1 INTRODUCTION

This tutorial presents the author's practical experience with writing Linux device drivers to control custom-designed hardware. The tutorial starts by providing an overview of the driver writing process, and describes several example drivers provided with this tutorial [4].

The reader is encouraged to experiment with those example drivers on their own x86 system, as it provides the best learning experience. Once the basics of device driver writing are covered, the tutorial then describes how to achieve optimal performance from the PCI bus. The ability of a user-space process to transfer data from multiple PCI boards is contingent on the implementation of both the hardware and driver. The requirements of both the hardware and software are presented.

The drivers in this tutorial are written for the Linux 2.6 kernel. The drivers have been built against: 2.6.9-11 (Centos 4.1), 2.6.13, and 2.6.14 for x86 and PowerPC targets. Details that are clearly described in the book 'Linux Device Drivers' [1], by Corbet, Rubini, and Kroah-Hartman are not repeated in this tutorial, so the reader is encouraged to obtain a copy.

2 THE BASICS

The Linux 2.6 kernel presents a number of generalized interfaces that the driver writer must first understand, and then implement for their specific driver. The best way to understand the interfaces is to write simple drivers that exercise a subset of the kernel driver interfaces. The following sections describe the interfaces used to implement character device drivers.

2.1 Kernel modules

The file simple_module.c implements a very basic kernel module. A device driver is a kernel module, but kernel modules are also used to add features to the kernel that have
nothing to do with device drivers. Welcome to your first generalized kernel interface.

The basic requirements of a kernel module are that they implement an initialization and an exit function. Those two functions are identified by the macros `module_init()` and `module_exit()`. The example also shows how to pass load-time parameters to the module, and how to setup logging in a module. The code sets up two logging macros; `LOG_ERROR()` and `LOG_DEBUG()`. The debug macro can be removed from the code at compile time (by not defining `DEBUG`), or can be compiled into the code and then enabled or disabled via the load-time parameter `simple_debug`. This method of adding log messages to code is easier to maintain (eg. disable) than a series of `printk()` calls littered throughout the code.

The following shows the driver usage; the // marks are comments, while the $ (user) and # (root) prompts show the commands you enter (bash shell syntax).

```
// build the driver
$ make
// add the driver tools to our path, eg. modinfo, insmod
$ export PATH=$PATH:/sbin:/usr/sbin
// login as root
$ su
// enter root password
// see what the driver does
# modinfo simple_module.ko
// clear the kernel message buffer
# dmesg -c
// install the module
# insmod simple_module.ko
// list all the installed modules
# lsmod
// remove the module
# rmmod simple_module
// view the log messages
# dmesg -c
simple_module: Module loaded.
simple_module: This is a debug message.
simple_module: Module unloaded.
// repeat, but disable LOG_DEBUG
# insmod simple_module.ko simple_debug=0
# rmmod simple_module
# dmesg -c
simple_module: Module loaded.
simple_module: Module unloaded.
```

So with the load-time parameter `simple_debug` set to zero, the `LOG_DEBUG()` message does not appear in the output. The module load and unload messages are generated using the `LOG_ERROR()` macro so that they are always generated.
2.2 Device drivers

The file *simple_driver.c* implements a simple device driver. What makes it a device
driver, and not just a kernel module? In *simple_init* the driver requests a range of major
and minor numbers (the numbers used to represent device nodes in /dev), it then allocates
memory for an array of device-specific *simple_device_t* structures, and then registers
the character device, *cdev*, member of each structure in the array with the kernel. Registration
of the character device requires a set of file operations, i.e., a kernel-level implementation
of the functions that get called when user-space calls system calls, eg. *open()*, *read()*,
*write()*, *ioctl()*, *lseek()*, *select()*, and *mmap()*. The file operations are stored as
function-pointers in a *struct file_operations*: if this code was written in C++, then
this structure would be the base-class, and your implementation of its functions would be
an inherited class.

The file *simple_driver_test.c* is a user-space application that tests the functions of
the driver. Install the module, type *ls /dev/simple* and once you see device nodes there,
run the test. After the test finishes, type *dmesg* to see the kernel-level messages triggered
by the user-space test.

Remove the driver, and reinstall it with load-time parameters, eg.

```
# insmod simple_driver.ko simple_device_count=3 simple_minor_count=2
```

This creates three devices each responsible for three minor numbers (functions on the de-
vice). *ls -al /dev/simple* will show the multiple devices created (and their major/minor
numbers). The relevance of devices and sub-functions will be clearer when we discuss PCI
devices.

