ATLAS Internal Note

Software Facilities for the TileCal ROD Development System

J. Roldan, J. Romance

Jesus.Roldan@cern.ch, Juanba.Romance@ific.uv.es
IFIC. Instituto de Física Corpuscular
Centro Mixto, Universidad de Valencia - CSIC
Avda. Dr. Moliner 50, E-46100 Burjassot
VALENCIA, Spain

Contents

List of Figures

List of Tables

1 Introduction

This document tries to define a first set of technical and functional specifications for the Front-End, ROD-Controller and ROD-Module software pieces which are in process of implementation at the Valencia ROD Laboratory. The system will provide a well defined environment both at hardware and software level where the ROD technologies can be evaluated for DAQ-1 integration and test-beam purposes. It's a working document and therefore updates are made as soon as possible.

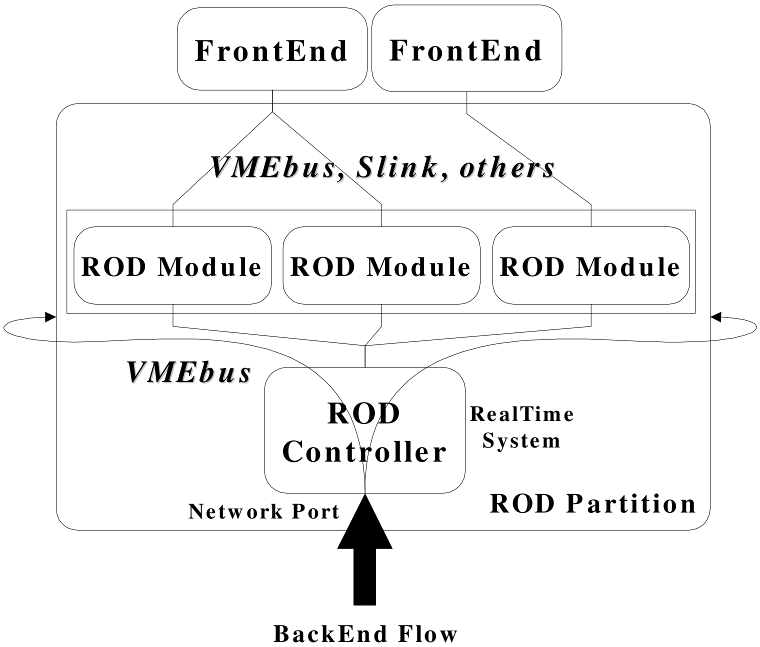

Figure 1: *ROD Environment.*

2 General Specifications

2.1 Cross Development

The ROD Module, ROD Controller and Front-End applications are multi-platform designed. The implementation must allow to abstract functionalities, from Hardware and OS constrains, providing a hierarchical and well distributed cross software package running in a multiprocessor environment. These packages are completely developed in the LinuxPC box using made-home cross compilers for the VME-LynxOS machines, based on Cygnus free tools[1]. The current supported systems are:

1. PentiumPC+Bit3 Interface Linux-2.2.1
2. PowerPC 2604 LynxOS-3.0
3. MVME167 68040 LynxOS-2.5.1
4. RIO2-8061 LynxOS-2.5.1 (In progress)

[1] http://sourceware.cygnus.com/mission.html

2.2 Driver Policies

The next common matters have driven the driver/packages design:

- The Latency of data transfers MUST be as low as possible, allowing a synchronous data transfer bandwith $\geq 25\frac{Mbytes}{sec}$ between each host. System Calls are only allowed for static operations i.e. memory alloc/map/unmap. Polling method is used for all the platforms to perform read/write DMA cycles. Previous studies of the DAQ-1 community shows that an ISR based data transfer implies a heavy system overload in high rate DAQ systems as the purposed here.
- All the hardware devices (either Slink or VME chipsets) are user space mapped using mmap (Linux) or smem-create (LynxOS) system calls. The concurrent access to hardware should be managed inside the libraries with POSIX1.0 tools (semaphores, condition variables, etc).
- The RAM involved in DMA transactions has to be a user space mapped. Therefore no typical Kernel to User/User to Kernel transactions are used to access DMA data buffers, double-buffering is only allowed under the ROD application.
- The ROD management of physical devices as RAM memory and PCI Chipsets MUST *live in peace* with the Kernel management of hardware devices no related with the ROD application.

2.3 I/O Ports. VME-Slink interfaces

Two ports are used for the data and control flow, VMEbus and S-Link.

- The *Data Flow* is directly involved with DMAs allowing a bandwidth $\geq 30\frac{Mbytes}{sec}$; it will be used to download bunches of data $\geq 500bytes$, i.e. VME data injection, Slink data injection, Synchronous data sampling, etc..
- The *Control Flow* is used to implement low bandwidth tasks, i.e. command transfers, control protocols, etc. It is implemented with VME raw pointers which have roughly a $5\frac{Mbytes}{sec}$ performance.

2.3.1 VME Port

The VME libraries[3] allow Raw and DMA user access to VME space using a well defined data-code interface. Four basic data types are provided, the address space for all of them is physically contiguous and the memory related is removed from the virtual memory OS management[2]:

- *VME Slave Objects.* On user request, a M bytes bunch of the physical system RAM is allocated and mapped to a raw user pointer. Two types of slave objects are provided: Dynamic and Static slaves. The first one creates the memory bunch associated to the object, performing the memory allocation *On Fly.* The second one[3] provides a link to the object memory bunch already created from external applications; i.e. VME Slave data pattern for applications running on MVME167 under LynxOS-2.5.1:

[2] Memory spaces will be always wrapped to 4096 bytes multiples

[3] Under Linux, current design only *Dynamic Slaves* are provided

```
typedef struct{
  word32  vme_addr;  /* VME Address                   */
  word32  phy_addr;  /* buffer Physical Address       */
  word32  size;      /* Device size in words          */
  word32 *v_ptr;     /* virtual pointer               */
  char    id[ 30 ];  /* Device ID                     */
}VMESlave;
/* Methods */
int Open_VMESlave   ( VMESlave*, char *mem_id, word32 size );
int Close_VMESlave  ( VMESlave* );
```

- *VME Master Objects.* As above, on user request M bytes from the VME bus are virtually mapped to a user raw pointer; i.e. VME Master Object for applications running on MVME2604 PowerPC under LynxOS-3.0:

```
/* Data Pattern */
typedef struct{
  word32  vme_addr;  /* VME Address           */
  word32  size;      /* Device size in words  */
  word32  space;     /* VME Address Space     */
  word32 *v_ptr;     /* virtual pointer       */
  char    id[ 30 ];  /* Device ID             */
}VMEMaster;
/* Methods */
int Dump_MasterInfo ( VMEMaster* );
int Alloc_VMEMaster ( VMEMaster*, word32 size );
int Free_VMEMaster  ( VMEMaster* );
```

- *VME DMA Objects.* Beside alloc and user-map affairs, a new function interface, DMAgo, provides an engine to perform Timed-Out DMAs to read-write data, using D64 VME block transfer capabilities where-when be available; i.e. DMA Object for LinuxPC-Bit3:

```
typedef struct{
  /* User Fields */
  word8        w_swap;   /* Word Swap                           */
  word8        nb_swap;  /* Byte Swap                           */
  word8        am;       /* VME Address Modifier                */
  word8        dir;      /* DMA Direction: VMEtoPCI/PCItoVME    */
  word32       size;     /* Requested Size                      */
  word32      *v_ptr;    /* Virtual Pointer                     */
  word32       vme_base; /* VME base address                    */
  /* Library Fields */
  char         id[ 45 ]; /* Device Identifier                   */
  RAM_Dev      ram;      /* RAM Device                          */
  sem_t        sem;      /* General Purpose POSIX1.0 Semaphore */
  word32       pg_index; /* Ptr to First Assigned Descriptor    */
  Bit3PT_Dev  *dpt_dev;  /* Ptr to Page Table Device            */
  Bit3CSR_Dev *csr_dev;  /* Ptr to CSR Device                   */
}DMA_Dev;
/* User Methods */
int DMA_Alloc    ( Bit3PT_Dev*, Bit3CSR_Dev*, DMA_Dev* );
int DMA_Free     ( DMA_Dev* );
int Dump_DMAInfo ( DMA_Dev* );
int DMA_Go       ( DMA_Dev*, word32 direction, word32 size );
```

