Software and DAQ for the CMS Silicon Tracker Front End Driver

J. Fulcher [1], L. Mirabito[4], J. Coughlan[2], G. Iles [1], R. Bainbridge [1], I. Church [2], E. Corrin [1], C. Foudas [1], E.J. Freeman [2],

G. Hall [1], R.N.J. Halsall [2], G. Iles [1], J. Leaver [1], M. Noy [1], M. Pearson [2], I. Reid [3],

M. Ageron[4], G. Rogers [2], J. Salisbury [2], S. Taghavi [2], I.R. Tomalin [2], O. Zorba [1]

[1] Imperial College, London, UK. [2] CCLRC Rutherford Appleton Laboratory, Oxfordshire, UK
[3]Brunel University, London, UK. [4]Institut de Physique Nucléaire, Lyon, France

Abstract

The CMS Silicon Tracker Front End Driver is a 9U 400mm VME64x card that digitizes and processes the analogue raw data, generated within the silicon tracker by the APV25[1] readout ASICs. The compressed data is then sent to the Data Acquisition(DAQ). The current tracker DAQ is fully based on the CMS communication and acquisition tool called XDAQ[2]. This paper discusses the software and DAQ methodology developed in order to initialize, control and readout data from the FEDs. Data illustrating the performance of the software and data readout are then presented.

I. Introduction

The CMS Silicon Strip Tracker(SST) Front End Driver (FED)[3][4] is a 9U 400mm VME64x card that processes the raw data from 192 APV25 silicon readout ASICs, corresponding to 0.2% of the total tracker. After multiplexing and streaming, the data from the front-end are routed via analogue optical links to the FEDs. 96 optical channels are then digitised to 10bit precision at 40MHz and processed in large FPGAs, before being collated into events and sent to the CMS DAQ via either VME or the SLINK-64 protocol.

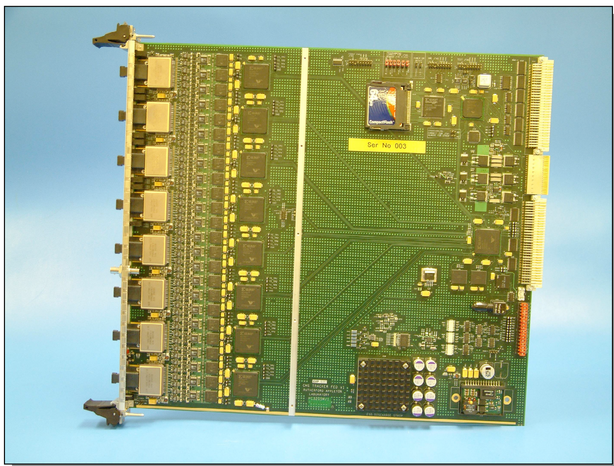

Figure 1: 9U VME FED

The software specific to the FED has been developed in an object-oriented manner using the programming language c++. The hardware components of the FED have been mimicked by software class equivalents and the data that are required for configuration of the FED are stored in a class providing a neat data container of necessary configuration information. The software structure is modular and layered, at each layer a level of abstraction is achieved, making the code manageable and extensible. Before delving into more detail about the software we consider the approach that has been taken.

II. Software Approach

Software in HEP has come a long way in the last few decades, going from punch card machine code through monolithic Fortran programs right up to the modern era of an object oriented modular approach in c++. The benefits of object orientation are well discussed and not the topic of this paper[5].

A. *Object Orientation*

The tracker itself has been designed in a modular manner and it makes perfect sense to carry this methodology though into the architecture of the software. For the parts of the Tracker system which require software interaction, which is nearly everything, (APV ASICs at the front end, PLL clock control, analogue optical laser drivers, temperature readout, clock and trigger distribution right up to the FED itself) it is important that the software is clearly defined, easily understandable and simple to integrate into the final software system. Therefore, the individual parts of the software have been based upon object representations of the subsystems and have been developed in the c++ language using standard object oriented axioms. In the case of the FED, this resulted in the development of a C++ class called *Fed9UDevice* which, although not the complete software required to operate the FED, nevertheless embodies the FED as a software object.