How did the device nodes magically appear in /dev? That's next.

2.3 Hotplug, sysfs, and udev

The simple driver initialization code, *simple_init*, also performs another step, it creates
a kernel object, *class_simple* or *class* depending on the kernel version, that creates entries
in the sys-filesystem, *sysfs*, in the directory /sys/class. Creation of the class object in
the initialization code, creates the entry /sys/class/simple_driver. Devices managed
by the driver are then added to the class object (see the code), creating the device nodes
under /sys/class/simple_driver, eg, if no load-time parameters are specified, the driver
creates one device, and the node /sys/class/simple_driver/simple_a0 is created.

Why create these class and device ‘objects’? The Linux 2.6 kernel supports the concept
of hot-pluggable devices, i.e., devices that can be plugged in while the system is turned
on, eg. a USB camera. In older Linux systems, if you plugged in a camera, you’d have
to look at the output of *dmesg* to see what the camera was detected as (if at all), and
then try and figure out how to get images off the camera! The Linux 2.6 system generates
‘hot-plug’ events every time a kernel object is created and destroyed, and these hotplug
events trigger the execution of scripts in user-space. The (appropriately written) scripts then automatically populate the /dev entries for a device. A nice feature of these scripts is that you can decide what name to give the device, eg., a camera detected as a USB mass-storage device might be detected as /dev/sda1 in a non-hotplug system, but with hotplug you can setup the camera name to be /dev/camera, much nicer!

The automatic creation of /dev entries relies on three related kernel infrastructures; hotplug, sysfs, and udev. The man page, man udev, gives details on how the scripts can be setup to create the /dev entries with specific permissions, and how to map a kernel name (eg. that used when the device was added to the class object in simple_init) to a user-space defined name. On Centos 4.1, the udev configuration files are kept in /etc/udev/, the line udev_log=no in in /etc/udev/udev.conf can be changed to udev_log=yes and hotplug events will be written to the system log. For example, as root type tail -f /var/log/messages, and then from another terminal install the simple_driver.ko, and you will see the logging of the hotplug events.

The default name given to a single device created by the simple driver is /dev/simple_a0. With no udev scripts in-place, the device node is created for use by root only, and is named identically to the string used in simple_init. The permissions on the device node can be changed by creating a udev script containing a single line;

```bash
# /etc/udev/permissions.d/20-simple.permissions
simple_*:dwh:mm:0660
```

This changes the permission on all nodes matching the pattern simple_* to the owner dwh, group mm, with permissions 0660. The name of the device entry can be changed, or a symbolic link to a device entry can be created, by adding another script, eg. the following creates a symbolic link to the first device entry

```bash
# /etc/udev/rules.d/20-simple.rules
KERNEL="simple_a0" SYMLINK="simple_00"
```

The udev man page gives more details on the options for device naming (eg. a user-supplied program can be run to generate the device name). The automatic creation of /dev entries helps reduce the contents of /dev to just those devices installed. It also provides flexibility to user-space in the naming of device nodes. For example, in the case of PCI devices it allows the PCI location, eg. bus:dev.fn to be remapped into a meaningful slot number, eg. instead of say a device named /dev/board_00:0c.0, the user-space name can be mapped to /dev/board2.

The class_simple interface, as described in the Linux Device Drivers book [1], was removed from the kernel in 2.6.13 (according to the ChangeLog for that kernel), and the API changed again slightly in 2.6.14. The parallel port user-space driver, ppdev.c, is a nice small (easily understandable) driver that uses the class interface. A diff of different kernel versions of this driver, can be used to determine the usage of any API changes (eg. whether a new argument can be assigned NULL).
2.4 Kernel timers

The driver `simple_timer.c` implements a single device that uses two different kernel mechanisms for delaying the calls `read()`, `write()`, and `select()`. The test program `simple_timer_test.c` tests the driver. The driver demonstrates the usage of timers and events.

2.5 Interrupts

The driver `simple_irq.c` implements a single device that uses the parallel port on an x86 PC. To test this driver, you might need to first remove the printer driver and parallel port driver, i.e., `modprobe -r lp, modprobe -r parport_pc`. The driver creates a kernel timer that fires every second. The timer handler writes a low and then high to all the data lines on the parallel port. If a data line, one of pins 2 through 9, is jumpered to the interrupt line, pin 10, then an IRQ will be generated every second. The IRQ handler unblocks a blocked `read()`, `write()`, or `select()`. If a data line is not jumpered to the IRQ line, then the blocked calls will timeout (2s) and continue anyway. The test program `simple_irq_test.c` tests the driver. The driver demonstrates the usage of timers, IRQs, and events with timeouts.

