# PCI Hardware Support in LIA-2 Control System

P. Cheblakov, D. Bolkhovityanov, BINP, Novosibirsk, Russia

## INTRODUCTION

There are several ways to drive PCI control electronics in Linux. This paper considers a userspace drivers approach.

These drivers communicate with hardware via a small kernel module, which provides access to PCI BARs and to interrupt handling. The module was named USPCI (User-Space PCI access). This approach dramatically simplifies creation of drivers, as opposed to kernel drivers, and provides high reliability (because only a tiny and thoroughly-debugged piece of code runs in kernel).

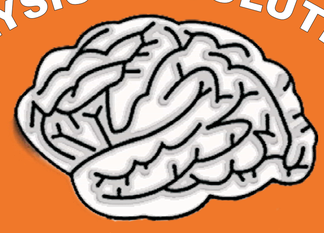
Control system hardware drivers differ dramatically from regular OS drivers in many aspects:

- They implement different model: interaction via "channels" or "records" instead of files, char/block semantics, ioctl()s etc.
- Control system hardware itself, such as DACs, ADCs, timers etc., also has very little in common with regular computer hardware, such as NICs, display cards, serial and SCSI/IDE/SATA ports.
- So, timing requirements and model of operation also differ radically.
- Usually what CS drivers do is read and write hardware memory, plus catch IRQs in a simple manner - what we used to do with CAMAC and VME.

But PCI subsystem in Linux kernel doesn't have a simple way to do it - it is oriented on regular hardware driver requirements.

So, we had to make a way to access PCI hardware in the same manner as CAMAC and VME.

## ANALYSIS OF SOLUTIONS



### *Loadable kernel modules*

It is also known as kernel drivers. A module, running in kernel space, has direct access to PCI and various kernel subsystems. Thus, it can support arbitrary complex hardware, and achieve maximum performance. However, this comes at very high price:

- Development of kernel code is an extremely complex task. Kernel programming requires exceptionally high skills, and debugging such code is very cumbersome.
- Kernel modules are closely tied to internal kernel interfaces, which often change between versions. So, any kernel upgrade requires modification and repeated testing of driver code.
- Similarly, any modifications due to changes in control hardware are also labour-intensive.
- Moving to another *nix platform is almost impossible.

### *UIO: user-space drivers*

UIO is a framework that allows implementing PCI drivers in userspace [3]. User space drivers eliminate most of disadvantages of kernel drivers. Advantages of userspace drivers are:

- you don't have specific kernel drivers for wide range of similar devices;
- much easier to develop and to port to other *nix operation systems;
- fault tolerance.

A tiny kernel-side driver to handle some basic interrupt routine is needed as part of every UIO driver. UIO is just a simple way to create very simple, non-performance critical drivers, which has probably been merged more with a "merge-and-see-if-it-happens-something-interesting" attitude than anything else. For now UIO doesn't allow to create anything but very simple drivers: no DMA, no network and block drivers [4].

### *WinDriver™*

The WinDriver™ [10] product line supports any device, regardless of its silicon vendor, and enables you to focus on your driver's added-value functionality, instead of on the operating system internals.

Its advantages are simple development of drivers and their portability between platforms.

But WinDriver is not free and it is closed-source.

### *Microdrivers*

The Microdrivers [5, 6] architecture was developed at University of Wisconsin-Madison. The main goal of microdrivers is to achieve a better fault tolerance without loose in performance. Microdrivers reduce the amount of driver code running in the kernel by splitting driver functionality between a small kernel-mode component and a larger user-mode component.

Microdrivers seek the middle ground between monolithic kernels and microkernels, and improve reliability while maximizing performance. In a microdriver, the functionality of a device driver is split between a kernel-mode component and a user-mode component. The kernel-mode component contains critical and frequently used functionality, while the user-mode component contains non-critical and infrequently used functionality.

However, this approach is still destined for “usual” devices, not for control system specific hardware. And yet this requires a kernel part for *each* driver.

### *Hardware access layer*

Many major hardware manufacturers implement their own, rather complex, infrastructures to access their control hardware. Usually these include a set of kernel drivers and multi-layer user-space libraries.

Examples include NI-KAL from National Instruments [7], IX-PCI, IX-ISA, IX-PIO from ICP-DAS [8]; COMEDI [9] was an open-source attempt to make manufacturer-agnostic implementation of such system.

Those infrastructures are often just components of larger systems, are usually oriented on specific manufacturer's hardware, and in many cases include closed-source parts.

Thus, this approach looks inadequate for our goals.

## USPCI APPROACH

USPCI (User-Space PCI access) implements approach, similar to WinDriver's: drivers run in user-space, and access hardware via a single tiny kernel module, which implements PCI I/O and simple interrupt management.

The main goal was to minimize drivers' complexity and design time. User-space drivers don't need to implement standard low-level interfaces, as regular OS drivers do. Since most control hardware don't require high throughput and ultra-low, real-time-like interrupt response latency, user-space implementation is quite adequate.

This approach has been widely used in Linux.

- Libusb [11] is a good example: it allows implementing user-space drivers for wide class of USB devices.
- X.org's 2D drivers operate in userspace, thus simplifying support of different unix-like OSes.
- CUPS and programs accessing the serial port like pppd are yet another example of userspace programs accessing the devices directly - the kernel doesn't implement any specific LPT printer or serial modem driver, those userspace programs implement the driver that knows how to talk to the printer.