B. *Modularity and Extensibility*

One major consideration when designing software projects of any kind is the fact that it, or parts of it, may well be required in future projects, either due to the inclusion of its corresponding hardware in another experiment or due to an architectural rethink on the part of the integrating framework. In either case it is important that the package in question is designed in a way that makes future extensions or utilisations as effortless as possible.

There are clear distinctions between the hardware interfaces and the purely software objects used for auxiliary tasks such as file handling and data manipulation, to name just a few. Therefore, it is prudent to separate these tasks into different software objects.

C. *The Layered Approach*

One method of dividing up larger chunks of software into manageable parts is to follow a layered approach. In this instance, one builds a base class, which can then either be inherited from and functionally augmented one layer at a time, or can be used as a data member of a higher order class, either approaches result in a project

which is both easy to divide up amongst developers and easy to adapt and extend in future iterations.

III. FED SOFTWARE

A. Overview of Architecture

The structure of the FED specific software can be seen in Figure 2; each grey ellipse represents a software class. An attempt has been made to demonstrate how each class fits into the context of another class, for example the *Fed9UHALInterface* class can be seen to be contained within the *Fed9UVMEBase* class. In this instance it is included as a member variable of this class. The red arrows show class inheritance; the main inheritance of note is that of the *Fed9UVMEDevice* and the *Fed9UDescription* from the *Fed9UABC* abstract base class; the purpose of which is to fix the interface of both classes to be identical. Thus, programmatically speaking, the software representation of the FED in the *Fed9UVMEDevice* and that of the container class for the FED configuration data, are identical and one could interchangeably call methods on objects of both class types without need to rewrite code.

B. Hardware interface abstraction

The VME crates are controlled by a PCI-VME interface. The controller of choice during development has been the SBS[6], although this will almost certainly change in the final system. The low-level driver software is hidden behind an abstract API - Hardware Access Library[7] (HAL) developed in CMS to perform basic input/output operation on the bus. In order to be immune to any future changes of this API, a FED HAL wrapper interface (*Fed9UhalInterface*) was developed, which provides a set of read/write commands that have been fixed for the lifetime of the FED software thus providing a stable basis upon which to build the project.

