



How to design a Linux device driver for a CompactPCI WAN adapter board

By Steve Schefter

Linux is rapidly becoming an operating system of choice for use on CompactPCI systems. In this article Steve discusses the relative advantages and disadvantages of intelligent vs. non-intelligent WAN adapter boards, taking into account the software and performance considerations. He then describes what is required to create a device driver for Linux, using a case study involving a CompactPCI WAN adapter board.

The Linux operating system is steadily increasing in popularity. Linux source code is freely available for download from the web, and can also be obtained through several companies, including Red Hat, Caldera, and SuSE.

Because Linux is freely available, exact numbers of users of the operating system are difficult to determine. Red Hat (the leading distributor of Linux) estimates that the number of Linux users is somewhere between 5 million and 10.5 million.

Linux has jumped from being the seventh most commonly installed version of UNIX (in DataPro's 1997 survey sample) to being the fourth most commonly installed version of UNIX, just 12 months later. According to Datapro, Linux trails only Solaris, HP/UX, and IBM's AIX in worldwide usage. In Germany, Brazil, and Australia Linux is being installed in large enterprises as often as (if not more often than) *any* other UNIX-type operating system.

At the same time there are many embedded software developers and system engineers that are questioning whether Linux is suitable for mission-critical applications, and also whether it is suitable for use in CompactPCI systems. The answer to both of these questions is "yes" because Linux provides system developers with a great deal of control over their computing environments.

Linux - Use the source!

Perhaps one of the best features of the Linux operating system is that it is available with *source code*. The Linux kernel source code is freely available from several websites. Source code availability eliminates the problems that device driver developers encounter when they call the routines and functions in "closed source" operating systems, only to discover that these routines:

- do not work as advertised
- have unanticipated side effects
- are simply poorly documented

By examining the Linux source code an engineer writing a device driver can find the answers to many questions that would need to be posed to the provider of a "closed source" operating system - thus avoiding the wait (and the project delay) that often occurs while waiting for answers.H

With closed source operating systems, functions provided by the operating system are described as *black boxes*. If the device driver writer is on a tight schedule and cannot wait for answers from the provider, he or she might be forced to disassemble routines just to obtain the vital information. However, with the Linux source code a developer can even temporarily pepper the kernel routines with print routines in order to obtain detailed troubleshooting information.

In addition to the Linux source code available on the web, Linux can also be obtained as a binary software distribution. Each of the following companies' distribution has its own unique feature sets, with some features geared towards specific types of computer systems.

- Red Hat is a leading North American distribution of Linux.
- SuSE is very popular in Europe.
- Slackware was one of the first Linux distributions, and is the most "Unix like" Linux distribution.

The Software Group Limited has developed Linux device drivers for its PCI WAN adapter board and for its CompactPCI WAN adapter board - the SG4C, as shown in Figure 1. These drivers have been optimized for inclusion in Red Hat, Slackware, and SuSE binary distributions.

