A VMEbus General-Purpose Data Acquisition System

A. Ninane, M. Nemry, J.L. Martou and F. Somers
Institut de Physique Nucléaire, Université Catholique de Louvain
Ch. du Cyclotron, 2 – B-1348 Louvain-la-Neuve, Belgium

November 8, 1991

Abstract — **We present a general-purpose, VMEbus based, multiprocessor data acquisition and monitoring system. Events, handled by a master CPU, are kept at the disposal of data storage and monitoring processes which can run on distinct processors. They access either the complete set of data or a fraction of them, minimizing the acquisition dead-time. The system is built with the VxWorks 5.0 real time kernel to which we have added device drivers for data acquisition and monitoring.**

The acquisition is controlled and the data are displayed on a workstation. The user interface is written in C++ and re-uses the classes of the Interviews and the NIH libraries. The communication between the control workstation and the VMEbus processors is made through SUN RPCs on an Ethernet link.

The system will be used for, CAMAC based, data acquisition for nuclear physics experiments as well as for the VXI data taking with the 4π configuration (100 neutron detectors) of the Brussels-Caen-Louvain-Strasbourg DEMON collaboration.

I. INTRODUCTION

Experiments differ in the way they produce data: they use different standards of hardware to digitize data (VME, VXI, CAMAC, ...); they generate data varying in byte length and counting rate. However, the last stages of data acquisition systems have many things in common: the data are analyzed on-line to control the experiment and are written on storage devices for further off-line analysis.

We have defined a common framework for a general-purpose data acquisition system. It meets the following requirements:

- the data source is open: the system can be enabled to acquire data from various instrumentation buses;
- the data sink is open: data can be analyzed on-line by concurrent processes and can be stored on different types of mass storage devices;
- the system is scalable: it can be used for low count rate nuclear physics experiments (20 byte events at 200 Hz) as well as in larger experiments such as the 100 neutron detectors of the DEMON collaboration [1] (300 byte events at 5 kHZ);
- the user sits at the highest level of the data acquisition system with the modern conveniences of workstations.

II. SYSTEM ARCHITECTURE

A. Distributed Hardware

The system is designed following a distributed architecture (Figure 1). The real-time data acquisition is performed by a VMEbus system. It allows to connect a wide variety of interfaces to external hardware as well as to run data acquisition processes by various processor boards. The user acquisition control and data handling is delegated to a standard workstation connected to the VMEbus system by an Ethernet link.

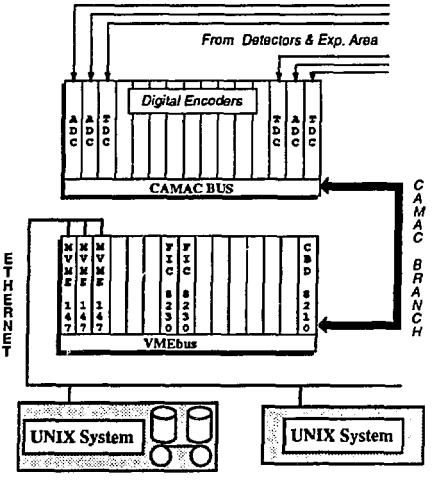

Figure 1: A Simple Distributed Architecture

In such an architecture, both parts are loosely coupled and may be evolved on their own. The user workstation or the VMEbus system may be replaced or upgraded without redesigning the entire system.

B. Software Architecture

Events are defined as a set of structured and correlated data which enter the system at random by interrupts. Events[1] are assembled into larger, configurable, structures called *blocks*.

A number of tasks can be implemented in the system to read data simultaneously from data *channels*. A channel is characterised by an access mode: `full` or `sample`. Tasks accessing data through a `full`-mode channel read and process all the data blocks. They can therefore lead to a considerable increase in the system dead-time. The influence on the data processing dead-time can be reduced by `sample`-mode channels which access only a sample of the data at the task's own processing speed. A data storing process works in the `full`-mode, while the `sample`-mode suffices for data monitoring. A `block type` parameter can also be assigned to a data channel: read operation on the channel will return only blocks of events of this particular type.

