Software developments for the Readout Unit Prototypes for CMS DAQ System

Bellato, M. (INFN Sezione di Padova)
Antchev, G.; Cano, E.; Cittolin, S.;Faure, B.; Gigi, D.; Gutleber, J.; Jacobs, C.; Meijers, F.; Meschi, E.; Orsini, L.; Pollet, L.; Racz, A.; Samyn, D.; Schleifer, W.; Sphicas, P (CERN)
Sinanis, N. J. (ETH, Zürich)
Erhan, S. (University of California, Los Angeles)
Ninane, A. (Université Catholique de Louvain)

CERN, 1211 Geneva 23, Switzerland
marco.bellato@pd.infn.it, eric.cano@cern.ch, alain.ninane@fynu.ucl.ac.be

Abstract

In the CMS data acquisition system, the readout unit is a fast buffering device for short-term storage of event fragments. It interfaces front-end devices and builder data network.

The current Readout Unit prototypes are based on two homegrown hardware boards, the Readout Unit Memory (RUM) and the Readout Unit I/O (RUIO). These boards are equipped with an input/output processor (IOP). Several OS environments for this processor are developed. The software running on those boards will have to setup and control the input and output processes. Fast IOP to host communications are experimented. A software test environment is specifically designed for test and validation of the complex memory management of the RUM.

I. THE HARDWARE ENVIRONMENT

A. Readout in the CMS data acquisition model

In the CMS data acquisition model, the Readout Unit (RU) temporarily stores data from a detector subpart, before forwarding it through the Builder Network to the Builder Unit. Each Builder Unit assembles event fragments for the Filter Units, which are in charge of processing events for Level 2 triggers and of forwarding events that passed Level 2 triggers to the computing services (see Figure 1).

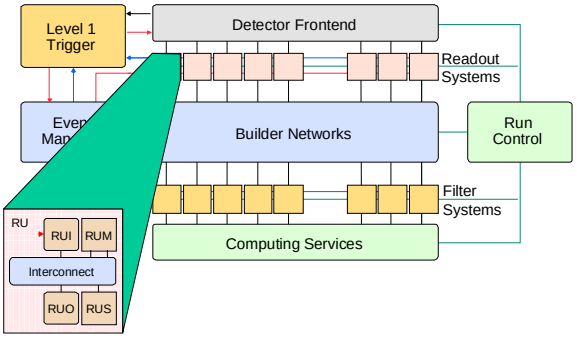

Figure 1 : CMS Data acquisition model

Each RU receives a fragment from each event that passed Level 1 trigger. Those fragments are stored temporarily in the RU. They will be forwarded to Builder Units on request, and the full event (which spans across multiple RUs) will be processed by the Filter Units, as defined in [1].

In order to be able to sustain a full link speed on both input and output at the same time, special hardware prototypes have been designed.

B. The Readout Unit IO and Readout Unit Memory boards

The Readout Unit Memory (RUM) and the Readout Unit Input Output (RUIO) are long sized PCI boards [2]. Those boards provide the Readout Unit with fast input and output capabilities. The RUM and two RUIOs can be connected together, forming a RU (interfaced through PCI links on the PCI input and output buses).

RUIO and RUM both contain a I/O processor from PLX (IOP 480). This processor is on a separate PCI bus, with an Ethernet interface.

The RU provides full bus speed on both input and output due to a design with PCI to PCI bridges and special memory management hardware. See Figure 2, [3], [4] and [5].

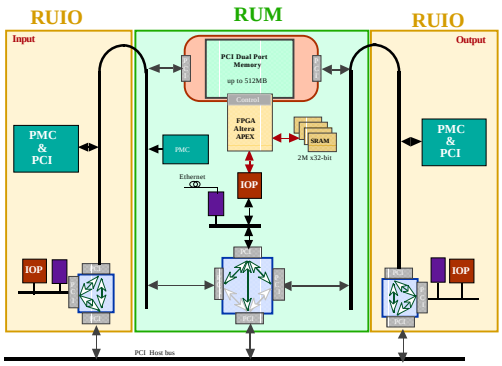

Figure 2 : Full RU structure

Currently the RUIO has been fully developed, and tested. The first prototype of the RUM is still in development.