2.3.2 Slink Port

A couple of Slink-Fiberchannel cards are available at our ROD lab. The first one (Link Destination card, LDC) is coupled to a Slink to PMC card, plugged both on a free PMC site of the VME PowerPC card. The second one (Link Source Card, LSC) is coupled to a PCI to Slink card, plugged both on a free PCI slot at the PC-Pentium mother-board. Two software packages have been developed, libSlinkLynxOS.a, libSlinkLinux.a, to properly manage the cards under the LynxOS-3.0 and Linux-2.2.1 operating systems. From the point of view of *Slink Programmers-Users*, a similar set of functions and data structures have been implemented, i.e. the slink data object and methods for Linux-code applications seems like below:

```
typedef struct{
  char       str[ 40 ]; /* Device Identifier String  */
  word32     link_stat; /* Link State                */
  int        fd;        /* File Descriptor           */
  S5933_REGs *amcc;     /* AMCC User Pointer         */
  word32     trailer;   /* Packet Trailer Word       */
  word32     header;    /* Packet Header  Word       */
}Slink_Dev;
/* Slink Functions */
int SLink_Open  ( Slink_Dev* );
int SLink_Close ( Slink_Dev* );
int SLink_Init  ( Slink_Dev* );
int SLink_Reset ( Slink_Dev* );
int SLink_Dump  ( Slink_Dev* );

typedef struct{
  char    str[ 40 ];     /* Device Identifier String  */
  word32 *buffer;        /* User Pointer              */
  word32  size;          /* Requested size in Bytes   */
  word32  pci_addr;      /* Memory Buffer PCI Address */
  int     fd;            /* File Descriptor           */
}DMA_Dev;
/* DMA Functions   */
int DMA_Open    ( DMA_Dev* );
int DMA_Close   ( DMA_Dev* );
int DMA_Raw_Write( DMA_Dev*, Slink_Dev*, word32 *fault, word32 size);
int DMA_Complex_Write ( DMA_Dev* , Slink_Dev* , DMA_Ctrl* );
```

FrontEnd and ROD-Module applications will use these structures only at object code RODslink.o, where the Slink devices are created and destroyed. However, physical references to Slink-objects will be also allowed on the threads which perform high-speed I/O operations, decreasing as much as possible software latency. The programs already developed for the Fiberchannel characterization provide us the necessary references to build RODslink.o . The available maximum bandwidth with the Slink port is $90\frac{Mbytes}{sec}$ for the PMC Card on the MVME2604 and $42\frac{Mbytes}{sec}$ for the complete Fiberchannel Link using a PCI card as data source[2].

2.4 Common Bones

Each ROD application is built using a common object map in order to save efforts in each specific implementation. The list below can be considered as the ROD applications *common*

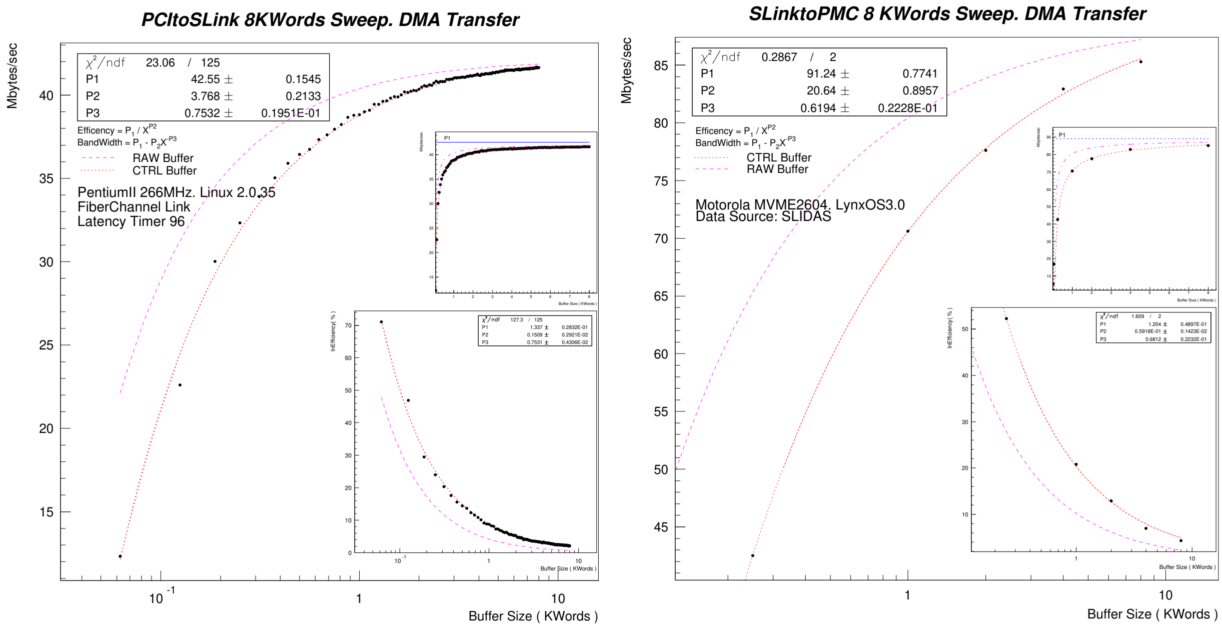

Table 1: *Right side, PMC Bandwidth using a SLIDAS* ($160\frac{Mbytes}{sec}$) *as data generator. Left side, BW with a real S-link implementation with Fiberchannel, LSC-PentiumPC LDC-MVME2604.*

bones. All of them are cross codes and the number of defines selecting each platform increases as one goes down to the hardware core. However, for each ROD application (FE, Module, Controller) some of them are partially recoded because the functionalities in each one are quite different.

- ROD.o: Interface between Main and the rest of the application
- RODhardware.o: Interface with semi-static and asynchronous I/O operations; i.e. rod modules booting, command transmission, Asynchronous sampling, etc.
- RODsoftware.o: Synchronous and Multi-threaded operations; i.e. data injection and production on FrontEnd/Controller, ROD-Machine on Modules, etc.
- RODvme.o: VME objects builder using the crate hardware resources database (HardwareROD.db.dat).
- RODslink.o: Same as above for the Slink stuff.

2.5 Control And Status Registers

Each ROD device (Modules, Controller and Front-Ends) have a CSR map which is used for slow-rate and control operations between itself and any external device. Local CSR space address is mapped into the crate VME address space and it is physically implemented on the system RAM memory of each machine. In this fashion, each local CPU will map its own CSR as a VME slave object, the external CSRs are implemented as VME raw master devices. Only the controller will build the whole CSR Master map in terms of ROD-Modules and Front-Ends, providing broadcast and single module/front-end commands.

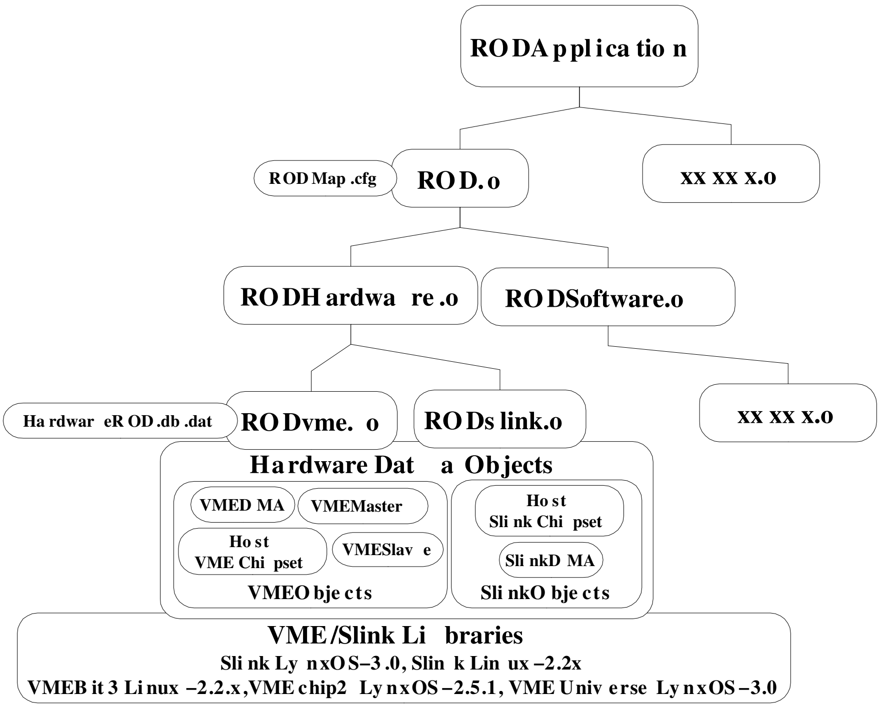

Figure 2: *Application bones. Object Distribution.*

The ROD-controller CSR Slave space is divided into M + N contiguous zones mapping the M modules and N Front-Ends of the ROD-crate. Each zone is sized to 64 words. By this way since 16 devices can be mapped at the controller level.