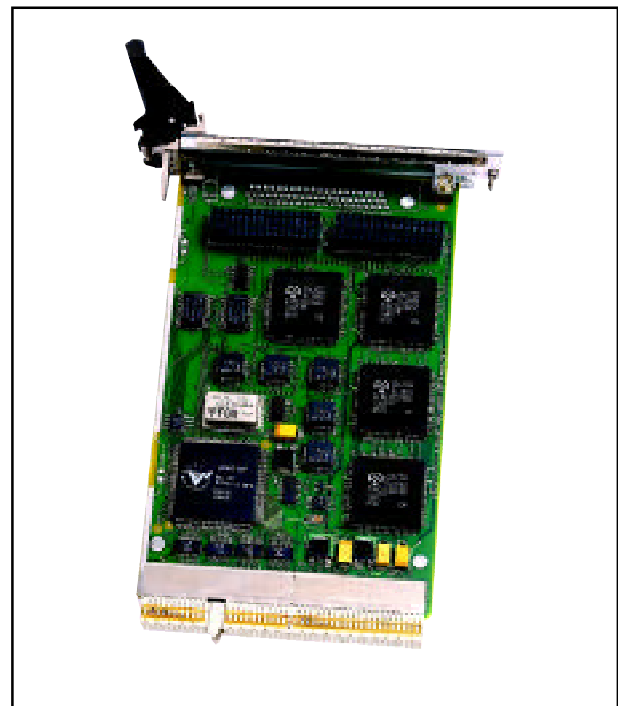


Figure 1



Intelligent WAN adapter boards

Should WAN adapter boards include an onboard processor? Historically, designers have provided an onboard processor to relieve the system CPU of two requirements:

- quick response to interrupts from the serial controller
- execution of the communications protocol layer software

The serial chip on an intelligent WAN adapter board is typically equipped with a DMA controller that writes the incoming data stream into onboard memory. The onboard processor does the protocol processing on this data stream and then interrupts the system CPU, which copies the processed data from the adapter board's onboard memory into system memory.

Can we do without the onboard processor?

Today's serial controller chips can handle their own time-critical processing, making it unnecessary to interrupt a processor. At the same time, the use of TCP/IP as a standard protocol, and the use of high-quality (relatively error-free) WAN data links have made it unnecessary to run sophisticated (and complex) error checking protocols over WANs. Both Frame Relay and PPP impose very little processing overhead, allowing engineers to design non-intelligent serial adapter boards with no reduction in functionality.

Bus master WAN adapter boards

However, using the system CPU to copy the incoming data from the adapter board's local memory to system memory imposes a load on it. Ideally, the adapter board should be able to transfer the incoming data stream directly from the wire to the system memory – eliminating the need for the system CPU to copy the data.

Similarly, the adapter board should be able to read the outgoing data stream directly from system memory and send it out over the wire. Our tests have shown that the use of a CompactPCI bus master adapter board increases the maximum serial transfer rate by about 30% over the use of a slave board.

Linux and memory management

An I/O device driver (which executes on the system CPU) accesses the system memory in a Linux-based system using *virtual* addresses. However, since the memory management unit (MMU) is between the system CPU and the CompactPCI backplane, it does not translate addresses generated by a WAN adapter board. As a result, a WAN adapter board accesses system memory directly, using *physical* addresses.

Consequently, whenever the device driver passes the address of a Linux kernel data structure (such as a data buffer) to the WAN adapter board, it must first convert the virtual address of the data structure to a physical address. Linux provides a special routine to do this:

```
physical_address = virt_to_bus(virtual_address)
```

Another problem often arises when using an MMU to do virtual-to-physical address translation. Suppose the device driver writes an 8-Kbyte string, byte-by-byte into contiguous memory. Obviously, each byte will be stored in the next monotonically increasing address – as seen by the CPU. However, if the mapping of virtual addresses to physical addresses is discontinuous (because of the MMU) there will be no guarantee that every sequential byte written into virtual memory will also be stored sequentially into the *physical* address space – there might be a jump in the address at each page boundary.

Linux solves this problem by providing a special option to the memory allocation routine that ensures the memory is contiguous in both virtual and physical spaces:

```
kmalloc(size_in_bytes, GFP_KERNEL | GFP_DMA)
```

`GFP_KERNEL` indicates a high-priority request, and `GFP_DMA` indicates that the memory needs to be physically contiguous so that it can be accessed by DMA and bus master devices.

The initialization routine

A device driver must initialize its adapter board and then configure the board to be compatible with the Linux kernel, for later use. This operation can be divided down into three basic tasks:

- accessing the configuration of each board
- declaring the board's resources to Linux
- setting up the back-end hardware

As the machine boots

Before the Linux kernel is loaded, the system BIOS queries each board (using slot-specific accesses) to determine its requirements. Based on these requirements, the BIOS then configures the PCI bridge chip on each board by writing parameter values into that chip's registers. This configuration involves the assignment of the board's:

- memory addresses
- I/O addresses
- interrupts

Note: Since CompactPCI boards might be installed or removed from the backplane while the system is powered down, the configuration of each board might be different when the machine is then rebooted. This makes it essential that the device drivers in the system *not* store board configuration information statically, such as in a file. Each driver must read the configuration parameters for its board each time the system is reinitialized.