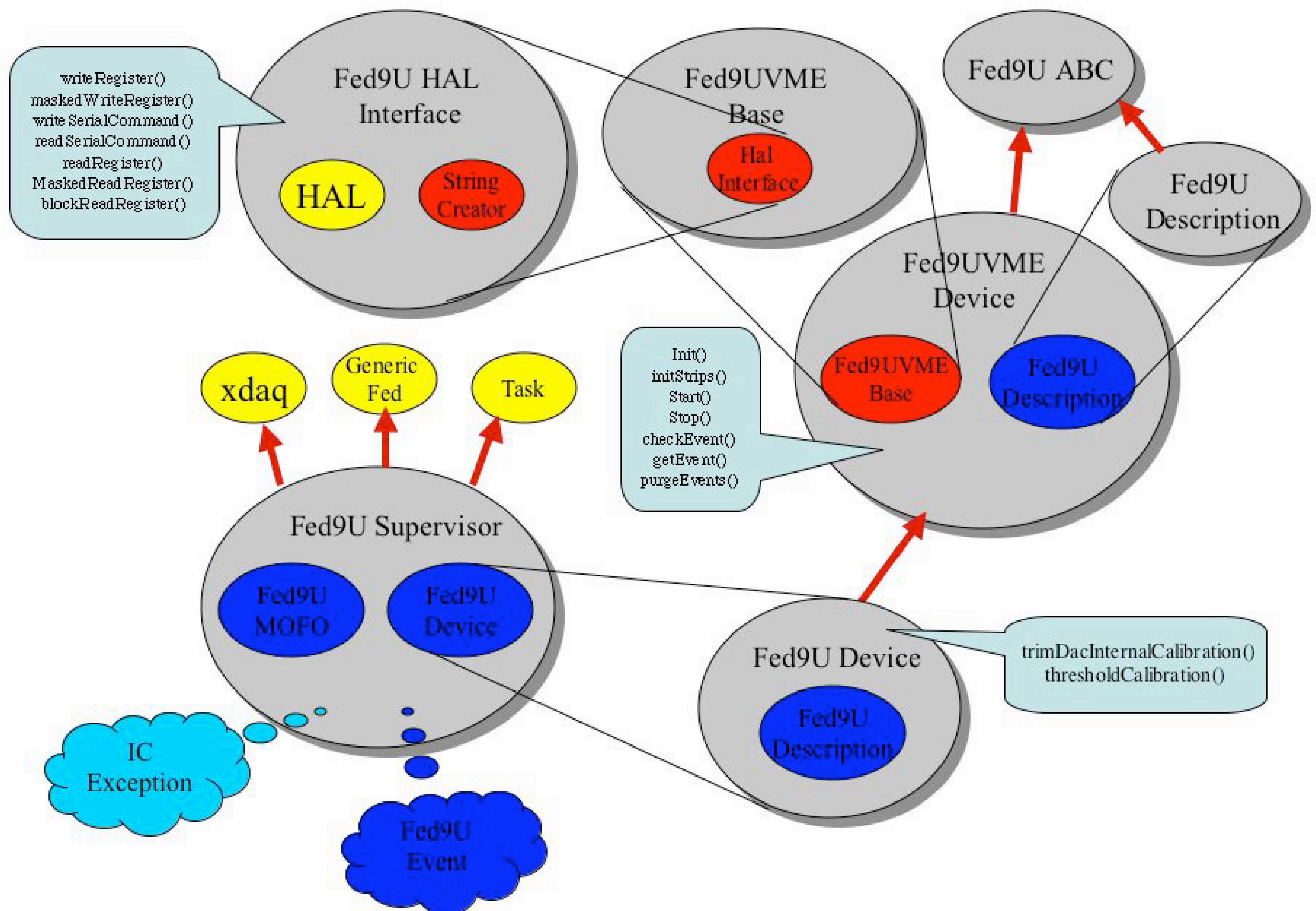

Figure 2: Schematic representation of the FED software project

Were the choice of controller to change, then the low-level software drivers for such a controller would be different from that of the SBS, this could result in a change in the HAL, or even a replacement for it. By setting a concrete interface to the hardware in the *Fed9UHalInterface*, we maintain the flexibility to easily adapt the software to any combination of controller and driver with a minimum overhead.

C. Low Level Hardware Communication

The next layer is the FED VME Base commands library (*Fed9UVMEBase*). All access to the FED is achieved via a large set of serial commands within the FED, the *Fed9UVMEBase* abstracts from these serial commands and provides a simplified API to the FED internal registers. An example interface would be a method such as setFineDelay(address, delay). The usage of this method is

self-explanatory enabling the user to set specific clock delay values on a channel by channel basis, but the actual implementation of this action within the *Fed9UVMEBase* is a more complicated process where one first constructs the serial command to send to the FED and then sends it via the VME bus. The purpose of this class is clearly to simplify the interface to the FED by abstracting from the complicated serial commands and exporting simple single method calls for each function.

D. Addressing Areas of the FED Hardware

In order to address the specific areas within the complicated architecture of the FED, a *Fed9UAddress* class was developed. This class encompasses all device space within the FED from the individual strips in each APV channel right through the APVs on each channel, the channel on each front end module and the backend and VME FPGA. The address internal to the *Fed9UAddress* class is stored in a format that gives the unique address as understood by the FED, but the user passes the address around in one object, and since nearly everything in the FED can be addressed by a *Fed9Uaddress*, it makes the API to the FED registers in the *Fed9UVMEDevice* class (see below) very neat. All methods that require an address of a strip, APV, channel or FPGA, to define exactly which object we are communicating with, take the *Fed9UAddress* specific to that location, instead of some methods taking an FPGA number, some taking a channel number, some taking a strip number and so on.

E. Fed Event Class

One hugely important aspect of the FED software is the handling of the output data packets. Each data packet is called an event since in the final system each of these packets is produced on the receipt of a level 1 trigger and contain all the data from 192 APVs for a specific bunch crossing. The purpose of the *Fed9UEvent* class is to provide a neat handle to the data and also to perform integrity checking on the data and provide status information about the timing of each channel.