RUM's memory management unit (MMU) will provide fast buffering between front end and builder network. It stores data from several events on its input (connected to detector front end), and delivers them on demand to the builder units through the builder network.

C. *Host environment*

The RUIO and RUM are 64 bits/66 MHz capable boards, but currently they only have been tested on 32 bits/33 MHz environments. The hosts used for the tests were Macintosh G3 and standard PCs.

II. OS DEVELOPMENTS

For exploiting the IOP RUIO, we needed an operating system to run on it. One of them, VxWorks is fully developed for the RUIO, and the other, Linux is still in progress.

A. *VxWorks*

To ease the monitoring and control of I/O and management processes taking part on the RUM and RUIO boards, a real-time OS has been ported for the PLX IOP480 processor. This is a highly integrated device comprising a PowerPC 401 core, a memory controller, a PCI bridge, an UART, two DMA controllers, an I_2O controller and the arbiters for both PCI and local bus. The RUM and RUIO also accommodate some RAM for OS operation and fast Ethernet controller for network connection.

A well-known real-time OS, VxWorks from Wind River Inc., has been chosen for a number of features, among which its deterministic high speed microkernel, scalability, comprehensive C++ and POSIX support, and a fully layered approach which make it processor and bus independent. The porting process is therefore limited to the design of a Board Support Package (BSP) that interfaces custom hardware to upper OS software layers.

The tasks of the BSP are handling of the processor boot sequence (processor init, memory controllers configuration, registers mapping, eprom handling, UART init, and stack init), written in assembly code; and configuraiton of all I/O devices (PCI bridge, memory maps and ethernet controller). A library of routines that map OS system calls to underlying hardware (interrupt dispatcher, timers and system clock handling, Flash memory and PCI access) is also part of the BSP. A set of software drivers (for the Ethernet chip, UART and Flash Ram) are then needed for allowing TCP/IP connectivity, Serial port operation and OS boot parameters storage and readout.

The BSP has been tested, debugged and validated on the RUIO hosted on a MAC G3 and fully exploits local and remote VxWorks functionality.

B. *Linux*

Linux is an operating system which is gaining more and more importance in the field of scientific research, as well as in industry. It is made of a kernel, standard Unix applications and tools, which vary according to the Linux distribution. Being ported to a wide variety of microprocessors: among others, Intel Pentium, Digital Alpha, SUN Sparc, IBM PowerPC, ... , Linux is available on a wide range of general-purpose platforms.

The wide availability and open-source characteristics of the Linux kernel makes it also a first choice candidate for embedded systems where application-specific components are bussed around a microprocessor. However, although the Linux kernel is standardized, the drawback of the open-source characteristic of Linux is that there is not yet an authoritative procedure to integrate new hardware and platforms.

The task of porting Linux to the RUIO therefore consists first of finding an already existing port of the kernel to a microprocessor close to the PowerPC 401 core. We have found that the PowerPC 403GCX on the IBM Oak evaluation board, where the Linux kernel 2.2.14 has been already ported, is a good starting point. The second task consists then of adapting the bootstrap process: it uncompress the Linux kernel from some area in the RUIO memory, where it has been previosuly uploaded by the host, and starts it.

The very first moments of the Linux kernel are executed in real memory until virtual memory mapping is enabled. At this stage, it remains to start all low-level software like the interrupts handlers, timers, scheduler, and serial line driver, ...

This work is still in progress: we are currently working on the port of the Tulip Ethernet controller of the RUIO. When this will be up-and-ready, the RUIO will mount a root file system populated with already existing PowerPC utilities from a network.

