Development and Implementation of a Linux- Xenomai based Hard Real-Time Device Driver for PCI Data Acquisition System (DAS) Card

Amit Bhaira Gaurav Rudra Tanmoy Bag Yusuf Husainy
Advanced Computing Training School (ACTS),
Centre for Development of Advanced Computing (CDAC), Pune

ABSTRACT
This paper elaborates the implementation of a PCI based device driver for a Data Acquisition System (DAS) Card using the RTDM (Real-Time Driver Model) skin over Xenomai kernel, which is integrated with the Linux kernel. A C language kernel module was written for the PCI bus based driver to provide hard real-time capabilities and determinism to any application accessing the DAS card. The PCI DAS card used consisted of 12-bit ADC, 12-bit DAC, Programmable Digital I/O lines (TTL compatible) and Timers/Counters. In order to test all the features of the DAS and the performance of the driver, a test system, consisting of a 3-axis analog accelerometer connected to the ADC of the DAS via a junction box and powered by its DAC, was been constructed. Additionally, a 3-axis digital accelerometer communicated with an AVR development board via I2C in order to generate conditioned input for the programmable digital I/O lines of the DAS card. A graphical tilt measurement application involving real time acquisition of the accelerometers data was implemented using OpenGL. Finally, the driver was thoroughly tested with this arrangement, and the interrupt latencies were noted to be around 4µsec.

General Terms
Data Acquisition Systems, Real-Time, Xenomai, PCI, RTDM.

Keywords
Real-Time PCI Device Driver, Xenomai, RTDM, Data Acquisition Systems

1. INTRODUCTION
There are numerous research laboratories and work places which are working on Data Acquisition Systems and most of the work is on PCs. Thus, these high speed PCI devices find their utility in such scenarios and this involves these real-time development frameworks into the picture. This device driver is responsible for the communication between the DAS hardware and the user space applications, in Xenomai environment, in hard real time. The application is provided with certain routines to configure the device as per requirement.

1.1 Data Acquisition Systems
Data acquisition, sometimes abbreviated as **DAQ** or **DAS**, typically involves sampling of the real world analog signals and waveforms and then processing them accordingly to make them suitable for digital computation.

The DAS hardware is responsible for interfacing the various signals to PC. It can be in the form of components that can be connected to the computer's ports (parallel, serial, USB, etc.) or cards that can be plugged into the slots (PCI, ISA, PCI-E, etc.) available in the mother board. The DAS application software accesses the input values from the hardware with the help of the device driver. This driver software allows the operating system to recognize the DAS hardware and conditions the incoming signals to serve the user space DAS applications .

1.2 Real-Time Driver Model (RTDM)
The Real-Time Driver Model (RTDM) [1][2] is a methodology to integrate the offered interfaces for developing device drivers and associated applications under Xenomai. It is intended to act as an intermediary between the application requesting a service from a certain device and the device driver offering it. It supports two different types of devices, namely Protocol devices (All message-oriented devices fall in this group) and Named devices (Devices registered with the real-time subsystem under a unique clear-text name).

1.3 Peripheral Component Interconnect
Peripheral Component Interconnect (PCI) [3][4], as its name implies is a characteristic specification that defines how to connect the various peripheral components of a system together meticulously in an organized fashion. The standard elaborates the behavior of the system components and the way they are electrically connected

PCI devices are automatically configured at boot time and the OS then allocates the resources and provides the allocation information to each device. Then, the device driver must be able to access configuration information in the device in order to complete initialization. Thus, this characteristic of auto-detection of PCI cards or interface boards turns out to be the most relevant subject to a PCI driver developer.

1.4 Device Drivers
A device driver [6] is a suite of kernel routines that drive a hardware device with the help of the programming interfaces defined by the canonical set of Virtual File System (VFS) functions (open, read, lseek, ioctl etc.). The computational loads levied by device drivers tend to be erratic as they are elicited by the events that occur in the device and may arbitrarily block or preempt other time-critical tasks. This attribute poses significant challenges in real-time systems, where schedulability analysis is crucial for ensuring real-time scheduling and achieving deterministic timing requirements without neglecting the device driver workloads at the same time.

