The Front-End Readout Control Software for the CODA Data Acquisition Toolkit

D.J. Abbott, P. Banta[a], J. Chen, W.G. Heyes, E. Jastrzembski, C. Timmer

Thomas Jefferson National Accelerator Facility (TJNAF)
12000 Jefferson Ave. Newport News, VA 23606 USA
[a]ADC Telecommunications Inc.
2240 Cambell Creek Blvd. Richardson, TX 75082 USA

The Readout Control (ROC) software for the TJNAF Data Acquisition System CODA is designed to run on a real-time system (e.g. VxWorks[1]) where it can be interrupt driven and have direct access to Front-End hardware. The ROC component is built on an interpreter for the Tcl scripting language with several extensions to allow for object-oriented scripting. Embedded C routines are used for many of the ROC's real-time critical functions. User specific compiled objects can be dynamically downloaded at runtime as part of its configuration. These Lists provide the interface to various hardware platforms (FASTBUS, CAMAC etc.) supported by the ROC software. We present details of the software architecture of the ROC and its real-time performance in several of the supported hardware platforms.

Key words: Data Acquisition Systems; Object-Oriented Methods; Software Tools;

1 Introduction

The CODA[2] Data Acquisition System in use at TJNAF (formerly CEBAF) consists of a set of software objects which communicate via a network-based, distributed, cross platform environment. With this flexibility, one is able to build an extensible data acquisition system capable of aggregate data rates in excess of tens of MBytes/sec, the limitations being largely due to network bandwidth and CPU speed. At the front-end of this system is the CODA Readout Controller (ROC).

At its core, the ROC is built upon an interpreter for the Tcl scripting language (v7.4)[3]. The interpreter has been extended by several packages including the *dp* (Distributed programming) package for client/server and socket support and the *itcl* package which allows for a more object-oriented approach to Tcl scripting. In our implementation the Tcl core handles most of the time non-critical functions such as file and database access, ROC configuration, and inter-process “control” communication, while embedded C routines have been written to handle the high speed data transport and buffer management.

By itself the ROC software can be used as a small “stand-alone” data acquisition system. It’s features include the ability to dynamically download at runtime two user specific compiled “Lists”. The Primary List contains the routines necessary to access and configure the front-end hardware as well as interrupt service routines that can be attached to multiple external trigger sources. The optional Secondary List contains routines that asynchronously take events generated from the Primary List and perform a user specific analysis (e.g. compression, sparsification, etc.). The events are finally passed to a third stage of the ROC where they are buffered and output to a user-specified option such as a disk file (local or nfs) or a network data link (TCP stream).

At TJNAF, Version 2.0 of the CODA Data Acquisition Toolkit is being tested for the first time in Experimental Hall B. This implementation will require over 20 front-end crates (FASTBUS and VME) running the ROC component. They are expected to require event rates up to several kilohertz and/or aggregate data rates over 10 MB/sec. Currently supported platforms include both the MVME 68040 CPUs from Motorola[4] and the PPC604 100MHz CPU from Radstone[5]. In this paper we present details on the software architecture of the ROC followed by results from tests on the real-time performance of the supported platforms including data transport using various network protocols (e.g. ATM, 100 BaseT).

2 Architecture

In Figure 1 the various software layers in the ROC code are illustrated. The low lying APIs in the Tcl layer are the same for all CODA components. The itcl class CODA defined in CODA_class.tcl may be inherited by any program which wishes to take on the characteristics of a CODA component. The file ROC_class.tcl contain methods which override some of the defaults defined in CODA_class.tcl. To increase performance many of the methods defined in both ROC_class and CODA_class are implemented directly in C via coda_component.c and roc_component.c.

The ROC must also be a RPC server to communicate with Run Control

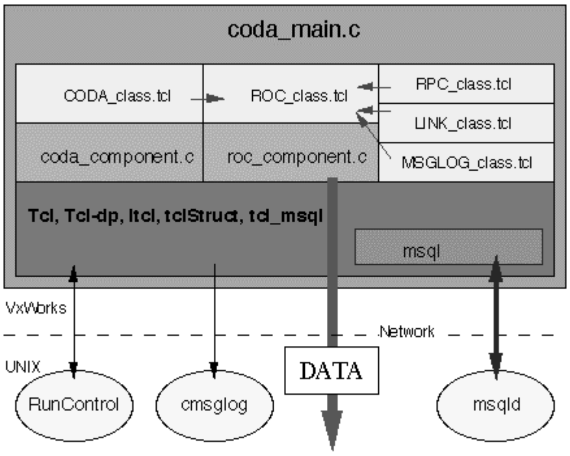

Fig. 1. CODA ROC Software Architecture

and other User processes as well as a client for the message logger. This is implemented by allowing the ROC to inherit three additional Classes defining methods for remote communications (via the Tcl dp package). These are RPC_class.tcl, LINK_class.tcl, and MSGLOG_class.tcl. Finally, all CODA components load configuration information via a central database *mSQL*[6] through a daemon running on the database host (*msqld*). Hence, the ROC includes the TCL API tcl_msql to provide this functionality.

At bootup the top layer (coda_main.c) simply calls initialization functions in the coda_component and Tcl core layers to setup interpreters and to create an object of class ROC. After booting the ROC is capable of accepting commands either locally (via an interactive Tcl shell) or remotely (typically from the Run Control server). It follows a standard state machine where first the ROC Downloads a Run configuration followed by a Prestart, Go, then Pause or End transition. When the ROC Downloads a configuration, up to two compiled objects are loaded in to local memory. These are the user's readout lists (ROL). They contain, at a minimum, the user specific code to be executed at each of the Run Control transitions. In addition, they contain trigger routines to be executed whenever a defined "Service" condition exists. These routines may either be interrupt driven or polled.

