COBRA Device Driver

D. W. Hawkins
dwh@ovro.caltech.edu
Document Revision: 1.9
March 31st, 2006

Contents

1 Introduction 3

2 The PLX-9054 Device Driver 4

3 Device naming using udev 5

4 The PLX-9054 virtual serial driver 7

5 The COBRA Device Driver 13
  5.1 Host-to-target transfers .................................. 16
  5.2 Buffer entries and the buffer list .......................... 16
  5.3 Transaction requests and the transaction list .............. 17
  5.4 Driver internals ....................................... 17
  5.5 The stdio and download devices .......................... 21
  5.6 COBRA slot numbering and device naming .................. 22

6 Driver test applications 23
1 Introduction

This document describes the COBRA Correlator System device driver for use under Linux kernel version 2.6. The correlator CPUs use the Centos 4.1 Linux distribution (2.6.9-11 kernel), and the code builds fine for kernels up to 2.6.15 (minor API differences exist between some of the 2.6 kernels). The driver design uses constructs that are specific to the 2.6-series kernels, so the driver is not backwards compatible with previous versions. The books ‘Linux Device Drivers’, 3rd Edition, by Corbet, Rubini, and Kroah-Hartman [1], and ‘Linux Kernel Development’, 2nd Edition, by Love [2] should be consulted for Linux 2.6 kernel details.

This document describes three drivers; the PLX-9054 driver, a PLX-9054 virtual serial driver, and the COBRA device driver. The PLX-9054, or PLX Technologies (www.plxtech.com) PCI-9054, is a PCI-to-local bus controller used on numerous adapter boards to provide a PCI interface. The PLX-9054 has a high-performance DMA controller, and registers that enable efficient host-to-target (system slot board to adapter board) communications. The PLX-9054 driver describes a generic driver for accessing the PLX-9054 control registers. The device operations are re-used in the other drivers. The PLX-9054 virtual serial driver demonstrates a host-to-target communications protocol and develops a virtual serial port over the PCI bus. The COBRA device driver extends from the virtual serial driver; maximizing performance over the PCI bus via the use of memory buffers, and the PLX-9054 DMA controller.
2 The PLX-9054 Device Driver

The COBRA boards interface to the PCI bus through a PLX Technologies PCI-9054 (PLX-9054) PCI-to-Local bus bridge. The local bus side of the device connects to the COBRA board system controller FPGA. The PLX-9054 converts the PCI bus protocols into a simpler local-bus interface. The system controller FPGA internally arbitrates accesses to the hardware by the DSP and PCI bus (allowing both the DSP and host CPU to read/write to any device on the board).

The PLX-9054 contains a set of internal control registers. These registers are accessible via a 256-byte region of memory space or I/O space located on the PCI bus. Some of these control registers can be loaded at boot time from an on-board EEPROM (this is how users customize the device for their specific application). When the COBRA boards are delivered, this EEPROM is blank, and so it must be programmed before the boards functionality is visible over the PCI bus. The PLX-9054 allows programming of the EEPROM through interface registers visible in the PCI configuration space area of the device. However, before programming the device using these registers, the EEPROM must be made writable. The write-enable register is located in the 256-byte region of control registers. Access to this region requires a device driver. Thus, to program the EEPROM, either a stand-alone PLX-9054 device driver is required, or the COBRA device driver must enable access to boards with blank EEPROMs. Both of these options have been implemented.

The PLX-9054 driver consists of three headers and two source files: plx_device.h, plx_ioctl.h, plx_registers.h, plx_device.c, and plx_driver.c. These files are built into a kernel module; plx.ko. The headers and device file contain the re-usable component of the driver; the file-operations used to manipulate the 256-byte PLX-9054 control registers. The driver implements read(), write(), and lseek() so that command-line applications can be used for accessing the device registers, and implements mmap() for custom application access. The user-space application plx_debug provides a simple interface for manipulating PLX-9054 registers. For example, the PLX-9054 registers on a board can be read back via

CMD> d 0 100

00: FF800008 00000001 10200000 00300600
10: 00000000 00000000 4B4300C3 FFFF0000
20: 01390000 013A0000 00000000 00000000
30: 00000000 00000008 00000000 00000000
40: 00000000 00000000 00000000 00000000
50: 00000000 00000000 00000000 00000000
60: 00000000 00000000 00000000 00000000
70: 905410B5 0000000A 00000000 00000000
80: 00000000 00000000 00000000 00000000
90: 00000000 00000000 00000000 00000000
A0: 00000000 00000000 00000101 10200000
B0: 00000000 00000000 00000000 00000000
C0: 00000000 00000000 00000000 00000000
D0: 00000000 00000000 00000000 00000000
E0: 00000000 00000000 00000500 00000000
F0: 00000000 00000000 00000000 00000000

and registers can be written to. This interface is very useful when developing drivers, as a user can simulate the host-side, or DSP-side of the communications protocol.
3 Device naming using udev

