FERMILAB-Conf-99/181

CDFVME – Software Framework for Testing VME Boards

C. Gay et al.

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

July 1999

Published Proceedings of the 11th *IEEE NPSS Real Time Conference*,
Santa Fe, New Mexico, June 14-18, 1999

CDFVME – Software framework for testing VME boards[1]

C. Gay[2], Y. Guo[3], S. Nahn[2], J. Patrick[3], S. Vejcik[3], M. Votava[3]

[2]Yale University, Physics Dept. PO Box 208120, New Haven, CT 06520-8120

[3]Fermilab, P.O. Box 500, Batavia, IL 60510

Abstract

New VME based boards are being produced for the Run II of the Collider Detector at Fermilab (CDF). These boards are being developed and tested at both Fermilab and offsite institutions. A software framework called CDFVME has been developed in which DAQ code can be easily written to control such boards in a test stand. The framework has been used to perform diagnostics at single board, multi-board, and multi-crate levels.

This software framework runs on Unix, Linux and Windows NT platforms with a Java GUI communicating via LAN to multiple intelligent front end VME crates. All distributed processes are managed by a custom CORBA based software. The system has been ported to Motorola 68K and PPC front end processors running the VxWorks real-time kernel [1].

I. INTRODUCTION

The nature of High Energy physics, the large bulk of data and high frequency of events, requires an enormous effort in developing hardware to record the data, involving many different collaborators spread around the world, each building their piece of the detector according to the design. For instance, roughly 60 different types of boards, numbering in total perhaps 1000, will be used in 150 VIPA crates in the final CDF Data Acquisition system. However, it is a rare piece of complex hardware which requires no software to see if it works; thus, software is typically developed along with the hardware for testing purposes.

More often than not, the result is many pieces of test code developed on incompatible platforms which only function when driven by the hardware developers who wrote them. These developers have invested much time in investigating software, rather than the actual hardware they are trying to bring up, which is unpalatable considering the level of importance placed on schedules in the High Energy physics world. Finally, there is no chance for integration with other bits of code for system level tests without a lot of rewriting and re-debugging.

The CDFVME framework was created with the goal of avoiding such chaos by making it easy to write and debug potentially complicated diagnostic code, using Java to eliminate platform dependence, and a common interface for easy integration of separately developed hardware. The use of one common framework, with aspects in common with the CDF final run control, consolidates expertise, such that software related problems can be solved by experts who know nothing about the hardware involved. The common GUI interface and common tools give users a familiar interface to all boards, whether they developed them or not.

Over the last year, the CDFVME framework has been used in developing board support packages for essentially all VME hardware development for CDF. Most prototype hardware has become finalized, and CDFVME code is being used to write checkout tests for production quantities of boards. It is also the basis of a “Toy Run Control” to provide a smooth transition from test stand software to full run control software.

II. CDFVME INFRASTRUCTURE

The hardware architecture is a collection of VME crates with custom built VME boards that collect data from the CDF detector. CDFVME matches with a distributed software architecture which contains a Java client running on Unix or Windows NT and servers on Motorola 68K and PPC front end processors (“servers”) in the crates running the VxWorks real-time kernel. The transport layer is using in-house CORBA-based distributed object package called ROBIN (Rpc and Object Broker)[2]. VME accesses are through in-house software package FISION (Fermilab’s VISION)[3].

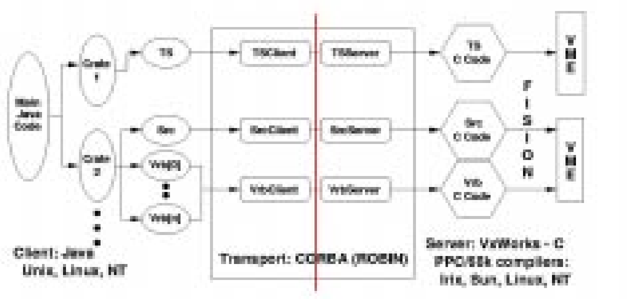


Figure 1: CDFVME Architecture. Java Objects have CORBA connections to the front end CPUs where the remote objects access the hardware.

The resulting software structure is shown in Figure 1. In general, the Java client instantiates objects (crate and board objects) dynamically according to configuration files specifying the node names of the front end servers and the board types in the particular slots. The transport layer dynamically loads server code and creates remote objects mirroring the client objects. Thus the client code is able to transparently manipulate the hardware through

[1] This work is sponsored by DOE contract NO. DE-AC02-76CH03000

its local references which are connected to VME backplane access code through the server implementation of the CORBA methods.