2. XENOMAI

Xenomai [7] is a development framework, providing generic RTOS services, which can be patched seamlessly with the Linux kernel, to adhere to numerous RTOS environments named as *skins* placed over the nucleus in order to add to the hard real time capabilities of the Linux Operating System. In this implementation, the RTDM skin has been used.

The Adeos/ I-Pipe allows Xenomai nucleus to handle all incoming interrupts first, before the Linux kernel has had the opportunity to notice them, and guarantees enforcement of proper priority management for its threads, regardless of their current execution domain. The Fig: 1 shows how the Nucleus provides real-time support to the user-space and kernel-space modules.

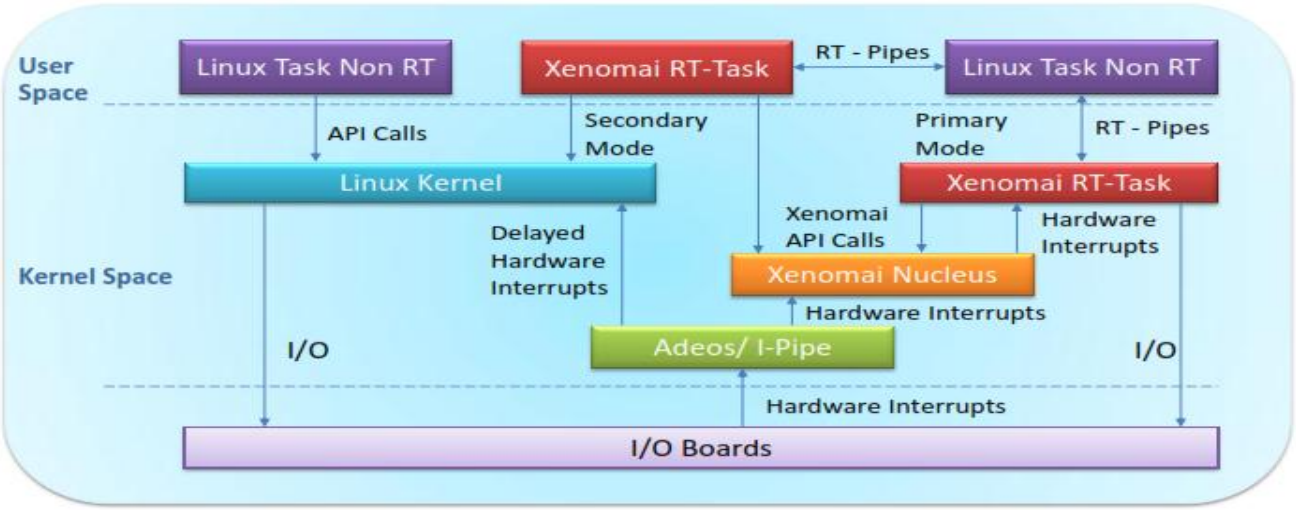

Fig 1: Xenomai kernel co-operating with Linux kernel

3. SOFTWARE ARCHITECTURE

This PCI driver handles multiple device instances and maintains the list of devices handled by the driver. A private object is maintained for each device to provide reentrancy. The driver accesses the configuration space [8] of the device to obtain the information about devices resources.

The user space application software comprised of five tasks for writing DAC, reading ADC, setting timer, programmable digital I/O and for the graphical display, which used dedicated interfaces defined in the driver for writing DAC, reading ADC and sending ioctl commands for setting ADC modes, DIO modes, Timer modes, updating data registers etc.

Routines were written for initializing, exiting, probing and removing the module. An interrupt handler [9] was also developed for the all the interrupts arriving on the irq line designated to the PCI card.