When peripheral boards are plugged into a PCI backplane, the ‘electrical’ location of a board is unique, and is given by the ‘bus’, ‘device’, and ‘function’ triple. The Linux command `/sbin/lscpi` on the host board displays this information in the form `<bus:dev.fn>`. For example, the following command lists the PLX-9054 devices found by a Linux host CPU plugged into single-segment compact PCI (cPCI) backplane;

```
$ lspci -d 10b5:9054
01:09.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
01:0a.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
01:0b.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
01:0c.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
01:0d.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
01:0e.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
01:0f.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
```

where 10b5 is the PLX Technologies PCI vendor ID, and 9054 is the device ID. This system has seven boards on PCI bus number 1. The output on another system with a PCI backplane consisting of three bridged segments is

```
$ lspci -d 10b5:9054
01:0b.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
01:0c.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
02:0b.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
02:0c.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
03:0b.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
03:0c.0 Bridge: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 0a)
```

where two boards have been plugged into each segment, i.e., on buses 1, 2, and 3.

The bus number of a peripheral board depends on the host CPU’s hierarchy of PCI-to-PCI bridges, e.g., a peripheral board may be seen on bus 1 by one model of host CPU, but on bus 2 by another model. When a user accesses a board, it is convenient to refer to a board by its ‘physical’ (or mechanical) ‘slot number’ where slot numbers in a cPCI crate start from 1 on the left hand side of a crate and increment from left to right. To avoid hard-coding the mapping of `<bus:dev.fn>` to slot number within the driver, the driver makes use of the Linux 2.6 `udev` user-space device node creation scripts. The PLX-9054 driver creates a kernel device name based on the device location. The default operation of the `udev` scripts is to create a device entry in `/dev` based on the kernel name. For example, for the 6 boards in the second system commented on above, the default `/dev` entries are;

```
# insmod plx.ko
# ls /dev/plx*
/dev/plx_01:0b.0_0 /dev/plx_02:0b.0_0 /dev/plx_03:0b.0_0
/dev/plx_01:0c.0_0 /dev/plx_02:0c.0_0 /dev/plx_03:0c.0_0
```

where the kernel name is of the form `plx_<bus:dev.fn>_0`. The trailing number, in this case _0, is used to indicate a device on a board; the PLX-9054 driver only defines one device, so this number could be left off, but the serial and COBRA drivers define multiple device nodes per board.
The `/dev` entry naming scheme is user-customized by adding a rules file to the `udev` scripts area. For the PLX-9054 driver, a rules file could take the form:

```
# /etc/udev/rules.d/20-plx.rules
#
# PCI <bus:dev.fn> to slot renaming rule
#
KERNEL="plx\*", PROGRAM="plx_slots -k %k -f pci.slots", NAME="%c"
```

This script says 'for each kernel device that starts with the characters plx, execute the program `plx_slots` passing the kernel name `%k`, and the file `pci.slots`, and create the `/dev` name based on the output of that program' (%c captures the string output of that program). The file `pci.slots` would be an ASCII mapping of PCI `<bus:dev.fn>` triples to slot numbers. The triple is encoded in the kernel name passed to the program, so it just extracts that from the input string, performs a triple-to-slot conversion, creates a slot-based string name, outputs it to stdout, and returns zero to indicate success to the `udev` system. Converting the default kernel names is not really useful for the PLX-9054 driver, since its really only used for testing and EEPROM programming. The COBRA driver makes full use of this concept, and its `udev` script is described later.
4 The PLX-9054 virtual serial driver

This section describes the design of an interrupt-based host-to-target communications protocol. The PLX-9054 virtual serial driver is a practical application of that protocol, and shows the Linux kernel mechanisms required to build a Linux character device driver.

A serial port is a good example of an interrupt driven I/O device; an interrupt is generated when a character is received, and an interrupt is generated when a character has been transmitted. Additionally, a serial port supports the concept of flow-control; the receipt of characters can be flow-controlled using the RTS signal, and transmission of characters can be flow-controlled using the CTS signal (where the CTS signal is controlled by the device being transmitted to).

The protocol for the PLX-9054 virtual serial port is:

- Host-to-target transmission:
  - The host writes a character to a mailbox register.
  - The host sends the target a receiver ready (RX_READY) interrupt by setting a bit in the PCI-to-local doorbell (P2LDBELL) register.
  - The target receives an RX_READY interrupt, clears the interrupt, reads the character from the mailbox register, and buffers it.
  - The target sends the host a transmitter empty (TX_EMPTY) interrupt by setting a bit in the local-to-PCI doorbell (L2PDBELL) register.
  - The host receives a TX_EMPTY interrupt, clears it, and allows further transmissions to the target.

- Target-to-host transmission:
  - The target writes a character to a mailbox register.
  - The target sends the host a receiver ready (RX_READY) interrupt by setting a bit in the local-to-PCI doorbell (L2PDBELL) register.
  - The host receives an RX_READY interrupt, clears the interrupt, reads the character from the mailbox register, and buffers it.
  - The host sends the target a transmitter empty (TXEMPTY) interrupt by setting a bit in the PCI-to-local doorbell (P2LDBELL) register.
  - The target receives a TX_EMPTY interrupt, clears it, and allows further transmissions to the host.