The Buffer System

The data acquisition system can be viewed as a *producer* task – the event's interrupts – and many *consumer* tasks – the data analysis and storage – running concurrently to fill and consume blocks of events. The producer and the consumer tasks share a common buffer system.

A *buffer* refers to a block of events. They are arranged in two doubly-linked lists [2] (Figure 2): the *free* list contains buffers that can be used directly by acquisition interrupts to store new events, while the *valid* list contains buffers already filled with events but not yet processed. Buffers can reside in both free and valid lists. This situation occurs when they have been processed by all `full`-mode channels but not by all `sample`-mode channels.

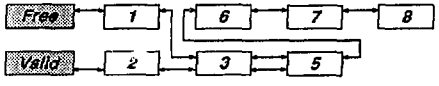

Figure 2: The two doubly-linked lists

Producer Part

At the beginning of a data acquisition, all buffers reside in the free list. When the data acquisition process starts, a buffer is extracted from the head of the free list and becomes the *current buffer*. It is filled with events up to its maximum size. It inherits the identification of the data reading channels interested to process it and is added at the tail of the valid and/or the free lists. A new current buffer is extracted from the free list head and the procedure continues.

[1] The event definition is not restrictive: an event can be `CAMAC` data of a single physical event but can also be a block of data preprocessed or filtered by other processors.

Consumer Part

A data reading channel scans the valid list until it finds a new unprocessed buffer and returns the data to its parent task. When the operation is completed, the buffer is marked and is moved within the linked lists according to three situations:

1. the buffer waits to be processed on another `full`-mode channel: nothing happens. It remains on the valid list and is safe from interrupts;
2. the buffer waits to be processed on `sample`-mode channels only: it remains on the valid list and returns at the tail of the free list;
3. the buffer has been processed on every channel: it is removed from the valid list and returned to the free list.

If the acquisition produces data at a faster speed than the consumers process them, the free list will be emptied; event interrupts are then disabled until a consumer process returns a buffer to the free list.

III. IMPLEMENTATION

The ideas presented above have been implemented in a `VMEbus` system running the `VxWorks 5.0 kernel`.

A. Hardware

The `VMEbus` system consists of three `Motorola MVME147` boards with *MC68030 microprocessors*. Each board has `SCSI` and `Ethernet` capabilities although they are not used on all of the boards. The system has been used so far with two different sources of data: `CAMAC` and `FIC8230` preprocessor.

CAMAC

The `CAMAC` crate is connected to the `VMEbus` by the `CES CBD8210` branch driver and the `CCA2` crate controller. The module allows the generation of `CAMAC CNAF` cycles as `VMEbus` memory mapped addresses. This elegant feature provides a fast access to the `CAMAC` bus and facilitates the software writing. The data acquisition system is interrupted at each physical event by `CAMAC LAMs`.

FIC Preprocessor

The `CES FIC8230` is a `VMEbus` board with a *MC68020* microprocessor. It runs a fast, specifically developed, kernel. The processor receives events from a `CAMAC` crate or from a `DMA` channel connected to local hardware. The microprocessor assembles events into blocks which are then written directly to the last stage of the data acquisition in a single interrupt.

B. Acquisition Software

To reach a high level of flexibility, the data acquisition system has been layered (Figure 3). The real-time kernel executes user tasks, which control the acquisition process and read the data through the kernel `I/O` system. At a

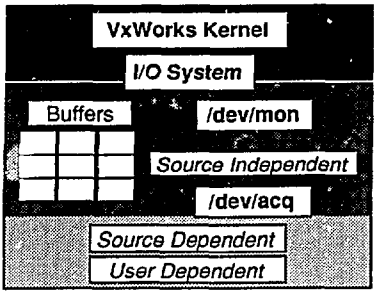

Figure 3: Acquisition System Layering

lower level, the data acquisition software itself has been structured in three layers:

1. the *source independent* layer comprises the data buffer system and its integration into the kernel I/O system;
2. the *source dependent* layer implements routines to connect the data source to the acquisition system;
3. the *user dependent* layer implements routines to render the acquisition process suitable to the user needs.