The ROD-Module will divide the local slave space into two zones: the Front-End and the Controller both sized to 64 words.

The Front-End will divide the local CSR as $1 + M$ contiguous zones, corresponding to the Controller space and the M ROD-Modules linked to itself.

The crate information is gathered from the ASCII database and the RODMap.cfg configuration files, both informations are available in a NFS shared directory. The Figure 3 shows the distribution map purposed.

2.6 Trigger Management

The trigger system will clock the data flow between either Controller-Module or Front end-Module. In the ATLAS scenario, this information is distributed to the ROD crate using a bus based high speed system (very likely a custom bus on the P3 VME connector), splitting the trigger information of the TTC system[4] .

A prototype of the external VME-TTC system using the TTC-receiver chips and optic fibers transmission is already available at other institutes and Test-Beam areas. However, such stuff is quite complex to distribute it to every laboratory. We have preferred to implement a Trigger interface using software tools, leaving an open source code ready to work in other scenarios.

[4]Timing, Trigger and Control distribution system

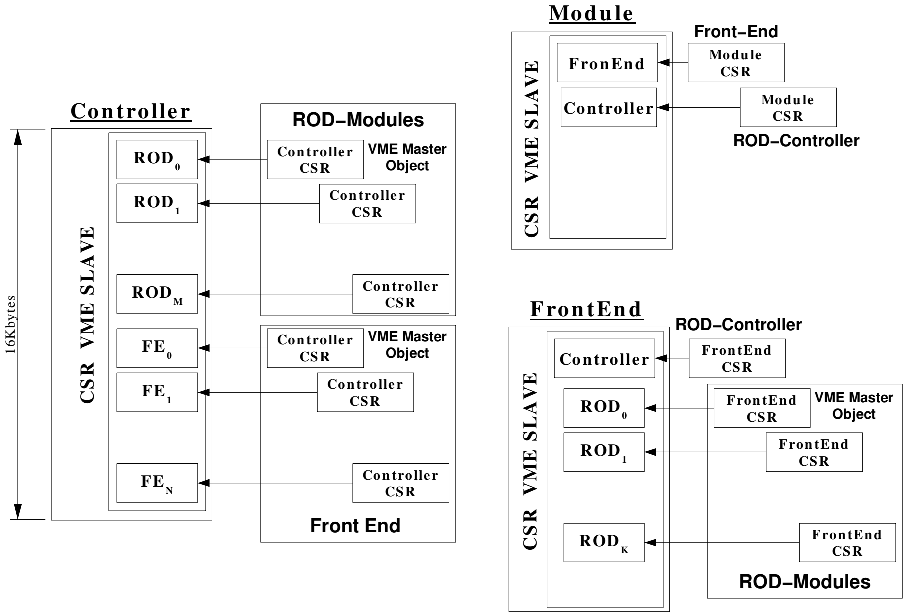

Figure 3: *VME-CSR slave map for the ROD-Crate.*

In our case, the trigger production is physically located at the ROD-Module and it's modeled with a POSIX1.0 thread. Inside the ROD-Module software, the trigger information is shared between the threads. For the external world, the trigger will be issued along VME path using the CSRs space. One of the first configuration tasks to do by the controller software is to select the trigger type on the ROD-Module device.

2.7 The Data Injection System

The ROD-module data injection can be defined for two data sources: The FrontEnd and the Controller. It is currently understood as a high speed synchronous operation mode driven by some control mechanism. A VME logical protocol has been defined and implemented to control the data exchange using either Slink or VME I/O ports. The design bears in mind the next rules:

- The trigger production is located on the ROD-Module.
- The protocol is designed virtually, and therefore can be modeled out of the ROD-environment. However, the real scenario will be the VME bus.
- The trigger flow is controlled and monitored with the local and remote CSR spaces of the devices involved in the data injection. *base* is an offset over the CSR base address.

ROD Module

Address	*Purpose*
base	Handshake word
base +1	Protocol Fault counter
base +2	Protocol Success counter

External Data Source

Address	*Purpose*
base	Handshake word
base +1	Module Fail counter
base +2	Module success counter
base +3	Local Fail counter
base +4	Local success counter
base +5	Local Trial counter

- Two VME modes are required. The **Target** mode assumes that module has not able to drive VME DMAs and therefore the DMA cycle will be driven externally. The **Driver** mode assumes that the module has capabilities to perform VME DMAs, and therefore the DMA cycle is driven locally on the module.
- The protocol is locally RAM polled in the OS user context, saving the VME bandwidth and sharing our poll time between other processes. The picture 4 shows the baseline operation for the *Driver mode*.

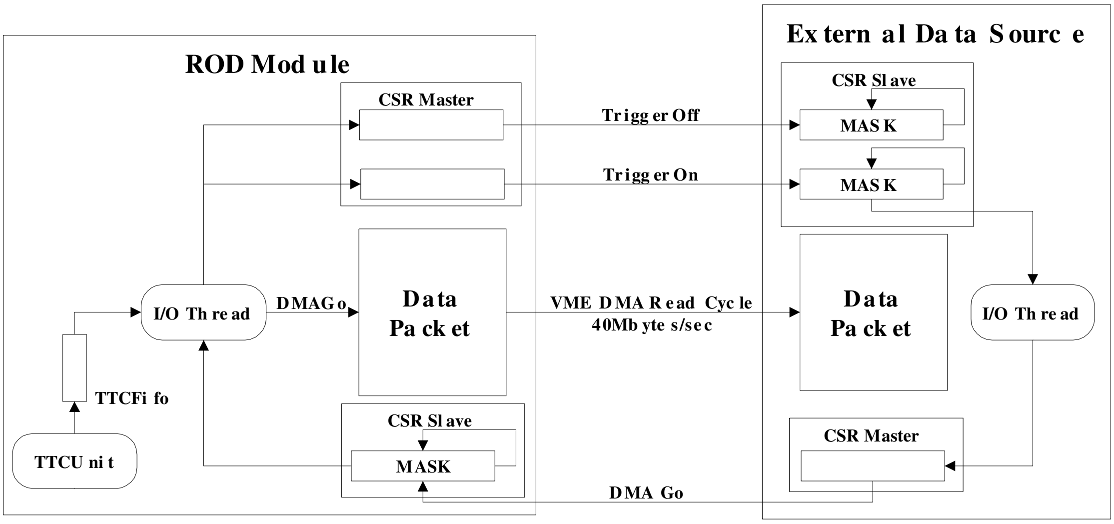

Figure 4: *Synchronous Flow for Driver Mode.*

3 RODModule

The main reference for the implementation of this software piece is the functional specifications document released at the end of 98[5]. However, the current hardware development of the RODs at the LArg is other of the baselines to start to produce code. Both calorimeters have similar data processing. The Front-end to Rod-module mapping in terms of $\frac{bytes}{sec}$ and readout channels per processing unit is also quite compatible. In fact, the TileCal ROD

[5]http://ific.uv.es/tical/rod/RODSpec.html

arquitecture described for the ATLAS experiment is already quite similar to the current LArg[4] prototype.

3.1 Implementation Specifications

The following ideas can summarize the purposed ROD arquitecture:

- *Multiple Data Input Flow.* The input Stage at ROD, gathers the data flow of four super-drawers in parallel using S-Link optical interfaces which are able to provide a theoretical bandwidth $\geq 1\frac{Gbit}{sec}$ (Fiberchannel version) per link. This is valid under normal data taking, however, is desirable to use the VME path to test module functionalities without front-end links under special runs. If a multiple input flow is required at some time, it can be prototyped on the MVME2604 mother board. This VME card provides by default one PMC slot, but it can be extended to three sites with a 6U daughter card (which we have available at lab). The same can be applied to the RIO2 processor using the PEB 6408 getting up to four PMCs per processor board. For the present, we only consider the case that a $\frac{1}{1}$ map is used.