The host-to-target transmission protocol is symmetric with respect to the target-to-host protocol. The interrupt hand-shake implements flow-control; eg. the host may not send more characters to the target until it has received a transmitter empty acknowledge, and vice versa for the target-to-host flow-control.

A single virtual serial port requires 2-bits in the PCI-to-local doorbell register 2-bits in the local-to-PCI doorbell register, and two mailbox registers. The PLX-9054 has 8 mailbox registers, and 32-bits in each doorbell register, so the PLX-9054 virtual serial driver uses all the mailboxes and creates four virtual serial devices per board. In keeping with the serial port analogy, the driver uses only one byte of each mailbox register.

Figure 1 shows a block diagram of the buses, interrupts, and registers involved in host-to-target communications via the PLX-9054 registers. Figures 2, 3, and 4 show host-to-target timing diagrams of increasing complexity. Figure 2 shows the host sending the target a character; this is the simplest case to understand. Figure 3 shows the slightly more complex case of simultaneous transmission of characters between the two processors. By assuming that the two processors operate at similar
Figure 1: Host-to-target communications with the PLX-9054. The host sends data to the target by writing to the mailbox 0 (MBOX0) register, and then setting a bit (RX_READY) in the PCI-to-local doorbell (P2LDBELL) register. This generates a local interrupt (LINT#) to the target processor. The target reads the mailbox register and sends an acknowledge by setting a bit (TX_EMPTY) in the local-to-PCI doorbell (L2PDBELL) register, which generates an acknowledge interrupt to the host. The target sends data to the host by writing to the mailbox 1 (MBOX1) register, and then setting a bit (RX_READY) in the local-to-PCI doorbell (L2PDBELL) register. This generates a PCI interrupt (INTA#) to the host processor. The host reads the mailbox register and sends an acknowledge by setting a bit (TX_EMPTY) in the PCI-to-local doorbell (P2LDBELL) register, which generates an acknowledge interrupt to the target. The registers in the PLX-9054 perform a task analogous to the transmit, receive, status and control registers in a serial port UART.

Figure 2: Host-to-target transmit timing. This figure shows the register accesses and interrupt assertions for the transmission of a character from the host to the target. Since the doorbell registers are written by one processor, and read and cleared by the other, the registers need to implement atomic bit setting and bit clearing operations. The doorbell registers use write-1-to-set and write-1-to-clear logic to ensure atomic operation.
Figure 3: Host-to-target transmit/receive timing. This figure shows the register accesses and interrupt assertions for the transmission of a character from the host to the target, at the same time the target transmits a character to the host. In this particular example, the timing of the host and target are similar.

Figure 4: Host-to-target transmit/receive timing. In this example, the host is faster than the target. The host has received and acknowledged the character sent by the target, before the target has entered its receive character ISR. The key thing to note from this timing is that the P2LDBELL register is written twice by the host, and bits in the register must accumulate. The doorbell register write-1-to-set semantics make this cumulative setting of bits race-free, i.e., the other processor will not miss an interrupt.
speeds, the diagram simply looks like Figure 2 repeated for each processor. Figure 4 shows a more realistic case where there is a speed difference between the processors, or at least a latency in one processor entering an ISR relative to the other. The timing of the P2LDBELL in Figure 4 demonstrates a critical hardware implementation issue; the host processor needs to report another event to the target, yet must avoid a race condition with the target performing a read and then clear of that register.

When dealing with inter-processor communications, it is important to consider race conditions with respect to shared variables. It is impossible to ensure that a read-modify-write operation on a shared variable performed by either processor will be successful, unless such an operation is atomic. The doorbell registers implement atomic bit operations; bits are set using write-1-to-set (where the 1 is in the appropriate bit position), and bits are cleared using write-1-to-clear. In addition to this, in the P2LDBELL only the host can set bits, and the target clear bits, and with the L2PDBELL only the target can set bits, and the host clear them. The write-1-to-set and write-1-to-clear semantics also make it possible for multiple processes on each processor to access independent bits in the doorbell registers. The PLX-9045 virtual serial driver creates four separate serial devices per board, and each of those devices are independent, but all access doorbell bits. Regardless of the driver implementation, multiple-process access to the doorbell register is safe, and the driver does not need use a lock to protect access (assuming of course, that the accesses are to control different doorbell bits).

Returning to Figure 4; there can be no race condition. If as shown in the figure, the target ISR is not entered until after the host has written both bits, then the target will read and clear both bits. If the target ISR had been entered just slightly prior to the host TX_EMPTY write, then the target will only read the RX_READY bit and clear it. If the target ISR then exits, LINT# will still be asserted, and the target will be interrupted, and will process the TXEMPTY acknowledge. The host and target can avoid some additional interrupts by writing their interrupt service routines such that the doorbell register is read in a loop until it is clear, e.g., on ISR entry the first loop will process the source of the interrupt, and the second iteration will process any bits that got set while the ISR was running.

With the functional operation of the hardware understood, the PLX-9054 virtual serial driver can now be described. The basic operation of the driver is:

- **write()** copies data into a transmit ring-buffer (circular-buffer), and if the transmitter is not busy, starts it transmitting characters from the buffer. If the buffer fills, the writing process is blocked (if the device is opened in blocking mode).