2.6 Data buffering

The driver `simple_buffer.c` implements a single device that also uses the parallel port on an x86 PC (so you will need to remove `simple_irq` to test it). This driver is similar to `simple_irq.c` with the change that IRQs write a time-stamp to an internal buffer, user-space `write()` writes to that buffer, and `read()` reads from the buffer. The following are some tests that can be performed using standard command-line tools:

1. Connect the parallel port IRQ to a data line. Install the driver;
   ```bash
   insmod simple_buffer.ko.
   ```
   Once the `/dev/simple` node is valid, type `cat /dev/simple`. A UTC timestamp will be printed every second.

2. Remove the parallel port jumper. Remove the driver. Install the driver and disable the timer and timeout;
   ```bash
   insmod simple_buffer.ko simple_timer_enable=0 simple_timeout_enable=0.
   ```
   On one terminal type `cat /dev/simple`, on another type
   ```bash
   echo "Hello" > /dev/simple".
   ```
   (You can also leave the timer enabled and it will just write messages to the log file).

3. Combine the first two tests (remove and re-install the driver without any load-time parameters); the IRQ will add a complete timestamp message every second, while write will add a complete string (whenever the user triggers a write). No messages will be interrupted, since each procedure locks the internal buffer.
The test shows that the driver works as one would expect, however, take a look at the source for the details. The internal buffer is a resource that is shared between `read()` (eg. one process), `write()` (eg. another process), and the IRQ handler (interrupt context). The driver uses a spin-lock to protect access to the buffer (and its associated buffer count and pointers). Without this protection, an IRQ could interrupt a write, and insert a timestamp into the middle of the string echoed into the driver. Of course in a real driver, the results could be more disastrous.

If the resource (buffer) being protected by the driver was only ever accessed by processes, then a semaphore can be used to protect it. Semaphores can be used to block a process, causing it to sleep while waiting for a resource. Spin-locks are not quite so forgiving. You are not allowed to sleep, or call a function that might sleep, while holding a spin-lock. Make sure to build your driver development kernel with `CONFIG_DEBUG_SPINLOCK` and `CONFIG_DEBUG_SPINLOCK_SLEEP` enabled, and the kernel will give you a nice reminder if you try to do something bad (eg. calling `kmalloc` while holding a lock).

The `write()` and `read()` operations of the driver need to copy data from (or to) user-space to (or from) a kernel buffer. However, a `copy_from/to_user` can sleep, so there is no way to copy directly to the spin-lock protected buffer! There’s also the following write sequencing issue; to write data into the buffer, you first need to check whether there is space, however, the spin-lock needs to be held to check the buffer state, so ideally you would hold the lock, check for space, release the lock, and then copy a matching amount of user-data to the kernel. But, since you are not holding the lock, an IRQ can come along and use up your space! The solution shown in the driver, is to first copy all the user data into a kernel buffer, and then hold the lock while checking for space. This allows the (sleepable) copy and allocation calls to be performed before holding the lock. Of course in the case of a full buffer and non-blocking write, the allocation and copy from user-space was a waste of time.

The code that holds the spin-lock, checks for a condition, and then goes to sleep on a wait-queue if the condition is not met, should look eerily familiar to anyone who has programmed with Pthreads; it is the same pattern of code as used with a mutex and condition variable. A mutex is used to protect a resource, while a condition variable is used to put a thread to sleep while waiting for some other thread to signal it that the condition has changed. The nice thing about this analogy is that you can write pthreads code to simulate driver buffering operations to ‘figure it out’ outside of the kernel.

The buffering used in the simple buffer driver is a bit contrived in that there are two ‘producers’ writing to the buffer, and one ‘consumer’. A more likely scenario for a driver would be to have a buffer contended for by a single producer (say the receive IRQ), and a single consumer (say read), and another separate buffer for a single producer (write) and consumer (transmit IRQ). But even in this situation, you can run into problems if the read from the buffer takes an excessive amount of time, blocking new data from the receive IRQ. One solution to this issue is to use two buffers for each producer-consumer pair; eg. the receive IRQ is initialized to point to an empty buffer, and receive IRQs fill the buffer until a read is issued, at that point IRQ buffer is passed to read, and the IRQ gets the second empty buffer. Once read has consumed the contents of the first buffer, if the second buffer in-use by the IRQ has new data, then the buffers are swapped again. In this scheme, the
lock only needs to be held to swap the buffers, and since read does not hold the lock once it has a valid buffer, a copy to user-space from the kernel buffer is allowed, removing the need to use an intermediate buffer as shown in the simple buffer driver. The kernel tty layer uses this form of buffering scheme and refers to it as flip-buffering (see `linux/tty.h`).

