CODA

CEBAF On-line Data Acquisition

User's Manual

Continuous Electron Beam Accelerator Facility

Newport News, Virginia, USA

(This page is intended as a binder edge insert. Cut the right 1” from this page and insert into the binder pocket).

C
O
D
A

CODA

CEBAF On-line Data Acquisition

User's Manual

Version 1.4

March 17, 1995

co-da \ 'kōd-ə\ *n* [It. lit., tail, fr. L *cauda*] : a concluding musical section that is formally distinct from the main structure

Document Date: March 17, 1995

The Southeastern Universities Research Association (SURA) operates the Continuous Electron Beam Accelerator Facility (CEBAF) for the United States Department of Energy under contract DE-AC05-84ER40150.

Release Notes for 1.4

1 General

CODA 1.4 contains significant performance enhancements and new features relative to the 1.2 release, as well as a number of changes designed to allow the experienced user the ability to more easily customize service communication and data flow. For a complete list of changes, see “New Features” below.

CODA 1.4 no longer supports the use of DataViews for the SlowControls display or graphics display in RunControl. Hence, CODA 1.4 may be used without either a DataViews development or run-time license.

A VxWorks target license is required (and can be purchased directly from CEBAF) for each single board computer (FASTBUS, VME or CAMAC) used as a readout controller; a development license is not required.

2 New Features since Version 1.2

- Support for HP-UX operating system (as of 1.3b).
- RunControl enhancements including: remote status display, graphical display history of event/data rates, switchable options for event logging, automatic halt of run after user specified event/data limit, automatic restart of a run after an end or service crash, and execution of multiple user written shell scripts during any of the transition events of RunControl (i.e. prestart, go, end, etc...).
- Improved network dataflow from ROC to EB using TCP streams over both ethernet as well as FDDI.
- Alternate data path for the FSCC readout via parallel link into VME memory.
- Support for the CEBAF trigger supervisor including both software and hardware modules.
- Support of “asynchronous” ROCs sending events to the event builder.
- ROC and Event builder support for syncronization events (issued by trigger supervisor).
- ROC event size limits have been increased. By default a single event can be 16KBytes (4096 FB data words). The user can adjust the ROC buffer’s high water mark to allow events as large as 32Kbytes.
- Support for multiple data spies on a single event stream.
- CAMAC readout supported (Library and ROC) for the CES VCC2117 Intelligent Crate controller with ethernet interface.
- Expanded Readout Control Language including a new crl interpreter *ccrl* for converting crl --> c code. There have been a number of FASTBUS readout options added to improve the execution speed of the trigger lists.

3 Problems Fixed

- There are no problems only “features” which are constantly being “enhanced” to better enable the user to improve his/her data acquisition system.

CHAPTER 1 Introduction to CODA 1-1
1.1 CODA 1.4 Capabilities 1-1
1.2 Future Evolution 1-2
Hardware Upgrades 1-2
Software Upgrades........ 1-2
Platform Independence 1-2
1.3 Reference Materials 1-3
1.4 How to Report Problems........ 1-3
1.5 Joining the CODA Mailing List........ 1-3

CHAPTER 2 CODA Architectural Overview 2-1
2.1 Data Flow........ 2-1
The Trigger System 2-1
Front End Crates 2-2
The Event Builder........ 2-3
On-line Analysis 2-3
Event Recording........ 2-3
Histogram Display 2-3
Diagnostic Event Dump........ 2-4
2.2 Run Control Architecture........ 2-4
State Machine Model of Run Control 2-4
User Definable Run Control Components 2-4
Run Control Configuration Concepts........ 2-5
Run Control User Interface........ 2-6
2.3 Data Analysis & Monitoring........ 2-6
Event I/O 2-6
Histogram Display 2-6
Formatted Event Dump 2-7

CHAPTER 3 Using CODA 3-1
3.1 Run Control........ 3-1
The Network Definition File 3-2
The Run Type Dictionary........ 3-3
The Run Type Configuration File........ 3-3
The Run Type Options File........ 3-4
Configuration Files Summary 3-4
Starting & Stopping a Run........ 3-5
The Options Menu in RunControl........ 3-6
RCDEFAULTS Environment Variable........ 3-7
3.2 The Trigger Supervisor 3-7
Configuring the Trigger Supervisor........ 3-8
Enables and Prescales........ 3-8
Memory Lookup Units 3-9
Readout Controller Interface 3-9
Timers........ 3-10
Synchronization Interval 3-10
3.3 Using the Read Out Controllers........3-11
Software Configuration........ 3-11
Using FASTBUS (the FSCC)........ 3-12
Using CAMAC 3-12
Using VME 3-13
3.4 The Event Builder 3-13
3.5 Running an Analysis Program 3-14

3.6 Using the Event Recorder ... 3-14
3.7 The Event Dump Utility ... 3-15

CHAPTER 4 Callable Routines ... 4-1
CAMAC I/O Library ... 4-2
FASTBUS I/O Library ... 4-6
Console Logger ... 4-8
Error Message Library ... 4-9
Event I/O Library ... 4-11
Histogramming ... 4-14
Run Control Communications ... 4-17
Spying: Data Acquisition RPC Library ... 4-20

CHAPTER 5 Utilities ... 5-1
ccrl ... 5-2
cdumphist ... 5-3
cefdmp/xcefdmp ... 5-4
cemsg ... 5-6
cnaf ... 5-7
coda_activate ... 5-8
codaf77 ... 5-9
makelist ... 5-10
vxmon ... 5-11

APPENDIX A Run Control Configuration File Formats ... A-1
rcNetwork ... A-1
rcRunTypes ... A-2
<runType>.config ... A-2
<runType>.options ... A-3
rcRunNumber ... A-4
rcDefaults ... A-4
rcExperiment ... A-4
rcPriority ... A-4

APPENDIX B CODA 1.4 Support for the Trigger Supervisor ... B-1

APPENDIX C Readout Controller Configuration File ... C-1
File Format ... C-1
Compiler Flags ... C-1

Code Sections C-2
Language Elements C-3
Example File C-7

APPENDIX D The CODA/EPICS Interface D-1
EPICS Requirements D-1
CODA Requirements D-1
CEIMON Requirements D-1
CEIMON Configuration files D-2

APPENDIX E CEBAF Common Event Format E-1
Event Format E-2
Physical Record Format E-5
Name Dictionary E-5

APPENDIX F CODA Event Bank Definitions F-1
Standard Physics Event F-1
Event ID Bank F-2
Readout Controller Data Banks F-2
Run Control and Sync Events F-2
Sync Event F-3
PreStart Event F-4
Go Event F-4
Pause Event F-5
End Event F-5

CHAPTER 1

1

Introduction to CODA

CODA (CEBAF Online Data Acquisition) is the data acquisition system for physics experiments running at CEBAF. This manual describes version 1.4 of CODA, which is the initial working version for CEBAF Hall C experiments. This release of CODA is appropriate for both detector prototyping/testing systems as well as moderately complex experimental systems requiring I/O throughputs of greater than 1Mbyte/sec (FDDI). It is intended as a “bridge” release to CODA 2.0 implementing a number of features necessary for running CEBAF online experiments but still considered as a system under development.

1.1 CODA 1.4 Capabilities

CODA 1.4 is designed to run on both an HP_UX (Hewlett-Packard/RISC) as well as an Ultrix (DEC/RISC) host, connected via ethernet or FDDI to multiple intelligent front end crates, FASTBUS and VME. CAMAC crates may be interfaced through VME or stand alone (using the CES VCC2117 controller). If events are acquired over ethernet, data rates are limited to roughly 250-700 kbytes/sec depending upon the single board computer used. Using FDDI (through a VME interface) practical data rate limits are roughly 2.5-3.0MBytes/sec. Event rates are limited by the front end latency and the size of the event, with rates as high as 2-3 kHz measured for reading a single module in FASTBUS over ethernet, and 16 kHz for reading a single register in VME.

The main run control portion of the software runs as a single process on a Unix workstation (single user operation), and can communicate with one or more FSCC’s (FASTBUS Smart Crate Controller) and/or one or more VME/CAMAC single board computers, up to 32 total. These front end computers, referred to as Read Out Controllers (ROC), run a multi-tasking real-time kernel called VxWorks.

Trigger information may be fed directly into each ROC or into the Trigger Supervisor (TS), a high speed custom trigger interface designed at CEBAF. The Trigger Supervisor supports up to 12 inputs with pre-scaling by up to 2^{24}. As many as 3 levels of experiment specific trigger logic may be connected to the TS for hardware event selection.

On-line analysis takes place as a single process running on a Unix workstation (not necessarily the same machine as is running Run Control). Event fragments from all ROC’s are automatically merged into a single event by an Event Builder (EB) and presented to the user’s analysis program. Events which pass the user’s selection criteria may then be written to a disk file or directly to tape.

Direct control/monitoring of experimental apparatus is no longer supported with CODA 1.4. The old Slow Controls package is currently being replaced with an interprocess communications interface with EPICS (Experimental Physics and Industrial Control

System). The EPICS system is now the supported slow controls interface at CEBAF for both the accelerator as well as the experimental halls. A number of EPICS control systems (including HV control/monitoring) are in development by the Data Acquisition Group. See Appendix D for more information.

1.2 Future Evolution

The next planned release of CODA for 1995 is Version 2.0. A number of enhancements will be implemented in order to support larger and more complex experimental systems. Some of these as well as future improvements are described below.

1.2.1 Hardware Upgrades

To remove the ethernet bottleneck, FDDI and other data links (i.e. ATM) will be supported to move data from VME crates to the event builder. (FDDI is now in operation at CEBAF). This will take place in two steps: first, links from FASTBUS into a VME crate will allow for building events under the control of a single board computer at VME backplane speeds (~10-15 Mbytes/sec), and forwarded to the event builder over FDDI or other available link. Later, a parallel event builder will be added to expand the bandwidth to over 100 Mbytes/sec. This may require custom hardware, but the preference is for future networking technology to meet this need.

Support for multiple workstations and processors in a processor farm will be added within a year. This will allow on-line analysis capability to be expanded to hundreds or thousands of MIPS.

CEBAF has begun testing CAMAC crate controllers with built in processors and ethernet interfaces. This will permit high performance, small data acquisition systems to be constructed from a workstation plus a single CAMAC crate with only an ethernet connection between them.

1.2.2 Software Upgrades

A version of the run control interface is now being tested which allows for multiple operators. That version (2.0) will also support running multiple copies of CODA (running different experiments) to co-exist on a single Unix machine.

Following that version, security and locking features will be added. In addition, graphical configuration editors will be added to aid in setting up CODA. Error logging and handling capabilities will be greatly enhanced by a new distributed message handler, which will include logging and report generation capability.

Data logging will be expanded to include support for labelled tapes (DAT and 8mm at first).

1.2.3 Platform Independence

Ports to additional workstations (other than ULTRIX and HP-UX) may be done by collaborating labs, with required changes merged into the sources at CEBAF (conditional compilation). Attempts will be made to keep CODA compliant with programming standards such as POSIX, ANSI C, etc., so that it should not be too difficult to port to additional hosts.

Support for additional FASTBUS, VME, and CAMAC interfaces will be added when there is a compelling need. Support for a real-time kernel other than VxWorks is not currently being considered.

1.3 Reference Materials

Other documents and manuals which will be useful in understanding and using CODA include:

1. CODA Application Notes
2. WWW (Mosaic) CODA Home Page
3. The Trigger Supervisor User's Guide
4. PAW Reference Manual (and associated CERNLIB manuals)

1.4 How to Report Problems

Please mail any bug or problem reports to coda@cebaf.gov, including a description of the problem and including any relevant configuration files which may be needed to reproduce the problem. Also include a description of the hardware used. Suggestions for improvement are also welcome.