Accessing the configuration information

There are three address spaces in a CompactPCI bus environment:

- memory space
- I/O space
- configuration space

On Intel-compatible processors, there are machine-level instructions to access only two of these:

- the memory space
- the I/O space

When the system CPU executes one of these machine-level instructions, the CPU signals the cycle type to the PCI bridge chip that connects the onboard PCI bus to the CompactPCI backplane. This bridge chip then signals the cycle type over the CompactPCI bus.

Because Intel-compatible processor chips have no machine-level instruction for accessing *configuration space*, the BIOS provides a special function call that configures the PCI bridge chip to signal a configuration-space access, as the BIOS function generates an access cycle over the CompactPCI bus.

Using the Vendor ID and the Device ID

The PCI Special Interest Group (PCI SIG) assigns a unique Vendor ID to each manufacturer. A device driver can query Linux

for all boards installed in the system having a specific Vendor ID, and a specific manufacturer-assigned Device ID. For example, the device driver might query Linux to find out whether an SG4C board manufactured by The Software Group is resident in the system as follows:

```
pcibios_find_device(TSGVENDORID, SG4CDEVICEID,
boardnum, &bus, &function)
```

Drivers that are capable of handling multiple boards can make repeated calls to this function, incrementing *boardnum* from zero, until the function no longer returns PCIBIOS_SUCCESSFUL.

The returned values of bus and function are unique to each board in the system. These values must be retained by the driver in order to perform subsequent reads from (and writes to) each board's configuration space.

Using these values, locations in the board's configuration space can be accessed as a byte, word (16 bits), or double word (32 bits) as follows:

```
■ pcibios_read_config_byte(bus, function, addr, &val)
■ pcibios_read_config_word(bus, function, addr, &val)
■ pcibios_read_config_dword(bus, function, addr, &val)
■ pcibios_write_config_byte(bus, function, addr, val)
■ pcibios_write_config_word(bus, function, addr, val)
■ pcibios_write_config_dword(bus, function, addr, val)
```

Declaring interrupts and the I/O address ranges to Linux

Although the device driver uses the six Linux functions shown above to read an adapter board's configuration, Linux itself does not try to "eavesdrop" on the resulting data transfers, in an attempt to "learn" about the interrupts and the I/O address range for each adapter board. (This was not attempted because Linux was written to run on ISA and other buses, where the BIOS cannot ensure non-conflicting interrupts and I/O address assignments.)

Therefore, when the initialization routine in each device driver is called upon start-up, that device driver must determine the resources required for each of its adapter boards and then *declare* these to Linux.

How the driver checks its adapter board's I/O range

To verify that its adapter board's I/O range does not conflict with any other adapter board, the driver can call the Linux function:

```
check_region(IOStart, SizeInBytes)
```

where *IOStart* and *SizeInBytes* indicate the I/O address range of the driver's own adapter board.

The driver finds out the base address (*IOStart*) of its adapter board by calling one of the *pcibios_read_...* functions, which were mentioned above. These functions read registers inside the PCI bridge chip on the adapter board, which will have been configured by the BIOS at the system boot-up time.

For CompactPCI systems (which allow the BIOS to configure all boards during power-up) this should always return 0, indicating that no other device driver has reserved this same I/O range.