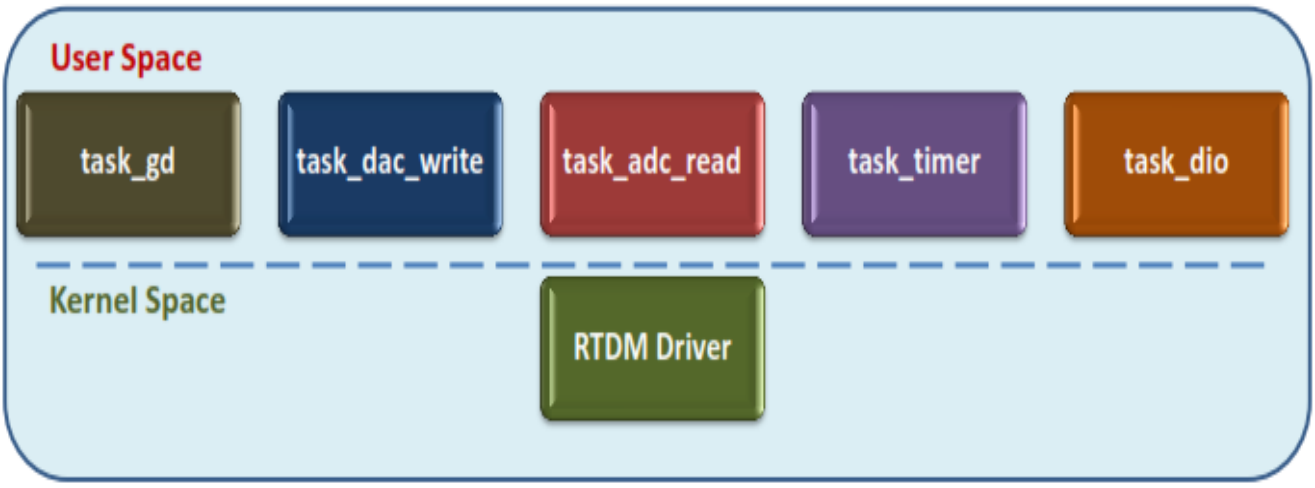

Fig 2: Layered Schematic of the software

4. IMPLEMENTATION AND TESTING

4.1 Implementation Overview

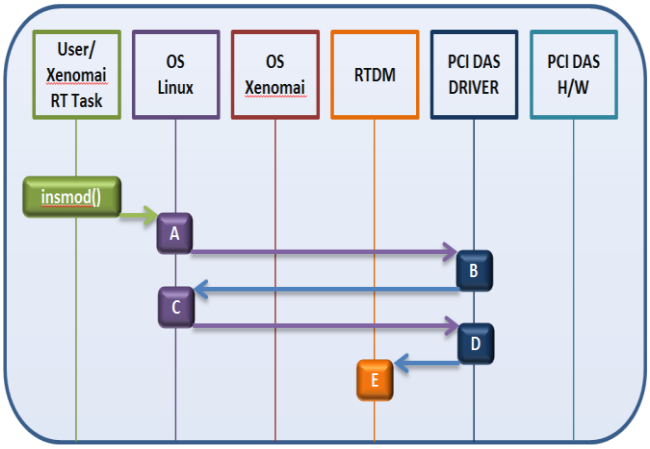

Fig 3: Sequence Diagram for Driver Module Insertion and Device Register

The Fig. 3 elaborates the sequence of steps followed to load the driver module, register and probe the device[9] and allocate resources accordingly. The blocks in the sequence diagram, with the corresponding alphabet tags, are described as follows:

insmod()

A. Loads the driver and calls the init() routine.

B. Calls pci_driver_register() to register itself with the Linux PCI System.

C. Gets driver info supplied by the driver, and starts searching for device (which was enumerated at boot time) with matching credentials. If found, calls driver probe() method by passing pci_dev info to it.