The simple buffer driver has (at least) two practical applications. If you install it and `cat` the timer generated timestamps into a file, a plot of the difference between consecutive timestamps minus 1 second, will show the error in the kernel’s ability to generate a 1 second delay. In a test on an HP Omnibook 6100 PIPII 1GHz laptop, the error was approximately -130µs (i.e., slightly less than 1 second). The test was started on a 1 second boundary, and over the space of 10 minutes, the timer was firing 100ms earlier than a 1 second boundary. The second test determines how good NTP operates. Install the driver with the timer and timeout disabled. Connect up the 1pps tick from your NTP server’s GPS unit to the parallel port interrupt of your PC, make sure your PC NTP daemon is running, and `cat` the IRQ generated timestamps. The observed error of the measured timestamp relative to that same timestamp rounded to the nearest second was about ±0.5ms. If the test PC (laptop) had its ethernet cable disconnected, or the NTP daemon was stopped, the error of the logged timestamps relative to the GPS 1pps tick would gradually increase (100 to 200µs over 10 minutes). If you had a method of generating a higher-frequency square-wave that was also locked to GPS, then you could determine the interrupt latency, and interrupt handling overhead, of the kernel by hammering the IRQ pin at a few kilohertz.

2.7 Summary

Writing a Linux kernel device driver consists of writing code that requests resources from the kernel, implements the generic interfaces presented by the kernel, and writing the custom code to control hardware. The interfaces to the kernel required to create even a simple driver can be overwhelming for a new driver writer. The simple drivers in this section should help illuminate some of the basics to make reading driver code in the kernel source, and the development of new drivers, a bit easier.

3 PCI DRIVERS

Peripheral Component Interconnect (PCI) is both an electrical standard and a software standard. The PCI familiar to most readers would be the cards you plug into your desktop PC. In industrial settings, a more reliable mechanical interface, and larger number of boards needs to be supported, and for that there is the compact PCI (cPCI) specification. The cPCI specification is based on the PCI specification, uses a different mechanical form factor, and its electrical specification allows for hot-swappable cPCI cards (the back-plane connectors have power pins of different lengths that connect in a particular sequence).
3.1 Kernel interface

Each PCI board in a system is configured at power-up by the BIOS (on an x86) or by Linux. If a board is hot-swapped into the system, then Linux has to configure it. Configuration of a PCI board consists of determining what I/O ports, or memory space, or IRQ the board wants, and then assigning an appropriate resource (e.g. non-overlapping I/O ports or memory, and most likely a shared IRQ). The Linux kernel maintains a list of PCI devices, with each device being represented as a struct pci_dev, and provides an API for accessing this structure. In older kernels, the kernel provided methods for walking along the list of PCI devices, so that the driver could find the device it was prepared to handle. The example driver pci_find.c demonstrates this deprecated technique (if the driver is loaded with PCI_ANY_ID in its device ID list, it will find all the PCI devices).

A PCI driver registers with the kernel a list of devices that it is prepared to handle, and a call-back function for when a device is added to the system, probe(), and another for when the device is removed, remove() (there are also power-management call-backs). The structure encapsulating the device list and function call-backs is the struct pci_driver, and the kernel registration functions are pci_register_driver() and pci_unregister_driver(). The example driver pci_probe.c demonstrates the PCI interface. If the list of devices used in the driver includes the PCI_ANY_ID device ID, then probe will be called on all unregistered PCI resources when the module is loaded. If the device ID list contains no IDs, then an ID can be added to the driver after its installed by echoing a deviceID:vendorID string into a /sys node, e.g. to add the ID 10b5:9054 to the installed driver’s ID list, the command is

```
# echo 10b5:9054 > /sys/bus/pci/drivers/pci_probe/new_id
```

A PCI driver needs to register with the PCI subsystem to be informed of the boards in a system, however, it also needs to create a character device interface (or network, etc). In fact, if the PCI board contains multiple functions, e.g. a serial port and a parallel port, it may register multiple character device drivers. The next section presents a simple PCI driver.

