Development of Real Time Linux Device Driver for PCI based Data Acquisition System

Jaimin Thakkar1
1M E (VLSI and Embedded Systems Design)
1Electronics and Communications Engineering
1Gujarat Technological University

Abstract---This paper elaborates development and implementation of hard real time device driver for PCI based data acquisition of analogical and digital data. Device driver was developed using the RTDM (Real-Time Driver Model) skin over Xenomai kernel. The system consists of two PCI based cards. First card consisted of 12-bit ADC, 12-bit DAC, Programmable Digital I/O lines (TTL compatible) and Timers/Counters. Second card consisted of Programmable Digital I/O lines and Timers/Counters only. In order to test all the features and performance, a test system consist of temperature sensors and output of DAC was connected to channels of ADC. Switches are connected to second card in order to generate conditioned input for the programmable digital I/O lines of the DIOT (Digital Input/output Timer card). Finally, both drivers were tested with this arrangement and with multiple application accessing drivers at same time, and the interrupt latencies were noted to be around 5μsec.

Keywords: RTDM, Real-Time Linux PCI Device Driver, Xenomai.

I. INTRODUCTION

The aim of this work is to develop deterministic data acquisition system with real time performance for time critical process such as nuclear reactor where data needs to be collected form lots of sensors. Linux can’t process Real Time Data with this type of heavy load condition. While Linux is not a real-time system, Xenomai environment is used to develop Real Time Data acquisition System. ESA PCI DAS (electro system associates peripheral component interconnect data acquisition system) and DIOT (Digital Input/output Timer system) cards are used to interfaces between the signal and a PC in this project. DAQ device driver is needed in order for PCI DAS card to work with a PC. The device driver performs low-level register writes and reads on the hardware, while exposing a standard API for developing user applications.

A. Data Acquisition Systems:

ESA PCI DAS (electro system associates peripheral component interconnect data acquisition system) is used to interfaces between the signal and a PC in this project. ESA PCI DAS (Data Acquisition System) is a high speed PCI bus based data acquisition add-on card for PC. It upgrades PC compatible computer system to high-speed data acquisition and process control workstation. Data acquisition is the process of sampling signals that measure real world physical conditions and converting the resulting samples into digital numeric values that can be manipulated by a computer.

B. Need of Real Time Data Acquisition System:

C. Large scale data processing and controlling systems are needed to store, access, retrieve, distribute, and process data at large facilities (like a nuclear reactor in Nuclear Power plant). The processes at such facilities are extremely complex; involving lots of detector elements that produce raw experimental data at rates up to a GB/sec. Action (or control) must be taken after getting data. Because of controlling tasks have time dead line; system must be deterministic to guarantee timing behaviour in the face of varying loads (from minimal to worst case). For solution of above problem we need Hard Real Time OS for deterministic performance which can be achieved by Xenomai.

II. DEVICE DRIVER

Driver software provides application software the ability to interact with a DAQ device. It simplifies communication with the DAQ device by performing low-level register writes and reads on the hardware. Typically, DAQ driver software exposes an application programming interface (API) that is used within a programming environment to build application software.

A Linux device driver is the set of kernel routines that makes a hardware device respond to the programming interface defined by the canonical set of VFS functions (open, read, lseek, ioctl, and so forth) that control a device. The actual implementation of all these functions is delegated to the device driver. Because each device has a different I/O controller, and thus different commands and different state information, most I/O devices have their own drivers.

The computational workloads imposed by normal device drivers tend to be aperiodic and unpredictable because they are triggered in response to events that occur in the device, and may arbitrarily block or preempt other time-critical tasks. Normal Linux task for data acquisition may not work deterministically under heavy load as described above. For solution of above problem we need Hard Real Time OS for deterministic performance which can be achieved by Xenomai.

III. XENOAMI

Xenomai is a real-time development framework cooperating with the Linux kernel, in order to provide a pervasive, interface-agnostic, hard real-time support to user-space applications, seamlessly integrated into the GNU/Linux environment.
RTDM layer of Xenomai is used for this system for real time driver development. Real-Time Driver Model (RTDM) is an approach to unify the interfaces for developing device drivers and associated applications under Xenomai. The RTDM is intended to act as a mediator between the application requesting a service from a certain device and the device driver offering it.