A. *Client Side Code*

The client side of CDFVME is written in pure Java, which has several advantages. Packages using CDFVME have been developed and can function on any combination of Linux, Irix, and Windows NT platforms. As an object oriented (OO) language, Java provides a natural mapping of several crates containing boards easily into an array of crate objects each containing arbitrary board objects. In addition, inheritance is very useful in providing general routines for every board, or writing tests that differ only in how data is collected or compared. Java also has a built-in GUI toolkit, making it easy to add GUI for board and test stand control. Finally, and perhaps most importantly, Java is a "safe" language for physicists without OO experience, with automatic memory freeing and reasonable error handling, providing a nice stepping stone to more modern programming techniques.

B. *Client-Server Transport*

The connections between the client CPU and the server CPUs are implemented using a software package called ROBIN. ROBIN is CORBA-2.1-based ORB and developed by Fermilab. ROBIN implements a dynamic server-side ORB which invokes static skeletons directly. The server and client skeletons are automatically generated with the ROBIN IDL compiler. The user simply adds server code in the marked places to implement the backplane access to the board register space and instantiates the generated client objects.

The IDL file implements the model protocol, which is a small number of routines which label specific atomic actions, such as "enable", executed on a target, which could be a bit, a register, a mode, etc. For example, to enable a mode called READOUT_MODE, the Java code would look like:

```
MyBoard.enable(MyBoard.READOUT_MODE);
```

This would get passed down to the server side code, which would then do whatever is necessary to set the hardware into that mode.

The implementation of the transport deserves some comment. On the client side, in order to hide the choice of transport and all the details of marshalling and unmarshalling arguments from the developer code, the ROBIN generated classes are extended rather than instantiated. Thus, replacing the transport would in principle be as simple as extending the board class from an alternate CORBA implementation. On the server side, ROBIN produces a stub that unwraps the arguments, which are then passed on to the server side C library functions (see below), of which there is one per IDL method. The separation of the stub from the library isolates the server code from the choice of transport, meaning less code rewriting should it be desirable to change the transport.

C. *Server Side Code*

The server side code consists of a C library of VME access functions, which contains both generic and specific routines for access to VME board registers. The separation of transport from server code also means more complicated functions can be built out of the atomic operations. The function is typically a switch statement on the target of the action. For the above example, the "enable" function called above may look like:

```
Switch (target) {
Case READOUT_MODE:
Enable(READOUT_BUFFERS);
Reset(OUTPUT_FIFO);
...
```

This example shows how the server code can call itself, forming a self-consistent driver layer for the boards.

The C library level also contains the VME register map and relevant bit masks, implemented as *#define* statements in include files. Should a register move or a bit field definition change during development, the change required to the software is limited to one place, not everywhere that quantity is used.

Finally, the one-to-one matching of the server side functions to IDL methods makes it easy to move code from the client side to the server side. The developer can prototype complex functions in Java, where there is more control and debugging is easier, and later move them to the server side with little or no editing, where they avoid the network and therefore run much faster. This sort of conversion can increase the speed of Bit Error Rate tests from ~10 KB/sec to ~10 MB/sec.

III. CDFVME SOFTWARE STRUCTURE

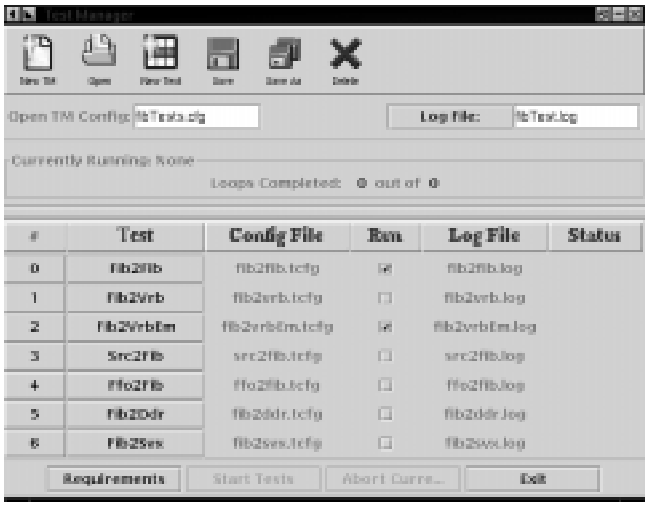
The CDFVME framework contains three packages; CDFVME_COMMON, which encompasses software elements common to all board developers, CDFVME_TEMPLATE, a working example and template for coding software for a particular type of board, and CDFVME_TESTSTAND, a working example of incorporating many board types into a test stand.