- *Parallel Data Processing.* After the front-end data event is buffered, the data processing is divided in two kind of actions: Synchronization event and Tile-Cal data processing. Synchronization is related with the BCID provided by the CTP and that one received in the Front-End event. The Tile-Cal data processing is related with the digitized waveforms received at ROD corresponding to each read-out channel set.

  An optimal filtering of the raw samples is planned for the TileCal event providing to ROB three values for each read-out channel, the energy, the event time, and a calculations confidence factor (similar to χ^2). However, this compression is carefully accomplished after checking the cell for overlapping pulses, high degrees of pile-up contamination, empty cells, etc. For special cases (High Energy events at the experiment beginning) is also planned that the raw samples be downloaded to next DAQ level for further analysis. A drawer/ROD map of $\frac{4}{1}$ is planned for ATLAS, therefore the above operations are performed in parallel for each drawer.

- *Event Builder.* After the TileCal processing, the ROD gathers results from each processing unit formatting the output results as the ATLAS DAQ standard packet, using the TTC information and the internal synchronisms.

- *Monitor, Control and Data Sampling.* Controller will be provided by statistics information and functional status under the current operation mode of ROD motherboard. Monitoring will follow LVL1 trigger nominal rate at ROD level, however, the crate controller request rate will be lower (1-5)%, except for real-time events as a Module-BUSY activation.

  Control operations are considered as an asynchronous operation which allow a complete configuration of the module working mode: Module-BUSY management, download codes to the programmable devices and LUTs to the Module memories, etc. All of them are performed from the external point of view, through the VME path. It is fundamental to define a simple VME bus based logical protocol to exchange control commands between the Controller and each ROD-Module.

Data Sampling is considered as a synchronous operation from the point of view of the Controller and the ROD-Module. It allows to sample the main DAQ data flow on the ROD crate at different percentages, it will rely on the Module working mode, i.e if ROB Link is enabled, (1-5)% of the ATLAS DAQ data flow should be sampled. In any case, one should not decrease the Module DAQ nominal Rate of 100KHz.

3.2 The RODModule Software Model

We have modeled a ROD-Module version based on POSIX1.0 thread tools which is depicted in the figure 5. POSIX threads provide the necessary mechanisms to create code pieces able to run concurrently as the ROD-Module operations do. The model can be also sourced with real data flow at Test-Beam or Laboratory environments without big hardware resources, using the released S-Link/VME software packages and hardware tools either Desktops PCs or VME workstations.

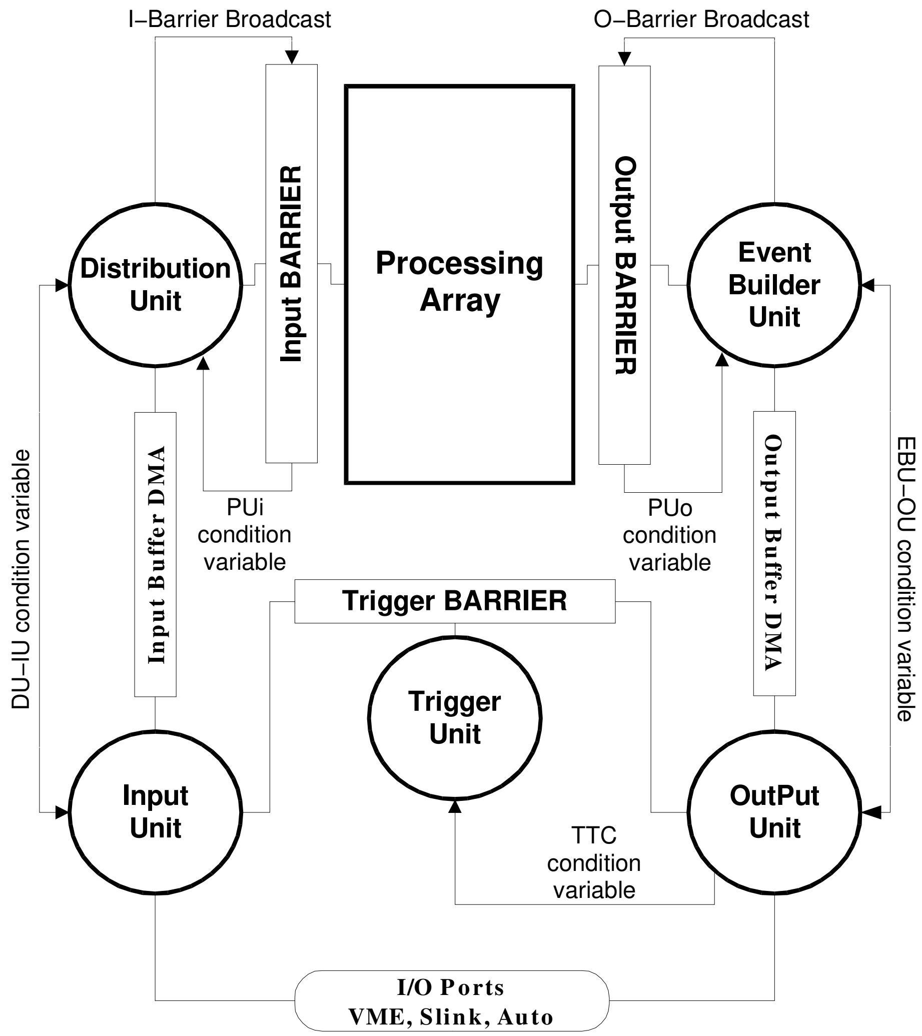

Figure 5: *ROD-Module Model. Unit Distribution*

3.3 Threads Distribution

The ROD-Module application is a fully multi-threaded program, due to the ROD definition. The ROD-Model is divided in several threads providing different kind of services. Basically, we have two type of operations:

- *I/O Management.* The I/O threads will use the hardware objects related with the synchronous data flow (either S-Link or VME ports) using either DMA objects or Slave objects.
- *Data Management.* The remainder threads and the main program are not related with high-speed I/O operations. They are dedicated to data distribution, processing, control, and monitoring operations.

The ROD-Module application is fitted in the common bones structure as can be shown in figure 6. The threads distribution is the following:

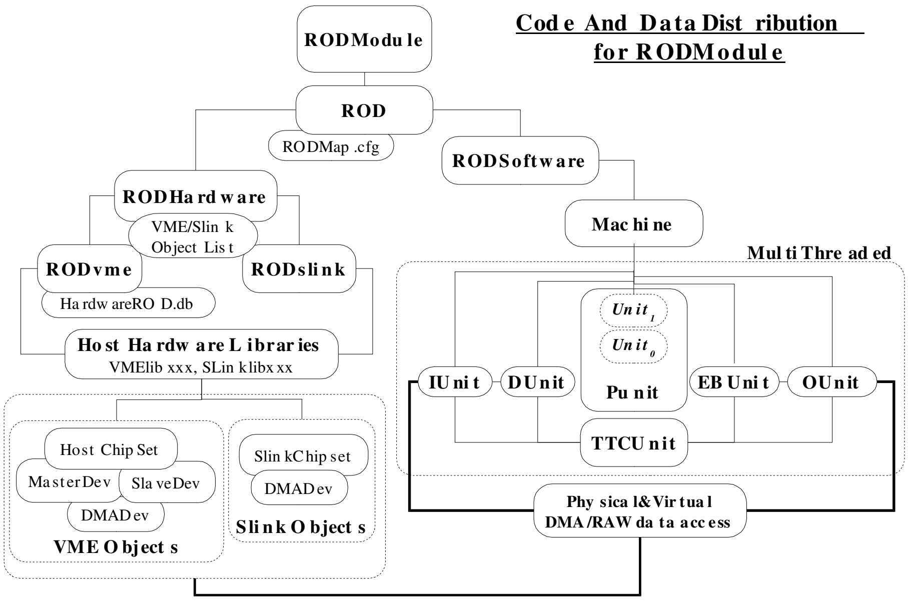

Figure 6: *ROD-Module Code and Data Distribution*

- *Main Thread.* It performs the On-Board controller roll. It is responsible of the software Boot/Shutdown and process the incoming commands on Local CSR issued from the crate controller. A slept switch-case decodes the command word, if some change is detected on the local VME CSR command address. The decoder states-machine follows a simple set of rules depicted in the picture 7 to verify user commands. The current command set expected from the crate controller is:
  - *RUN.* Start-Up the Data-Injection resources with the present SetUp
  - *STOP* Kill the Data-Injection resources.
  - *PAUSE.* Trigger Veto which stops the trigger generation.
  - *CONTINUE.* Reinits the trigger generation