IV. LAYOUT OF THE DRIVER

As this driver is to be added to the running kernel dynamically, it has to be designed in form of a module. After compilation a kernel object will be created (.o file) which can be added using insmod utility. Insmod will call the module’s init routine. Following are the routines implemented in driver:

A. Module_Init: In this routine, the driver is registered using pci_register_driver with a pointer to a structure describing the driver (struct pci_driver). pci_register_driver () leaves most of the probing for devices to the PCI layer and supports online insertion/removal of devices (thus supporting hot pluggable PCI, CardBus, and Express-Card in a single driver). pci_register_driver () call requires passing in a table of function pointers and thus dictates the high level structure of a driver. Real time (RTDM) devices are created using rtdm_dev_register(), one each for ADC, DAC, Timer and DIO.

B. Module_Exit: This routine is the counterpart of the init routine. Whatever allocation was done in init is de-allocated here. Then the driver structure is de-allocated. MMIO/IOP resources released. One important point to note regarding the RTDM device is that the function rtdm_dev_unregister () is needed to be called four times (for this system). Once for every device created.

C. Probe Method: This probing function gets called (during execution of pci_register_driver () for already existing devices or later if a new device gets inserted) for all PCI devices which match the ID table and are not "owned" by the other drivers yet. This function gets passed a "struct pci_dev *" for each device whose entry in the ID table matches the device. The probe function returns zero when the driver chooses to take “ownership” of the device or an error code (negative number) otherwise. Once the driver knows about a PCI device and takes ownership, the driver generally needs to perform the following initialization in probe method:

  - The device is enabled (using pci_enable_device ()).
  - Request MMIO/IOP resources for the specified device (usingpci_request_regions)
  - Required information is read from configuration space of PCI device and then stored into the private object of the device. pci_(read|write)_config_(byte|word|dword) used to access the config space of a device represented by struct pci_dev *.

D. Remove Method: The remove () function gets called whenever a device being handled by this driver is removed (either during deregistration of the driver (rmmod utility) or when it’s manually pulled out of a hot-pluggable slot). Following things needs to be done in remove method:

  - Release MMIO/IOP resources using pci_release_regions ()
  - Disable the device using pci_disable_device ().

E. Open Method: This method is common for all four devices.

F. Four Write Methods: Four write methods are implemented for each device (ADC, DIO, DAC, and Timer).

G. Four Read Methods: Four read methods are implemented for each device.

H. Two IOCTL Methods: One IOCTL method for ADC and another for DIO. IOCTL method is used for operations like setting ADC mode, get ADC status and start conversion of ADC.

1) Following Are The Data Structures Are Used In Device Driver:

   a) Struct Pci_Driver: This structure describes the pci driver. The pointer to this structure is required during pci driver registration. This structure contains pointers to id_table, probe method and remove method.

   b) Struct Rtdm_Device: Four rtdm device structures used to represent four devices ADC, DAC, DIO and Timer. In the rtdm_device structure, there is an entry for each of the interface routines the driver implements, such as open_rt() and open_rnt(). As their suffixes tell us, those routines are invoked by RTDM depending on the calling context.

   c) Struct Pci_Device_Id: The ID table is an array of struct pci_device_id entries ending with an all-zero entry; use of the macro DEFINE_PCI_DEVICE_TABLE is the preferred method of declaring the table but customized macro is used. Each entry consists of:

      - Vendor and device ID to match
      - Subsystem vendor and device ID to match
      - Device class, subclass, and "interface" to match.
      - driver_data: Data private to the driver.

Four private structures for individual Device were used to manage data of individual devices.

V. CHALLENGES FACED

A. Re-Entrancy: Calling a function/method of the driver again from another real time task while the same is still under run creates re-entrancy problem. It means that calling same function of driver from two different task create data corruption problem in driver.