How the driver declares its adapter board's I/O range

Since Linux does not know what range of I/O space the BIOS assigned to each adapter board during bootup, each device driver

must *declare* the base address and the size of I/O space occupied by its adapter board. It does so by "requesting" that region of I/O space. This request is accomplished by calling the Linux function:

```
region_request(IOStart, SizeInBytes, "SG4C")
```

Note: This *dynamic* method for allocating the I/O space (which allocates all of the I/O space from *scratch*, each time the system is booted up) is different from the method used in other operating systems. Many non-Linux operating systems have a *static database file* that lists all of the I/O address ranges that have been allocated to I/O devices at some time in the past. When installing a new I/O adapter board into the system, the new driver's *install script* queries this static database file to see if there is an address conflict between some I/O device already in the system and the address where the install script wants to put its own I/O adapter board. Assuming there is no conflict, the install script then adds the new I/O address range to this database file. The difference here is that the database is *static*, and is filled in by the install script once, instead of being built each time the system boots by each driver's initialization routine.

Since Linux does not use a static database, each device driver must redeclare (to Linux) the range of I/O space that its adapter board is using each and every time the machine boots. As each driver does this, an entry is added to the file */proc/ioports*, allowing the system administrator to see what areas of the I/O space have been assigned by the BIOS.

Note: The file */proc/ioports* is *not* a static database file. It is *rebuilt* each time the machine boots, as each driver calls the Linux *region_request()* function.

How the BIOS configures system interrupts at boot-up

PCI and CompactPCI interrupt request lines are active-low, open-drain, level-triggered signals. To interrupt the CPU, an I/O device drives the interrupt line low. When there are no devices holding the interrupt line low, a pull-up resistor holds it high. This allows boards to share an interrupt request line.

The process by which the BIOS assigns system interrupts is very BIOS-specific. Some allow the user to specify which interrupt will be used on a *per-slot* basis. Others allow the user to specify, for each *interrupt number*, whether it should be used for PCI devices or legacy (ISA) devices. The BIOS then puts all of the interrupt numbers for PCI devices into a *pool* of numbers. If that pool is not large enough to assign a unique interrupt number to each PCI device the BIOS will reuse some numbers, and *interrupt sharing* will result. In general, the BIOS makes no determination of board types when assigning shared interrupts. An I/O board might share an interrupt with another board of the same type, or with a board of a completely different type.

How the driver declares the interrupt resources used by its adapter board

A driver declares the particular interrupt used by its adapter board by "requesting" that interrupt using:

```
request_irq(IRQnum, SG4Cintr, SA_SHIRQ, "SG4C",
deviceID)
```

SG4Cintr is the name of the interrupt handler routine for the SG4C adapter board.

SA_SHIRQ is a *flag* that declares that the driver is willing to share interrupts.

Note: Other flags are defined in */usr/include/asm/signal.h*.



As well as reserving the interrupt, `request_irq()` lets Linux know which interrupt service routine to call when an adapter board generates an interrupt.

If the driver indicates that it is willing to share interrupts, then it must provide a unique `deviceID`. Hard coding this `deviceID` into the device driver (or just picking a `deviceID` at random) might cause problems. A better approach for choosing a unique number is to use the address of some data structure that is used only by the driver.

If a driver calls `request_irq()` indicating that it is *unwilling* to share its interrupt number with some other driver, and if that interrupt number has already been requested by another driver, the call will return the error `EBUSY`. [`EBUSY` is also returned if the *other* device driver indicated that it was unwilling to share the interrupt when it previously called `request_irq()`.]

Note: When this happens it is typically *not possible* to reconfigure the interrupt numbers to resolve this conflict. The interrupt numbers are assigned by the BIOS at boot-up, and there is no mechanism to repeat the system configuration process once the boot-up process completes.

If `request_irq()` returns a 0, then the driver has been granted its request for the interrupt indicated by `IRQnum`. The driver must then *unmask* the interrupt by calling:

```
enable_irq(IRQnum)
```

The memory space occupied by the adapter boards

As described above, the device driver “declares” the interrupt numbers and the I/O addresses of its board by sending “requests” to Linux. However, Linux provides no mechanism that allows a driver to declare the memory space occupied by its adapter board.

This means that Linux provides no bookkeeping to ensure that memory overlaps do not occur, and generates no `/proc/mem` file to allow the system administrator to see what areas of the memory space have been assigned.