F. Fed VME Device Class

The next layer up is the *Fed9UVmeDevice*, which serves three main purposes. Firstly it incorporates the *Fed9UAddress* class as the interface to hardware addresses of parts of the FED, secondly it implements the API defined within the *Fed9UABC* base class, providing as simple an interface as possible to all aspects of the FED's configurable hardware. Finally it wraps groups of base commands to provide the high level API to the FED including methods such as init(), start(), stop(), checkEvent(), getEvent(), purgeEvents(). During the construction of this class, a *Fed9UDescription* object is required, from which all the configuration data for the FED is passed through into the FED registers. All required configuration tasks are performed within the two main initialisation methods: init() and initStrips(). Therefore, it is almost never necessary to call individual methods on a per register basis when using the FED in the final system. However, this is clearly possible when debugging the FED system, and the highest-level debug software (see below), *FedDebugSuite.exe*, provides a simple interface to the full register by register API.

G. FED High Level API

The *Fed9UDevice* class is derived directly from the *Fed9UVmeDevice*. It is actually this class that is instantiated within the DAQ software directly providing a handle on the FED. As well as exporting all the methods from the previous layer, it adds more complicated tasks such as calibration of input parameters and timing of the ADC in each of the 96 channels at the front end.

H. FED XDAQ Supervisor

The *Fed9USupervisor* is responsible for the interface to the XDAQ framework in which the DAQ software is embedded. Each FED in the final system is represented by a unique instantiation of the *Fed9USupervisor*, within each there is a pointer to the *Fed9UDevice* object through which all software communication with the FED is performed. The API to the *Fed9USupervisor* is predefined by the XDAQ framework basically consisting of: Configure(), Enable(), Disable(), and a SOAP[8] command handler which is used to set run time states of the FED on the fly, such as changing the readout mode from Scope to Virgin Raw, or Zero Suppressed.

I. FED Debug Software and GUI

During testing and development stages of the project, it was necessary to have a comprehensive debug tool for reading and writing on a per register basis, and performing simple readout loops for data handling analysis. To this end the *FedDebugSuite.exe* and the *FedGui.exe* were developed. The Debug Suite is a command line based application which provides a simple interface to the configuration data in the *Fed9UDescription*, and also a direct interface to the *Fed9UVmeDevice* API, providing a hands on control panel from which one is able to directly read from and write to configuration registers inside the FED. The GUI provides a very simple user interface for reading editing and saving FED descriptions to XML files. The later generation of the GUI now also gives a direct handle on the registers in the FED using the same user interface as that for creating descriptions, this was achievable very easily since the API for the *Fed9UDescription* and that of the *Fed9UVmeDevice* were identically predefined by the *Fed9UABC*. By simply calling the interface on a pointer to the base class, one is able to invoke the corresponding method on either a FED description or an actual FED hardware object, without need for adaption of code. Figure 3 shows the basic GUI to the front-end registers in the FED.