- **read()** copies data from a receive ring-buffer (circular-buffer), and if the buffer is empty, blocks until data is available (if the device is opened in blocking mode). The RX_READY ISR receives characters and adds them to the receive ring-buffer.

The transmit and receive ring-buffers are accessed by both the user-space process (to write to, or read from, the buffers) and the kernel-space interrupt service routine. The Linux kernel offers several options for locking resources, the two most common being spin-locks and semaphores. A spin-lock disables interrupts, and can be used for locking a resource shared between a process and an interrupt or timer handler. A process can sleep while trying to access a semaphore, so it cannot be used in an interrupt or timer handler. The obvious locking candidate for this application is the spin-lock, however, the use of spin-locks is generally frowned upon in the Linux kernel, as disabling interrupts can reduce responsiveness. The PLX-9054 virtual serial driver uses a semaphore for protecting access to the ring-buffers, and uses another Linux feature to avoid the use of locks in the interrupt handler; a kernel work-queue. A work-queue is essentially a kernel thread reading from a message queue, where the messages in the queue are work functions. When a work function is added to the queue, the thread becomes schedulable, and when scheduled, will dequeue and call the work function.

A detailed description of the major functionality of the PLX-9054 virtual serial driver is then (this description should be read in conjunction with a copy of the source code):
• Write

  – A user-space call to `write()` invokes the driver file-operation `serial_write()`.
  – The driver write routine acquires the transmit semaphore to lock accesses to the transmit variables by either the kernel work thread, or other user-space processes that also call write.
  – The driver checks for space in the transmit buffer, and if there is none and the file was opened non-blocking, releases the semaphore and returns. If there was no space and the file was opened blocking, then the process releases the semaphore and is put to sleep on a wait-queue.
  – If a write process is sleeping and is woken due to space, then it re-acquires the semaphore.
  – Once there is space in the transmit buffer, the amount of data to write is checked against the amount of space, and the user-space count is truncated if required. User-space data is then into the transmit buffer.
  – Since a `copy_from_user` space can sleep, a spin-lock can not be held while performing this operation, this is one reason why a semaphore is preferred for locking.
  – The driver then checks the transmit-busy flag; if the transmitter is not busy, then a transmit-start work request is added to the driver work-queue.
  – The semaphore is then released and the write call completes.

• Transmit work

  – The transmit-start work function, `serial_transmit_start`, acquires the transmit buffer semaphore, reads a character from the buffer, releases the semaphore, sets up a timeout timer, writes the to the PLX-9054 mailbox and sends an `RX_READY` interrupt to the target.
  – The transmit-done work function, `serial_transmit_done`, is scheduled by the interrupt handler when it receives a `TX_EMPTY`. The work function acquires the transmit buffer semaphore, determines whether there is enough space in the buffer to wake blocked writers, and if there is more data to send, schedules a transmit-start work request. If there is no more data in the buffer, it deasserts the transmit-busy flag, so that a subsequent call to write can schedule a transmit-start work request. The semaphore is then released.
  – The transmit-error work function, `serial_transmit_error`, is scheduled by the write-timeout timer handler if a `TX_EMPTY` is not received within the driver transmit timeout (a load-time parameter, defaulting to 5 seconds). The work function acquires the transmit buffer semaphore, flushes the buffer, clears the transmit-busy flag, wakes up writers, and releases the transmit semaphore.
  – Since the work functions are run in the context of a kernel thread, there is no way to return an error code to the user in the case of a transmit timeout. A flag could be set that returns an error to the next write, but the next write might actually succeed, so that is not an ideal solution. The solution used in the driver is to simply drop the data, log a timeout, and continue. It is up to the user-space program to use a request-response protocol to ensure data was received by the target.
- **Read**
  - A user-space call to `read()` invokes the driver file-operation `serial_read()`.
  - The driver read routine acquires the receive semaphore to lock accesses to the receive variables by either the kernel work thread, or other user-space processes that also call `read`.
  - The driver checks for data in the receive buffer, and if there is none and the file was opened non-blocking, releases the semaphore and returns. If there was no data and the file was opened blocking, then the process releases the semaphore and is put to sleep on a wait-queue.
  - If a read process is sleeping and is woken due to data, then it re-acquires the semaphore.
  - The driver checks whether the receive buffer is full. The work function receiving data will not send an acknowledge to the target when the receive buffer is full. This flow-controls data until `read` makes space in the receive buffer. Prior to removing data, the read function sets a flag to send an acknowledge based on the full-state of the buffer.
  - The amount of data requested by the user is checked against the amount of data available, and the user-space count is truncated if required. Data from the receive buffer is then copied to the user-space buffer, and the semaphore is released.
  - Since a `copy_to_user` space can sleep, a spin-lock can not be held while performing this operation, this is one reason why a semaphore is preferred for locking.
  - If the buffer was full prior to the read call, a transmitter-empty acknowledge is sent to the target.