Fortunately, in CompactPCI systems the PCI BIOS ensures exclusive memory assignments so no bookkeeping is needed. However, in ISA machines (where the 640K to 1M region can be very crowded, and where boards are often configured in hardware with static jumpers) this is a serious problem.

Virtual addresses

Since the device driver accesses the memory map with virtual addresses, it must be provided with some way to convert the physical address of its adapter board to a virtual address in order to access that board. As on other operating systems, Linux provides a routine to convert the base address of the adapter board to a virtual address:

```
vremap(physical_address)
```

Note: `vremap()` has been renamed `ioremap()` in the recently released Linux version 2.2.

However, there are two important restrictions when calling `vremap()`.

First, `vremap()` cannot be used to convert low physical addresses, such as memory-mapped ISA devices in the 640K to 1M region. The `vremap()` routine defines “low physical

addresses” as those addresses below the top of system RAM, which is determined dynamically by the kernel. (This restriction simplified the code of the `vremap()` routine.) Fortunately, this is no problem for CompactPCI systems, where the PCI BIOS generally maps the devices up near the top of 4-Gbyte memory space.

Second, the physical address passed to `vremap()` must be on a *page boundary*, which is generally a 4-Kbyte boundary. If the PCI bridge chip on the adapter board requests some multiple of 4 Kbytes for the adapter board (such as 4 Kbytes, 8 Kbytes or 16 Kbytes) the BIOS will place the adapter board on a 4-Kbyte boundary. However, if the bridge chip requests some other size, the BIOS will place the board on a multiple of whatever size the PCI bridge chip requests. For example, if the bridge chip asks for 2 Kbytes, the board will be placed on a 2-Kbyte boundary, which might or might not be a 4-Kbyte boundary. Since `vremap()` only accepts physical addresses on page boundaries, the driver will need to:

- *compute the offset* to the previous 4-Kbyte boundary
- pass the physical address of the previous 4-Kbyte boundary to `vremap()`
- *add the previously computed offset* to the virtual address returned by `vremap()` each and every time the board is accessed

Initializing the back-end hardware

Typically, the driver will need to configure some onboard chips, in addition to the PCI bridge chip. As described above, the PCI BIOS will assign a base I/O address and/or a base memory address to the adapter board, to allow the device driver to access it.

However, just because the BIOS has *assigned* an I/O base address and a memory base address, this does not guarantee that the resources on the adapter board can be accessed. In fact, the adapter board should be designed to *disable* access to onboard resources until the device driver’s initialization routine enables them, as the system boots. This is necessary because (for unknown reasons) some CompactPCI host systems fail to complete their power-on self-test if the memory on the adapter board is enabled. (One possible explanation might be that memory walking during the boot-up process overwrites memory-mapped registers in the I/O devices. This could cause an I/O device to do something unexpected, such as generating an interrupt before the interrupt handlers have been set up.) Also, letting the host system have access to the components on an uninitialized board can be a bit dangerous.

As mentioned earlier, the device driver accesses the memory on the adapter board (whether it is RAM or some memory-mapped register) using a *virtual* address. However, I/O-mapped registers are located in I/O space, which has no distinction between virtual and physical addresses. The Linux kernel provides the following routines that allow the caller to read bytes, words, or longwords directly from the I/O space.

- `inb(addr)`
- `inw(addr)`
- `inl(addr)`
- `outb(val, addr)`
- `outw(val, addr)`
- `outl(val, addr)`

Caution: If you use any of these `out-()` routines, watch out for the *order* of the two parameters. They are *backwards* compared to the order found on most operating systems!

Running the initialization routine

So far this article has described some of the things that the initialization routine in a device driver must do. But how does the `init()` routine for each device driver get executed in the first place?

When you have the source code for the operating system this is easy. You just edit some kernel C file to insert a direct call that gets executed at kernel boot time. For example, you might insert a call within the function `start_kernel()` which is found in `/usr/src/linux/init/main.c`

Other entry points

In addition to the initialization routine, device drivers under Linux have a set of entry routines similar to those found in many other operating systems. These include:

- `open()`
- `close()`
- `read()`
- `write()`
- `ioctl()`

In order for the device driver to be useful, there has to be some way for the application program to call upon these routines. That is, there must be some way of mapping a function call from application space (where the application program runs) to a driver routine in kernel space (where the driver runs).