1.5 Joining the CODA Mailing List

CEBAF is now running a mail server which supports user subscribable mailing lists. To improve communications with CODA users and interested parties, a new mailing list has been created. To join the list, send a message to mailserv@cebaf.gov containing the line:

```
SUBSCRIBE CODA-L
```

To remove yourself from the mailing list send the line

```
UNSUBSCRIBE CODA-L
```

To get help on other capabilities of the mail server, send the line

```
HELP
```

Mail sent to the address coda-l@cebaf.gov will be forwarded to all subscribed users. This will be one important mechanism that the CODA developers will use to post problem fixes and announcements of new versions.

In addition to the mailing list, the CEBAF Data Acquisition Group maintains a WWW site accessible via the CEBAF home page (http://www.cebaf.gov/). General information about CODA, current developments and releases etc. can be obtained from this site.

CHAPTER 2

CODA Architectural Overview

2

This chapter gives an overview of the system architecture for CODA. Operating instructions for each component of the system will be given in the next chapter.

The system can be broken down as follows:

1. trigger system
2. front end crates (digitizers, etc.)
3. event builder
4. on-line processing
5. event logging
6. host computer and X-windows displays

This chapter will discuss first the data flow from trigger to tape. Next an overview of the run control architecture will be given (how a user controls data acquisition), and finally the set of offline data analysis/monitoring tools will be described.

2.1 Data Flow

Valid triggers cause the digitizers(ADC/TDC) to convert and be read out in the front end crates. Data from each crate is then transmitted to the event builder and then to an on-line analysis program. Events which pass a software filter (if any) may be written to an event file. The following sections describe this process in more detail.

2.1.1 The Trigger System

The trigger system contains the logic capable of making very fast decisions about when to acquire a physics or calibration event. These decisions are generally made using a small subset of the detector signals. Each experiment will, of necessity, design trigger electronics customized to that experiment. In order to simplify system integration, a common trigger interface, called the Trigger Supervisor, has been designed. For small systems or test setups, the Trigger Supervisor may be omitted.

The Trigger Supervisor

Frequently, trigger systems are built with 2 or 3 levels of event selection logic. The first level makes a very simple and fast decision based on hits in scintillators or other fast detectors. To keep deadtime low, this logic must operate at the highest rate (physics + background) expected by the experiment. When an event passes this level 1 trigger, more complex computations are performed by the level 2 logic. The level 2 trigger need only be capable of handling (with low deadtime) the output of the level 1 logic.

The CEBAF Trigger Supervisor (TS) is capable of interfacing with up to 3 levels of experiment specific trigger logic, with 12 independent level 1 inputs. Eight of the 12 inputs can be prescaled (4 by up to 2^{24}), and each can be individually enabled or disabled. When any input is accepted, all 12 trigger inputs are latched to form a 12 bit address. Using this address, three items are retrieved from local fast memory: the event class type (how many levels of triggering for that event), a level 1 mask (How many of 8 total outputs fire, useful for conditionally generating gates), and a readout list number (physics event type, forwarded to the readout controllers).

A sequencer is then started which handshakes with any higher level trigger logic. Once a trigger passes the highest level trigger and the digitizers have completed their work, the event type (readout list number) is forwarded to all front end crates instructing them to read out that event. Simultaneously, if the multi-event buffers (described below) are not full, the TS enables level 1 triggers again. (See the Trigger Supervisor User's Guide for a detailed description of the capabilities and use of the Trigger Supervisor.)

Custom Triggers

Alternatively to the Trigger Supervisor experimenters may produce a completely custom trigger system which interfaces to each ROC (readout controller) using a subset of the TS-ROC cable protocol. A hardware trigger interface card has been developed for both the FSCC and VME ROCs to facilitate this implementation. This subset allows up to 4 trigger bits to be latched upon receipt of a strobe signal. Then each ROC signals its availability to receive the next trigger via an acknowledge (output) signal.

See the application notes for further details, including example NIM hardware for implementing the necessary external logic for a single FASTBUS ROC.

This mode of operation is expected to be used primarily in small systems not requiring the full capabilities of the Trigger Supervisor.

2.1.2 Front End Crates

Signals from the detectors are processed by boards in FASTBUS, VME, VXI, or CAMAC crates. These boards are then read out by a processor referred to as a "read out controller" (ROC). Generally, a single ROC handles only one crate of electronics (for performance reasons).

The ROC serves 4 main functions:

1. communicate with the Trigger Supervisor
2. read event data
3. send event fragments to the event builder
4. perform setup and initialization functions on its crate of electronics at the request of the host computer.

The ROC for FASTBUS is the FSCC (FASTBUS Smart Crate Controller), a Fermilab designed board manufactured by BiRa. It is a sequencer based single board design with an on-board 68020 and an ethernet interface. The VxWorks operating system has been ported to the FSCC at CEBAF, and it has passed performance tests. Established measurements confirm a 20Mbyte/sec handshaked data transfer rate and up to 40 Mbyte/sec read rate for high performance slaves implementing pipeline transfers. A slot addressing overhead of <10 μsec under processor control has been measured with currently supported FB library routines (sequencer controlled AS/AK lock time has not been measured). In addition to the ethernet interface there is a sequencer controlled parallel output port capable of transferring data (10-20 meters) at 40Mbytes/sec into a VME based dual ported memory.

For VME and VXI, single board computers running VxWorks (68K based) are required. These boards are available from a large number of manufacturers with VxWorks support. CEBAF is currently using Motorola’s MVME167and MVME162 68040 boards. For a VME FDDI interface CEBAF has implemented the Rockwell CMC1150 series with a VxWorks Driver support package from Ross Microsystems.

For users requiring CAMAC front end crates, the Kinetic Systems VME interface (model KS 2917) is supported (requires KS 3922 crate controller). Direct CAMAC readout through the CES VCC2117 Smart Crate Controller (with ethernet interface) is also supported. Since all CAMAC I/O is done through the CAMAC standard routines, any other interface for which these routines are available will also work (e.g. the CES branch highway interface).

2.1.3 The Event Builder

Each ROC operates independently, reading and buffering its event fragments and then sending it via ethernet/FDDI to the event builder process on a Unix workstation. This process can handle multiple connections (up to 32 ROC’s), and builds the event into the CEBAF common event format (see Appendix E). Event fragment numbers are checked to detect missing data. If synchronization events are implemented (through the Trigger Supervisor) then the event builder can automatically resyncronize the event stream flagging the previous section of “bad” events.

The Event Builder supports spying on the data stream from multiple sources (i.e. copy an event from the stream), as well as inserting user specified events into the event stream, through a remote procedure call (RPC) interface.

2.1.4 On-line Analysis

The on-line analysis program is written by the experimenter using utility routines from CODA to read and write events. HBOOK routines from the CERN program library are recommended for histogramming applications because of the large number of compatible utilities and routines.

It is possible to improve performance by incorporating both the Event Builder and the Analyzer into a single program. The command *coda_ebana* is the default Event Builder/Analyzer provided with CODA.

2.1.5 Event Recording

All events are written in a CEBAF standard event format (see Appendix E). Currently users can log event data to standard file system named output files. Rudimentary support for writing directly to tape is provided by using the tape (8mm or DAT) device file as the logging file name (i.e. /dev/rmt/0mn).

2.1.6 Histogram Display

The CERN program PAW (Physics Analysis Workstation) may be used to interactively display HBOOK histograms either live (ULTRIX only) or stored on disk. This very large and capable program includes data manipulations, peak fitting, and many other features. (See the PAW Reference Manual for more information).

2.1.7 Diagnostic Event Dump

CODA contains a diagnostic utility (*cefdmp*/*xcefdmp*) to print the contents of an event in an easy to read format. Recursively embedded structures are displayed along with ASCII titles obtained from an event tag dictionary customized by the user.

The X-windows version of this dump utility presents the structure of the event in a graphical form, with data optionally shown in pop-up windows.

2.2 Run Control Architecture

The data acquisition system is controlled by a single Run Control process running on a host machine or workstation. This program views the experiment as consisting of a number of subsystems; each subsystem in turn contains one or more components. The following subsystems are implemented:

1. user trigger system
2. trigger supervisor
3. readout controllers (1 or more)
4. event builder
5. analysis (currently a single Unix process)
6. output device
7. EPICS Interface (not yet implemented)
8. data acquisition run (this conceptual subsystem describes the behavior of a data acquisition run)
9. arbitrary user subsystem

Each subsystem may consist of 0 or more components (i.e. there may be 3 ROC's making up the "readout controllers" subsystem). The minimum system is one ROC, one event builder, and one analyzer.

2.2.1 State Machine Model of Run Control

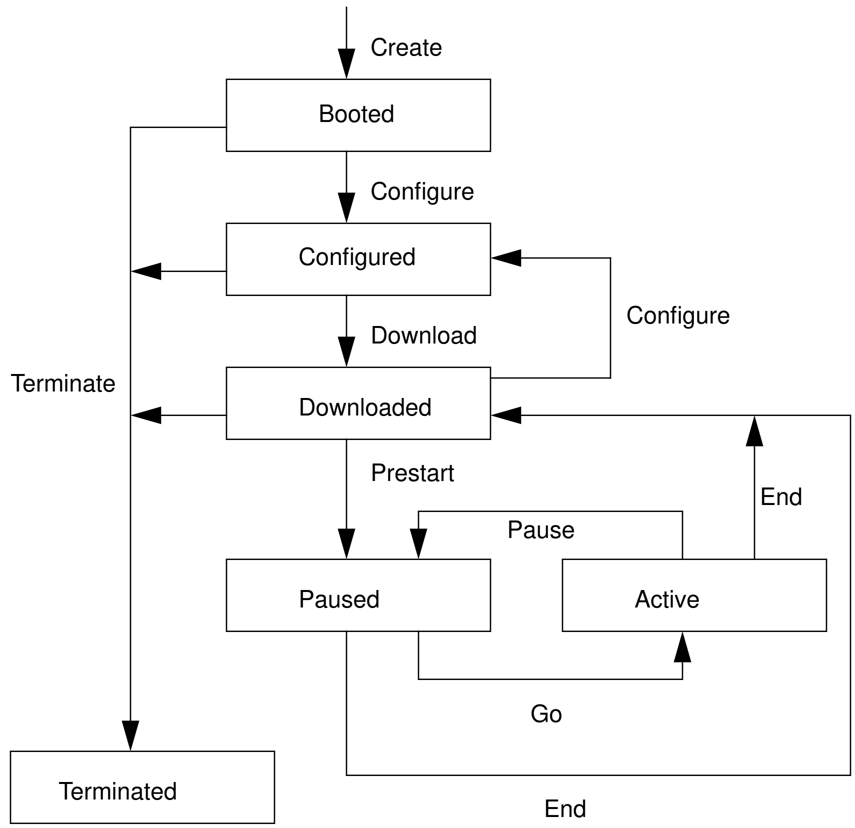
Each component is a collection of software and possibly hardware which at any point in time is in one of several possible states. The state transition diagram shown in Figure 1 describes the behavior of a single component, and of the system as a whole.

Each command (Configure, Download, Prestart, Go, Pause, End, Terminate) is automatically propagated from the system down to each component in a user definable priority order. If any component fails to successfully make the state transition, the system as a whole remains in the previous state. Many transitions are performed asychronously through intermediate states not shown in the figure.

2.2.2 User Definable Run Control Components

Most of the components in a CODA system are pre-defined by CODA and only require configuring (e.g. setting trigger supervisor options). However, CODA allows for two different user created subsystems and components: the user's trigger subsystem & components, and the arbitrary user subsystem & components. Each of these components is defined to Run Control as a remote procedure callable (RPC) service. (The CODA/EPICS interface will also be defined in this context.)

FIGURE 1 State Transition Diagram



The RPC server defines a pre-defined set of entry points which correspond to the set of state transitions (Create, Config, Download, PreStart, Go, Pause, End, Delete). Values returned by these RPC routines indicate whether the requested action completed successfully, failed, or is still pending (in which case Run Control will periodically request a status update until it succeeds, fails, or a time-out elapses).

Details of how to create a user defined component will be given in a future application note.

2.2.3 Run Control Configuration Concepts

The Run Control process is configured through a set of ASCII files, which may be created and edited with any text editor. These files are located in a directory pointed to by the environment variable RCDATABASE. In the simplest case, three files are necessary. (For more complete descriptions, see Chapter 3.)