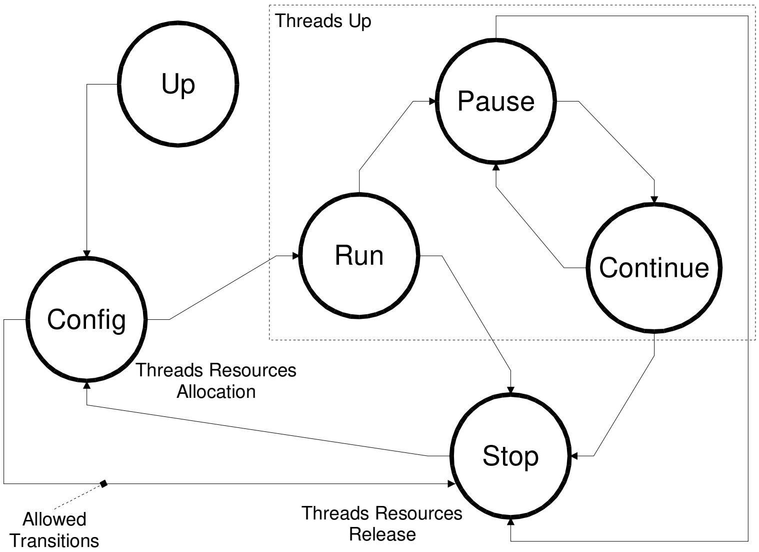

Figure 7: *ROD-Module Controller States Machine*

- *CONFIG.* Config the ROD-Module according the 3 long words located at CSR space offsets bs +2 (Input Flow), bs +3 (Output Flow), bs +4 (Trigger Word). The words are formatted following the patterns shown at Table 2.

dummy_{8b}	IOType_{2b}	$\mathrm{FormatID}_{2b}$	IOMode_{2b}	$\mathrm{SRCType}_{2b}$	NoSmpl_{8b}	NoChan_{8b}	dummy_{8b}
Output Flow (Formatted as above)							
Trigger word_{4b}							

Format-ID

Symbol	*value*
Calibration	0
ATLAS	1
Monitor	2
ByPass	3

Trigger-Word

Symbol	*value*
NoneTRG	0
TTC	1
autoTRG	2
ctrlTRG	3

I/O-Mode

Symbol	*value*
Target	0
Driver	1

SRC-Type

Symbol	*value*
Controller	0
FrontEnd	1

I/O-Type

Symbol	*value*
Slink	0
VME	1
Auto	2

Table 2: *ROD-Module Config packet.*

- *SHUTDOWN.* Halt Rod-Module Software

This command set is updated periodically as soon as the new requirements are defined or changed.

- *TTCUnit.* TTC Engine is fully dedicated to the trigger interface, having a veto flag available to stop the TTC engine. Several working-modes are possible[6]:

[6] Only Auto-Trigger and Controller has been implemented living the other cases to future work

  - *Auto-Trigger.* Under this mode, the TTC engine internally generates and increases the event number and BCID counters, clocking the whole module operation and the read-out.
  - *ControllerTRG.* As above, but located inside Controller application, it will use the VME CSR space to exchange the trigger data.
  - *TTC.* Defined to interface the module with real trigger hardware devices as the TTC receiver for the PMC version[6] or the TTCvi VME card[7].

- *IUnit.* The *Input Unit* is designed to manage the *Input Data Flow* in the ROD-Module application. We consider this operation mode as a synchronous mechanism. The IU engine has virtual (Master And Slave CSRs) and physical access (DMAs, VME Slave buffers) to the VME and S-Link objects related. Three working-modes for the IUnit are available:
  - *Auto mode* internally suplies a fixed data pattern at the input buffer. This mode is useful when the external data sources are not available or to measure latencies in the ROD-Module application without I/O flow.
  - *VME mode* is provided for special runs, prototyping, etc. The VME DMAs are performed either inside the module (IUnit must be configured to work in *DRIVER* mode) or externally (IUnit configured as *TARGET*).
  - *S-Link mode.* This mode will be the normal one used by the Front-End application. It will use as a DMA-Slink-buffer the Input Buffer of the ROD-Module also mapped into the VME address space, in order to save memory resources.

- *DUnit.* The *Distribution Unit* has been implemented to be an interface between the processing threads and the IUnit. Different working modes could be applied to the processing matrix. Up to now, the input buffer processing is an open matter. Beside this, we expect to need an extra pre-processing overall data related with the synchronous ROD operation, and it should be done before moving data to the PU buffers. By this way, DUnit is understood as a decoupling item of such kind of operations.

- *PUnit.* The *Processor Unit* is a set of M concurrent threads which in general, can be configured from two points of view:
  - Miscelaneous processing. The Front-end event is considered as a single item, and each PU thread is configured with different tasks and algorithms to be performed on the whole event. Therefore, the FE event will be fully splitted to the PU input buffers.
  - Homogeneus processing. The Front-End event is divided in several pieces, each one routed from the ROD-Module input buffer to the PU input buffers. So, the whole data processing will be shared between M threads.

- *EBUnit.* The *Event Builder Unit* performs the data exchange between the PUs and the DMA output buffer, where the ATLAS event is built using the information provided by the TTCUnit. The same specifications as the DUnit can be applied to the EBU. It is not clear still how the output results are distributed on the output stage of the PUs matrix. Such matters must be decoupled from the point of view of the PUs and the Output Unit.

- *OUnit.* The *Output unit* is very similar to the IUnit because it manage the same kind of ports playing in a similar way. Very likely, it could be quoted in terms of VME and Slink objects management. The work already done is only related with the internals of the model. By this way, the ROD-Module output buffer will be either a data source or data target using both I/O ports VME and Slink. These operations could be performed by an external CPU (Controller, ROB, Custom ROB, etc) which should be also identified on a configuration parametter for the OUnit.

3.4 Present Status

We have checked the ROD model behaviour and implementation, without I/O flow under different OSs (OSF-1, Linux, LynxOS 2.5.1, and LynxOS-3.0) and hosts (Digital Alpha, PC desktops, MVME2604, MVME167 and RIO2-8061). In the figure 8 is shown a debugging trace of the RODMachine recorded on the MVME2604 under LynxOS-3.0. Each machine state is coded and time stamped using gettimeofday, therefore the record is an estimation of the real machine behaviour in timing terms. The latencies introduced by the module synchronisms and for 2 PUs are roughly 468 μsec for the RIO2-8061, 274 μsec for the PC-Linux and 174 μsec for the MVME2604. For the RODCrate the ROD-Model has been also fully checked mapping it in the MVME2604 card where the Slink and VME ports are availables, and therefore performing real I/O data cycles. To validate the cross-development the module has been also successfully mapped on the PC-Linux and MVME167 workstations using the VME port as the data flow channel.