- **Receive work**
  - The receive work function, `serial_receive_work` is scheduled by the interrupt handler when it receives an `RX_READY` interrupt from the target.
  - The receive work function acquires the receive semaphore, reads the character from the mailbox, adds it to the receive ring-buffer, wakes any waiting readers, and releases the semaphore.
  - If the receive buffer is not full, then the transmitter-empty acknowledge is sent to the target. If the buffer is full, then the acknowledge is held-off until a user-space read makes space in the receive buffer.

That's the heart of the driver as seen by a user-space process. The driver has some other nice features worth describing. The driver makes use of the Linux kernel PCI subsystem, and registers itself with the kernel as handling boards with the PLX-9054 PCI vendor and device ID. The kernel calls the `serial_probe` call-back when a device with a matching device and vendor ID is detected. The probe routine creates a board structure, `serial_board_t`, that is used to manage the resources and devices for a board (i.e., an array of four `serial_device_t`s), then acquires the PCI board resources and initializes the serial device resources. When a user-space application calls `open()` the driver determines which board is being opened, and then which device on a board. The device specific information is then set in `file->private_data` for device file operations to use. When the driver is removed, the PCI subsystem calls the `serial_remove` call-back, which releases the resources acquired by a board. With multiple boards in a system, the probe and remove calls occur for each board. The makefile for the driver can be edited to turn on debug messages. The kernel log file can then be viewed (using `dmesg`, or `tail -f /var/log/messages`) to see the driver operational sequences. The driver source code is well commented, and contains additional details on the driver functionality.
5 The COBRA Device Driver

Figure 5 shows a conceptual block diagram of the COBRA device driver. The block diagram shows the relationship between the /dev nodes accessed by user-space applications and the devices implemented by the COBRA the driver. Each COBRA board consists of 7 device nodes:

- /dev/cobra_plx# PLX PCI9054 interface device
- /dev/cobra_control# COBRA control device
- /dev/cobra_stdio# COBRA stdin and stdout device
- /dev/cobra_stderr# COBRA stderr device
- /dev/cobra_monitor# COBRA monitor device
- /dev/cobra_data# COBRA data device
- /dev/cobra_download# COBRA download device

Although the COBRA device functionality from user-space is different, the driver implementation for each device is very similar. The COBRA driver is implemented by the following files:

- plx_device.h, plx_ioctl.h, plx_registers.h PLX-9054 headers.
- plx_device.c PLX-9054 device file-operations.
- cobra_sdram.h PCI shared-memory (SDRAM) layout.
- cobra_ioctl.h User-space and driver ioctl codes.
- cobra_driver.c COBRA driver.

The COBRA driver uses the same doorbell-based host-to-target communications protocol as the PLX-9054 virtual serial driver. However, the mailbox registers are not used, instead data is transferred via PCI shared-memory regions. This provides a performance improvement in that blocks of data can be transferred for every host-to-target interrupt handshake (this is analogous to adding a FIFO buffer to a serial port). Figure 6 shows the 4MB SDRAM shared-memory layout. The COBRA driver also adds the option of using the PLX-9054 DMA controller to transfer data to or from the board. This feature offers the most significant performance improvement, but requires additional complexity in the driver. One reason for describing the PLX-9054 virtual serial driver first, is that it makes the functionality of the COBRA driver easier to understand.
Figure 5: COBRA device driver conceptual block diagram. The block diagram shows the relationship between the /dev nodes accessed by user-space applications and the devices implemented by the COBRA driver.
Figure 6: COBRA SDRAM layout. The 2MB download region is used by the host to boot the target DSP, and to download the correlator FPGAs. The other regions are used for host-to-target communications, with each region being divided into a write or read buffer (where the direction is defined relative to the host). The stdio region is divided into four buffers; write and read for stdio, write and read for stderr.
5.1 Host-to-target transfers

The additional complexity of the COBRA driver relative to the PLX-9054 virtual serial driver is due to the use of the PLX-9054 DMA controller. There are two DMA controller channels on the PLX-9054. The COBRA driver only uses channel 0. There is no advantage in using the other channel, since the PCI bus is a shared bus and so two simultaneous transfers can not occur. Two scheduled transactions could be initiated, and the priority logic in the PLX-9054 would sequence them, but this minor optimization would likely add more complexity to the driver, for little added benefit.

Each COBRA board contains multiple devices that can independently initiate read or write host-to-target transfers. However, with only one DMA controller, those transfers need to be queued (serialized). The driver implements this queuing by defining a transaction request structure, and a transaction request queue (list, actually). For example, when a user writes data to a device, the data is buffered, and if a write transaction is not in progress, a write transaction request is added to the transaction queue, the request includes a call-back that enables the device to be notified when the transaction completes. The transactions can be performed by either the CPU or the DMA controller, when the request is complete, the call-back is called, allowing write to schedule another write transaction. A user-space read blocks waiting until there is a read buffer. A receiver ready interrupt from the target causes a read transaction request to be added to the transaction queue. When the read transaction is processed and completed, the transaction call-back adds the buffer read from SDRAM onto the read buffer queue, and the user-space read then consumes the data from that buffer.