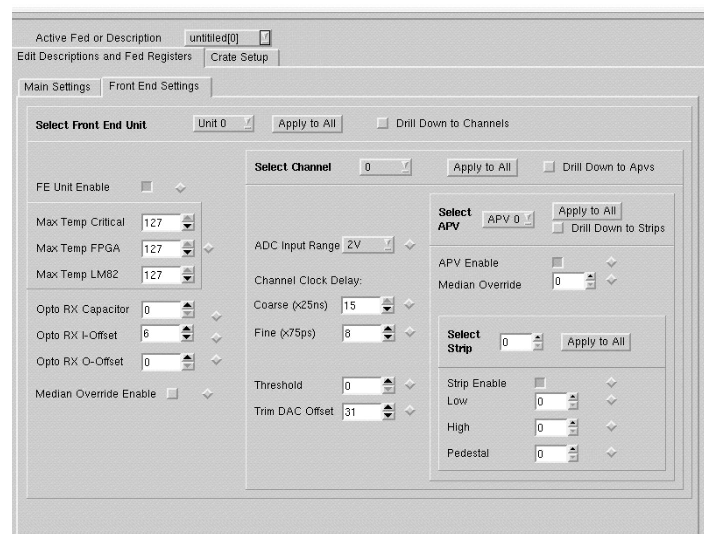

Figure 3: Graphical user interface to fed debug software

J. Error Handling

One very important aspect of any software project is the handling of errors. This rather well avoided subject should be paid great attention especially in projects that involve such an integrated approach to hardware and software. If a piece of hardware is not functioning as expected for one of many reasons, it is important that the error is tracked and handled in a manner that can expedite its resolution. To this end the *ICException* class was developed, which provides ASSERT and VERIFY macros for ease of coding - such macros provide a neat framework in which to test for potential errors. Example syntax would be:

```
VERIFY(x<100)(x).msg("x < 100").err().code(CODE)
```

The exception that is thrown from such macros is an object of type *ICException* (or a derived version), in which all the required information about the error, where it occurred, when, what the values of the error were, a description of the error, and error code for programmatic resolution wherever possible.

IV. TRACKER DAQ SOFTWARE

The DAQ itself consists of independent web server applications capable of loading plug-ins containing the actual applications (FED Supervisor application, Trigger application, Event Builder, Filter Unit and so on). The web server approach allows to have distributed computing with easy configuration by the mean of SOAP messages. The data collection is achieved using an asynchronous event builder connecting the data source (FED) to the analysing applications (Filter Unit). Data transfer is performed using[9] message packets. The integration in this framework of the FED was eased by the use of several adaptor classes interfacing it to the Event Builder. This adaptor method allowed a fast integration of this new hardware in the existing data acquisition system used for test beams and detector commissioning. Additionally the *FED9USupervisor* application implements connections to the configuration Data Base and to the error management system. Preliminary measurements show a comparable performance of the card in standalone running or in the DAQ framework.

A. Requirements

The tracker data acquisition[10] has to achieve two main tasks:

It should first provide an infrastructure to read Si detectors during the test and assembly phase of the project. It implies scalability in terms of data flow since setups can handle from one to several hundred detectors. The final system will require 440 FEDs reading out a total of 75,000 APV chips corresponding to 37,500 optical channels. Clearly, scalability of the software is essential.

Secondly, it should provide tools to commission the tracker before it is included in the central control and data acquisition of CMS and to configure it. This constraint forces the adoption of the final architecture as the default framework in which to develop the software and to customize the final parts to the needs of the Tracker.

Consequently the XDAQ environment developed for the CMS online software was chosen in early 2001 to facilitate tracker data acquisition tasks. It offers both an application deployment and configuration environment based on Web technologies (SOAP). It also provides a communication API based on I2O to exchange messages and/or data between applications. Tthe default *xdaqApplication*, from which all applications such as the *Fed9USupervisor* are derived, implements a state machine representational of the different stages of a physics run: *Configure*, *Enable*, *Halt*, *Pause* and *Resume*, which are event-driven (SOAP messages) and facilitate coherent configuration and operation of the various applications comprising the DAQ.

B. Tracker Implementation

Preliminary to any data acquisition, all hardware devices have to be mapped to a *xdaqAplication*. Nominally they are: *FecSupervisor*, which manages the FEC[11] and handles chip configuration of on detector devices and the trigger and clock distribution and timing; the *Fed9USupervisor* which handles digitization of detector analog signals. And the *LTCSupervisor* which manages the LTC handling trigger and clock formatting. Finally, the data from the FEDs must be collected, analyzed and stored, which is achieved in the event builder described below

C. The Event Builder

Together with the XDAQ environment, the CMS online software group provides a software prototype of the final event builder. On an Event Manager (EVM) request, the data sources - Readout Units (RUs) push their data fragment to the Builder Units (BUs). Filter Units (FUs) can then request lists of fragments to process. The whole process is asynchronous. Well-defined message interfaces are provided to feed the RU with FED data and to get merged events from the FUs. The trigger flow is controlled by feeding triggers to the EVM.