D. Enables the PCI device, requests for appropriate "BAR" region of configuration space and then registers the device with the RTDM using rtdm_dev_register(). While registering, it provides the addresses of the rt VFS functions for read(), write() and iocti() and nrt functions of open() and close().

E. RTDM maintains the addresses, and any future calls to these driver routines from the user/fs will be routed to the appropriate driver routine.

The Fig. 4 describes the sequential flow for the open(), ioctl() and read() routines through the various layers with the help of an instance from the driver i.e. ADC read. The blocks in the sequence diagram, with the corresponding alphabet tags, are described as follows:

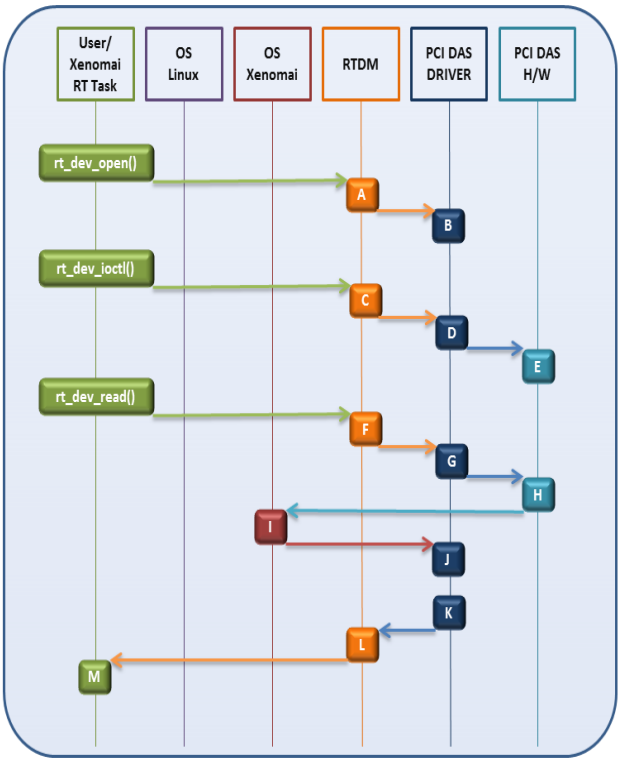

Fig 4: Sequence Diagram for Open, Read and Ioctl routines

rt_dev_open()

A. Calls driver's open() method.

B. Does some basic h/w initialization and registers ISR for DAS Card.

rt_dev_ioctl()

C. Calls driver's ioctl() method.

D. Initializes ADC for use.

E. Required ADC mode and ADC channel is set.

rt_dev_read()

F. Calls driver's read() method.

G. Enables PCI interrupt, issues start conversion request to ADC, and then waits for hardware interrupt.

H. Starts conversion of accelerometer data, issues end of conversion signal which will generate interrupt.

I. Xenomai will call driver's ISR.

J. Copies data from ADC to local buffer and signals event on which read() was waiting.

K. In driver's read() method, after ISR signals event, read() will come out of wait state and ADC data is copied from local buffer to user buffer via RTDM by calling rtdm_safe_copy_to_user().

L. RTDM will ensure read/write to user-space buffer is safe or not, then only will copy ADC data to user-space.

M. User-space application gets ADC data, computes tilt and displays it graphically.

This explanation can be applied for other methods like exit(), remove(), write() etc. as well.

4.2 Test System

Components:

- DAS Card
- Junction box
- 3-axis analog and digital accelerometer
- Connecting wires.
- AVR development board

The **PCI DAS** card used consisted of 12-bit ADC, 12-bit DAC, Programmable Digital I/O lines (TTL compatible) and Timers/Counters. Thus, to test all its features and the corresponding routines of the driver, 3-axis analog and digital accelerometers were used.

The x, y and z axis values of the analog accelerometer were supplied to three channels of the ADC of the DAS card through a junction box. The driver provides these values to the user space application which computes the tilt [10][11] on x and y axes, and displays them as an animated graphical scene.