This mapping is done using the contents of the `/dev` directory. Each entry in the `/dev` directory has a major and a minor number, which are visible if you do a long listing (`ls -l`) of the device.

The kernel keeps a table of *major numbers* and their associated drivers. If an application opens a device with major number 50, the kernel calls the `open()` routine within the device driver that corresponds to major number 50.

So how does the kernel build this table? Each device driver registers itself, usually in its `init()` routine by calling:

```
register_chrdev(SG4CMAJOR, "sg4c",
sg4c_file_operations)
```

...where `SG4CMAJOR` is an integer indicating the major number that the driver wants assigned to itself.

The middle string (`sg4c`) in this list of three parameters is an arbitrary name to associate with the major number. The structure `sg4c_file_operations` is a structure (defined in `/usr/include/linux/fs.h`) that includes pointers to each of the I/O functions provided by the driver: `read()`, `write()`...

Note that a driver need not implement all of the five functions. It might provide NULL pointers for the functions that are not needed, or it might provide a pointer to a routine that returns an *error* for those functions that should never be called.

Caution: Be careful about what number you choose for your device driver's major number. If your driver attempts to register its functions with a major number that has already been taken, its registration will fail. There are some predefined major numbers that are used by common drivers found on Linux systems. These are listed in `/usr/src/linux/Documentation/devices.txt`. Also the major numbers used on a particular system can be displayed by running: `cat /proc/devices`

A safer approach is to pass `register_chrdev()` a major number of 0. This function will then return an unused major number.

However, your application will then need to get this major number from your device driver and then create an entry in `/dev` with this dynamically assigned major number.

Linking into the Linux kernel

Using the above information, a device driver can:

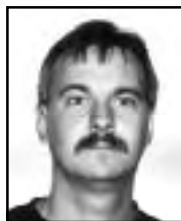
- set up the hardware
- provide all of the I/O functions that an application needs
- handle interrupts generated by the board

All that is left is to make the driver's binary part of the kernel. Binaries for drivers are normally held in a subdirectory off of `/usr/src/linux/drivers`. Once you have created a subdirectory for your drivers, and have put their binary files into that subdirectory, you will also need to add the name of this new subdirectory to the `SUB_DIRS` variable in `/usr/src/linux/drivers/Makefile`. This file also includes samples of how to add binary modules to Linux, instead of having the driver compiled into the kernel.

Finally, the driver object code itself is usually held in an archive – a collection of object modules. Again, this will be within the subdirectory off of `/usr/src/linux/drivers`. You will also need to add the name of the archive to the variable `DRIVERS` in `/usr/src/linux/Makefile`

Summary

This article has covered the basics of creating Linux device drivers for a CompactPCI WAN bus-master board. The use of Linux for WAN applications provides a very cost-effective alternative to standard commercial operating systems. Because the source code for Linux is readily available, it's also easier to troubleshoot the system. And, best of all, there are numerous references and a lot of information on the web to assist developers who are working with Linux. Ω



Steve Schefter is the Engineering Manager at The Software Group Limited. He has been with the company for nine years since graduating from Systems Design Engineering at the University of Waterloo. Since joining TSG, Steve has designed hardware and written device and protocol drivers for many distributions of UNIX. Favorite out-of-office activities include squash, canoe camping, and snowboarding.

For questions about this article or more information about The Software Group's PCI or CompactPCI SG4C WAN adapter boards, or their Linux, UNIX, or WindowsNT device drivers, you can contact Steve at the address shown below.

The Software Group Limited
642 Welham Road
Barrie, Ontario, Canada
L4N 9A1
Phone: 705-725-9999
Phone: 888-WANWARE
Fax: 705-725-9666
Email: sales@wanware.com
Website: www.wanware.com