III. SOFTWARE DEVELOPMENTS

The exact implementation of the Readout Unit is not yet fully defined. In order to be able to experiment with different layouts of RU (see Figure 3) with parts of hardware and parts of software, we develop communication schemes between host and IOP, sharing the same bus. A good candidate for this protocol is the I_2O message-passing scheme, as the IOP480 implements all the necessary registers for this.

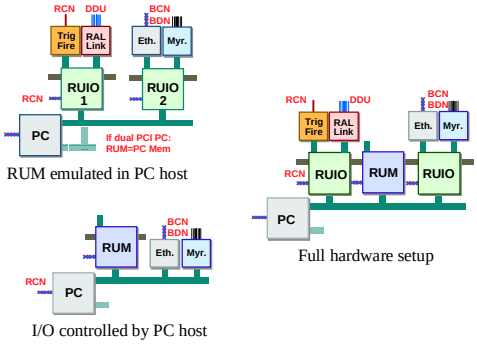

Figure 3: possible layouts for RU

A. I_2O-like host-IOP communication

The I_2O [6] (Intelligent I/O) architecture specification describes a communication scheme between host processors (typically a workstation) and input/ouput specialized processors (IOP) on expansion boards. I_2O specifies that host and IOP communicate through message frames. The format of the message frames is defined in the specification, as well as the means used to transfer those messages from host memory to IOP memory and back. Here we just used the hardware protocol, i.e. the message passing scheme, not the message format.

In the specification, this communication scheme allows the drivers for devices controlled by IOP to be split into two parts. One part on the host side, and one on the IOP side, hence the IOP can relieve the host from most of the I/O chore.

In the I_2O philosophy, the driver running on the host is also relieved of knowing the internals of the add-on board, as the specification defines messages for each type of peripherals that can be controlled by the IOP.

The host/IOP message passing scheme is implemented through two PCI registers on the IOP. Those register allows the host to access four queues, in which message frame addresses are stored. (see Figure 4)

In the following, we will use inbound and outbound from the IOP point of view. On the IOP memory resides four FIFOs, which contain pointers to free frames (inbound and outbound), and posted frames (also inbound and outbound). The preferred model for I_2O message passing is pushing messages in pre allocated frames in destination's memory.

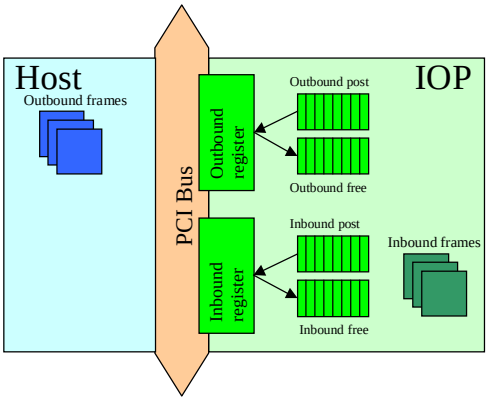

Figure 4 : Message passing FIFOs

Thus, when posting a query, the host retrieves the address of a free frame in the IOP memory by reading the Inbound Queue Register (this pops the first free frame from the inbound free queue). It gets the address of a free frame in the IOP memory. The host writes the message directly in this location, then writes the frame address in the Inbound Queue Register (this pushes the frame address in the inbound post queue). This event (inbound post queue not empty) triggers an interrupt on the IOP side. The IOP just has to read the frame address in the inbound post queue, and process the frame.

Symmetrically, when the IOP has to send a frame to the host, it pops a frame address out of the outbound post queue, and DMA's the frame into the host memory. It then writes the frame address back in the outbound post queue, and the host is warned by an interrupt that the outbound post queue is no longer empty (the host still has to read the frame address on the PCI bus).

The big advantage of this scheme is to provide a write-only method (except when host reads frame addresses, but it's only one or two word per frame transferred). Using write-only scheme on a PCI bus gives best performance, especially in our case where there is a PCI-to-PCI bridge between host and IOP.