The first configuration file is the Run Control Network Definition File, *rcNetwork*. It lists each component (RPC service) in the system by name, and gives its internet address. Certain components are pre-defined by CODA and are not included in this file.

The second configuration file, *rcRunTypes*, defines a set of “run types”, giving a correspondence between run type names and run type numbers. (Prior to starting a run, the user specifies what type of run is desired by entering a run type name. Examples might be “physics” or “calibration”.)

The third configuration file, the Run Configuration File, specifies a configuration string (generally the name of another file) for each component to be included in the run. The name of this file is the name of the run type followed by “.config”, e.g. *physics.config*. There must be one such file for each defined run type. Each component listed in this file becomes part of any run of that type. (Calibration runs, for example, may only require a subset of the ROC’s).

The configuration string in the Run Configuration File is used as an argument to the Config routine of that component. For example, a user may define a component named ROC3 of type ROC (readout controller). The configuration string will be sent to that readout controller during the transition to the Configured state. The CODA standard ROC code interprets this string as the name of a file containing the user’s data acquisition readout lists (NOTE: in CODA 1.4, this must be the full name of the compiled object module). In general, the meaning of the string is dependent upon the component server code.

2.2.4 Run Control User Interface

The user communicates with the Run Control process via a Motif (X11) windows display. The top level window contains push buttons to control a run (state transitions), as well as menu selections to implement or disable various experiment control options. Information on the status of most of the subsystems in CODA can be accessed, each subsystem interface is created on demand as the user requests a subsystem screen by clicking on the appropriate menu selection.

2.3 Data Analysis & Monitoring

The data analysis program, under control of the run control program, reads events from the event builder, performs some analysis, and optionally writes the events to the event output subsystem.

2.3.1 Event I/O

Five routines are provided for event I/O: evOpen, evClose, evRead, evWrite and evIoctl (see Chapter 4, Event I/O routines). These routines currently perform I/O to files, and will eventually be able to read events from the Event Builder and write them to the Event Recorder. Events are checked on output for conformance to the CEBAF event format at the outermost level.

2.3.2 Histogram Display

Support is provided for HBOOK routines[1] using shared memory (ULTRIX Only). This allows histograms to be displayed using PAW[2] (Physics Analysis Workstation) as they are being accumulated. Histograms may be written to disk either by PAW or by the analysis program, and read back again later with PAW or another analysis program. The *cdumphist* utility allows to user to periodically dump histograms to a disk file.

1. HBOOK is a package of histogramming routines, and is part of the CERN program library.
2. PAW is also part of the CERN program library.

2.3.3 Formatted Event Dump

For diagnostic purposes, there is an event dump utility "*cefdmp*", which can dump event records obtained from a readout controller, the event builder, the event logger, or an event file. This program uses information contained in the event, along with an event tag dictionary (which gives names and titles for each piece of an event), to display an easily readable dump of a single event. This is helpful to verify operation of the readout controllers (and the corresponding readout lists and hardware), as well as to verify the output of an analysis program.

An X-windows version of this utility, *xcefdmp*, can also display the internal structure of the event graphically, and displays selected pieces of the event in pop-up windows.

CHAPTER 3

3

Using CODA

This chapter explains how to use each piece of the CEBAF On-line Data Acquisition (CODA) system. Examples of configuration files may also be found in the CODA online examples directory ($CODA/examples/).

3.1 Run Control

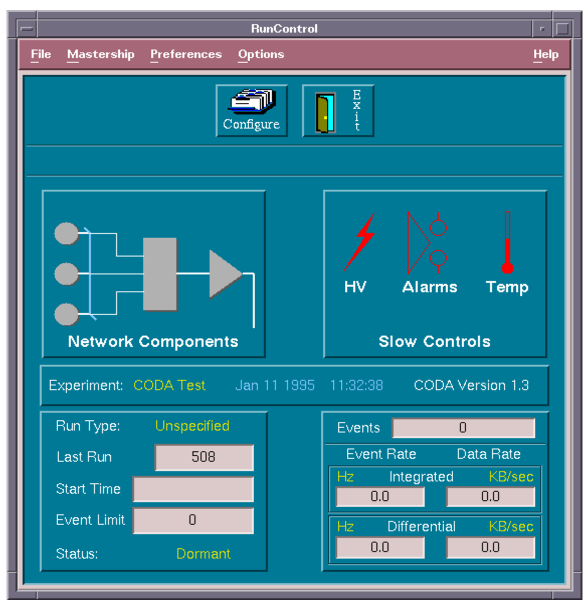
In order to gain access to the Run Control program, the user must execute the following command, either interactively or in a ".login" file[1]:

```
setup  coda/1.4
```

Once this is done, Run Control is started by typing:

```
RunControl
```

FIGURE 2 Run Control top level window (Motif Interface)



1. "setup" is a CEBAF script for gaining access to optional software. If setup is not available, follow instructions in the README file in the CODA distribution.

At this point, Run Control reads a file of default settings (defined by RCDEFAULTS environment variable), and then it displays the top level window (see FIGURE 1.1). Next, two configuration files are read: the Run Control Network Definition file (file name: rcNetwork), and the Run Type Dictionary file (file name: rcRunTypes). These files are pointed to by the environment variable RCDATABASE, which is the name of the directory in which the files rcNetwork and rcRunTypes are to be found. If this environment variable is not defined, it defaults to rcDatabase. (Note that the current setup script for CODA defines RCDATABASE to point to an example directory of demo files.)

3.1.1 The Network Definition File

Each entry in the network file defines one distributed component (other than the Run Control process itself) in the data acquisition system. Each entry has the following format:

```
componentName  number  type  hostName  [command]
```

The *component name* is an arbitrary alphanumeric name by which to refer to this component, and is used in other files. The component *number* is a number from 0 to 31 used in data structures created by the component (e.g. readout controllers insert this number at the head of their bank of data). The component *type* is one of the supported types given below:

TS -- trigger supervisor

ROC -- readout controller

EB -- event builder

ANA -- analysis

ER -- event recorder

UC -- user defined component

LOG -- console logger

The *host name* is either the IP host name (e.g. xyz.cebaf.gov) or IP address (e.g. 123.45.67.89), and is used along with the component type to find the program on the network. Each component type is assigned an RPC program number (not user configurable), because there can only be one program of a particular number on each host (RPC restriction), Run Control can only communicate with at most one component of each type on a single machine. (This restriction will be removed in CODA 2.0.)

The optional *command* may be used to automatically start the program containing the component. If RunControl fails to connect to the component during the “Download” transition (described below), it spawns the listed command, passing it the first four fields as arguments. That is, it effectively executes the following:

```
command  componentName  number  type  hostName
```

command may simply be an executable file, or may be a script to start a program. If the component is to run on a different machine than RunControl, *rsh* is used to create the process on the remote machine. If *rsh* is used, remember to place appropriate entries in the user’s .rhosts file (see your system administrator if you need help doing this). For an example of a script which invokes “rsh”, see the file *coda_activate* in the directory *$CODA/bin*. This script may be used to start any program on another machine using the following as *command*:

```
$CODA/bin/coda_activate -p ~/mydir/myprog
```

where the argument after the -p is the filename of the program to start.

RPC programs (components) may be started manually (not recommended), or they may be started by the inet daemon on a machine, or they may be started by Run Control using the command given in the network file. In order to have them started by the inet daemon, the program number and file name must be added to the inet database (/etc/services on Ultrix).

Example rcNetwork file:

```
ROC0 0 ROC  hostname0
ROC1 1 ROC  x.cebaf.gov
MYEB 7 EB   xyz  $CODA/bin/coda_activate -p ~/work/ana
MYA  7 ANA  xyz  $CODA/bin/coda_activate -p ~/work/ana
```

Note that it is necessary to include an entry for the event builder . It should have the same arguments as for the ANA (this restriction of analyzer and event builder on the same node will be lifted in CODA 2.0). It is, however, not necessary to include an entry for a console logger. One will be added automatically on the same machine as RunControl.

By default the console logger saves all log messages in a file called coda_console.log in the same directory as that from which RunControl was activated. This default may be overridden by specifying the logger in the rcNetwork file. For example, the following line disables the logfile:

```
LOG 0 LOG $HOST $CODA/bin/coda_activate -nolog
```

For more detailed information, refer to the section on coda_activate in Chapter 5.

3.1.2 The Run Type Dictionary

The Run Type Dictionary contains one entry per run type. Run types are arbitrary names assigned to a collection of hardware, software, and configuration options. For example, a particular experiment may require 2 distinct sets of running conditions, one for physics, and one for calibration. In this case, run types of “physics” and “calibration” could be defined in the dictionary. Each entry has the following form:

```
runTypeName  runTypeNumber
```

where *runTypeName* is a user defined alphanumeric string, and *runTypeNumber* is a user assigned integer that will be used to refer to this run type in calls to system routines.

Example file:

```
physics 1
calibration 2
test 3
```

3.1.3 The Run Type Configuration File

For each run type in the dictionary, there must be a corresponding configuration file of the same name as the run type, with the extension “.config”. So for the example above, there would be three files, “physics.config”, “calibration.config” and “test.config”.

These files are called Run Type Configuration files; each of these files contains configuration information for each component participating in that type of run. Each entry in the file has the following format:

```
componentName  configString
```

The *configString* is a string which is passed to the component to configure it for this type of run. For most components, this string is in fact the name of a component specific configuration file. These configuration files have formats which are specific to the component types (TS, ROC, etc.), and will be discussed in the sections of this chapter that deal with those subsystems. These files are parsed by the components in parallel, speeding up starting a run. Currently, the EB component requires no argument.

Example file:

```
ROC0 drift_chamber.o
EB
ANA data_logging_filename
```

3.1.4 The Run Type Options File

For each run type there is a run options file with an extension of ".options" (e.g. "physics.options"). This file contains values for any options which may vary from run type to run type. The format of this file is

```
optionName  optionValue
```

Currently, one supported *optionName* is "runNumber", which is used to control assigning a number to the next run. If the corresponding *optionValue* is "increment", the current run number (stored in file rcRunNumber) is incremented as the run is started. If the option value is "noincrement", the value in rcRunNumber is used and not modified. The default (when no file or entry is present) is "increment".

Example file:

```
runNumber increment
```

In addition to the runNumber option, the user can in the options file define user supplied scripts to be executed at the various transitions during the stopping and starting of runs. The format of each line is

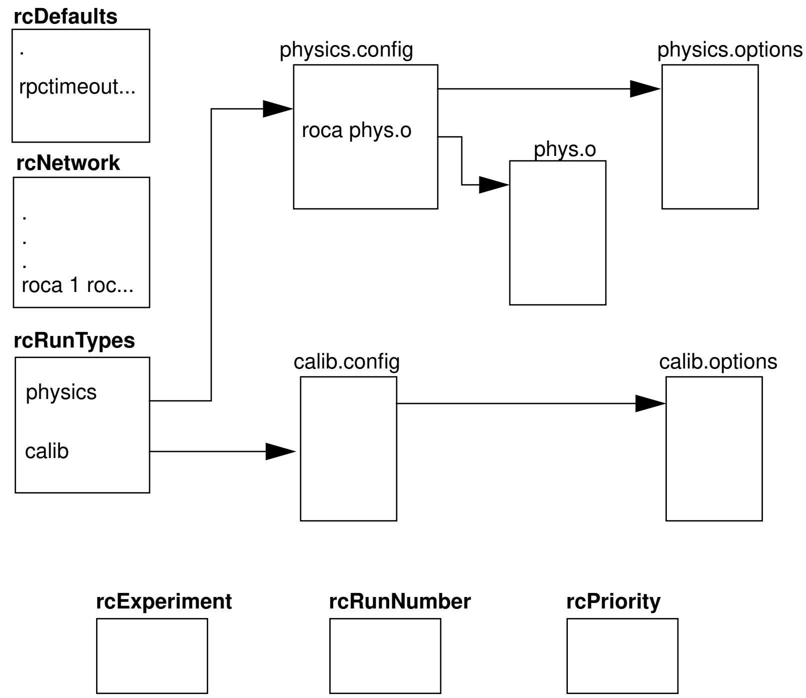
```
transitionName  priority  scriptName
```

The transitionName can be one of: download, prestart, go, pause, or end. The priority is an optional number defining the precedence the script takes in the transition sequence (the default is 29. See Appendix A for details on priority numbers). The scriptName specifies the name (including path) for the executable shell script. RunControl will pause the transition until the execution of the script has been completed - successfully or unsuccessfully.