As a result, driver code, must be re-entrant—it must be capable of running in more than one context at the same
time. Data structures carefully designed to keep multiple threads of execution separate, and the code will take care to access shared data in ways that prevent corruption of the data. Re-entrant RTDM driver can be written by keeping all data in device context structure which is associated with every open device instance. RTDM takes care of its creation and destruction and passes it to the operation handler when being invoked. The size of this data is provided via rtdm_device.context_size during device registration.

B. Shared Interrupts: In this system interrupt of PCI DAS card and internal PCI based USB system was the same. So sharing of interrupt between Linux and Xenomai creates performance issues. This problem can be solved by using APIs of Adeos layer. Adeos layer \(^{[14]}\) is used in the Xenomai architecture. 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.

Adeos guarantees that no stack overflow can occur due to interrupts piling up over any given domain and it also stalls the current stage before firing an interrupt handler. Driver should enable re-enable it using rthal_irq_enable(), so that further occurrences can be immediately logged, and will get played immediately after the current handler invocation returns.

So, to get real time performance interrupt handler of driver was changed like this way:

```c
/*
   This interrupt handler has been installed using rthal_irq_request(), so it will always be invoked on behalf of Xenomai (primary) domain.
*/
if (check status of Interrupt status)
{
    //do work
    //clear Interrupt status bit in //Interrupt control register

    rthal_irq_enable(irq);
    rthal_irq_host_pend(irq);
    /* this irq has been marked as pending for Linux */
}
void linux_handler()
{
    /*
    This interrupt handler has been installed using rthal_irq_host_request(), so it will always be invoked on behalf of the Linux (secondary) domain, as a shared interrupt handler (Linux-wise).
    */
    /* process the IRQ normally. */
}
```

VI. PERFORMANCE

Analog sensors, LEDs and input switch are connected with PCI DAS and DIOT cards for measuring performance of real time driver. The benchmarking tests are needed to prove the determinism and low latency of real time driver over a non-real time driver. The benchmarking test includes calculation of following parameters.

A. Interrupt Latency: The difference in time between the generation of an interrupt and execution of an ISR is called as interrupt latency. Difference between point A and C in Fig. 2 gives total Interrupt latency. ADC generate the interrupt after every from analog to digital conversion. A deterministic interrupt latency of this system was around 5μsec.

B. Event Latency: The difference of time at which event is signal led and time at which task (which was blocked on that event) is scheduled is called as event/scheduling latency. In Fig. 2 difference between D and E point gives Event latency. After getting interrupt from ADC, waking of waited event was done in interrupt handler. Real time Event was used to wake up real time task waiting for ADC reading. Event latency was measured and it was around 4μsec.

C. Overall Performance: In this system ADC on PCI DAS card takes 10 μsec for conversion of analog to digital system\(^{[7]}\). So combining this hardware latency with interrupt and event latency gives total latency of around 20μsec. So using this real time driver this system can take 50000 samples per second form ADC.

![Fig. 2: Latency Measurement](image)

VII. CONCLUSION

So we can conclude that deterministic and high throughput (50000 samples per second from ADC) for Real Time Data acquisition system using PCI DAS and DIOT card can be achieved by writing RTDM driver and real time task under Xenomai environment. The implementation discussed in the paper would be helpful for the readers, not only to work with the RTDM skin but would also provide better way to solve challenges similar to which is discussed in this paper.

VIII. ACKNOWLEDGMENT

This work would not have been successful without the kind support and help of many individuals and organizations. I am highly indebted to Mr. Rajesh Sola for their guidance and constant supervision. I am also thankful to Mr. Babu Krishnamurthy for giving suggestions and critical comments. I thank the Gujarat Technological University and CDAC for providing necessary resources regarding dissertation work.

REFERENCES

Conference on Electrical Engineering and Informatics
17-19 July 2011, Bandung, Indonesia


[9] pci.txt from kernel documentation found under<ksrc>/Documentation/PCI/pci.txt


All rights reserved by www.ijsrd.com 515