A. *CDFVME_COMMON*

CDFVME_COMMON encompasses software common to all CDF test stands. It embodies the software definitions of boards and crates, provides an easy way to put different combinations of boards together into multi-board and multi-crate systems, and provides a base from which all customized test stands extend. It also contains useful Java utilities, base classes for single board and multi-board tests, and a Test Manager to run sets of these tests in a batch-style configuration.

The first task of CDFVME_COMMON is the definition of common interfaces and classes extended in the other products. This package provides the base Board class, which every board extends, such that all boards can be grouped into Board arrays. The Board class also provides a standard set of fields such as slot number, server name, etc. which can then be used as part of a common look and feel for displaying the board status. The package also provides the "global objects" which any board package can use independent of what test stand it is in. This includes the base configuration object containing all crates and boards, with various methods to access them, the main window/status reporter object, and a crate configuration editor. Users can extend these objects and customize in the test stands without losing the ability to incorporate any board.

Keeping in mind that all the production boards need extensive testing, perhaps one of the more useful facilities provided in the package is the Test interface and Test Manager class. A Test Manager is a GUI interface to incorporate and edit an arbitrary list of tests, select the hardware to be tested and configuration or each test, and run them in batch. The tests must implement the Test interface in the package, which defines how the Test Manager finds out what hardware a test requires and how to select it. There are several levels of implementation of the Test interface that the code developer can choose from depending on how much of the default implementation they wish to use. The entire structure is aimed at providing a flexible and easy to use interface for developing and running tests for production checkout. Figure 2 is an example of a Test Manager.



Open TM Config: fibTests.cfg

Log File: fibTest.log

Currently Running: None

Loops Completed: 0 out of 0

#	Test	Config File	Run	Log File	Status
0	Fib2Fib	fib2fib.tcfg	✓	fib2fib.log	
1	Fib2Vrb	fib2vrb.tcfg	☐	fib2vrb.log	
2	Fib2VrbEm	fib2vrbEm.tcfg	✓	fib2vrbEm.log	
3	Src2Fib	src2fib.tcfg	☐	src2fib.log	
4	Ffo2Fib	ffo2fib.tcfg	☐	ffo2fib.log	
5	Fib2Ddr	fib2ddr.tcfg	☐	fib2ddr.log	
6	Fib2Svx	fib2svx.tcfg	☐	fib2svx.log	

Figure 2. A series of Fib tests in a Test Manager.

The package also contains useful widgets and other Java objects. One major example of this is the Crate class, which contains code to read and write any register given the base address and the accessing mode. This allows users to start probing their hardware before writing any code, and similar tools exist for block data I/O, scripting of single word accesses, etc. Along the same lines are useful widget extensions, such as text fields that only allow input of numerical characters, or read-only checkboxes to display status. As a final example, there are Utility classes that provide easy wrappers for procedures that are somewhat awkward in Java, such as opening a file. These and other constructs in CDFVME_COMMON make it easier for the Java novice to develop useful interfaces and tests for their hardware.