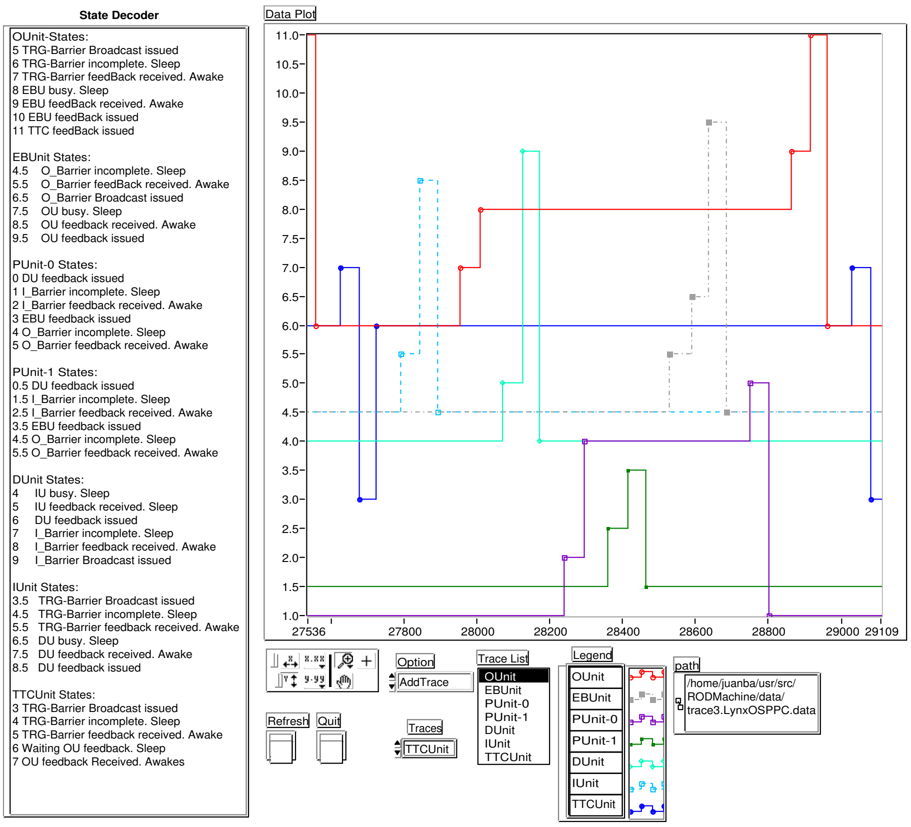

Figure 8: *ROD-Machine Debug Track*

4 RODController

The ROD controller ATLAS system for the TileCal will be a farm of VME workstations connected to the network. Each controller will fully use TCP-IP networking services providing external access (for instance from the *TileCal Run Control*) and VME capabilities to exchange data and control commands with the ROD-crate items (ROD-Modules, FrontEnd, TTC-interfaces, etc..).

4.1 Implementation Specifications

The ROD-crate controller issues for the current prototype are the following:

- **Booting and Shutdown of the ROD-crate:** ROD-Modules And Front-End Applications
- **ROD-Module and FrontEnd Configuration** using the VME bus
- **VME Data Injection** on ROD-Modules for special runs
- **A TCP-IP services layer** to interface the Tile-Cal run control with the *Main Controller thread*
- **ROD-Busy Monitoring** using a VME interrupt

The implementation is far to be fully developed for the present, because we have focused the work in the ROD-Module software design. As the module functionalities and operation modes are not completely defined, some issues can not be still designed.

4.2 Current Status

We have fitted the functionalities described above in the common bones object structure presented at 2.4, next sections show the work and items already going on.

4.2.1 ROD-crate Map

The VME ROD-Crate map is built at RODhardware.c following the data-base prototypes HardwareROD.db.dat and RODMap.cfg ASCII files.

Machine$_{id}$	Space$_{id}$	Object type	Base address	AMcode	Space size	Memory$_{id}$
localhost	CSR-Space	slv-VME	$00f23000_h$	$0f_h$	16384	MEM_31_00623000
localhost	Input-Buffer	slv-VME	$00f49000_h$	$0f_h$	16384	MEM_31_00649000
localhost	Output-Buffer	slv-VME	$00f69000_h$	$0f_h$	16384	MEM_31_00669000
147.156.2.75	CSR-Space	mst-VME	$0043f000_h$	09_h	16384	MEM_18_0043f000
147.156.2.75	Input-Buffer	dma-VME	00443000_h	$0f_h$	16384	MEM_18_00443000
147.156.2.75	Output-Buffer	dma-VME	00447000_h	$0f_h$	16384	MEM_18_00447000
147.156.2.81	CSR-Space	mst-VME	03000000_h	09_h	16384	dummy
147.156.2.81	Input-Buffer	dma-VME	03004000_h	$0f_h$	16384	dummy
147.156.2.81	Output-Buffer	dma-VME	03008000_h	$0f_h$	16384	dummy

Table 3: *RODHardware Data Base (HardwareROD.db.dat).*

```
# File format
# Single entries: Controller
# Multiple entries: Front_End_Section, Module_Section
# Section_xxx[tab]:[no_itm]

# Entries Format
# Controller[tab]ip_number
# fe[tab]ip_num[tab]No_mod[tab]IO_type[tab]mod-ip[tab] .....
# module[tab]ip_number[tab]FE_ip_number

# Values for IO_type: Both(Slink-VME), VME, Slink.

Front_End_Section:1
PC_fe1  147.156.2.81    1       Both    147.156.2.73
end

Module_Section:1
PPC_Module      147.156.2.73    147.156.2.81
end

Controller_Section:1
VME68K  147.156.2.75
end
```

Figure 9: *RODcfg ASCII file.*

4.2.2 ROD-Module And Front-End Booting

Using the ROD-systen map and the crate information, the controller builds an fullfills below structures to manage the Front-End and ROD-Module sets.

```
typedef struct{
  char   ip_addr[20];
  word32 io_type;
}Mod_Link;

typedef struct{
  char      id[20];
  char      ip_addr[20];
  int       no_modules;
  int       lives;
  int       status;
  word32   *l_csr;
  word32   *r_csr;
  word32    evt_size;
  Mod_Link *modules; /*+ Modules vector */
}Front_End;

typedef struct{
  char    id[20];
  char    ip_addr[20];
  word32  o_index; /*+ Index of Output Buffer Object, shortcut for DMAs +*/
  word32  i_index; /*+ Index of Input  Buffer Object, shortcut for DMAs +*/
  int     lives;
  int     status;
  word32 *l_csr;
  word32 *r_csr;
  word32 *r_ibuf;
```

```
  word32 *r_obuf;
  word32  evt_size;
  word32  fe_ptr;  /*+ Pointer to fe_set[i]. Beware!!! you need casting   +*/
}ROD_Module;

/*+ 02/June/1999: Virtual pointers to local Slave Buffers +*/
/*+ 07/October/1999: FrontEnd Vector +*/
typedef struct{
  VME_Devs   *vme;         /*+ RODVMECrate Configuration        +*/
  ROD_Module *module_set;  /*+ RODModule Vector                 +*/
  word32      mod_size;    /*+ Module size                      +*/
  Front_End  *fe_set;      /*+ FrontEnd  Vector                 +*/
  word32      fe_size;     /*+ FrontEnd size                    +*/
  word32     *l_ibuf;      /*+ VirtualAddresses of Local Buffers +*/
  word32     *l_obuf;
  word32     *l_csr;       /*+ VirtualAddresses of Local CSR[0]  +*/
}ROD_Hardware;
```

With above data patterns, the detection protocol follows the next rule for both sets. On the local CSR map depicted in figure 3 the controller masks the first location for each item with the ROD_DEAD value. For each one, a timed loop checks the initial mask value until it changes or time is over. If the $item_i$ does not change the mask value, the item will be marked as DOWN. Otherwise, the controller handshakes the cycle writing CONTROLLER_ALIVEs on the VME Slave CSR $offset_0$ of the $Module_i$, which is previously masked by the rod-item software. This procedure is iterated over the module and Front-End whole set.

4.2.3 Module-FrontEnd Commands

VME Single and Broadcast primitives have been implemented either to communicate control commands or to receive status information of the modules and-or front-ends. Their fully use Master-Slave VME CSR spaces on the local and remotes machines as the ATLAS DAQ will do. For each command issued from the ROD-controller, an ACK and STATUS call-back from the module/front-end software is expected on the local controller CSR, at offsets base +1, base +2, where base corresponds with the external $device_i$. If ACK is not received the $device_i$ lives control parameter will be decreased once. When this counter reaches zero value, the device is marked as DOWN avoiding future references to the object. The primitives to send single and broadcasting commands are shown below.

```
int  Module_BroadCast   ( ROD*, word32 cmd );
int  Module_Command     ( ROD*, word32 module, word32 cmd, char *stat );
int  FEnd_BroadCast     ( ROD*, word32 cmd );
int  FEnd_Command       ( ROD*, word32 module, word32 cmd, char *stat );
```

Some commands related with the Controller data injection, STOP, PAUSE, RUN, CONTINUE, are internally filtered at controller level to avoid violations of the state machine depicted at 7. If the data injection is related with the front-end, the Module-commands are also forwared to the Front-Ends automatically.