3.1.5 Configuration Files Summary

A summary of the configuration files for CODA is given below. Further details are available in Appendix A.

FIGURE 3 CODA Configuration Files



3.1.6 Starting & Stopping a Run

At the top of the Run Control window is a set of push buttons used to control a run. When the program is first started, Run Control is in the “booted” state, and only 2 buttons appear: “Configure” and “Quit”. Pushing the “Configure” button causes Run Control to first prompt for a run type, giving as options entries from the Run Type Dictionary. Next, the “.config” file is opened, and configuration strings are read.

In each of the following states, buttons appear at the bottom left which are appropriate for that state. Commands which cause state transitions are automatically propagated out to all subsystems and components, and will cause actions defined by those subsystems to be executed. In the rest of this discussion, some of the actions are mentioned, but for a full discussion of what happens to a particular subsystem during a state transition, see the corresponding subsection in this chapter for that subsystem.

In the “configured” state, 3 buttons appear: “Download”, “Quit”, and “Configure”. Pushing “Download” causes each component to perform any actions necessary to apply the selected configuration (update hardware, etc.). It is at this point that data acquisition readout lists are downloaded into the readout controllers, for example. If this step completes successfully, Run Control is in the “downloaded” state.

In the “downloaded” state, 4 buttons appear: “Prestart”, “Auto Start”, “Abort”, and “Configure”. Pushing “Prestart” causes each component to perform any actions necessary to prepare for data acquisition, and enters the “paused” state. Pushing “Auto Start” is equivalent to pushing “Prestart” followed by “Go”. Pushing “Abort” returns Run Control to the “configured” state (previous menu).

In the “paused” state, “Go” and “End” buttons appear. The first enables triggers and moves to the “active” state, and the second returns to the “downloaded” state.

In the “active” state, “Pause” and “End” buttons appear. The first disables triggers and transitions to the “paused” state, and the second disables triggers and returns to the “downloaded” state (ends the run).

Graphical feedback on these state transitions may be obtained by clicking the mouse in the bottom lefthand rectangle of the main RunControl view. This brings up an additional Run Control window showing the state transition diagram and all allowed transitions, as well as indicating the current state. All state boxes in this view are buttons which allow the user to control the run from this window alone.

3.1.7 The Options Menu in RunControl

There are a number of optional features of Run Control the user may find useful when communicating with service components and starting and stopping of Runs during an experiment. These can be accessed without editing RCDATABASE files through the Options menu in Run Control. Below is a list of available Options and a brief description of each.

RPC Timeout	Determines the time in seconds the Run Control rpc communication will wait for a reply. The rpc request is issued 3 times. The timeout refers to a single request.
Set Run Number	Automatically edits the rcRunNumber file with a user input run number and updates the Run Control display.
Automatic Reboot	Specifies a timeout for Run Control to wait before automatically starting up a new Run.
Set Event/Data Limit	Allows the user to set both (or either) a number of events limit or a total data limit before automatically ending the run.
Log Events to a File	The user may optionally specify the name of a file in which to write events or turn off data logging altogether.
Remote Run Display	The user may specify an X Windows server to display an enlarged Run Status Box (no functionality).

Choosing the following options is identical to clicking on one of the 4 front panel windows in RunControl (Slow Controls is not implemented in CODA 1.4).

Select Active Component	Provides a list of the active components for the particular runType chosen. Selecting a given component will display the event and data rates from that component in the lower right panel of Run Control. (Same as clicking the upper left front panel of Run Control.)
Show Run Status	Brings up the Run Status Button Display showing the current state of Run Control and providing the available transition buttons to control the Run. (Same as clicking the lower left front panel of Run Control.)
Show History	Brings up a graphical display of the event and data rate history for the component chosen using the Select

Active Component Option. (Same a clicking the lower right front panel of Run Control.)

3.1.8 RCDEFAULTS Environment Variable

Several of the default initial settings for Run Control may be overridden by specifying an optional defaults file. This file is identified to Run Control through the RCDEFAULTS environment variable. If this variable exists, Run Control treats it as specifying a text file containing initial settings. The format of the file is similar to that of the other configuration files, each line specifying a setting name and its initial value. An example RCDEFAULTS file is:

```
! A comment line
buttonfeedback    true
online            true
rpcupdate         true
rpctimeout        3
verbosereporting  true
```

The allowed setting names, which may be specified in upper or lower case, are:

buttonFeedback	Determines whether the various buttons on the Run Control command panels show feedback describing their actions when the mouse cursor enters them. The default is true.
online	Determines whether Run Control is online. A value of false will result in state transitions being performed without any communication with the components described in the rcNetwork file. This is useful for diagnostics. The default is true.
rpcUpdate	Determines whether updating of variables from components in the rcNetwork file occurs. The default is true.
rpctimeout	Determines the time in seconds the Run Control rpc communication will wait for a reply. The rpc request is issued 3 times. The timeout refers to a single request.
verboseReporting	Determines how verbose the status messages in the scrolling status region are. The default is true (verbose).

3.2 The Trigger Supervisor

Current (CODA 1.4) Trigger Supervisor support is through communication with a special type of readout controller (ROC). This ROC behaves as any other ROC except for two differences. It possesses an rpc program number that identifies it as a trigger supervisor (as opposed to the ROC program number). This defines the priority with which it communicates with Run Control. Also, it is configured to be an “Asynchronous” ROC meaning that it will not issue Prestart ,Go, Pause, and End events to the event builder. Therefore, the event builder will ignore this ROC with respect to building physics events. Any events generated by this component and sent to the EB will be treated as user specific and be passed through untouched to the analyzer.

In the VME crate which holds the Trigger Supervisor module the *coda_ts* code must be running on the resident CPU (i.e. MV162 or MV167). The programing of the trigger supervisor is done though a compiled downloadable readout list to the running program. Configuration of the trigger supervisor in the Run Control database is identical with the normal readout controller only the type TS is specified in rcNetwork file.

The functionality and ease of configuration are being improved for the release of CODA 2.0, allowing the user the ability to program the TS through a simpler file format as well as through a graphical interface invoked form RunControl. In the following sections a description of the available parameters and their allowed values are described. For more information, see the Trigger Supervisor User's Guide. In Appendix B an example of a TS readout list is provided and the TS programming described.

3.2.1 Configuring the Trigger Supervisor

In order to include the Trigger Supervisor into a setup, there must be an entry in the Run Control network file with a type of TS giving its IP hostname/address. In addition, the configuration file corresponding to the run type in use must have an entry for the TS component (see the Run Control section for a discussion of these files). The configuration file entry gives the name of a file containing values for the Trigger Supervisor parameters in the following format:

```
parameterName  parameterValue
```

When the Trigger Supervisor is downloaded, this configuration file is loaded into the hardware. Parameters omitted from the file take their default values. Changes may be made to the file's settings using the Trigger Supervisor control screen.

TABLE 1 Trigger Supervisor Parameter Defaults

Name	Default Value	Comments
tsPreScale1,2,...12	1	(12 separate lines)
tsEnable	0xFFF	(all lines enabled)
tsEnable1,2,...12	1	(alternative to tsEnable)
tsStrobe	0	
tsTriggerClass	'L1OK=1; TC1=1; TC2=0; TC3=0'	
tsRocCode	'RC0=1; RC1=0; RC2=0; RC3=0'	
tsL1Accept	'L1A1=1; L1A2=1;... L1A8=1'	
tsRocEnable1,2,3,4	1,0,0,0	(single ROC)
tsRocLock	1	(force lock step operation)
tsRocLock4	0	
tsClearPermitTimer	0	(disabled)
tsLevel1Timer	0	(disabled)
tsLevel2Timer	0	(disabled)
tsBusyTimer	0	(disabled)
tsClearHoldTimer	0	(disabled)
tsSyncInterval	0	(no SYNC events)

3.2.2 Enables and Prescales

The Trigger Supervisor has 12 trigger inputs. The first 8 of these are prescalable: the first 4 by 2^{24}, the second 4 by 2^{16}. Prescales are entered as decimal numbers into the appropriate box on either the pop-up box or the parameters screen. Each trigger input may be enabled or disabled by clicking on the corresponding toggle button. The inputs may be used as triggers, or a separate trigger strobe input may be used to latch the 12 inputs. The strobe setting is selected by a separate push button.

3.2.3 Memory Lookup Units

The Trigger Supervisor has 3 memory lookup units (MLU's), addressed by the 12 trigger inputs (appropriately latched). The trigger class MLU is used internal to the Trigger Supervisor to determine how many levels of user trigger to wait for. The readout list MLU gives the readout list number to forward to all readout controllers. The gate output MLU controls which gate output bits are set for each trigger pattern.

MLU contents are specified as logic equations, one equation for each bit of output. Equations for each bit are separated by semicolons. Input bit names are t1,t2,...t12, and can be negated with a leading slash (/). Output bit names are specific to each MLU.

TABLE 2 Trigger Supervisor MLU Output Bit Definitions

MLU	BIT	Description
Trigger Class	L1OK	If set, this is a valid trigger pattern, else re-enable triggers
	TC1	If set, this is a class 1 trigger
	TC2	If set, this is a class 2 trigger
	TC3	If set, this is a class 3 trigger
Readout Code	RC0,1,2,3	Binary code to send to ROC's for this trigger pattern. Each bit must be programmed separately.
Level 1 Accept	L1A1,2,3,...,8	Drives front panel level 1 accept outputs. Each bit must be programmed separately.

Equations are expressed as sums of products, with parentheses to nest terms:

```
rc2 = t1 + t2*t3*/t4 + t5*(t6+t7*/t8)
```

3.2.4 Readout Controller Interface

The Trigger Supervisor supports up to 32 readout controllers (ROC's) on 4 branches of up to 8 ROC's each. Each branch transmits readout codes from the Trigger Supervisor readout code fifo's to the ROC's on that branch. The fifo's can hold up to 7 codes, allowing the Trigger Supervisor and front end modules (ADC's and TDC's) to be up to 8 events ahead of the ROC's. The 4 branches operate independently, so that all ROC's on a branch are processing the same event (and must wait for the slowest on that branch), but ROC's on different branches may be operating on different events.This pipelined mode of operation reduces the system dead time for systems that contain front end modules capable of buffering multiple events (generally 8).

Six parameters control the operation of this interface. The first four of these, "tsRocEnable1,2,3, and 4", indicate which of the 8 possible ROC's are in use on the corresponding branch. These 8 bit integers may be specified as a decimal number, or as a hex number with a leading "0x" (e.g. 0xff).

Two additional parameters control whether pipelining is enabled or not. If tsRocLock is set, all pipelining is disabled. If tsRocLock is 0 and tsRocLock4 is set, then pipelining is disabled only on branch 4. If both are 0, pipelining is enabled on all 4 branches. (See the Trigger Supervisor User's Guide for more information about pipeline operation).

3.2.5 Timers

There are 5 timers in the Trigger Supervisor to assist in customizing the operation of its state machine. Their names, maximum values, and meanings are given in the following table.

TABLE 3 Trigger Supervisor Timers

Name	Maximum Value	Description
tsClearPermitTimer	2.6 ms	Longest time during which a level 2 or level 3 reject will produce a fast clear signal. This is meant to reflect the time interval during which front end modules are willing to accept a fast clear.
tsLevel2Timer	2.6 ms	This timer is used with class 1 triggers to define a time delay between the level 1 accept and level 2 accept signals generated by the TS.
tsLevel3Timer	2.6 ms	This timer is used with class 1 and 2 triggers to define a time delay between the level 1 accept and level 3 accept signals.
tsBusyTimer	2.6 ms	The time period following level 1 accept that can represent the conversion or dead time of the front end module. The TS will not re-enable triggers after an accepted trigger for at least this time.
tsClearHoldTimer	5.1 us	Width of the TS's CLEAR output signal.