USPCI is a small and simple kernel module, implementing following functionality:

- /dev/uspci char device;
- access to address space of PCI-device (I/O ports & memory);
- data read and write in any PCI BAR of selected device (regardless of type – memory and I/O ports are accessed in the same way);
- 8-, 16-, 32-bit I/O support (64-bit is implemented, but yet untested);
- interrupt handling, with shared IRQ support (PCI address and mask+condition are specified for IRQ checking);
- interrupt counting;
- interrupt flags accumulation;
- target PCI device can be chosen by either PCI address (BUS:DEVICE.FUNCTION) or device ID (VENDOR:DEVICE:INSTANCE or VENDOR:DEVICE:SERIAL).

### *USPCI API and workflow*

The USPCI module is accessible from user-space via */dev/uspci* char device. Most operations are performed via *ioctl()* interface.

The following steps should be performed first to interact with PCI device:

- Open the */dev/uspci* via *open()* syscall.
- Select target PCI device – either by bus ID, or by vendor:device ID. This is achieved via USPCI_SETPCIID ioctl.
- Optionally specify a method of IRQ checking (required for shared IRQs) and IRQ acknowledgement. USPCI_SET_IRQ does this.
- Optionally specify a list of operations to perform upon close of file descriptor (such as stopping, IRQ masking, etc.) - USPCI_ON_CLOSE.

Than device's resources can be read and written via USPCI_DO_IO ioctl; very similar to how NAF is done in CAMAC or bus I/O is performed in VME. Usually all these *ioctl()*s are wrapped by a more friendly library.

In order to “finish” interrupt processing the USPCI_FGT_IRQ is called, which returns interrupt count and accumulated interrupt flags; these are reset afterwards.

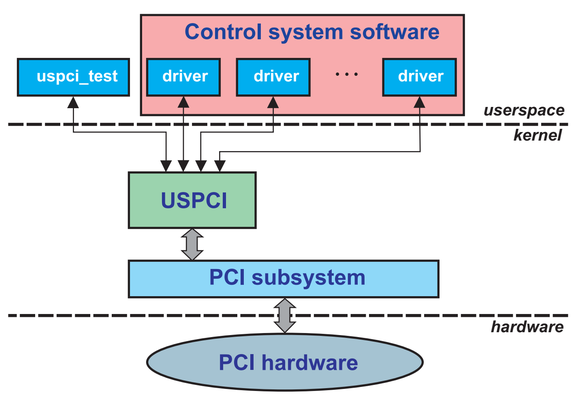
USPCI implements “poll” interface, which allows using file descriptor, associated with device, in *select()* and *poll()* syscalls.

To finish interaction with device, its file descriptor must be *close()*'d. Since *nix closes all file descriptors upon program termination, this, together with ON_CLOSE specification, enables to leave device in a safe state in all cases, including user-space driver crash.

Since USPCI kernel module is simple and short, it was easy to test and is stable.

*uspci_test*

Command-line interface program was developed for the purpose of device debugging. This program provides an access to all USPCI functions and logic. The use of the program allows analysing of device operation at development of userspace drivers.




Block-diagram of USPCI functionality.

## DEPLOYMENT

The considered driver was implemented successfully on LIA-2 [1] linear accelerator at Budker INP.

LIA-2 control system is based on cPCI crates with x86-compatible CPU boards running Linux (CentOS-5.2 with custom-built 2.6.25 kernel. Slow control electronics (modulator controllers, slow DACs/ADCs) is connected via CAN, while fast hardware – such as 4MHz and 200MHz ADCs and 200MHz timers – are cPCI/PMC boards.

LIA-2 control system software is based on CX [2]. Being highly modular, CX doesn't include CAN and PCI support “out of the box”. So, PCI hardware support had to be created from scratch.

USPCI Linux kernel allows developing of userspace drivers that is an optimal approach for control system building.

The drivers for following BINP-made devices were developed with the use of USPCI: timer DL 200 ME Fast, timer DL 200 ME Slow and ADC 200 ME. These drivers are successfully implemented in CX control system for LIA-2. The driver for PISO-ENCODER600 of ICP-DAS Co., Ltd production was also developed [12].

## REFERENCES

[1] P. Logachev, A. Akimov, P. Bak et al., “Perfomance of 2 MeV, 2 ka, 200 ns linear induction accelerator with Ultra low beam emittance for X-ray flash radiography”, IPAC'11, Kursaal, San Sebastian, Spain, September 2011, WEoAA02

[2] D.Yu.Bolkhovityanov, A.Yu.Antonov, R.E.Kuskov, “Present Status of VEPP-5 Control System”, PCaPAC2006, Newport News, VA, USA

[3] UIO: user-space drivers, http://lwn.net/Articles/ 232575/

[4] Linux 2 6 23, http://kernelnewbies.org/Linux_2_6_23

[5] Vinod Ganapathy, Arini Balakrishnan, Michael M. Swift, Somesh Jha, “Microdrivers: A New Architecture for Device Drivers”, HotOS XI, San Diego, California, USA, May 2007

[6] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, Somesh JhaThe, “Design and Implementation of Microdrivers”, ASPLOS XIII, Seattle, WA, USA, March 2008

[7] http://joule.ni.com/nidu/cds/view/p/id/2372/lang/en

[8] http://www.icpdas.com/download/linux.htm

[9] COMEDI, Linux control and measuremnt device interface http://www.comedi.org/

[10] WinDriver™, http://www.jungo.com/

[11] libusb, www.libusb.org/

[12] PISO-ENCODER600, PCI Bus, 6-axis Encoder Input Card, http://www.icpdas.com/

Paper ID: WEPMN017

Presented by Pavel Cheblakov: P.B.Cheblakov@inp.nsk.su