4.2.4 Asynchronous ROD-Module sampling

It is performed on controller by means of Async_Get_Event and Get_Module_Stat interfaces. The user parameters module and bs_addr are the module index to be sampled and the user pointer where the ROD-Module output buffer will be VME mapped. The input pointer should be NULL pointer, because the function routes it to the DMA buffer associated to the VME object. We have preferred to use direct access to buffers instead of double buffering techniques which add latency. The VME data reading is performed using DMAs over the VME object associated. For the Module CSR monitoring a simple memcpy is performed with Get_Module_Stat. Status info can be decoded at user level using Decode_Module_Stat function.

```
int  Async_Get_Event    ( ROD_itm*, word32 module, word32 *bs_addr );
int  Get_Module_Stat    ( ROD_itm*, word32 module, word32 *state   );
void Decode_Module_Stat ( word32 *status );
```

4.2.5 ROD-Module configuration

The module configuration is user interfaced with ROD_Config function.

```
int  ROD_Config         ( ROD*, word32 module, word32 config_data );
```

The CONFIG command is also automatically forwarded to the Front-End if it is selected to perform the Data-Injection. The Configuration packet is implemented using bit-fields on 32bit words and full words assembled both into a word32 array. Below is shown an extract of the *configuration bones*, some of them cover the values presented at the table 2 for the input-output stages of modules.

```
cfg[ 0 ].all             = __ConfigPackSize__ ;
cfg[ 1 ].all             =   CLEAR;
cfg[ 1 ].bit.io_mode     = __Target__ ;
cfg[ 1 ].bit.src_type    = __FrontEnd__ ;
cfg[ 1 ].bit.no_channels = __NoChannels__ ;
cfg[ 1 ].bit.no_samples  = __NoSamples__ ;
cfg[ 1 ].bit.io_type     = __Slink__ ;
cfg[ 1 ].bit.format_id   = __Monitor__ ;

cfg[ 2 ].all             =   CLEAR;
cfg[ 2 ].bit.io_mode     = __Target__ ;
cfg[ 2 ].bit.src_type    = __FrontEnd__ ;
cfg[ 2 ].bit.no_channels = __NoChannels__ ;
cfg[ 2 ].bit.no_samples  = __NoSamples__ ;
cfg[ 2 ].bit.io_type     = __Auto__ ;
cfg[ 2 ].bit.format_id   = __ByPass__ ;
cfg[ 3 ].all             = __AutoTRG__ ;

if( ROD_Config( rod, 0, ( word32 )( &cfg[ 0 ] )) != FAIL ){
  Module_Command( rod, 0, RUN, status );
  sleep( 10 );
  ...
  ...
```

4.2.6 VME Data Injection

It is considered as a multi-threaded operation in the ROD-controller program flow. The tasks have been divided in four threads which are fired up at RODsoftware.c using Run_Software function interface , the VME resources required are indexes over io_data array. The different modes are filtered at the set-up before the data injection starts. By this way, the protocols can be perfomed directly, decreasing the latencies. Below is an extract of how-to the ROD_Software structure is broken:

```
/* ----------------- Data Types To Manage VME Data Injection --------------- */
typedef struct{
  char             io_id[ 20 ]; /*+ Thread Identifier                    +*/
  word32          *buffer;      /*+ Driver Mode - VME data Objects       +*/
  word32           dma_obj;     /*+ Target Mode                          +*/
  word32          *l_csr;       /*+ Local  CSR vp.                       +*/
  word32          *r_csr;       /*+ Remote CSR vp.                       +*/
  word32           cfg;         /*+ Module Configuration Word            +*/
  word32          *stat;        /*+ Common Flag {STOP,PAUSE,CONTINUE}  +*/
  word32          *evt_buf;     /*+ Event Buffer Ptr                     +*/
  /* Synchronization tools */
  int              io_ready;
  int             *ready;       /*+ ptr to Productor-Monitor flag        +*/
  pthread_mutex_t *io_mtx;      /*+ To Lock ready flags and cv           +*/
  pthread_cond_t  *io_cv;       /*+ Syncronism                           +*/
  pthread_t        phy_th;      /*+ Physical Thread                      +*/
}IO_Type;
int Input_Engine        ( void *par );
int Output_Engine       ( void *par );
int In_Module_Handshake ( IO_Type*, word32 count );

typedef struct{
  word32           cfg;         /*+ Module Configuration Word            +*/
  word32          *stat;        /*+ Control Flag {STOP,PAUSE,CONTINUE} +*/
  word32          *evt_buf;     /*+ Event Buffer Ptr                     +*/
  /* Synchronization tools */
  int             *io_ready;    /*+ ptr to IO flag                       +*/
  int              ready;       /*+ Local Flag                           +*/
  pthread_mutex_t io_mtx;       /*+ To Lock ready flags and cv           +*/
  pthread_cond_t  io_cv;        /*+ IO condition variable                +*/
  pthread_t       phy_th;       /*+ Physical Thread                      +*/
}Productor;
int Production_Engine ( void *par );

typedef struct{
  word32           cfg;         /*+ Module Configuration Word            +*/
  word32          *stat;        /*+ Control Flag {STOP,PAUSE,CONTINUE} +*/
  word32          *evt_buf;     /*+ Event Buffer Ptr                     +*/
  /* Synchronization tools */
  int             *io_ready;    /*+ ptr to IO flag                       +*/
  int              ready;       /*+ Local Flag                           +*/
  pthread_mutex_t io_mtx;       /*+ To Lock ready flags and cv           +*/
  pthread_cond_t  io_cv;        /*+ Syncronism                           +*/
  pthread_t       phy_th;       /*+ Physical Thread                      +*/
}Monitor;
int Monitor_Engine ( void* par );

/*+ ROD-Software Object +*/
typedef struct{
  IO_Type    in;
  IO_Type    out;
```

```
  Monitor    mon;
  Productor  pro;
  word32    *stat;
}ROD_Software;
```

state field will monitor the software-thread state, the memory address is referenced from the threads using *stat pointer. Each thread-engine will monitor this flag to break the endless loop if some mechanism shutdowns externally the VME data injection procedure. The interface between the data consumers and producers is implemented with shared buffers and pthread synchronization primitives (mutex and condition variables). The I/O Input-Producter connection has been fully implemented and the I/O Output-Monitor flow has been left as future work.

4.3 Future Work

The presented services and behaviours have been checked and driven on the ROD-Controller application using an embbeded standalone system. However, it is necessary to implement a clear interface with the BackEnd Software, which is network based. A TCP-IP services layer is an issue which should provides the already available local services over the network. This software piece should also run concurrently with the rest of ROD-Controller threads. Two implementation approaches are currently under study: the first one is based on a raw TCP-IP socket custom library which could use the forward socket channel to ACK commands and to publish the modules status with the external back-end software. The backward channel could be used in a select fashion by the main RODcontroller thread, waiting incomming commands from the network. The second one based on some open-source implementation of CORBA, which provides a remote mapping of the local procedures on the network-client.

5 Front-End

For the present, the Front-End will be physically considered as a VME module with the possibility to have also S-link(LSC) capabilities. In our case, the PC-Linux box have such kind of interfaces. The rest of the platforms availables only have the VME port to perform the data and control exchange. Two basic scenarios are possible to map the Front-End on the Modules: one to one, and one to many. For the present, we have restricted the tests carried out to one-one version. However, we bear in mind that the many-one mapping should be necessary and fully checked. The object distribution for the Front-End application is the one already used for the ROD-Controller.