The first four timers are programmed in increments of 40 ns, and the last in increments of 20 ns. E.g., setting tsClearPermitTimer to 50 gives a 2 us fast clear permit interval. Setting any of the timers to 0 disables the feature.

3.2.6 Synchronization Interval

In pipeline mode, the Trigger Supervisor is designed to periodically pause and allow all readout controllers to catch up (i.e. empty the readout code fifo's on the trigger branches and the corresponding event fifo's on the front end boards), and verify that all ROC's have not dropped or gained an event. This is done by counting events up to the counter tsSyncInterval, and then automatically generating a special code telling the ROC's to check their modules for any left over data (there should be none). Any ROC detecting an error reports this to the error logger. It is expected that off-line analysis programs will use this log to discard blocks of events in which a synchronization error occurred.

The synchronization interval should be small enough so that the probability of an error in the window is very small. Making it too small, however, generates more sync events, and therefore may increase the dead time slightly. The largest value possible is 65535, and a value of 0 disables sync event generation.

3.3 Using the Read Out Controllers

The readout controllers are responsible for receiving a trigger code, executing the corresponding readout list, and passing the event fragments on to the event builder. (In CODA 1.4, this is accomplished by an IP socket connection over ethernet or FDDI.) In addition, ROC's cooperate with the Trigger Supervisor in checking event fragment synchronization, and cooperate with Run Control in producing pseudo events to be sent up the data chain upon change of state.

In CODA 1.4, three hardware ROC's are supported: the FSCC (FASTBUS Smart Crate Controller)[2], 68K VME boards running VxWorks for direct VME readout or interfaced to CAMAC via any interface for which the CAMAC standard routines are available, and for 68K based CAMAC smart controllers for which the CAMAC standard routine library is available (such as the VCC2117)[3].

3.3.1 Software Configuration

Getting a readout controller operating consists of the following steps: (1) booting the VxWorks operating system (done at power on either through boot script or direct from ROMs), (2) downloading the CODA libraries and daemons, and (3) downloading the user specified readout code (part of the "download" step in Run Control).

Each of these steps require the VxWorks node to have access over the network to the appropriate file systems. Generally this is done via an *rsh* command, requiring an appropriate entry in a *.rhosts* file. See your local system manager for help in setting this up. In addition, for Ultrix nodes running the BIND service, it will be necessary to put the VxWorks node names into the name server. In all cases, it will be necessary to have the names of the machines running RunControl and the Event Builder entered into the host tables on VxWorks. This is done by the hostAdd command (usually in the boot script for that board):

```
hostAdd "myhost","123.45.67.89"
```

It is also necessary to have the portmap daemon running on each machine to be used by CODA in order for the remote procedure calls to function. Since this is not the default for Ultrix nodes, the file /etc/rc.local will have to be edited to include the appropriate commands (see your system manager for assistance). It is possible to test for the presence of the portmap daemon by typing the following at the Unix prompt:

```
/etc/rpcinfo -p myhostname
```

where 'myhostname' is the name of the machine you wish to test. This command will either produce a listing of RPC programs active, or will report that the portmap daemon is not running.

At CEBAF, the account used by the VxWorks single board computer to access files has the following entry in its *.cshrc* file:

```
setenv CODA /path/to/version/of/coda
```

2. Sold by BiRa. See also "An Intelligent Readout Controller for FASTBUS, The Fermilab FSCC", Cancelo et al., Conference Record of the 1990 IEEE Nuclear Science Symposium, 292.

3. Sold by CES. Ortec is the local distributor.

and a boot script filename containing, for example, the following (for the FSCC):

```
cd "$CODA/VXWORKS68K51/bin"
ld < coda_roc
taskSpawn "ROC",110,spTaskOptions,8000,roc_coda
```

should be available and specified in the VxWorks CPU boot RAM.

Readout code is specified to CODA as a file containing commands for that controller (either FASTBUS macros or CAMAC macros). This file is compiled into a downloadable object file by the *makelist* utility. For each ROC, the name of its compiled readout file is entered in the configuration file for that run type (see section 3.1.3 on page 3-3). When the "download" command of Run Control is executed, the files are downloaded and linked into the polling or interrupt service routines.

NOTE: In a future version of CODA, the compilation will be done automatically, but for CODA 1.4, the user must manually compile the list using the *makelist* script in $CODA_BIN:

```
makelist myfile.crl 5.1
```

(the 5.1 indicates a VxWorks version number). Although currently only a single list is supported, the ROC code will eventually be able to specify separate lists for each of the 16 event types (binary encoded on the trigger cable, 0 to 15); lists for state transitions (download, prestart, go, pause, resume, end) are currently supported. Commands in the list may write to control registers in I/O modules (e.g. to select a particular data register), write data into the output stream (to insert user defined markers), or transfer data from the I/O modules to the data stream. See Appendix C, *Readout Controller Configuration File (Language Summary),* for details of writing a readout list.

The output into the event stream of each readout list is a single bank containing 4 byte integer data. The tag for the event is the readout controller number contained in the rcNetwork file. The "num" field in the header contains an 8 bit event counter used to synchronize event fragments. For details of the format of these banks, see Appendix E, *CEBAF Common Event Format.*

3.3.2 Using FASTBUS (the FSCC)

The FSCC has 4 front panel trigger inputs plus a front panel strobe input; these are differential TTL inputs, and therefore can not be driven directly by the Trigger Supervisor's trigger cables. Instead, the trigger cable connects to a small I/O card on the back of the FSCC, and a short cable connects from there to the front panel (the user must make this modification on the FSCC to enable use with the trigger supervisor).

If it is desired to run the FSCC without the Trigger Supervisor and I/O card, connect the trigger strobe to pins 11 & 12 (differential TTL), and connect the trigger type bits to pins 13-20. If only the trigger strobe is connected, readout list 1(i.e. trigger type 1) will be executed for all events. ROC trigger acknowledge (busy reset) is output on pins 1-2.

3.3.3 Using CAMAC

CODA currently supports one or more CAMAC crates with Kinetic Systems 3922 crate controllers interfaced to a VME crate with a 2917 VME-KBUS interface. The KBUS cable is connected at one end to the 2917, and is daisy chained to all the 3922's. Each 3922 must be set to a different crate number, and the VME interface must have its VME address set to 0xFF00 (factory default). All crates connected in this way will appear on

CAMAC "branch" 0. The VME crate should have the CPU board in slot 1 (left most) and the 2917 in slot 2 (or a higher numbered slot if all intervening slots pass IACK. This is required if the 2917 is operated in interrupt mode).

In addition to the VME interface to CAMAC, CODA 1.4 supports the CES VCC2117 ethernet smart CAMAC crate controller. This board will allow small systems to be constructed consisting of only a CAMAC crate and a workstation, separated by an arbitrarily long ethernet network. There is, however, no Trigger Supervisor interface to this configuration. In principle, any other interfaces for which the CAMAC standard routines are available would also work.

To run a CAMAC crate without the Trigger Supervisor, simply include CAMAC instructions in a "prestart" list to enable LAM's on one module in one of the CAMAC crates. Arrange signals to that module will produce a LAM on every event; the readout controller is programmed to automatically execute readout list 1 on a LAM interrupt. Support for multiple LAM's is under consideration.

3.3.4 Using VME

CODA 1.4 supports use of any 68K based VME CPUs running VxWorks 5.1 (such as the Motorola MV162 and MV167 boards). Kernels provided with these specific boards support A16/D16, A24/D24, A24/D32, and A32/D32 addressing on the VME bus. At this time there is minimal readout language support for addressing modules (except for specific modules such as the 2917 CAMAC interface), hence, the user is left to use imbedded C code in his readout list to address their specific modules. See Appendix C for a description on how to do this.

3.4 The Event Builder

The event builder is implemented as routines which are capable of accepting event fragments via IP sockets from multiple ROC's. Like other components in the data acquisition system, it is defined to Run Control through the rcNetwork file (section 3.1.1 on page 3-2). There must be an EB entry in this file that matches an ANA entry. (RunControl no longer explicitly creates one.)

Currently, the event builder and analyzer must be a linked single process on the same machine. This restriction will be relaxed in future releases of CODA allowing multiple analyzers on different CPUs to be utilized.

Once started, the event builder waits for connection requests from readout controllers and the analysis program. As event fragments arrive from the ROC's, internal fragment numbers are checked for consistency. Fragments are then merged into a single data structure in the event format described in Appendices E and F. (Each fragment is contained in a separate bank, and the event builder creates an additional bank to contain standard event information such as the event number.)

Events are built as long as data fragments are pending or until a high water mark in the event builder is exceeded. Then, events are passed to the analysis program, which may optionally inquire how many events are pending for analysis (via a trailer word attached to the event).

In addition to the above task, the event builder provides a number of other services available via its remote procedure call (RPC) interface:

- get a copy of an event (with selection criteria)
- push an event into the event stream (User/EPICS events)
- get status information (number of events built, etc.)

The first of these services can be used by the event dump utility to examine events prior to analysis. The second service is useful to injecting miscellaneous information into the data stream, either to control the analysis program or change calibration constants, or simply for archiving.

3.5 Running an Analysis Program

The analysis program is part of the data acquisition pipeline, receiving events from the event builder and (optionally) writing events to an output file.

The analysis program is started automatically when the “download” command is issued in Run Control. It performs whatever initialization it requires, opens the connections to the event builder and the event output, and then reads an event from the input connection.

Alternatively, the analysis program may declare itself to CODA as an analysis program, and then pass flow control to a CODA library routine. This routine calls user supplied subroutines each time an event is received from the event builder, and handles a few other tasks such as opening the output data file as specified in the user’s *.config* file. See the Run Control Communications library in Chapter 4 for additional details.

The first event obtained will be a “prestart” event followed by a “go” event. (See Appendix F for the definitions of these events.) The “prestart” event will contain the run number and run type number, which the analysis program may use to finish any necessary initialization. After the “go” event will be 0 or more data events interspersed with 0 or more “pause” and “go” event pairs, and finally an “end” event.

Other types of events may also appear in the data stream if another program is in use which pushes events into the event builder.

The analysis program is allowed to read and write any files it requires. Summary information may be written in any form -- this is left completely up to the implementer. If histograms are desired, it is recommended that the CERN HBOOK routines be used, as it permits the histograms to be viewed in real time by PAW (Physics Analysis Workstation, a CERN program, ULTRIX only).

In order to simplify compiling and linking an analysis program, a script has been provided which automatically links to the CODA libraries and the CERN libraries. This script (located in $CODA/bin) may be used directly, or may be copied and customized as needed:

```
codaf77  [compiler-options]  <filename>.f
```

3.6 Using the Event Recorder

While the analysis program may write its events directly to disk or tape, it may also choose to write events to the Event Recorder process. This process accepts events from the analysis program and copies them to a file specified in the <runType>.config file. If this file name contains the string “<runNumber>”, it is replaced by the current run number (decimal).

Like the Event Builder, the Event Recorder has a remote procedure call interface which allows the following operations:

- get a copy of an event (with selection criteria)
- push an event into the event stream
- get status information (number of events written, etc.)

Thus it is possible to spy on events at all stages in the data acquisition / analysis pipeline. The utilities *cefdump* and *xcefdmp* (see below and Chapter 5) are available for this purpose.