5.2 Buffer entries and the buffer list

The driver read and write file-operations copy data to or from user-space in terms of a linked list of buffers. When a user-space write occurs, the length of the write is checked and truncated to the maximum length allowed by the device, a buffer list entry and pages are allocated, and the user-space data is copied into the buffer pages. When the target writes to the host, a similar action is performed, and the target data ends up on the host read buffer list. When user-space reads the data, the driver removes a buffer from the read list, and copies data to user-space until all data in the buffer is read (multiple user-space reads are allowed).

The driver represents a buffer list with the buffer_list_t structure, and entries on the list with the buffer_entry_t structure. The list is designed using the Linux kernel generic linked-list structure. The write component of each device maintains a write buffer list, while the read component maintains a read buffer list. The lists each have a maximum buffer size (i.e., the maximum size of a write or read), and a maximum buffer list length (which is set to 16). With two lists per device, and six devices per board, the number of allocated list entries in the lists is 192 per board. In a cPCI crate with 20 boards, the total number of entries in the lists is 384 (the maximum number of entries is slightly higher than this, since write will allocate an entry before adding to the list, and an entry is remove from the list to read it). Since these entries will be allocated and freed on a regular basis, some form of caching is required to avoid memory fragmentation. The Linux kernel kmem_cache is used to create a cache of buffer_entry_t structures, and buffer entries are allocated and returned to this cache.

Buffer list entries and buffer pages are allocated in write(), or as the result of a receiver ready interrupt, and subsequent work function. The list entries are allocated from the buffer cache, while the buffer pages are allocated using alloc_pages(). Allocation and freeing buffers in units of pages allows the kernel page-caching to track pages, rather than having to code this into the driver. Kernel pages are referenced in terms of a page pointer, and do not necessarily have a corresponding kernel address. The driver write and read routines obtain kernel addresses for the pages in a buffer by temporarily mapping the pages, a page at a time, using kmap(), and releasing the mapping using kunmap(). Data is written to a buffer starting at the first page of the buffer; write writes data from user-space, while receiver ready processing writes data from SDRAM. Since the kernel buffers
are page-order (power-of-2) allocated, they will typically be larger than the data they hold (so the buffer entry contains a length parameter to track the amount used). Buffer pages are freed for a write when a write transaction has been acknowledged, and for a read when a user-space process consumes the data in a buffer.

A user-space read causes the driver to remove a buffer entry from the read list. The read call then maps the first page in the buffer to a kernel address, and starts copying data from the buffer to user-space. The kernel address is unmapped and remapped as page boundaries are crossed. If the user-space read count is less than the buffer length, then the driver needs to track how much data has been read. A buffer entry contains an offset parameter that starts at zero, and tracks the progress of a read. Subsequent user-space read calls will continue consuming data at the correct buffer offset. When the contents of a read buffer are consumed, the buffer entry is freed back to the buffer entry cache, and the buffer pages are freed back to the kernel.


5.3 Transaction requests and the transaction list

A transaction request describes an operation requested by a device, or a work function, eg. a write request, or a read request. The driver represents the transaction list with the *transaction_list_t* structure, and requests on the list with the *transaction_request_t* structure.

When a write transaction is added to the transaction request list, a new transaction request is allocated from the request cache, the write data buffer pages are converted to a transaction request DMA scatter-gather list (the transaction request entry becomes responsible for the buffer pages), and the write buffer entry is then freed (returned to the buffer entry cache). When the write transaction completes, either due to the CPU copying the data to the board, or the DMA controller transferring the data to the board, the transaction call-back is executed; the call-back schedules a write transaction done work function. The write transaction done work function frees the transaction entry, and buffer pages.

When a receiver ready interrupt occurs, a read request work function is scheduled. The work function creates a read transaction request and adds it to the transaction request list. When the request is processed, the length of the read is read from a device specific location in SDRAM, if the length is valid, then pages are allocated, and the read transaction is performed. The call-back for the read transaction allocates a buffer entry, converts the transaction request pages into buffer entry pages, adds the buffer entry to the read list, and then frees the transaction request entry. A previously blocked user-space read would then unblock, and proceed to read data from the buffer.

