specify the type and the access mode of the memory area.
PCI_ROM_ADDRESS	32bit	bit 11-31 is the start address of the device's ROM area. (write bit 1 to enable ROM)
PCI_MIN_GNT	8bit	minimal latency time (vendor specific)
PCI_MIN_LAT	8bit	maximal latency time (vendor specific)
PCI_INTERRUPT_PIN	8bit	This entry denotes the IRQ pin that should be used 1=INTA 2=INTB 0=disabled
PCI_INTERRUPT_LINE	8bit	This entry specifies the interrupt line on which the device IRQ is mapped (usually IRQ 0-15)

Table 1: The Linux PCI-Configuration macros

Macro	Description
PCI_COMMAND_IO	Enable I/O area
PCI_COMMAND_MEMORY	Enable Memory area
PCI_COMMAND_MASTER	Enable Busmastering
PCI_COMMAND_SPECIAL	Enable response to special cycles
PCI_COMMAND_INVALIDATE	Use memory write and invalidate
PCI_COMMAND_VGA_PALETTE	Enable video palette access
PCI_COMMAND_PARITY	Enable parity checking
PCI_COMMAND_WAIT	Enable address/data stepping
PCI_COMMAND_SERR	Enable SERR
PCI_COMMAND_FAST_BACK	Enable back-to-back writes

Table 2: The Linux PCI-Command Bit-Settings

Macro	Description
PCI_STATUS_66MHZ	Support 66 Mhz PCI 2.1 bus
PCI_STATUS_UDF	Support User Definable Features
PCI_STATUS_FAST_BACK	Accept fast-back to back
PCI_STATUS_PARITY	Detected parity error
PCI_STATUS_DEVSEL_[MASK|FAST|MEDIUM|SLOW]	DEVSEL timing
PCI_STATUS_SIG_TARGET_ABORT	Set on target abort
PCI_STATUS_REC_TARGET_ABORT	Master ack of
PCI_STATUS_REC_MASTER_ABORT	Set on master abort
PCI_STATUS_SIG_SYSTEM_ERROR	Set when we drive SERR
PCI_STATUS_DETECTED_PARITY	Set on parity error

Table 3: The Linux PCI-Status Bit-Settings