Similarly the digital accelerometer communicates with the AVR development board through I2C and sends the desired x, y and z axis values to the DAS through the programmable digital I/O lines which in turn is used by the same user-space application.

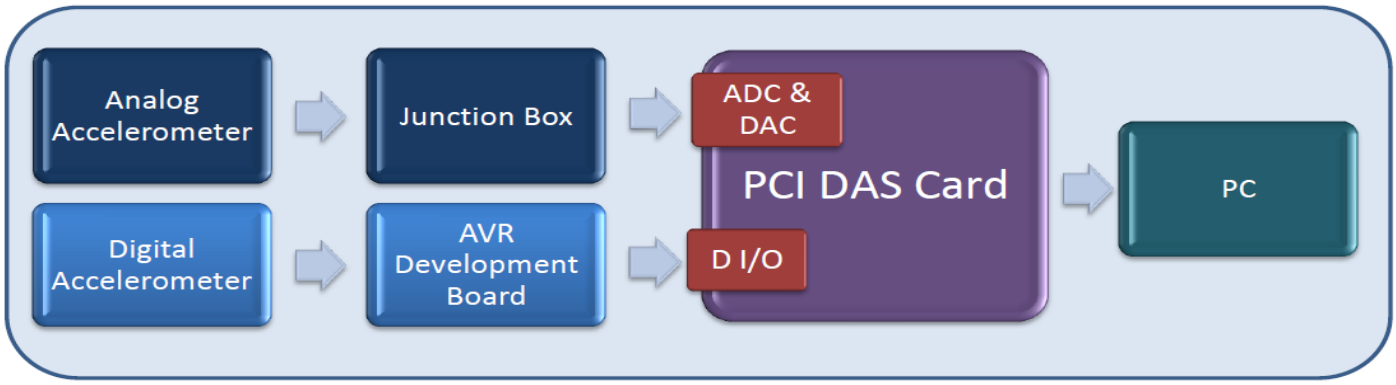

Fig 5: Schematic diagram of the Test System Implementation

4.3 Observations

It was observed that the real-time performance of the driver was excellent with a deterministic interrupt latency of around **4µsec**.