5.4 Driver internals

This section provides a detailed description of the driver code, and should be read in conjunction with a copy of the source code; *cobra_driver.c*. Figure 7 shows the life-cycle of buffer entries, pages, and transaction requests for write and read transactions on a device in the COBRA driver. The *read()* and *write()* calls on the left of the figure occur in the context of the user-space calls (kernel operations triggered by the user call), whereas the details on the right are activities performed in the context of a kernel thread; the work-queue processing thread. There are six devices on each board that use the COBRA driver file-operations, each device has a set of read and write buffers, each device uses a pair of doorbell interrupt bits, and there is one transaction list per board. So, this means that there are six write lists, read lists, receiver ready interrupts, and transmitter empty interrupts that are generating events that schedule transactions on one transaction queue. This complexity enables the sharing of the DMA controller.
Figure 7: COBRA resource life-cycle. The figures show the life-cycle of buffer entries, kernel pages, transaction requests, and kernel work-queue function queuing for (a) write, and (b) read. See the text for more details.
Figure 7(a) shows the life-cycle of a write transaction;
1. A user-space write() call occurs, and triggers the allocation of a buffer entry, and pages to store the user-space write data.
2. Data is copied from user-space into the buffer.
3. The buffer is added to the write-list, and a write-request work function is scheduled. The user-space write is then complete.
4. When the write-request function is scheduled, it removes a buffer entry from the write list, allocates a transaction request, converts the pages in the buffer entry into a write transaction request, and frees the buffer entry. The transaction is then added to the transaction list, and a transaction-start work function is scheduled.
5. When the transaction-start function is scheduled, it removes a request from the transaction list, and performs the write transaction; if the device uses DMA for transfers, then the DMA controller is programmed to perform the write, otherwise, the CPU performs the transfer to SDRAM.
6. When the transfer is done, the transaction request call-back is called (which schedules a write-ready work function), and a transaction-done work function is scheduled. In the case of a DMA transfer, these operations are performed in the DMA interrupt handler, whereas for a CPU transfer, the CPU performs the calls after it has performed the transfer.
7. The write-ready work function is called when data is in SDRAM, so the write transaction request and write buffers are freed, and the target (DSP) is sent a receiver ready interrupt. A write-timeout timer is then started; in case the target is not running, and therefore will never reply with the transmitter empty acknowledge.
8. The transaction-done work function is called when a transaction completes, it checks whether there is another request in the transaction list, and if there is, schedules a transaction-start work request to continue transaction processing.
9. When the target acknowledges the receipt of data by sending a transmitter empty interrupt, the interrupt handler schedules a write-done work function. When the write-done work function is scheduled, it checks to see if there is another entry in the device write list, and if there is, schedules a write-request work function.
10. If the target does not acknowledge the receipt of data, then the timeout timer fires, and the handler schedules a write-error work function. The write-error work function flushes the contents of the write list (since a timeout is most likely due to a non-responsive target, and further writes will also fail).

Comments on write locking are now in order. There are three locks of interest; the write transaction semaphore, the write list semaphore, and the transaction list semaphore. When a user-space write is called, the write transaction semaphore is acquired; to block other user-space writes. Once the user-space data is copied into a buffer, the write list semaphore is acquired, and if the list is full, released. With the write list semaphore released, the kernel worker thread can remove a write entry (while holding the write list semaphore), schedule a write request (while holding the transaction list semaphore), and then unblock the blocked user-space write. Since the blocked user-space write has blocked other user-space writes by holding the write transaction semaphore, when it is woken by the kernel thread, it will find space in the write list, and the write will complete. The write list semaphore is also held when modifying the write busy flag. The busy flag tells the user-space write whether it needs to kick-off a write request, and is cleared (while holding the semaphore) by the kernel thread when it empties the write list. The write list semaphore is used to protect the busy flag, as its state changes based on the state of the write list.
Figure 7(b) shows the life-cycle of a read transaction. Although the transaction is shown starting at the receipt of a receiver ready interrupt, it really starts with a call to read that then blocks. If a device is not open for read, then receiver ready interrupts are simply acknowledged, and the data is dropped. So, given a process blocked in a call to read, the read life-cycle is;

1. A receiver ready interrupt occurs, and a read-request work function is scheduled.

2. The read-request work function allocates a transaction request, adds it to the transaction request list, and then schedules a transaction-start work function.

3. When the transaction-start function is scheduled, it removes a request from the transaction list. The length of the transfer is read from SDRAM, and buffer pages are allocated. The read transaction is then performed; if the device uses DMA for transfers, then the DMA controller is programmed to perform the read, otherwise, the CPU performs the transfer from SDRAM.

4. When the transfer is done, the transaction request call-back is called (which schedules a read-done work function), and a transaction-done work function is scheduled. In the case of a DMA transfer, these operations are performed in the DMA interrupt handler, whereas for a CPU transfer, the CPU performs the calls after it has performed the transfer.

5. The read-done work function allocates a buffer entry, converts the pages in the transaction request into a read buffer entry, frees the transaction request, adds the read buffer entry onto the read list, and wakes up blocked readers. If the read list is not full, then the target is sent a transmitter empty acknowledge.

6. The transaction-done work function is called when a transaction completes, it checks whether there is another request in the transaction list, and if there is, schedules a transaction-start work request to continue transaction processing.

7. A buffer entry is removed from the read list. If the read list was full, then the target is sent a transmitter empty acknowledge (since there is now space on the read list).

8. Data is copied from the start of the buffer into the user-space buffer. If the user-space read does not consume the contents of the buffer, then the buffer entry offset is updated.

9. When the user-space read empties the buffer, the buffer entry and pages are freed.

Comments on read locking are now in order. There are three locks of interest; the read semaphore, the read list semaphore, and the transaction list semaphore. The read semaphore is used to protect the read entry. The read entry is an entry that has been removed from the read list, and is in the process of being consumed by user-space reads. The semaphore is held in the driver read, and poll file-operations. The read list and transaction list semaphores are used to protect those lists. When read is called, if the read entry is null, then the read list semaphore is acquired, and held, while an entry is removed from the list. The read list semaphore is also held when the kernel thread adds read data to the read list. The transaction list semaphore is held by the kernel thread when scheduling a read transaction, and when processing transactions.
5.5 The stdio and download devices

The description of the COBRA driver to this point has treated each device as a binary data stream between the host and target. This is not quite correct.