The interface to this builder is achieved through three applications:

1. The *TrackerSupervisor* controls the trigger flow, blocking Level 1 triggers at the *LTCSupervisor* level and sending software triggers to the EVM. It can then make dedicated acquisition loops, scanning chip settings (*FecSupervisor*) for batches of events during a commissioning run.

2. The *dataSender* collects data from different FEDs, formats them in RU readable form and send them to one RU.

3. The *RootAnalyzer* collects events from the Filter Units, writes them to ROOT files and performs commissioning analyses.

The *dataSender* application provides a buffer to *Fed9USupervisor* via a *dataBufferAccepter* interface (which the FED application must derive from,in order to have access to the buffer) in which the FED data is placed. Each buffer is then forwarded to the next application (FED or *dataSender*) in the collection ring. Since the acquisition is asynchronous, the *TrackerSupervisor* verifies the receipt of the buffer at the RUs before sending more software triggers to the EVM. The communication between the different tracker applications is achieved with the I2O *TrackerCommandMessage*. Sending and receipt of messages is handled by two additional classes the *TrackerCommandSender* and *TrackerCommandListener,* which the tracker applications inherit from.

In the final system the FEDs will be read out using S-Link, which is a fast link which continually pushes data out to a receiver unit. In this situation the readout is slightly different, no longer requiring the *Fed9USupervisor* to perform the readout, and the addition of an application to handle the receiver unit routing of data directly through the *dataSender*.