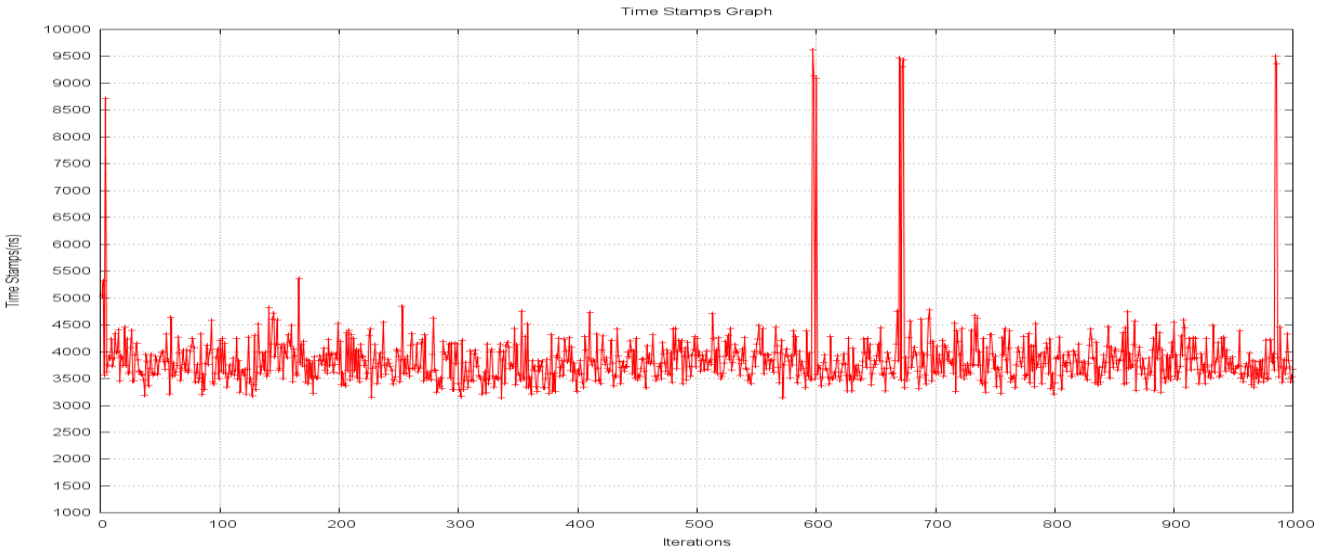

Fig 6: Interrupt latency results of the kernel module

4.4 Challenges Faced

Some of the major challenges that were confronted and addressed during the experiment have been listed below:-

1. The vender id and the device id were required for registering the PCI device, which was not specified by the manufacturer. This was addressed by locating this information from the configuration space or at the following locations of the kernel:
   /sys/bus/pci/devices/0000:04:00.0/vendor
   /sys/bus/pci/devices/0000:04:00.0/device

2. In order to register the interrupt handler (using *rt_request_irq*), the interrupt line number (IRQ number) was a prerequisite. However, different values of IRQ number were observed, from the Configuration space and the PCI device structure that is supplied by the system. Then according to the kernel documentations available, it was concluded that memory (MMIO), and I/O port addresses should NOT be read directly from the PCI device configuration space. Instead the values in the *pci_dev* [9] structure should be used, as the PCI "bus address" might have been remapped to a "host physical" address by the arch/chip-set specific kernel support. Thus the confusion was resolved and the IRQ number provided by the system through *pci_dev* structure was used to install the interrupt handler.

3. Moreover, the IRQ line was shared with a Linux device so the Linux device had to be disabled from the BIOS settings.

5. CONCLUSION

The paper provides an insight to develop a PCI device driver for Linux- Xenomai with a splendid real- time performance, for systems involving acquisition of data through PC. The implementation discussed in the paper would be helpful for the readers, not only to work with the RTDM skin but would also provide a picture to experiment similarly with the other existing skins over Xenomai. And as today most Scientists and Engineers are using personal computers for laboratory research, industrial control and test and measurement, such intricate PCI kernel modules built on open source platforms would prove a lot beneficial.

6. ACKNOWLEDGEMENT

This work would never have been so successful without the supervision, technical guidance, suggestions and critical comments of Mr. Babu Krishnamurthy. Sincere gratitude to him. Earnest appreciation and tribute to Mr. Rajesh Sola and Ms. Bhawwna Aggarwal for all their assistance and support. Lastly, a mention to the various internet forums and the kernel documentations provided by Linux and Xenomai for their valuable contributions throughout the research process.

7. REFERENCES

[1] Xenomai documentation at www.xenomai.org.

[2] J. Kiszka, “The Real-Time Driver Model and First Applications”, svn.gna.org.

[3] Doug Abbott, “PCI Bus Demystified”, Elsevier, 2004.

[4] “PCI 9052 Data Book”, PLX Technology, 2000.

[5] Daniel P. Bovet, Marco Cesati, “Understanding the Linux Kernel”, O’Reilly, 2006

[6] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, “Linux Device Drivers”, O’Reilly 2005.

[7] P. Gerum, “Xenomai – Implementing a RTOS emulation framework on GNU/Linux”, 2004, 2008.

[8] Don Anderson, Tom Shanley, “PCI System Architecture”, Mindshare, 1999

[9] Robert Love, “Linux Kernel Development”, Pearson Education, 2010.

[10] Mark Pedley, AN3461 Application Note, “Tilt Sensing Using a Three-Axis Acclerometer”, Freescale Semiconductor, 2013.

[11] AN3182 Application Note “Tilt measurement using a low-g 3-axis accelerometer”, ST Microelectronics, 2010.