The DSPs used on the COBRA boards use 32-bit little-endian data types for everything; including the (normally 8-bit) character type. The COBRA device structure, cobra_device_t, contains a stdio_device flag that is used to flag a device as being a stdio, or ASCII, device. When a stdio device is written to, the data from the user-space write is unpacked; each byte is unpacked into a 32-bit little-endian word. When a stdio device is read from, when a read-buffer is removed from the read-list it is repacked; the least significant byte of each 32-bit little-endian word is packed into a byte. This packing and unpacking is done on the host to relieve the DSP of the task, since it is performing more critical real-time tasks.

The download device is also slightly different than the other devices. The other devices are communications channels between the host and target. The download device is not. The download device is used only by the host to move DSP boot images into SDRAM, and to move FPGA configuration files into SDRAM. The download device needs to use the DMA controller, as writes to SDRAM are painfully slow (about 20 times slower than using DMA). The main difference between the download device and the other devices is the layout of SDRAM; the host-to-target devices write a length parameter followed by the data, whereas the download device simply writes the data. The other difference is that there is no receiver ready interrupt to trigger a read transaction for a download device. Instead, a read to a download device calls its own read-request work function, which ultimately leads to a read-buffer being added to the read list. The download device also implements the seek function. A user-space program sees the download device as a readable, writeable, seekable 2MB file. Command line tools such as echo, cat, and dd can be used to transfer arbitrary data to and from the 2MB region (this is useful for testing, and generating logic analyzer traces).
5.6 COBRA slot numbering and device naming

The COBRA driver uses the udev scripts to create device names. Each combination of CPU and cPCI crate requires a bus triple-to-slot number mapping file, eg. sza.slots is used for a Force CPU in an SZA crate, whereas carma15.slots is used for the CARMA 15-antenna system that spans two cPCI crates, linked with PCI-to-PCI bridges. The application cobra_slots takes as arguments, the kernel name for a COBRA device, and a slots definition file. The kernel name provides the application with the bus triple and the device number; the bus triple is converted to a slot, and the device number is converted to a name, eg. plx, control, stdio, stderr, monitor, data, and download.

If the COBRA driver is installed without the 20-cobra.rules script in the udev area, the /dev nodes are created with the default kernel names, eg., on a crate containing two boards, the following 14 device nodes are created

```bash
# insmod cobra.ko
# ls /dev/cobra*
/dev/cobra_03:0b.0_0 /dev/cobra_03:0b.0_5 /dev/cobra_03:0c.0_3
/dev/cobra_03:0b.0_1 /dev/cobra_03:0b.0_6 /dev/cobra_03:0c.0_4
/dev/cobra_03:0b.0_2 /dev/cobra_03:0c.0_0 /dev/cobra_03:0c.0_5
/dev/cobra_03:0b.0_3 /dev/cobra_03:0c.0_1 /dev/cobra_03:0c.0_6
/dev/cobra_03:0b.0_4 /dev/cobra_03:0c.0_2
#
```

Removing the driver, placed the rules script into the appropriate udev directory, and reinstalling the driver causes the /dev nodes are created with the following names

```bash
# rmmod cobra
# cp 20-cobra.rules /etc/udev/rules.d/
# insmod cobra.ko
# ls /dev/cobra*
/dev/cobra_control2 /dev/cobra_download3 /dev/cobra_stdio2
/dev/cobra_control3 /dev/cobra_download3 /dev/cobra_stdio3
/dev/cobra_data2 /dev/cobra_monitor2 /dev/cobra_stdio2
/dev/cobra_data3 /dev/cobra_plx2 /dev/cobra_stdio3
/dev/cobra_download2 /dev/cobra_plx3
#
```

The boards are located in slots 2 and 3 of a 19-slot SZA crate.
6 Driver test applications

Several user-space tests were written to test each driver:

- `plx_debug.c` PLX-9054 register control interface
- `cobra_debug.c` COBRA 8MB region control interface
- `cobra_loopback_test.c` write-read loopback test
- `cobra_protocol_test.c` Host-to-target protocol test
- `cobra_dsp_boot_sdram.c` Boot the DSP from an SDRAM image
- `cobra_reset.c` Reset a COBRA board
- `cobra_slots.c udev` name generator

The tests check the majority of the driver functions. The DSP boot tool is used to load DSP interrupt-based, or RTOS-based test applications.

Command line tools such as `echo`, `cat`, and `dd` can be used to test the devices too. For example, assuming the DSP has been booted with the loopback test, the following demonstrates a loopback test on the host for a board in slot 1:

```
# cat /dev/cobra_control1 &
# echo "hello" > /dev/cobra_control1
hello
```

The `cat` command performs a blocking read on the control device, and then `echo` writes to the device. The DSP then loops the write data back to the host, generates a receiver ready, and the driver buffers the read data, and unblocks `cat`, which writes `hello` to standard output. A logic analyzer connected to the PCI bus can be used to view the transaction at the bus level.

References