3.2 A generic PCI memory and I/O driver

The command lspci can be used to list the PCI devices in a system. The verbose option, -v, can be used to list more detailed information. For example, the ethernet controller on the HP Omnibook laptop gives the following;

```
# lspci -s 02:08.0 -v
02:08.0 Ethernet controller: Intel Corporation 82801CAM (ICH3)
PRO/100 VM (KM) Ethernet Controller (rev 41)
  Subsystem: Hewlett-Packard Company: Unknown device 001a
  Flags: bus master, medium devsel, latency 66, IRQ 10
```
Memory at d0200000 (32-bit, non-prefetchable) [size=4K]
I/O ports at 3440 [size=64]
Capabilities: [dc] Power Management version 2

`lspci` displays the configuration space header of the PCI device (`-x` will give it in hex). The resources of interest, are the 4K-bytes of non-prefetchable memory at physical address 0xD0200000, the 64-bytes of I/O ports at I/O address 0x3440, and the IRQ 10. Using `cat` on the `/proc/iomem` and `/proc/ioports` folders shows that the e100 driver has claimed the memory and I/O resources for this device, and `cat /proc/interrupts` shows the eth0 interface has claimed the IRQ. The kernel provides PCI resource accessors and memory, I/O port, and IRQ registration functions that drivers use to request and acquire the resources on a PCI board.

When developing a custom board with a PCI interface, the first step in debugging is to simply read from, and write to, registers on the board. Polling of interrupt status registers can even be used to check that the basic interrupt features of the board work, without having to implement interrupt handlers. Reading and writing registers on a PCI board requires a driver. But since the memory and I/O ports on a PCI board are all reported in the same way, it is possible to build a generic PCI memory and I/O driver. The driver `pci_io.c` (PCI I/O) is such a driver.

The PCI I/O driver initialization routine registers a PCI driver for a list of PCI IDs. The driver probe call-back is called when the kernel finds a device the driver supports. The probe method obtains the PCI board resources, obtains a range of minor numbers (one per memory or I/O region), and registers a character device driver for the board; so each board is a represented by a character device, and the memory and I/O regions on a board are controlled by that same device. The driver uses the `udev` system to automatically create `/dev` nodes of the form; `/dev/pci_<bus:dev.fn>_<bar>`, where `<bus:dev.fn>` is the bus-device-function triple as shown by `lspci`, and `<bar>` is the base address register (BAR) in the PCI configuration space in which the resource was claimed (there are 6 possible BARs numbered 0 through 5).

When a PCI I/O driver `/dev` node is opened, the `open()` method determines which BAR is being accessed, and passes that to the other driver methods in the `file->private_data` area. The read/write/mmap methods then use the BAR details to implement the interface to user-space. User-space applications can determine the properties of the BAR via a number of `ioctl()` calls. The user-space application `pci_debug.c` provides a terminal-like interface for reading blocks of memory from a PCI board, and writing 8-bit or 32-bit values to registers on a board.

### 3.3 Endianness

The PCI bus is defined in terms of little-endian byte format (showing a bias toward the PC architecture). However, the PCI interface is found on both little- and big-endian processors. Demonstrating how this can affect user-code requires a specific example. PLX Technologies Inc. (www.plxtech.com) manufacture a range of PCI-to-I/O bridges. These
bridges take the complex protocol of the PCI bus and translate it into a simpler read-write protocol. Since the bridges can be used in many applications, they have a number of device control registers, and these registers can be loaded from an EEPROM at boot time (to load say a custom PCI ID). The PLX PCI9054 (or PLX-9054) is a 32-bit 33MHz PCI-to-I/O bridge. The bridge is used on custom boards developed by the author, and is available on reference kits from PLX. The PLX-9054 has 256-bytes of control registers. These registers are accessible from the PCI bus via BAR[0] as 256-bytes of non-prefetchable memory and BAR[1] as 256-bytes of I/O ports. The PLX NET2270 PCI-RDK is a reference kit from PLX for evaluating a USB 2.0 controller. The baseboard of that kit uses the PLX-9054 to implement the PCI interface. The \texttt{pci\_io.ko} driver was loaded into an x86 desktop system containing the NET2270 kit, and the registers were read using the \texttt{pci\_debug} utility (BAR[0] and BAR[1] give the same results); the 256-bytes of control registers read back as

\texttt{CMD> db 0 100}

\begin{verbatim}
00: 00 00 F0 FF 01 00 00 00 00 20 01 00 05 30 00
10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40: 78 56 34 12 EF CD AB 89 00 00 00 00 00 00 00 00
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60: 00 00 00 00 00 00 00 00 00 00 01 01 0F 7E 18 0F
70: B5 10 54 90 0A 00 00 00 78 56 34 12 EF CD AB 89
80: 43 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90: 00 00 00 00 43 00 00 00 00 00 00 00 00 00 00 00
A0: 00 00 00 00 00 00 00 00 10 10 00 00 00 20 01
B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C0: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E0: 00 00 00 00 00 00 00 00 50 00 00 00 00 00 00 00
F0: 00 00 F0 FF 01 00 10 00 C3 03 00 00 00 00 00 00
\end{verbatim}