1) *Host side implementation*

The host side implementation was developed with Linux. We used a very simple interrupt service routine that pops frame addresses out from PCI and delivers them to a kernel thread, thus the message processing is not done at interrupt time.

2) *IOP side implementation*

The IOP side of the I_2O implementation was written for VxWorks. (see section II). It is quite symmetric with the host side one.

3) Status

In our software prototypes, we don't use the message protocol defined in the I_2O specification, but only the message-passing mechanism. We just used a simple "receive and reply" protocol that implied a nearly immediate message processing.

The current implementation encounters some problems, which are still under investigation, involving DMA engine and I_2O message queues.

As a conclusion, we can say that the I_2O message passing protocol seems very promising for bus efficiency, but the implementation we have didn't give its full performances at the time of writing.

IV. TEST ENVIRONMENTS

Different test environments have been setup to validate the various parts of the hardware.

A. Bridge validation

The first step in the RUIO validation was bridge validation. This was simply done by running a driver for the RUIO Ethernet interface on the host (running Linux). By loading the network, we could get a good validation of the bridge (as well as the Ethernet part), because the bridge was accessed in read and write simultaneously on both sides (the Ethernet chip is a bus master). We just had to create important transfers on the network interface, and to analyze the bus behavior.

B. VxWorks development environment

Apart from the messaging layer specification, the I_2O specification defines a standard interface to a real time OS. Part of this interface has been developed for VxWorks. The main interest of this interface is to define access to low level (bus, adapters, and interrupts…). We could then imagine writing drivers portable across platforms with no other adaptation than recompiling. This is of great interest for all the peripherals that will be tested individually before being integrated in the DAQ demonstrator with links to detector front end and builder network. Those components are developped and tested in parallel. See [7] and [8].

C. RUlib and Labview test environment

A number of hardware components comprising the DAQ prototype system need custom software for debugging and testing. RU library, which is based on an object oriented shared library and NI Labview human interface, has been built for this purpose. The use of OO programming allows easier evolutions and "add on".

RU library is the latest evolution of an OO class library already used for tests in the past [9]

RU library objects are based on a class named "BaseObject" from which almost every object inherits. BaseObject defines common features for every object such as getting environment variables, debug printing command, log file facilities… Two generic classes inherit from this "BaseObject": "PCIMaster" and "PCIBoard" which respectively define common feature for PCI masters and PCI boards. The PCI board class accesses the hardware with its PCI Master. The generic PCI board class has member functions that delegate the real accesses to the PCI master. Therefore, a given PCI board derived class works unmodified on any type off host (if PCI master is available).

The Labview application provides the user with a panel allowing all kinds of accesses to the bus : memory space, configuration space and I/O space. For each board, we developed a specific class, which provides specific functionalities, and the corresponding Labview panels.

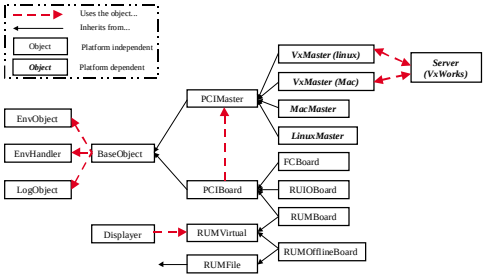

Figure 5 : Objects organization in RUlib

Currently, the library is supported on Linux and Mac OS environments. We can also access the PCI bus of an embedded system running VxWorks, through TCP/IP, thanks to a server. The client side works on both MacOS and Linux.

1) Mac OS environment

For configuration space and IO space, we use Mac's PCI API. Memory space accesses are done directly by the library.

2) Linux environment

A specific driver (a kernel module) has been developed to allow accesses. Communication between kernel space and user space is done through a character file with major number 123. This module is in charge of searching PCI boards, PCI accesses, and of communication with the library. Configuration accesses are made through IOCTL calls that wrap kernel functions. IO space and memory space accesses are made from user space using memory mapping, in order to have better performances by dramatically reducing the number of system calls.