Real-Time Kernel

The VMEbus processors run the VxWorks 5.0 real-time kernel. This software has been selected for its:

- clear separation between system development and code management tasks – on a UNIX system – and real-time tasks – on VMEbus processor boards – which fits naturally in our distributed architecture;
- platform independency on both sides: many UNIX platforms and many VMEbus processors are supported;
- high networking capabilities with IP family of standard protocols (TCP, UDP, RPC, NFS, ...).

The kernel provides all real-time primitives: semaphores, events, message queues, control of preemption, priority-based scheduling, ... as well as the standard C library. It supports the notion of *device driver* which provides a common interface to devices or pseudo-devices through calls to the standard I/O C library.

Multiprocessor Extension

The VxWorks 5.0 is basically a single-processor kernel. To use the full power of the VMEbus system and to obtain the requested scalability, multiprocessor (MP) features have been added. The granularity of the MP-architecture is situated at the task level.

The system has a *master* processor and many *slave* processors. The master creates and owns the shared ressources while the slaves manipulate them. We have implemented MP-devices and MP-semaphores.

MP-devices: Device structure has been splitted into a private and a shared part. The private part is the standard VxWorks device structure refered to the local I/O system. The local structure contains a reference to the shared part of the device.

MP-semaphore: The MP-semaphore has been implemented with a shared flag protected by a *spinlock* variable [3]. The spinlock is accessed by indivisible cycle machine instructions to eliminate contentions. The MP-semaphore has a private, standard VxWorks, semaphore in each of the participating processors. Tasks waiting for the MP-semaphore *sleep* on the private semaphore inside their processor. A remote *wakeup* has been implemented with the help of the MVME147 mailboxes.

System drivers

The data buffer system is accessed by two MP-device drivers integrated in the VxWorks I/O system. They reflect the producer–consumer relationship.

1. /dev/acq: The acquisition device controls the *production* of the data. ioctls are used for example to start and stop the acquisition by enabling and disabling the interrupts in the master board.
2. /dev/mon: The monitoring device implements the *access* to the data. Tasks open this device to get a channel and read data.

The lower part of the acquisition device driver is connected to the data source by four routines:

1. acqStart(): implements commands to initialize the source when starting an acquisition process;
2. acqIntr(): is executed at each event interrupt;
3. acqRestart(): restarts the data source at the end of the event interrupt handling;
4. acqStop(): executes commands to finish the data acquisition process.

Each routine has an user defined part, which accesses the user modules participating in the data acquisition.

Because of the VMEbus limitation of a single interrupt handler on a given level, the data acquisition process can be executed only on a single processor, the master, while the data processing tasks run on several slave processors.

Network Servers

The remote control and data analysis from an user workstation is executed by *Remote Procedure Calls* (RPCs) *servers* running in the VMEbus system.

- acqServer: executes ioctls on the /dev/acq device to control the data taking;
- monServer: controls the /dev/mon device to grant access to channels for remote data reading tasks;
- acqSysServer: supervises global parameters and procedures such as system directory, system reboot, ...

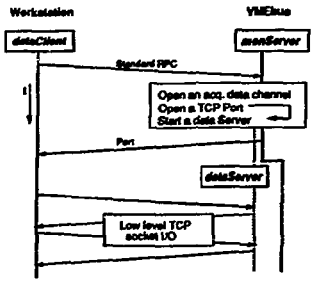

Figure 4: RPC-Less Data Tranfer

Because of their widespread acceptance, SUN 4.0 RPCs have been selected. To avoid retry problems, RPCs use the TCP/IP underlying transport protocol. The machine-dependent data format problem, unavoidable in a heterogeneous distributed environment, is solved by the *External Data Format* (XDR) layer of the RPC protocol.