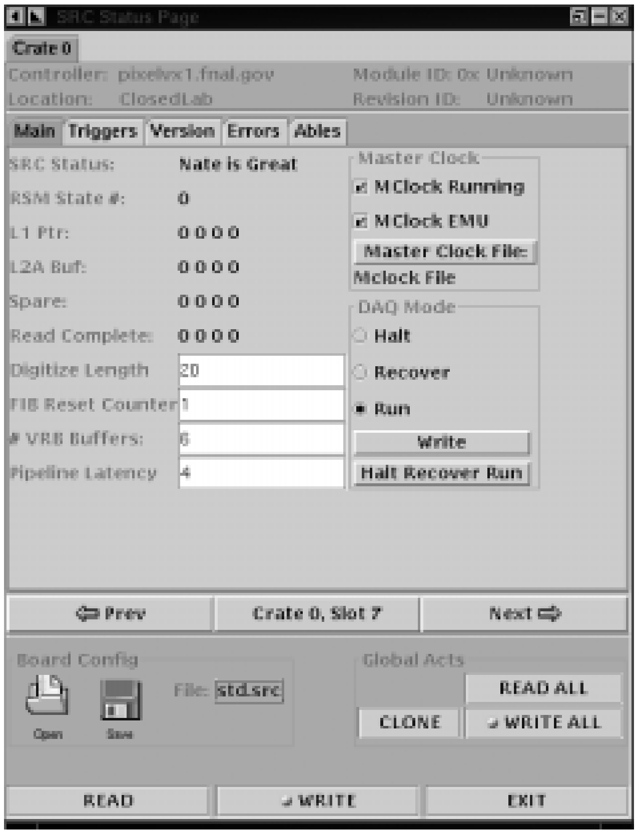
B. CDFVME_TEMPLATE

CDFVME_TEMPLATE is a template of what could be called a *Board Support Package* i.e. the board-level code that provides all control, monitoring and test functions for each VME module type. It contains the base structure for implementing board-specific code on both the server and client side, as well as the IDL file which generates the CORBA links between these. "Out of the box", it is coded with a complete working example using a fictitious board, with print statements for the user to trace through the code while trying it out. It comes with a script to convert the demonstration code to the start of a real board package.

Implementation of a board requires two actions. The first is the creation of server routines and IDL targets and functions to access the particular board register space. This is straightforward but tedious work of defining bits and addresses and making VISION calls to the board.

The second stage of development is implementation of pure Java functions which manipulate the "state of the board", that is, the values of all registers, bits, etc that define the board state. Several routines are required to move this board state between static storage, Java variables, display GUI, and the actual registers on the hardware. Furthermore, the user must implement some GUI to display the board's state. Mandating all this allows general code and GUI to manipulate and display any board state, load it from or save it to a file, read or write it to the board etc. Figure 3 is an example of a board display.

Other than the board implementations, the board support package is the place to put other specific board code, such as tests and calibration classes that use the board. The package then gets propagated as a product that other people can use simply by pointing their classpath at the location of the code.


SRC Status Page

Crate 0

Controller: pixelvx1.fnal.gov
Module ID: 0x Unknown
Location: ClosedLab
Revision ID: Unknown

SRC Status: Nate is Great
RSM State #: 0
L1 Ptr: 0 0 0 0
L2A Buf: 0 0 0 0
Spare: 0 0 0 0
Read Complete: 0 0 0 0
Digitize Length 20
FIB Reset Counter 1
VRB Buffers: 6
Pipeline Latency 4

Master Clock
✓ MClock Running
✓ MClock EMU
Mclock File

DAQ Mode
○ Halt
○ Recover
● Run

Crate 0, Slot 7

Board Config
File: std.src

Figure 3. Board Display. The board specific details are encapsulated within a standard frame, which includes navigation between boards and transferal of the state between the GUI, hardware, and static storage.

C. *CDFVME_TESTSTAND*

CDFVME_TESTSTAND is simply a starting point for creating a test stand that integrates a collection of arbitrary boards into a DAQ system, corresponding to what hardware is in the crates. It consists of some trivial extensions of CDFVME_COMMON classes and a Board Types interface, which contains information about the boards that the test stand will use. Incorporating other boards into a test stand is as simple as adding them to this interface, and including them in the configuration files describing the crates. At the same time, the simple implementation can be added to, providing multi-board code, adding extra Test Managers and tests for many boards, etc. In this way the CDFVME framework makes it very easy to integrate independently developed boards and build test stands.

Perhaps the best example of a modified test stand is the Toy Run Control. This is just a simple test stand with an extra interface for sending “Run Control” commands to the front end CPUs. The Toy Run Control will import many boards and use their board support packages when responding to run control commands, as a first step towards integration. It also can support development of database access to replace the ASCII configuration files, development of Error Messaging and Monitoring systems, etc. The goal of the Toy Run Control is to provide a platform to develop pieces of the Full Run Control while utilizing well known and well debugged test stand code from the CDFVME framework.

IV. SUMMARY

The scale and complexity of hardware development, in high energy physics, involving many people working independently on pieces of the system, requires a flexible software interface that is not overly complex, can run on multiple platforms, and makes it easy to combine pieces as the DAQ system comes together. The CDFVME framework satisfies these requirements, giving a coherent platform for developers and users to create and combine software to test the hardware from the first probing of registers until final integration at the detector.

REFERENCES

[1] VxWorks is Wind River’s real time operating system. More information is at http://www.wrs.com

[2] ROBIN information can be found at http://www-b0.fnal.gov:8000/ROBIN.html

[3] The FISION implementation of VISION: http://www-b0.fnal.gov:8000/vme/VISION.html