The PLX-9054 databook defines most registers as 32-bits, so its more convenient to view the bytes as 32-bit words, i.e.,

\texttt{CMD> d 0 100}

\begin{verbatim}
00: FFF00000 00000001 01200000 00300500
10: 00000000 00000000 42030140 00000000
20: 00000000 00000000 00000000 00000000
30: 00000000 00000008 00000000 00000000
40: 12345678 89ABCDEF 00000000 00000000
50: 00000000 00000000 00000000 00000000
60: 00000000 00000000 0F010100 180F767E
70: 905410B5 0000000A 12345678 89ABCDEF
80: 00000043 00000000 00000000 00000000
90: 00000000 00000043 00000000 00000000
\end{verbatim}
The register at offset-address 40h and 44h are mailbox registers that are loaded from an EEPROM. The load values are arbitrarily loaded on the NET2270-RDK as 0x12345678 and 0x89ABCDEF.

The AMCC 440EP PowerPC processor is a highly integrated CPU with multiple high-speed buses; SDRAM, peripheral, and PCI. The Yosemite evaluation kit is sold by AMCC to evaluate this processor (http://www.amcc.com/Embedded). The processor is being evaluated by the author for use in a new design. Plugging the NET2270 PCI board into the PCI slot on the Yosemite board, loading the pci_io.ko driver (compiled for the PowerPC of course!), and running the user-space debug application, gives the identical result as the x86 for read-back as bytes, but the following for read-back as 32-bit integers;

```
CMD> d 0 100
00: 0000F0FF 01000000 00002001 00053000
10: 00000000 00000000 40010342 00000000
20: 00000000 00000000 00000000 00000000
30: 00000000 08000000 00000000 00000000
40: 78563412 EFCDAB89 00000000 00000000
50: 00000000 00000000 00000000 00000000
60: 00000000 00000000 001010F 7E760F18
70: B5105490 0A000000 78563412 EFCDAB89
80: 43000000 00000000 00000000 00000000
90: 00000000 43000000 00000000 00000000
A0: 00000000 00000000 10100000 00002001
B0: 00000000 00000000 00000000 00000000
C0: 02000000 00000000 00000000 00000000
D0: 00000000 00000000 00000000 00000000
E0: 00000000 00000000 50000000 00000000
F0: 0000F0FF 01001000 C3030000 00000000
```

Yikes! Its all backwards (take a look at the values at 40h and 44h). This is analogous to the well-known endianness issue that occurs with network packets. Network data requires conversion from host byte-order to network byte-order, which is big-endian, whereas PCI byte-order is little-endian. Endianness issues in the driver can be avoided using cpu_to_le32() and friends (asm/byteorder.h). However, a user-space application that accesses memory-mapped registers or uses read() or write() needs to be aware of the endianness of the CPU relative to the little-endian PCI bus.
3.4 CPU and DMA transactions

The PLX-9054 was designed into custom compact PCI boards used at Caltech’s Owens Valley Radio Observatory in the Correlator System (http://www.ovro.caltech.edu/~dwh/correlator). A 32-bit 33MHz PCI bus has a theoretical bandwidth of 132MB/s, i.e., 4-bytes every 33MHz clock cycle. Testing of the correlator boards using CPU initiated reads and writes showed a PCI bus bandwidth as low as 1MB/s! Reference [3] contains the detailed test results. This section demonstrates the issue using results obtained with a PCI bus analyzer.

Figure 1 shows the PCI bus waveforms for a single-read and burst-read. In a single-read, the PCI bus initiator (master) starts the transaction by asserting the address, read-command, and frame signal, on the next clock it tristates the data bus, asserts the byte-enables, and asserts the initiator ready signal, and then waits for the target to indicate it is ready. A burst-read is similar, with the main difference being that the master asserts the frame signal for the duration of the transaction, deasserting it to indicate that the next clock will be the last transaction. If a bus master was to issue single-reads to a target as shown in Figure 1(a), the maximum bandwidth would be; 4-bytes at 33MHz every 5-clocks, i.e., 26.4MB/s. However, if a bus master issues burst-reads to a target as shown in Figure 1(b), then after the initial latency between the address and the first data, the data transfer occurs 4-bytes every clock, i.e., at 132MB/s, the maximum bus bandwidth. Figure 1 does not show the overhead of the read at the host CPU. The figure shows the waveforms on the PCI bus, but the PCI master may be a PCI-to-PCI bridge, with the CPU several PCI segments away. At best, the PCI master is some form of processor-to-PCI bridge on the CPU. The
additional overhead of the host-to-PCI bus interface will reduce the maximum bandwidth achievable with CPU initiated reads. PCI write transactions are similar to reads, with the difference being that the command is a write, and the initiator drives the data on the PCI bus, with the target reading it.