3) VxWorks facilities

A specific program has been developed for remotely accessing PCI boards on a VxWorks host, through network. This allows the user to debug boards on a nice

UI. The program is in charge of searching for PCI boards, executing PCI accesses, and of communication with the library through network (it's a TCP server). From library point of view, on Linux, communication is made through TCP sockets and on Mac OS using open transport's API.

V. FUTURE WORKS

A. RUM logic validation

RUM logic will be partly validated using RU library facilities. A general class "RUMvirtual" defines common features for the RUM board. "RUMboard" object implements a real hardware component whereas, "RUMoffline" object allows offline analyses of dumps generated in case of error during automatic tests. "RUMsim" object is in charge of simulation.

For a real hardware RUM board, "RUMboard" provides accesses to the different memories in the RUM, RUM's command functions, RUM's configuration. "RUMsim" is in charge of emulating the input and output, sending level 1 and 2 triggers. It is a virtual event manager. Knowing what the software sends, we can then check the behavior of hardware by several test functions. These functions mainly consist on reading the MMU memory and data memory, and checking pointer logic and even event data on thorough tests.

Offline analyses are available through "RUMOffline" object. RUM's MMU memory can be stored in a file, which can later be checked by test functions to search for occurred troubles. This allows RUM board to run endlessly (for instance, during the night), and then to save MMU memory when an error occurs, and reset the board to re-run the test. Test and analysis can be performed later, since data is saved in a file. This is used to find very infrequent error.

B. RUM-RUIO integration

The next step in the prototyping and testing of the RU is to integrate two RUIOs, and RUM in order to get a complete RU. The RUIOs will enable links (from detector and to builder network) to utilize the full bandwidth of the input and output busses. In this setup, the commands can be received either by the IOP on the RUM (using the Ethernet interface) or from the host.

If host receives the commands, it will pass them to the RUM's IOP through the message-passing scheme. Optionally, the host can also access the RUM registers directly.

We also plan to assign monitoring tasks to the IOPs.

C. Readout column demonstrator

Once the RU will be validated in its full form, it will be integrated into the DAQ demonstrator, for testing actual event building over a scaled-down network. The current candidate for this builder network is Myrinet. The test of the Myrinet network is described in [7].

D. Xdaq application

The I2O message passing scheme can be used as a fast transport protocol for the XDAQ. The XDAQ is a software toolkit developed for the testbeam [10]. This toolkit allows the user to choose the network interface by using different objects. We will integrate the I_2O messages as one of those transport objects.

REFERENCES

[1] The CMS Collaboration, *The Compact Muon Solenoid*, CERN, Technical Proposal, No 7, LHCC 94-38 December 1995

[2] http://www.pcisig.org

[3] G. Antchev, D. Gigi 20 - 24 September 1999, Colorado, USA LEB 5th Workshop on LHC Electronics,. *dual-port memory with reconfigurable structure*.

[4] G. Antchev et Al. *Readout Unit Prototype for CMS DAQ System*, Poster session, this workshop.

[5] http://cmsdoc.cern.ch/~dgigi/ruio.htm and http://cmsdoc.cern.ch/~dgigi/rum.htm

[6] http://www.i2osig.org

[7] F. Meijers, *Evaluation of Myrinet for the Event Builder of the CMS experiment*, MUG 2000, First Myrinet User Group Conference, Sep 2000, Lyon, France.

[8] A. Racz and L.Pollet, *A 400 MB/sec data link based on GE componants and reconfigurable hardware platforms*, This workshop.

[9] T.Ladzinski et al *CMS Data Links and Event Builder Studies*, Third Workshop on Electronics for LHC Experiments

[10] J. Gutleber, *A software development toolkit for the CMS Data acquisition*, Poster session, CHEP2000 7-11 February 2000, Padova Italy