NOTE: The Event Recorder is currently not implemented in CODA 1.4

3.7 The Event Dump Utility

The event dump utility (cefdmp) can display, in a readable ASCII format, events from the following sources:

- event file
- readout controller (fragment)
- event builder
- analysis output

command format

```
% cefdmp [filename] [-o objectname ] [-t tag] [-u uniquetag]
      [-s start] [-e end] [-d dictionary] [-x]
```

(To check for list of currently available options, type cefdmp with no arguments).

options

x	hex dump for integers (default is decimal)
o	specify name of CODA object from which to get events / fragments
t	select specified tag number or name (full path in event)
u	select specified unique tag, independent of location in event
d	specify dictionary for obtaining tag names and titles
s	number of first record in file to dump
e	number of last record in file to dump

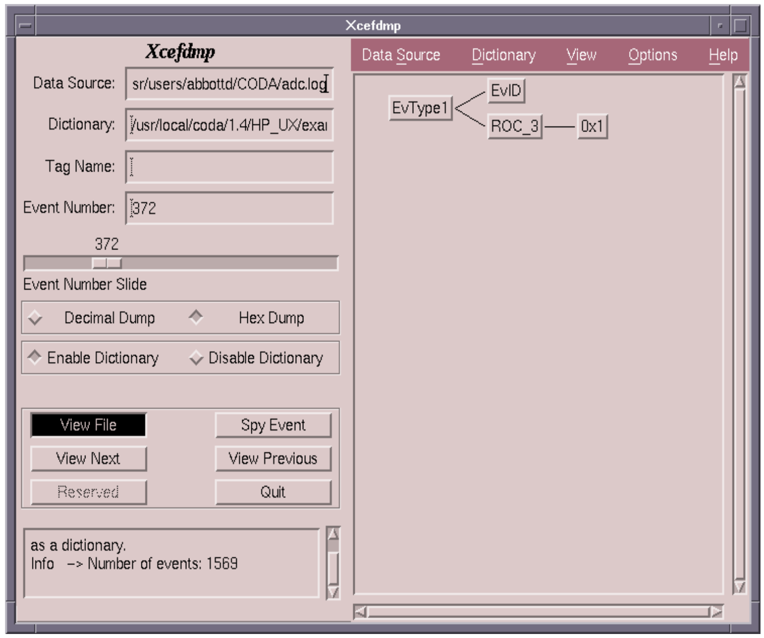
example

```
% cefdmp myfile.dat -t physics.drift
```

This command reads physics events from the file and dumps the drift chamber portion. See Chapter 5 for more examples and details.

An X-windows version of this program also exists, started via the command *xcefdmp*. Options are specified by menus, and the data structure is presented graphically. (See figure below)

FIGURE 4 Xcefdmp window



CHAPTER 4
4

Callable Routines

The capabilities of CODA may be expanded by writing applications which interface with existing CODA routines and programs, often via remote procedure calls. Library routines are available which hide most of the details of these remote calls; these routines and other callable routines are described in the following sections. As an aid to developers of analysis programs, a small subset of the HBOOK histogramming routines are also documented here (all HBOOK routines are available through the CERN library).

Similar routines are grouped together in alphabetical order. Routines are available for:

- CAMAC I/O
- FASTBUS I/O
- Console Log Routines
- Error Messages
- Event I/O
- Histogramming (abstracted from CERN HBOOK manual)
- Run Control communications
- Spying: Communications with Data Acquisition Components

These routines may be called from C or Fortran. Most of the routines (except the CAMAC and HBOOK routines) return as function values a success or error code.

CAMAC I/O Library

An implementation of the CAMAC standard routines (IEEE-758 1979) has been provided both for writers of readout lists, and for applications running on the host workstations. For host based applications, these routines use a remote procedure call (RPC) interface to a dedicated server (caSrvr) running on the CPU to which the CAMAC crate is attached (generally a VME or CAMAC single board computer running VxWorks). The current RPC implementation does *not* support connecting a user routine to a LAM interrupt.

The RPC library is a level B implementation (single actions). The local implementation (C only) includes 10 additional block I/O routines and multiple action routines, corresponding to a level C implementation without the execute-on-LAM feature, and with high address limits ignored in the address scan routines (scans to the end of the crate, or until the requested count is satisfied). The RPC interface for these block I/O routines will be added in a later release. Typical applications for the CAMAC library include CAMAC diagnostic programs, and programs ported from other IEEE-758 compliant systems.

Fortran calling conventions conform to the Fortran implementation standard for IEEE-758. Two additional functions are added, including *caopen,* which is used to establish the connection with the server. There is no IEEE standard C binding, so the C interface is modelled after the Fortran interface, with read-only parameters passed by value.

Standard Functions

- cdreg() Define a register
- cfsa() Execute a single function (32bit data)
- cssa() Execute a single function (16bit data)
- cccz() Crate initialize
- cccc() Crate clear
- ccci() Control crate inhibit
- ctci() Test crate inhibit
- cccd() Control crate demand
- ctcd() Test crate demand
- ctgl() Test graded lam
- cdlam() Define lam
- cclm() Control lam
- cclc() Clear lam
- ctlm() Test lam

Standard Block Functions (local only)

- cfga() General multiple action
- cfmad() Address scan
- cfubc() Controller synchronized block transfer
- cfubl() LAM synchronized block transfer
- cfubr() Repeat mode block transfer

- csga() 16 bit general multiple action
- csmad() 16 bit address scan
- csubc() 16 bit controller synchronized block transfer
- csubl() 16 bit LAM synchronized block transfer
- csubr() 16 bit repeat mode block transfer

Extended Functions

- caopen() Establish connection to hardware
- ctstat() Test status

C interface

```
#include "ca.h"
void cdreg(int *ext,int b,int c,int n,int a);
void cfsa(int f,int ext,int *data,int *q);
void cssa(int f,int ext,short *data,int *q);
void cccz(int ext);
void cccc(int ext);
void ccci(int ext,int logic);
void ctci(int ext,int *logic);
void cccd(int ext,int logic);
void ctcd(int ext,int *logic);
void ctgl(int ext,int *logic);
void cdlam(int *lam,int b,int c,int n,int a,int *inta);
void cclm(int lam,int logic);
void ctlm(int lam,int *logic);

void cfga(int fa[],int exta[],int intc[],int qa[],int cb[]);
void cfmad(int f,int extb[],int intc[],int cb[]);
void cfubc(int f,int ext,int intc[],int cb[]);
void cfubl(int f,int ext,int intc[],int cb[]);
void cfubr(int f,int ext,int intc[],int cb[]);
void csga(int fa[],int exta[],int intc[],int qa[],int cb[]);
void csmad(int f,int extb[],int intt[],int cb[]);
void csubc(int f,int ext,int intt[],int cb[]);
void csubl(int f,int ext,int intt[],int cb[]);
void csubr(int f,int ext,int intt[],int cb[]);

void caopen(char *server,int *success);
void ctstat(int *istat);
```

Fortran interface

```
integer*4 ext,b,c,n,a,f,data,q
integer*4 logic,lam,inta[2],success,istat
character*20 server       ! length unimportant
...
call cdreg(ext,b,c,n,a)
call cfsa(f,ext,data,q)
call cccz(ext)
call cccc(ext)
```

```
call ccci(ext,logic)
call ctci(ext,logic)
call cccd(ext,logic)
call ctcd(ext,logic)
call ctgl(ext,logic)
call cdlam(lam,b,c,n,a,inta)
call cclm(lam,logic)
call cclc(lam)
call ctlm(lam,logic)
call caopen(server,success)
call ctstat(istat)
```

Arguments

server = (character) IP name or IP number of the server machine
success = (returned) 0 if failure, 1 if success
ext = external representation of an address given by b,c,n,a
b = branch number (must be 0)
c = crate number
n = slot number
a = address in slot
inta = additional lam definition information (ignored)
f = function code
data = data to read or write
q = status of q line at the end of specific operation
istat= state of the q and x lines at the end of this operation
- 0 = normal
- 1 = no Q
- 2 = no X
- 3 = no Q & no X

logic = logical state
- 0 = reset
- 1 = set

fa = array of function codes
exta = array of external addresses
intc = array of integers
qa = array of q responses
cb = control block; cb[0] = requested count; cb[1] returns actual count
extb = array of external address; extb[0] = starting; extb[1] ignored
intt = array of truncated (16 bit) data words

Description

The CAMAC standard routines are described in IEEE Std 758-1979 appendix D: *IEEE Standard Subroutines for CAMAC.* Most operations involve defining an external address to operate upon followed by operating on that address. For example, to read register 0 of slot 3 of crate 1 branch 0, it is necessary to call cdreg to obtain the external address of the b-c-n-a quadruplet, and then call cfsa with that address, specifying the function code (typically 0). Once an address has been defined, it can be used multiple times.

Example 1

```
integer*4 ext,data,q,status
call caopen("myserver",status)
call cdreg(ext,0,1,3,0)     ! point to register
call cfsa(0,ext,data,q)
if (q.ne.0) type *, "bad q at cnaf=1,3,0,0:",q
call cfsa(16,ext,1234,q)    ! write 1234 to register
call cfsa(0,ext,data,q)
if (data.ne.1234) then
  type *, "bad compare: wrote 1234, read",data
endif
```

Example 2

```
! (CAMAC readout list example using crate scan)
int ext, cb[2];
cdreg(&ext,0,1,12,0);     /* start scan at slot 12 */
cb[0] = 100;              /* limit to 100 words */
cfmad(0,ext,DATAPTR,cb);  /* output goes to CODA buffer */
DATAPTR += cb[1];         /* bump CODA output pointer */
!
! OR use 16 bit transfers:
csmad(0,ext,DATAPTR16,cb);
DATAPTR16 += cb[1];
```

FASTBUS I/O Library

An partial implementation of the FASTBUS standard routines (IEEE-1177 1989) has been provided for the FSCC and the VxWorks kernel. These routines may be called in a readout list, in a seperate loadable user routine, or directly from the VxWorks shell on the FSCC. They reside in the fastbus library ($CODA/VXWORKS68Kxx/lib/libfb.o) which must be downloaded to the FSCC (this can be done in a VxWorks startup script).

Standard Functions

• fparb()	arbitrate for mastership
• fpad()	address module in data space
• fpac()	address module in control space
• fpsaw()	secondary address write
• fpsar()	secondary address read
• fpr()	single word read
• fpw()	single word write
• fparel()	address release
• fprel()	address and bus release

Standard Block Functions

• fprb()	block read
• fpwb()	block write

Additional Routines

• fb_init_1()	FSCC Initialization

C interface

```
#include "fb_fscc_macro.h"
#include "fb_types.h"
#include "fb_status_macros.h"
#define IFERROR(str,pa,sa)
  if (fb_errno!=FB_ERR_NORMAL)
    printf(FB_ERRTXT[FB_GET_ERROR_VALUE(fb_errno)],
    fb_errno,str,NULL,pa,sa);

void fb_init_1();
int fparb();
int fpad(int pa);
int fpac(int pa);
int fpsaw(int sa);
int fpsar();
int fpr();
int fpw(int data);
int fprb();
int fpwb(int *data,int len);
int fparel();
```

```
int fprel();
```

Arguments

pa= primary address (slot number or logical address)

sa= secondary address (for either CSR or DATA registers)

data = value or pointer to an array of values

len = number of data values to write

fb_errno = contains fastbus error number after each routine call

Description

These routine are primarily ment for convienience in accessing the FASTBUS port on the FSCC. They are particularly useful for identifing modules with problems in the crate or for debugging a “hung” DAQ system. In the example below from the the VxWorks shell the FSCC is initalized and a module in slot 16 is addressed (in Control space), cleared (write of 0x40000000), and its module ID (ID = 0x104f0000) read out. Finally, the address lock and bus are released (fprel() returns fb_errno = 0x800bc119 which is normal completion of a fastbus operation).

Example

```
->
-> fb_init_1
value = 3140068 = 0x2fe9e4
-> fparb
value = 0 = 0x0
-> fpac(16)
value = 0 = 0x0
-> fpw(0x40000000)
value = 0 = 0x0
-> fpr
value = 273612800 = 0x104f0000
-> fprel
value = -2146713319 = 0x800bc119
->
```

Console Logger

The following 2 routines allow messages to be sent to the console log process, which displays them in an X-window display as well as copying them to a log file.