The RPC mechanism is well suited to remotely control the system, but its layers introduce a time overhead that is too large to transfer a bigh rate of data. For this purpose, we are using a less ressource consuming protocol (*Figure* 4). The client contacts, with a standard RPC, the server to open a data channel on the acquisition side and a data communication port on the network side. A new specific data server, receiving both I/O descriptors, is created. The network port descriptor is returned back to the client who can establish a faster and reliable point-to-point connection (TCP/IP) to the data server to read events.

C. User Level Tasks

Users can run data processing tasks in the VMEbus system, they simply access the data through the monitoring device in the same way as for any other device. By this way the user can analyse data, build histograms, ... Data can also be copied to a disk or a tape cartridge in the VMEbus crate.

Users may want to access the data directly from processes running in their workstation. They can use a library of subroutines which takes care of the communication with the network servers in the VMEbus system. Users must provide four routines:

1. monStart: begins a data monitoring task;
2. processBlock: is executed for each block of events;
3. monRefresh: asynchronous user's request handling;
4. monStop: completes the data analysis.

Processes respond to the SIGHUP and SIGINT signals. The SIGHUP handler executes asynchronously the routine monRefresh() to get intermediate results while the SIGINT handler completes prematurely the data reading process.

A workstation process must indicate the VMEbus board and device it wants to read and the data channel mode. An example is the ddVME command (Figure 5), based on the well-known UNIX dd to copy data.

```
% ddVME if=vmeacq11:/dev/mem of=data.01 mode=full count=20
Warning: obs set to 1024 bytes
Connecting to 130.104.3.120(8979) ... done
Process 16478 started
ddVME: 40/0 blocks ---> 20 blocks of 1024 bytes
Rate = 24980 bytes/s
Process ddVME terminated
```

Figure 5: Example of a workstation task

Workstation Interface

An X Window interface helps the user to configure the VMEbus system and to control the acquisition processes. The interface, written in the C++ language, uses the HIR-CL and InterViews object libraries [4]. It implements Macintosh-like menus whose items are activated or deactivated according to the experiment current status. It uses dialogue boxes to get the data acquisition and monitoring parameters and to show their status. The main C++ class includes a method to dispatch user's requests and sends RPC requests to the VMEbus servers.

The user interface is able to listen to the data acquisition system and transmit its messages to the user. For that purpose, we have modified the InterViews events handler, making it sensitive to asynchronous messages from the VMEbus system.

Conclusion

We have developed a simple architecture for a data acquisition system in which real-time data acquisition, monitoring tasks and system control have been loosely coupled. This provides the flexibility of the system. The system is now fully integrated within our network of workstations and is used in experiments around the Louvain-la-Neuve Cyclotron.

References

[1] G. Bisard, G. Costa, D. Durand, Y. El Masri, G. Guillaume, F. Hanappe, B. Heusch, A. Huck, M. Mossynski, J. Peter and B. Tamain: The Belgian-French Neutron Multidetector DEMON, *Proceedings on the International Conference on New Nuclear Physics with Advanced Techniques*, Iearapetra, Crete, June 23-29, 1991

[2] The data acquisition system and its buffer system is inspired by the UNIX kernel and the buffer cache, see M.J. Bach, *The Design of the UNIX Operating System, Prentice-Hall, 1986.*

[3] Lawrence M.Ruane, Process Synchronisation in the UTS Kernel, *USENIX Computing Systems, Vol 3, 1990.*

[4] M.Sibomana, Y. Longrée, P.Mareschal, M. Nemry, A. Ninane and F.Somers, Direct Manipulation User Interfaces, in *New Computing Techniques in Physics Research, éd. du CNRS, 1990,* and references therein.