Figure 2 shows a diagram of the event readout and buffer management scheme. Each readout list allocates a pool of buffers in which to hold events (1 buf/evnt). Also created are input and output queues. These queues are implemented as linked lists of descriptors to the pool buffers. When the trigger routine of the Primary ROL gets executed (usually as an ISR generated by an external source) a buffer from its pool is acquired and data from the front-end hardware is stored. The buffer descriptor is then placed on the output queue. The Secondary ROL meanwhile polls its input queue for "events" and executes its trigger routine when it is not empty. Here the user has several options. He can copy the input event into a local pool buffer, modify it, and pass the new

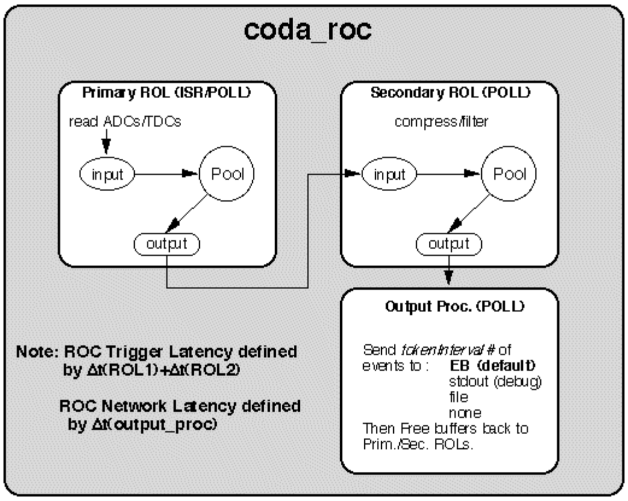

Fig. 2. CODA ROC Buffer Management

buffer onto the Secondary ROL output queue while freeing the old buffer, or he can use the existing buffer from the Primary ROL and pass it directly to the output queue.

In all cases event descriptors end up on a third stage queue where the ROC can output them to a user defined destination. Currently the options include: to a binary file, to the console, or over a network data link (socket) to a CODA Event Builder. After the data have been sent the buffers are freed back into the ROL pool from which they originated.

3 Performance

There are two primary performance issues related to the ROC that we have investigated. The first is the question of trigger latency, or how long it takes to execute the rol trigger routines. In Table 1 below are listed execution times for “minimal” trigger routines (i.e. 48 Byte events) for the two primary CPU architectures used at the Lab (Motorola MVME162 (25MHz) and Radstone PPC604 (100MHz)). The times shown are the Interrupt Latency, Primary, and Secondary ROL execution times.

Table 1
CODA ROC Trigger Latency

CPU Type	ISR Latency	ROL1 Execution	ROL2 Execution
MVME162	12μsec	65μsec	18μsec
PPC604	20μsec	40μsec	7μsec

For purposes of dead time calculations only the ISR latency plus the ROL1 execution time need to be considered. Execution of the Secondary ROL trigger is asynchronous to the Primary ISR. This is useful as it allows secondary event processing to go on between Level 1 event triggers. However, if the maximum input L1 trigger rate exceeds the total trigger latency defined by the sum of all three times, then buffers will fill up in a 100% dead condition.

The second issue studied was the network performance. Most of the tests were performed with the PPC604 CPU as it had a number of network adapter options through on-board PMC slots. In all cases using the built-in 10BaseT Ethernet both the MVME162 and PPC604 were able to saturate the network around 1 MByte/sec. For the PPC604, however, we tested both an ATM interface (with our own VxWorks driver) as well as a 100 BaseT interface (with a driver provided by Radstone). Table 2 below shows the sustained data transfer rates for the PPC604 using both the ROC, sending data to an CODA Event Builder, and a dedicated client process (Blaster) that measures the maximum data throughput to a UNIX host. In the ROC test a 2 KByte event size was used while for the Blaster test the event size was 8 KBytes.

Table 2
Network Performance (PPC604)

Network Task	10 BaseT	100 BaseT	ATM
coda_roc	0.9 MByte/sec	4 MByte/sec	5 MByte/sec
Blaster	>1 MByte/sec	7 MByte/sec	10 MByte/sec

The Blaster client was also at an advantage in these tests as it had implemented TCP Zero Copy buffers (zbufs) on the VxWorks side. The ROC does not use this option yet. The zbuf option can generate a 20-30% performance improvement to network tasks on the PPC604 depending on the application.

4 Conclusion

The new CODA Readout controller has proven to be a high performance and flexible real-time tool for accessing the front-end hardware and distributing data. Our performance tests with the PPC604 have shown that the ROC is more than capable of handling the experimental requirements at Jefferson Lab. Currently, a working system with eight ROCs has been implemented at the Lab. This includes seven PPC604 CPUs each running in a Struck SFI/340[7] FASTBUS interface reading out up to 10,000 detector channels. An expansion to over twenty ROCs is scheduled for the coming months.

References

[1] VxWorks and Tornado are registered trademarks of Wind River Systems, USA.

[2] G. Heyes et.al. The CEBAF On-line Data Acquisition System *Proceedings of the CHEP Conference* (April 1994) 122–126.

[3] John K. Ousterhout *Tcl and the Tk Toolkit* (Addison-Wesley, Massachusetts, 1994)

[4] Manufactured by Motorola Corporation, USA.

[5] Manufactured by Radstone Technology PLC, England.

[6] mSQL was developed as part of the Minerva Network Management Environment. Copyright (c) 1993-1995 David J. Hughes.

[7] Manufactured by Dr. B. Struck, Tangstedt, Germany.