Routines

```
daLogOpen()              Open a connection to the logger
daLogMsg()               Write a string to the logger
```

C Interface

```
void daLogOpen(char *host,char *pname,int pnum);
void daLogMsg(char *format,arg1,arg2,...);
```

Fortran Interface

```
character*20 host, pname, line
integer pnum
call dalogopen(host,pname,pnum)
call dalogmsg(line)
```

Arguments

host = name of the machine on which the console task is running

pname = label to prepend to all messages from this process

Description

daLogOpen establishes a remote procedure call link to the console log task running on the machine specified by *host*. The arguments *pname* and *pnum* (normally the component name and number) are saved and prepended to any messages later sent by *daLogMsg*.

The C interface to daLogMsg behaves like the printf routine, and in fact the arguments are passed directly to sprintf to create the string to be sent to the logger.

The Fortran interface does not handle variable argument lists, and so only a single character variable may be passed (use internal writes to format a line prior to calling *daLogMsg*).

daLogOpen is called automatically by the *rcService* routine, so analysis programs using this routine may call *daLogMsg* without first calling *daLogOpen*. In this case, *pname* is set to the name of the analysis component, as found in the *rcNetwork* file.

Example

```
...
status = myfunction(arg1,arg2)
if (status.ne.1) call daLogMsg("error in my function")
...
```

Error Message Library

Many of the packages in CODA return a status value which indicates success or failure, and if not successful, it indicates what the problem was. C header files exist for testing for specific error returns from each package or facility. Generally, though, it is sufficient to note that an operation failed, and to report the error to the user. The routines in this library can either print the error message corresponding to the error code, or return the character string to the user for further formatting.

Functions

- ceMsg() Retrieve the CODA error message
- cePmsg() Print the CODA error message

C interface

```
#include <cemsg.h>        /* prototypes & defines */
char *ceMsg(int code,int flag,char *string,int strlen);
void cePmsg(char *prefix,int code);
```

Fortran interface

```
integer*4 code, flag
character*80 string
character*20 prefix
...
call cemsg(code,flag,string)
call cepmsg(prefix,code)
```

Arguments

code = integer status returned by CODA routine
flag = option specifying which part of the error message to extract
CEMSG_ALL = 0 = all
CEMSG_NAME = 1 = error code name
CEMSG_MSG = 2 = message string
CEMSG_SEV = 3 = severity
CEMSG_FAC = 4 = facility name generating the error
CEMSG_CODE = 5 = error code minus severity and facility
string = character variable to hold requested text
prefix = character string to print before the error text, if any

Description

Error codes are 32 bit numbers in which the high 2 bits indicate severity as follows:

0 = informational
1 = warning
2 = error
3 = fatal

In this way, negative numbers indicate errors since the most significant bit is set. (The normal return for all routines is zero.) The next 14 most significant bits are used to encode a facility number. The low 16 bits encode an error number for that facility:

30	29 16	15 0
sev	facility	code

Status message names are of the form S_FAC_BRIEFERRMSG, where FAC is an abbreviation for the facility to which the status message belongs, and BRIEFERRMSG is a short name of the error message. The codes S_SUCCESS = 0 and S_FAILURE = -1 are shared by all facilities.

When using these routines, an object module containing a table of all known error codes and the corresponding status message names and strings is automatically linked into the application (if new error codes are defined, the application will need to be re-linked to recognize them).

The routine *ceMsg*() may be used to extract information from this table. If only the severity is requested, one character strings are returned (I, W, E, or F). If all is requested, a string of the following form is returned:

```
Error: S_SC_BADVALTYP Bad value type specified
```

(If the high bits were 01, then the message would start ‘Warning:’, etc.).

The routine *cePmsg* prints an error message in the same format as above, prefixed by the string specified as the first argument in the call.

C example

The following examples makes use of the fact that the ceMsg routines returns a pointer to the requested error message string:

```
#include <cemsg.h>
int status;
...
status = scWriteInt(name,val);
if (status!=S_SUCCESS)
  printf(“failed to write %s: %s\n”,name,
         ceMsg(status,CEMSG_MSG,NULL,0));
...
if (status) cePmsg(“error in xyz”,status);
...
```

Fortran example

The following example prints an error message for each call that returns an error or fatal error, but not for warnings:

```
integer*4 status,scwriteint
...
status = scwriteint(name,1234)
if (status<0) call cepmsg(“error in xyz”,status)
```

Event I/O Library

Events in the CEBAF common event format can be read from and written to a file on disk or tape, or from the Event Builder process, or to the Event Recorder process. File I/O is performed using the C run time library routines *open*, *read*, *write*, and *close*.

NOTE: CODA 1.4 allows only file I/O.

All logical records are assumed to be in the CEBAF bank format (see Appendix E). In this format, the first longword of each event is a longword count of the number of words to follow. The routines below do not attempt to validate a record either on input or output except for reads from a file, which will verify that each logical record's starting point is consistent with information in the block header.

Functions

- evOpen() open a file or live event stream
- evRead() read a logical record (event)
- evWrite() write a logical record (event)
- evClose() close a file or stream
- evIoctl() control file or stream

C Interface

```
#include "evfile_msg.h"
evOpen(char *filename,char *flags,int *handle);
evRead(int handle,int *buffer,int buflen);
evWrite(int handle,int *buffer);
evClose(int handle);
evIoctl(int handle,char *request,void *argp)
```

Fortran Interface

```
integer*4 status
integer*4 evOpen,evRead,evWrite,evClose,evIoctl
character*32 filename
character*1 flags, request
integer*4 handle, buffer(8192), buflen, argp
data buflen /8192/
status = evOpen(filename,flags,handle)
status = evRead(handle,buffer,buflen)
status = evWrite(handle,buffer)
status = evClose(handle)
status = evIoctl(handle,request,argp)
```

Arguments

filename = name of file to open

flags = 1 character indicating read (R) or write (W). Opening an existing file for write first truncates the file. Opening a non-existent file for read results in an error.

handle = variable to hold a pointer to the open file

buffer = buffer into which an event is read (evRead) or from which an event is written(evWrite). The first word of the buffer will/should contain the event length in longwords excluding the length word. On a read if the event is longer than the buffer length, only buflen words are written to the buffer, and an error is returned.

buflen = length of caller's buffer in longwords

request = I/O request option:

"B" or "b": change the blocksize of physical records

argp = value for I/O request

Description

EvOpen is implemented as a call to the C *fopen* routine. By default it creates a new file with a blocksize of 8K longwords (this can and should be extended if large event files will be generated at high data rates). Each routine returns success or error as the function value. In some cases, errors are returned from the underlying I/O system (i.e.from errno). The error message as a string can be obtained via a call to ceMsg (see Error Message Library on page 4-9).

Each call to evRead copies the next event from the file referenced by the handle argument into the caller's buffer. The first element of the array will receive the event length; if this is less than *buflen* then array contains the entire event, otherwise as much of the event as will fit is copied into *buffer* and a warning is returned.

Each call to evWrite copies an event from *buffer* to the file referenced by *handle*, automatically buffering the events into physical records, with events allowed to span records.

When evClose is called, any partial physical record is flushed to output, and the file is closed.

The default blocksize can be changed by a call to evIoctl. The call must be made immediately after opening a new file for write access (i.e. in the online analyzer the call should be made in the users "prestart" routine). In Fortran, the call would be

```
status = evioctl(handle,"b",16384)
```

Example

The following C code fragment copies the first 1000 events from one file into a second file, and forces the new file to have a blocksize of 1024 longwords.

```
int blocksize = 1024;
...
if (evOpen("input","r",&handle)==S_SUCCESS) {
  if (evOpen("output","w",&handle2)==S_SUCCESS) {
    evIoctl(handle2,"b",&blocksize);
    for (i=0;i<1000;i++) {
```

```
        if(evRead(handle,buffer,buflen)!=S_SUCCESS) break;
        if(evWrite(handle2,buffer)!=S_SUCCESS) break;
      }
      evClose(handle2);
    }
    evClose(handle);
  }
```

Using variations on this code, simple utilities for extracting events of a particular type or matching particular criteria may be written.

Histogramming

The CERNLIB HBOOK package supports 1 and 2 dimensional histograms, as well as N-tuples (short arrays of length N treated as N-dimensional scatter plots). This section documents the most useful HBOOK routines for convenience. For more details and descriptions of other routines, see the HBOOK User's Guide.

Subroutines

- HLIMIT(nwords)
- HLIMAP(nwords,name)
- HBOOK1(id,chtitl,nx,xmi,xma,vmx)
- HBOOK2(id,chtitl,nx,xmi,xma,ny,ymi,yma,vmx)
- HBOOKN(id,chtitl,ndim,chrzpa,nprime,tags)
- HFILL(id,x,y,weight)
- HFN(id,array)
- HRESET(id,chtitl)
- HRFILE(lun,chtop,chopt)
- HROUT(id,icycle,chopt)
- HREND(chtop)

Arguments

The above routines are written in Fortran, and must follow fortran calling conventions. (To call from c, append an underscore to the end of the name, and for each character variable append an additional argument which is the length of the string, by value.)

nwords = number of words in COMMON /PAWM/ array(nwords)

name = name of the shared memory in which to place the histograms

id = histogram number

chtitl = histogram title

nx = number of channels in 1st dimension

xmi = minimum x value to histogram

xma = maximum x value to histogram

vmx = upper limit on single channel content (dictates underlying data type)

ny = number of channels in 2nd dimension of 2d histogram

ymi = minimum y value to histogram

yma = maximum y value to histogram

ndim = number of dimensions in N-tuple

chrzpa = memory / disk allocation control: if blank ' ', then arrays of length nprime will be allocated and filled, continuing until the HBOOK common area is exhausted; if chrzpa contains the name of a directory as given by argument chtop to HRFILE, then a single array of length nprime will be allocated, and copied to the RZ file each time it is full

nprime = size of primary allocation for an N-tuple

tags = array of character strings giving the names of each element of the N-tuple

x = x value to histogram

y = y value to histogram for 2d histogram (ignored for 1d)

weight = value to add to channel contents referenced by x,y

array = array containing one N-tuple to append to the set of N-tuples

lun = logical unit number of an RZ direct access file (must be opened external to the HBOOK routines)

chtop = character string name of the RZ directory to use for I/O

chopt = character string containing options; for HRFILE, the default (‘ ‘) is to open an existing disk file, ‘N’ creates a new file, and ‘U’ updates an existing file; for HROUT, must be ‘ ‘

icycle = (returned) version number of histogram on disk

Description

The routine HLIMIT is used to set the number of words in the Fortran common block /PAWC/ that may be used by HBOOK for working storage. The argument to this routine must not exceed the declared value.

The routine HLIMAP replaces HLIMIT, and places the histograms into a shared memory whose name (4 characters) is given by the second argument. PAW can access this shared memory by the command “global <name>”.

HBOOK1, HBOOK2, and HBOOKN are used to create new histograms/n-tuples which are later referenced by their number.

HFILL and HFN are used to insert data into the referenced histogram.

HRESET zeros the channel counts (or n-tuple count), and optionally changes the title.

HRFILE establishes a temporary unique correspondence between the logical unit LUN of a previously opened direct access file and the RZ top directory name contained in CHTOP. If several direct access files are opened by HRFILE, they are identified by the top directory only. After the call, the current RZ directory is set to the name in CHTOP.

HROUT writes one or more histograms to the current RZ directory on the direct access file. Using id=0 writes all histograms from the current directory in memory to the current directory on disk.

HREND removes the association between the RZ directory name and the direct access file. The file remains open.

Using the direct access routines above requires an open statement of the form

```
open(unit=lun,file='myfile.dat',form='UNFORMATTED',
     access='DIRECT',status='UNKNOWN',recl=XXX)
```

where XXX is 1024 for VAX/VMS and 4096 for most other machines.