Figure 2(a) shows a logic analyzer trace of a CPU initiated PCI read of 8-bytes (two 32-bit words). Larger transfers showed an identical initial sequence, with additional transfers looking identical to the second transfer in the figure. The PCI reads transfer 4-bytes every 40 clocks, i.e., 3.3MB/s (performance gets progressively worse after further PCI-to-PCI
bridges). The \texttt{FRAME#} signal in Figure 2(a) is only asserted for a single clock, showing that the PCI-to-PCI bridge controlling this bus is not issuing PCI burst transactions. This is likely due to the fact that the x86 host is also not performing a burst transfer. This is a well-known problem with the x86 architecture, and is the subject of a PLX knowledge-base article (see reference [3] for copies of comments from that, and other, PLX articles). The transaction sequence in Figure 2(a) occurs regardless of whether the device driver \texttt{read()} method is called (which uses \texttt{memcpy_fromio()}) or by using \texttt{mmap()} and direct pointer manipulation from user-space (the test was performed using the PCI I/O driver and the user-space test \texttt{pci_transaction.c}). Although the slow transfer of data over PCI is clearly a limitation of the hardware, if the device driver only offers this method to transfer data, then its also a driver problem. The hardware solution to the problem is to implement DMA, so its up to the driver writer to use it.

The PLX-9054 direct memory access (DMA) controller acts as a PCI bus master, and can perform two types of DMA: block transfers and scatter-gather transfers. A block transfer is a single bus transaction, whereas a scatter-gather transfer consists of multiple bus transactions. Programming a DMA controller is fairly simple; you supply a source address, a destination address, a size, a control register setting, and then trigger the transfer. Figure 2(b) shows the results of a DMA transfer between two boards on the same PCI bus segment. The DMA controller on the source board performs a PCI write to the second board. This models what is required to DMA data to host memory (to replace a slow CPU initiated PCI read). The DMA controller registers on the first board were programmed by hand via a simple terminal program running on the on-board DSP (communication was via the serial port, to avoid transactions on the PCI bus). The PCI address of the destination board determined from \texttt{lspci}. Hand-programming of registers using simple read/write programs on hosts and targets is a great technique for figuring out the workings of a new piece of hardware without having to debug driver issues.

Figure 2(b) shows a transfer between two devices on the same bus, with no other devices on that bus. This gave the PCI master exclusive access to the bus, allowing it to obtain the maximum achievable PCI bus bandwidth. The source and destination addresses in the tests were from, and to, burst-capable SDRAM on both boards. Additional captures of bus writes would sometimes show wait-states inserted on the PCI bus whenever the source ran out of data briefly, or the destination FIFO became full while waiting for writes to complete (eg. due to SDRAM refresh cycles pre-empting the PCI transfer). DMA transactions that read data from one board to the other were also tested. In those tests, the target inserted initial wait-states until the PLX-9054 read FIFOs contained data, and then data was burst over the PCI bus in the same fashion as the PCI write-burst.

Figure 3 demonstrates that its not just the host CPU that can reduce the PCI bus bandwidth. The figure shows DMA transactions between two PLX-9054 boards, but with a PCI bridge between them. The PCI bus analyzer probe was located on the same bus segment as the PLX-9054 acting as a bus master. The waveforms in Figure 3 show that the PCI bridge, acting as a target, can not sustain the full 132MB/s PCI bus bandwidth (the bridge does perform burst-mode transactions, but only for short bursts). Moving the logic analyzer onto the bus containing the target PLX-9054 board (where the PCI bridge acts
Figure 3: Correlator board PCI transactions through a PCI-to-PCI bridge. (a) shows a DMA write transaction from the PLX-9054 to a board on the other side of a PCI bridge. The bridge (acting as a target on the PCI bus probed) accepts about 20 clocks of data and then issues a *disconnect* (asserts STOP#). The PLX-9054 then tries to continue the transaction, but receives a *retry* from the bridge, and then on the next attempt, the transfer continues. This process repeats until the transfer of 256-bytes completes, taking about 3.75µs (65MB/s). (b) shows a DMA read transaction for the same hardware configuration. The bridge issues retries to the PLX-9054 until it can transfer a block of 8 words (32-bytes), at which point the bridge issues a disconnect. The figure shows how a transfer of 128-bytes is broken up into four 8 word (32-byte) transfers. The transfer of 128-bytes takes 5µs (24MB/s).