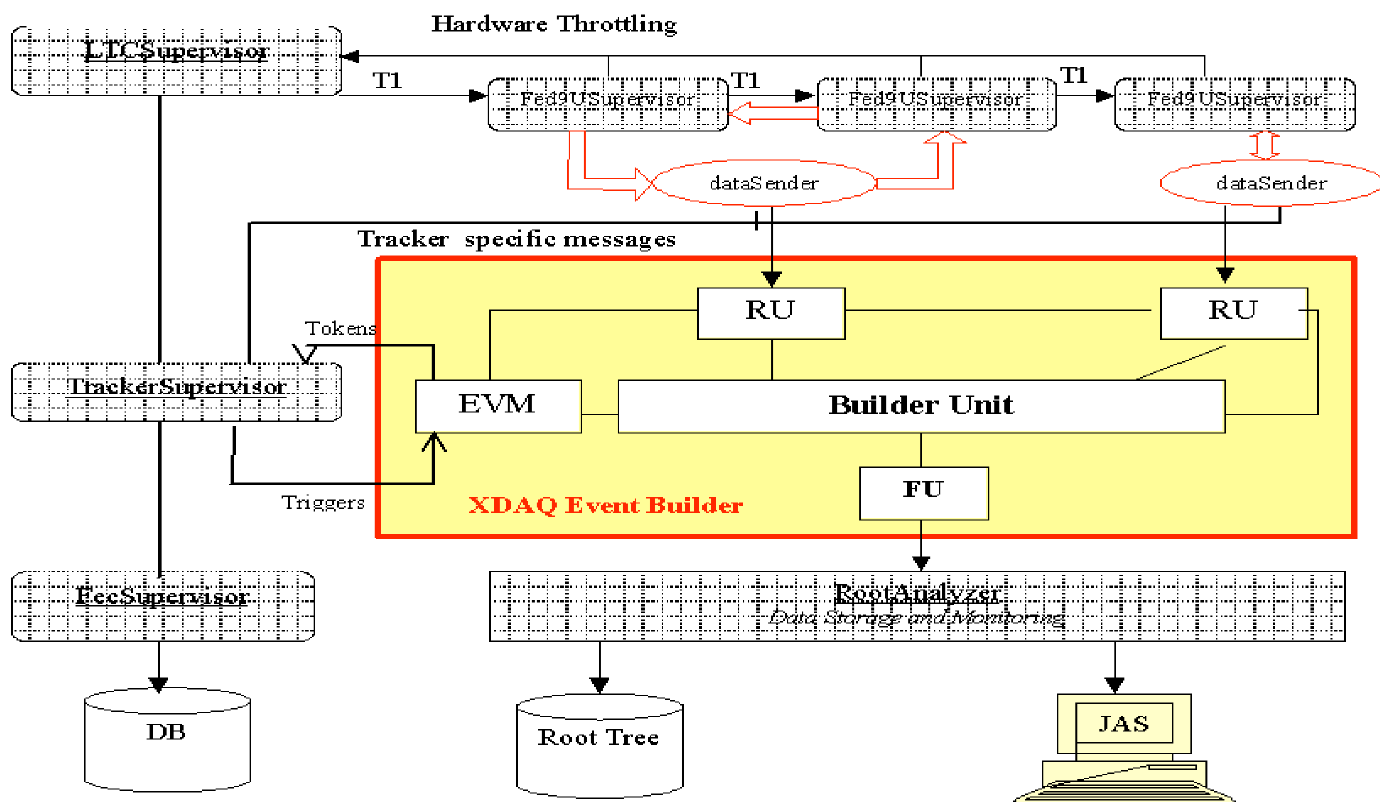

Figure 4: Schematic representation of the CME Tracker DAQ software

V. CONCLUSIONS

The software has been tested in test beams at CERN and has been proven to work using both VME and S-Link readout. Preliminary measurements in the laboratory show that over VME the software is capable of sustaining an event readout rate of up to 150 Hz corresponding to 5.7Mb/s and with S-Link over 200Mb/s.

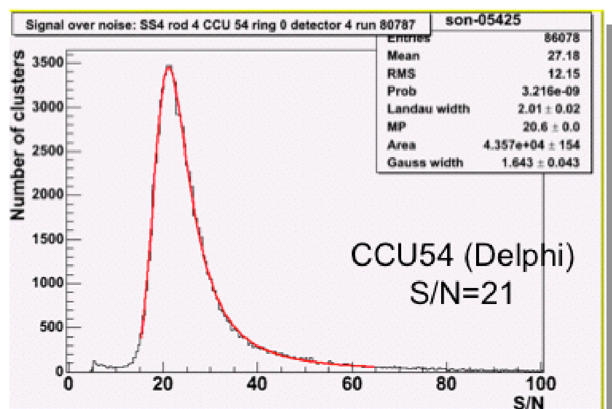

Figure 5: Example Landau distribution from data taken by DAQ in a test beam of July 2004 at CERN. APVs were set to deconvolution

Figure 5 shows an example of a Landau distribution of energy deposition in a number of Tracker modules during a test beam at CERN in July 2004. These data were taken using the system described in this paper with the front end APVs running in deconvolution mode. A signal to noise level of 21 can be measured. This is a small example of a large amount of data that has now been taken using this system. Confidence in the reliability of the software is high and the project is in a good state to move towards a final system for the commissioning of the Tracker.

VI. REFERENCES

[1] M.J.French et al. "Design and results from the APV25, a deep sb-micron fron-end chip for the CMS tracker", Ncl. Instr. And Meth. A466 (2001) 359-365

[2] J.Gutleber et al., "Architectural Software Support for Processing Clusters", IEEE International Conference on Cluster Computing (Cluster 2000), November 28 - December 2, 2000, Chemnitz, Germany, IEEE Conference Proceedings http://xdaq.web.cern.ch/xdaq/

[3] G. Iles et al. "*Performance of the CMS Silicon Tracker Front-End Driver*" These proceedings

[4] J. Coughlan et al. "*The Manufacture of the CMS Tracker Front-End Driver*" These Proceedings

[5] C++ Object oriented reference

[6] SBS620 PCI-VME Bridge, http://www.sbs.com/products/457

[7] C. Schwick. The Hardware Access Libraries. http://cmsdoc.cern.ch/~cschwick/software/documentation/HAL/index.html

[8] D. Box, et al., "Simple Object Access Protocol (SOAP) 1.1", W3C Note 08, May 2000;

[9] I20

[10] L.Mirabito et al., "Tracker data acquisition for beamtest and integration", CMS IN 2003/021

[11] P. Gras et al., *"Front-End Electronics configuration System for CMS"*. WEDT005 physics/0112049