5.1 Modules Map

The ROD_hardware data structure for the FrontEnd application is depicted below.

```
/**+ ROD hardware Data Types ++++++++++++++++++++++++++++++++++++++++++++++++ */
/**+ ROD Controller ++*/
typedef struct{
  char          id[20];
  char          ip_addr[20];
  word32        o_index;  /**+ Index of Output Buffer Object, DMAs shortcut  +*/
```

```
  word32        i_index;  /*+ Index of Input  Buffer Object, DMAs shortcut  +*/
  word32       *l_csr;    /*+ VAddr(ctrl_base) of Local  CSR, VME Slave     +*/
  word32       *r_csr;    /*+ VAddr(ctrl_base) of Remote CSR, VME Master    +*/
  int           lives;
  int           status;
}ROD_Ctrl;

/*+ ROD Module +*/
typedef struct{
  char         id[20];
  char         ip_addr[20];
  word32       o_index;  /*+ Index of Output Buffer Object, DMAs shortcut   +*/
  word32       i_index;  /*+ Index of Input  Buffer Object, DMAs shortcut   +*/
  word32      *l_csr;    /*+ VAddr(m_base) Local CSR, VME Slave Segment_i   +*/
  word32      *r_csr;    /*+ VAddr(m_base) Remote CSR, VME Master Segment_i +*/
  word32      *r_ibuf;   /*+ VAddr(m_base) of Ibuffer, VME DMA              +*/
  word32      *r_obuf;   /*+ VAddr(m_base) of Obuffer, VME DMA              +*/
  word32       evt_size;
  int          lives;
  int          status;
  int          io_type;
}ROD_Module;

typedef struct{
#if defined( LinuxPCBit3 )
  ROD_slink   *slink;
#endif
  ROD_vme     *vme;      /*+ RODVMECrate Configuration                      +*/
  word32       mod_size; /*+ Module size                                    +*/
  ROD_Module  *mod_set;  /*+ RODModule Vector                               +*/
  ROD_Ctrl     ctrl;     /*+ ROD Controller Resources                       +*/
  word32       o_index;  /*+ Index of Local Slave Objects                   +*/
  word32       i_index;
  word32      *l_ibuf;   /*+ VAddr(base) of Local Buffers, VME Slaves       +*/
  word32      *l_obuf;
  word32      *l_csr;    /*+ VAddr(base) of local CSR, VME Slave            +*/
}ROD_Hardware;
```

5.2 Control Flow

The Front-End application receives commands on the VME local-CSR from the ROD-Controller. The switch-case structure is quoted from the ROD-Module main due to both are strongly coupled in behaviour terms. All the controller-commands are handshaked following the same scheme that the ROD-module software. The states-machine already developed for the ROD-Module 7, is applied to the switch-case control structure.

5.3 Data Injection

The Data Injection procedure is managed inside the RODSoftware.o object, which is designed as a multithreaded object. Production and Input engines share a buffer where the data is pushed by the producter. A mutex, a condition variable and a busy flag couple both program flows. The input engine is responsible to exchange data and trigger flows with the target module using the VME and Slink data objects related. Both engines shares with upper layers a backward pointer to a status flag, to PAUSE, CONTINUE, RUN and STOP the data injection procedure.

The interface with upper layers is done using the Open_Software, Close_Software, Config_Software and Run_Software functions, but the real multithreaded action starts in fact on a Run_Software user call. All of them use the ROD_Software substructure side of the super-structure ROD. The external data transaction is performed with In_Handshake_VME and In_Handshake_Slink, exchanging both the trigger flow throught the CSRs mapped in the VME address space.

We overlapp the data production with the external DMA (either Slink or VME mapped) cause both operations are deocupled. The protocol use memory locations from lUHandshake, +1 to +6, where lUHandshake is some offset starting from the l_csr[0] which is also another offset over the local slave CSR space. Some of their are used to monitor trigger statistics and error counters related with the data injection. The VME handshake is already able to work in TARGET and DRIVER modes. A local user polling is used to control incoming triggers produced, up to now, at the ROD-Module host.

6 Building ROD-Applications using cross-sources

The cross-source files are distributed along a directories tree as can be shown on the next pages, the @ code means links to external directories which relying on the implementation of low-level resources as are the VME, Slink or Memory interfaces which are platform dependent. The main script to compile the ROD-software is runmake. It resolvs the host and target dependencies using a different number of input parametters.

The host-target is specified in the first input parametter, environments and paths dependencies related with the host will be tuned. Up to now, the values MVME2604, LinuxPC and MVME167 can be applied. The rod-target is specified at the second position, front_end, controller, module and all should be applied. A third input parameter is used to build and maintain the object and binary directories starting from the /home/user-x/bin path, the values SetUp, all and clean should be used. It is also possible to build a specific object passing its name also as third parametter. The scripts and makefiles maintain updated as well, the objects related with the low-level layers which are platform-dependent, we have preferred to use links to such objects to reduce disk-usage and interferences with other developments.

6.1 Examples

This Section ilustrate how-to produce ROD-Applications, using the scripts provided in the package.

- To do a CONTROLLER on the MVME167-lynx-2.5.1 stuff
  1. create an alias of source runmake called doit
  2. doit MVME167 SetUp. It creates the objects and binaries tree
  3. doit MVME167 controller SetUp. It checks and-or builds links to the VME and Memory stuff
  4. doit MVME167 controller all
- To do a MODULE for the LinuxPC box

1. Same as above if you don't have one yet
2. doit LinuxPC SetUp
3. doit LinuxPC module SetUp
4. doit LinuxPC module all
5. To remove the module stuff: doit LinuxPC module clean

- To do whatever for the MVME2604 PowerPC

1. Idem that above
2. doit MVME2604 SetUp
3. doit MVME2604 all SetUp
4. doit MVME2604 all
5. To remove all doit MVME2604 all clean

- To Get Help: doit help, doit –help,

```
*Source Tree                   *Object And Binary Tree
RodCode                         home/{user}/bin/
|_____runmake.csh                      |__LinuxCode
|_____Makerules                        |    |
|_____MVME2604-ppc-lynx-3.0.csh        |    |
|_____MVME167-68k-lynx-2.5.1.csh       |    |_rod
|_____i686-linux-2.2.12-20.csh         |        |
|_____src                              |        |__controller
|       |__controller                  |        |   |__OBJs
|       |           |__inc@            |        |   |    |__ROD.o
|       |           |__Makefile        |        |   |    |__RODhardware.o
|       |                              |        |   |    |__RODSoftware.o
|       |__front-end                   |        |   |    |__RODvme.o
|       |          |__inc@             |        |   |    |__libROD.a
|       |          |__Makefile         |        |   |    |__libRODhardware.a
|       |                              |        |   |    |__libVMEBit3.a@
|       |__module                      |        |   |__RODController@
|              |__inc@                 |        |
|              |__Makefile             |        |___fe
|                                      |        |    |_OBJs
|                                      |        |    |   |__ROD.o
|                                      |        |    |   |__RODhardware.o
|_____include                          |        |    |   |__RODSoftware.o
            |                          |        |    |   |__RODvme.o
            |__controller              |        |    |   |__libROD.a
            |        |__src@           |        |    |   |__libRODhardware.a
            |        |__UIO-2.5.1@     |        |    |   |__libVMEBit3.a@
            |        |__UIO-3.0@       |        |    |   |__libSlinkLinux.a@
            |        |__Universe@      |        |    |___FrontEnd@
            |        |__VMEchip2@      |        |
            |        |__Bit3Linux@     |        |___module
            |__front-end               |             |_OBJs
            |        |___src@          |             |   |__ROD.o
            |        |___UIO-2.5.1@    |             |   |__RODhardware.o
            |        |___UIO3.0@       |             |   |__RODSoftware.o
            |        |___Slink-Linux@  |             |   |__RODvme.o
            |        |___VMEchip2@     |             |   |__Machine.o
            |        |___Universe@     |             |   |__IUnit.o
            |        |___Bit3Linux@    |             |   |__DUnit.o
            |                          |             |   |__EBUnit.o
            |__module                  |             |   |__PUnit.o
                    |____src@          |             |   |__OUnit.o
                    |____UIO-2.5.1@    |             |   |__TTCUnit.o
                    |____UIO3.0@       |             |   |__libROD.a
                    |____Slink-LynxOS@ |             |   |__libRODhardware.a
                    |____VMEchip2@     |             |   |__libRODMachine.a
                    |____Universe@     |             |   |__libVMEBit3.a@
                    |____Bit3Linux@    |             |__RODModule@
                                       |__LynxOSCode
                                            |__MVME2604(Idem than LinuxCode)
                                            |__MVME167(Idem than abv.)
                                            |__RIO2-8061(Idem than abv.)
```

References

[1] TileCal ROD pages. *http://ific.uv.es/tical/rod*

[2] S-link studies for the TileCal RODs. *http://ific.uv.es/tical/rod/slink*

[3] TileCal ROD, Software Reference. *http://ific.uv.es/tical/rod/CodeProduction.html*

[4] The Liquid Argon ROD system. *http://wwwinfo.cern.ch/ efthymzp/rod/larg-rodpage.html*

[5] S-link at CERN. *http://www.cern.ch/HSI/s-link*

[6] A TTC interface por the PCI bus. *http://www.cern.ch/TTC/TTCsrSpec.pdf*

[7] The VME interface to control the ATLAS TTC systemhttp://www.cern.ch/TTC/TTCviSpec.pdf