as a master) showed the PLX-9054 (target) board received or delivered all data requested by the master. Since the tests in Figure 2 show that the PLX-9054 boards can burst at near full-bandwidth, the problem clearly lies with the bridge. PCI-to-PCI bridges do have configuration registers, so it would be possible to fine tune a system, however, this is not
a job for the PLX-9054 board’s device driver, since every system can have different PCI bridges.

The PLX-9054 DMA controller PCI bus read command defaults to memory-read-line (MRL), whereas the CPU read command is memory-read (MRD), and a third option is memory-read-multiple (MRM). The DMA controller was programmed to use each of the memory read command types; MRL and MRD transferred data in blocks of 8-words, while MRM transferred in blocks of 16-words. A transfer of 256-bytes took \( 9 \mu s \) for MRL/MRD (27MB/s), while for MRM it took \( 7 \mu s \) (34MB/s); so the doubling of the burst size did not double the bus bandwidth. The PLX-9054 DMA controller PCI bus write command defaults to memory-write (MWR), but there is also a memory-write-and-invalidate (MWI) write command. The use of the MWI command changed the transfer such that data was transferred in blocks of 16 words (whereas the MWR transfers were slightly larger than this). For a transfer of 256-bytes, the two commands performed similarly (eg. see the MWR performance in Figure 3(a)).

4 **A ‘REAL-WORLD’ PCI DRIVER**

The experience presented in this document was gained during the development of the Caltech-OVRO Broadband Reconfigurable Array (COBRA) Correlator System. The hardware developed for that system is documented at [www.ovro.caltech.edu/~dwh/correlator](http://www.ovro.caltech.edu/~dwh/correlator). The hardware is currently in use on several radio astronomy projects, eg. the SZ Array ([http://astro.uchicago.edu/sza/](http://astro.uchicago.edu/sza/)) and the CARMA array ([http://www.mmarray.org](http://www.mmarray.org)).

The ePCI digitizer and correlator boards used in the correlator system contain a PLX-9054 PCI interface, a Texas Instruments DSP, Altera FLEX10K FPGAs, and on the digitizer, 1GHz analog-to-digital converters. The digitizer output routes to the FPGAs on the digitizer board, where data is digitally filtered, delayed, and routed to front-panel high-speed connectors. The data travels over LVDS cabling (Ultra-SCSI cables) to the correlator boards, where FPGAs cross-correlate and average the data. The on-board DSPs retrieve auto- and cross-correlation results from the FPGAs, perform FFTs, further corrections, and average the data for 100ms to 500ms. Data is then transferred to a Linux host. The system uses a GPS based NTP server with a 1pps output. The 1pps signal is used to derive a hardware heartbeat, so that the 100ms and 500ms transfers are aligned with real-time. The Linux hosts run NTP pointing to the NTP server, and check that data from boards arrives within a 50ms window relative to a 100ms or 500ms boundary.

The Linux driver used in the COBRA system is shown graphically in Figure 4 [2]. The driver implements several character device interfaces to the board; a terminal-like interface with standard-input, output, and error, a read/write control interface, a read-only data interface, and a read-only monitoring interface. The reason for using multiple devices, rather than a complex scheme of I/O control was determined by the usage of the driver. For example, one objective was to enable the use of standard command line tools like `cat`, `od` (octal dump), `echo`, and `dd`. These tools know nothing of I/O control calls, so need to be directed to a device node of a specific ‘personality’. The COBRA control system
Figure 4: COBRA device driver block diagram. The block diagram shows the relationship between the /dev nodes accessed by user-space applications and the files that implement the driver.
code controls up to 20 boards in a single sub-system, and data must be collected from each board at about the same time. The standard method for dealing with multiple sources of data is to use the \texttt{select()} call, which uses file-descriptors. So by separating out the data device and monitor device functionality at the driver-level, a user-space server can run a thread containing a \texttt{select()} call that collects all the data from all boards, and serves that data up to clients. Then another thread, or another process even, can run a monitor server containing a thread calling \texttt{select()} on all the monitor file descriptors.

References