Integration with PAW

In order to view “live” histograms with the PAW utility from CERN, initialize the HBOOK package with a call to HLIMAP, followed by calls to HBOOK1 or 2, followed by calls to HFILL:

```
parameter nwpaw=100000
common /pawc/ipacw(nwpaw)
```

```
...
call hlimap(nwpaw,'CLAS')
call hbook1(1,'spec1',100,-3.,3.,0.)
call hbook1(1,'spec2',100,-3.,-3.,0.)
...
call hfill(1,x1,0.,1.)
call hfill(2,x2,0.,1.)
...
```

Then, inside PAW, connect to the shared memory using the same name as in the call to hlimap (arbitrary 4 character name):

```
PAW > global clas
PAW > cd //clas
PAW > hi/pl 1
```

Run Control Communications

The following routines facilitate building a program to serve as a component in a CODA data acquisition system. They may be used to build an analysis program, a user trigger component, or a general user component.

Functions

`rcService()`	Start specified service
`rcExecute()`	Execute the service(s)

C Interface

```
#include “services.h”
void rcService(void *service_name);
void rcExecute();
```

C Callbacks

```
int usrDownload_(char *config);
int usrPrestart_(int *run_num,int *run_type);
int usrGo_();
int usrPause_();
int usrEnd_();
int usrDump_();
int usrEvent_(int *buffer,int *buflen,int *flag);
```

Fortran Interface

```
external service_name
call rcService(service_name)
call rcExecute()
```

Fortran Callbacks

```
integer function usrDownload(config)
  character *(*) config

integer function usrPrestart(run_num,run_type)
  integer*4 run_num,run_type

integer function usrGo()
integer function usrPause()
integer function usrEnd()
integer function usrDump()

integer function usrEvent(buffer,bufmax,flag)
  integer*4 buffer(buflen), buflen
  integer*4 flag
```

Arguments

config = variable containing the configuration string (minus output filename)

run_num = run number of this run

run_type = run type number, from rcRunTypes

buffer = buffer containing 1 event, in CODA event format (first word is exclusive length of the event)

bufmax = declared / allocated length of buffer. Generally greater than the length of the event, and gives the maximum size of an output event for analysis programs which expand the event.

flag = flag to indicate whether to write the event to output or not. Set to 0 to reject (filter) the event. Set to 1 to cause event to be output. Value is 1 on entry, so that events by default are recorded if an output file is opened.

Description

The routine *rcService* declares a remote procedure call service to the network; i.e., it announces itself to the Run Control program or any other CODA utility as being an event builder and/or an analysis program. The allowed service names are:

```
RC_SERVICE_EB
RC_SERVICE_ANA
```

After declaring the desired service(s) (typically both), the program calls rcExecute to begin processing commands and events. If rcExecute is successful, it never returns; it waits for messages from Run Control or other utilities.

For each change of state received from Run Control, a user change of state callback routine (e.g. usrGo()) is called by rcExecute, passing along any relevant information received from Run Control. If the user provides these routines they are called; if they are not provided, null routines are linked in from the library.

If the requested service is RC_SERVICE_ANA, then when the download command is received, rcExecute opens a link to the Event Builder (if the service RC_SERVICE_EB was also started, this link is just internal pointers).

Also at download, Run Control passes the configuration string (everything after the object name) from the users .config file to rcExecute. The first word of this string (up to the first white space) is interpreted by rcExecute as an output filename. If this word is uppercase NOLOG, then no output file is used. Any characters after the first word are passed to the users usrDownload routine as the *config* argument.

Following any go command, events are read from the Event Builder and the usrEvent routine is called for each event received which is not a change of state event. If an output file is open, all change of state events and any event for which usrEvent returns flag not equal to zero is written to the output file (flag is set to 1 prior to the call). The analysis program is free to edit the event, subject to the constraint that the event not grow in size beyond *bufmax* longwords.

rcExecute also calls *daLogOpen* when RunControl first establishes a connection to the service. The program name and number used in this call are the component name and number sent by RunControl. If the user desires to log messages from his application to a separate console log, *daLogOpen* may be called prior to calling *rcExecute*.

The usrDump callback is invoked by the CODA utility $CODA/bin/cdumphist, which was originally used to tell an analysis program to dump its histograms to a file so that they could be read by PAW. This feature is no longer needed when using shared memory histograms, and the usrDump routine may be removed in a future release.

Example

The following is an excerpt from the program $CODA/examples/ebana.f, which incorporates the event builder service into the analysis program:

ULTRIX:

```
...
external rc_service_ana
external rc_service_eb
...
call rcService(rc_service_eb)
call rcService(rc_service_ana)
call rcExecute()
...
```

HP-UX:

```
...
common /rc_service_ana/rc_service_ana
common /rc_service_eb/rc_service_eb
...
call rcService(rc_service_eb)
call rcService(rc_service_ana)
call rcExecute()
...
```

Spying: Data Acquisition RPC Library

Many of the distributed components in CODA have RPC interfaces to control and monitor their operation, and to insert and retrieve events or event fragments. The following routines are available for building custom applications to interface to CODA.

Functions

• daCopyEvent()	Copy 1 event or fragment from the named component
• daCopyRegister()	Register a request for multiple events (connect)
• daCopyNext()	Copy the next event
• daCopyPoll()	Copy the next event if one is available
• daCopyUnregister()	Terminate the connection with the component
• daInsertEvent()	Insert 1 event into the event stream
• daReadInt()	Read the value of a named variable

C Interface

```
int daCopyEvent(char *component,int type,int array[],int arlen)
int daCopyRegister(char *component,int type);
int daCopyNext(int array[],int arlen);
int daCopyPoll(int array[],int arlen);
int daCopyUnregister(char *component,int type);
int daInsertEvent(char *component,int event[]);
int daReadInt(char *component,char *item,int *value);
```

Fortran Interface

Arguments

component = null terminated ASCII name of a component found in the rcNetwork file
type = type of event desired, -1 returns any type
array = address of array to receive one event
arlen = maximum length of array in longwords
event = address of an array containing one event in CEBAF event format
item = null terminated ASCII name of the item to be read or modified; see list below (case insensitive)
value = address of integer to hold the item data

Description

The above routines locate the desired component on the network using information contained in the rcNetwork file in the directory pointed to by the environment variable RCDATABASE, or in the current directory if RCDATABASE is not defined. If the rcNetwork file is not located, only the special components “EventBuilder” and “EventRecorder” can be accessed, and only if they are running on the current host.

Each call to daReadInt results in a remote procedure call to the selected component, passing the item name as the argument. The returned data is copied into value; status is returned as an integer function value. See the table Supported Items on page 4-21 for a list of currently supported item names for each type of component.

Each call to daCopyEvent reads a copy of the next event of the specified type via RPC into the specified array. Up to arlen longwords are copied; if the event is longer, the remainder is discarded and an error is returned. The first word of the array will contain the event length (may be greater than arlen).

Each call to daInsertEvent inserts the data from the array ‘event’ into the data stream. The data is checked to verify that it is in CEBAF event format. The first word of the array must contain the event length (i.e. outermost structure must be a bank).

TABLE 4 Supported Items

Component Class	Item Name	Description
ROC	nevents	Number of events seen by this ROC
	nlong	Number of longwords of data sent by this ROC (includes outermost bank length words)
EB	nevents	Number of events seen
	nlong	Number of longwords of data sent to the analysis program
	nkbytes	Number of Kilobytes sent to analyzer rounded down to an integer value
	remkbytes	Number of additional bytes over the integer value nkbytes
	rocmask	Mask of ROC numbers included in this run (ROC numbers are from 0-31)
ER	nevents	Number of events written to output
	nlong	Number of longwords written to output
	nblocks	Number of physical records written to output
	blocksize	Blocksize of physical records

CHAPTER 5 Utilities

5

CODA contains a number of useful utilities for controlling and monitoring experimental apparatus, and for examining event data stored in files. The following utilities are documented in this chapter:

- ccrl — CODA readout list interpreter to convert crl to c code
- cdumphist — Utility for invoking usrDump routine in analysis program
- cefdmp — Utility to dump events from files or from live processes
- cnaf — CAMAC I/O utility
- coda_activate — Script for starting Run Control network components
- codaf77 — Script for compiling and linking fortran applications
- facmsg — Print a coda facility error message
- makelist — Script for compiling readout lists (calls ccrl
- vxmon — VxWorks remote system monitor

ccrl

The *ccrl* utility converts CODA readout list language files (.crl files) into c code which can then be passed to the GNU 68K cross compilers

Syntax

```
% ccrl filename.crl [outfile]
```

Arguments

filename = filename containing readout list code to be converted
outfile = optional output file name (default is filename.c)

Description

ccrl uses lex and yacc to parse the input file and identify keywords for conversion into c code. It generates c code specifically for downloading into coda_roc running on a VxWorks host. It does not generate stand alone code. The user can, however, imbed his own c code into his readout list in addition to using the defined readout list language. For more information on CRL language see Appendix C.

ccrl is used by the CODA utility *makelist* in generating the downloadable VxWorks objects used by the ROC.

Example

To take the readout list physics.crl and convert it to c storing it in a temporary file for viewing (i.e. temp.c).

```
% ccrl physics.crl temp.c
```

cdumphist

The *cdumphist* utility is a simple program to cause an analysis program to execute its usrDump routine.

Syntax

```
% cdumphist target
```

Arguments

target = hostname of the machine running the analysis program

Description

cdumphist makes an RPC call to the daDump routine of the analysis program running on the specified host. If this program has called rcExecute, the call is passed to that program's usrDump routine.

This utility was written as a simple way to signal an analysis program to write all its histograms to disk for viewing by a display program. It is no longer necessary for that function since Hbook histograms may be placed in shared memory and viewed live (Ultrix Only).

cefdmp/xcefdmp

The event dump utility (cefdmp) can display, in a readable ASCII format, events from the following sources:

- event files
- readout controllers (fragments)
- event builder output
- analysis program output

Syntax

```
% cefdmp [filename] [-o object] [-t tag] [-u uniquetag]
          [-s start] [-e end] [-d dictionary] [-x]
% xcefdmp [same]
```

Options

o	specify the source object name (CODA component name)
t	select specified tag number or name (full path in event structure)
u	select specified unique tag, independent of location in the event
s	number of first record in file to dump
e	number of last record in file to dump
d	specify dictionary for obtaining tag names and titles
x	hex dump of integers (default is decimal)

Description

The cefdmp utility can dump selected portions of particular events from either a file on disk or from special processes which comprise the data acquisition pipeline.

For file dumps, the filename is specified as an argument to cefdmp. One event is dumped, and then the utility waits for a carriage return before dumping the next event. If *-s nnn* is specified, (*nnn*-1) records are skipped before the dump starts. If *-e mmm* is specified, the dump continues without prompting until record *mmm*. Output is to standard output, and so in this case can be piped to other programs or re-directed to disk:

```
% cefdmp myfile -e 20 > dump20.lis
```

Events are tree structured, and every node and leaf on the tree contains an identifier or tag. Any node or leaf may be specified either by a path (set of tags) starting at the root of the tree, or by a tag which only occurs at a single point on the tree. The -u option is used to specify a unique tag. (Using unique tags when defining event structures makes it easier to use this utility, but is not required.)

Tags are stored in the event as integers, but may be referenced by names stored in a tag dictionary (see Appendix E). The name of this dictionary file may be specified with the -d option, or defaulted to the value of the environment variable EVTAGS, or the file *evTags* in the directory pointed to by the environment variable RCDATABASE. The following command dumps the drift chamber portion of physics events (events whose outermost identifier is not *physics* will be skipped):

```
% cefdmp myfile -t physics.drift
```

Events may alternatively read from processes in the data acquisition pipeline. Currently, the readout controller (ROC), event builder (EB), and the analysis program (ANA) all will support this feature, allowing spying on the data stream at the crate level, prior to analysis, and after analysis. The data source is selected using the *-o* option followed by the name of the object as found in the run control database file. Assuming that the environment variable RCDATABASE is defined, the following example extracts physics events after analysis:

```
%cefdmp -o myana -t physics
```

cemsg

Error codes from the various facilities within CODA have associated error message strings which may be printed interactively with the *cemsg* command.

Syntax

```
% cemsg errornumber
```

Description

The program takes its first argument as a decimal (hex if 0x is prepended) status code, looks it up in a compiled table, and prints the full error message to stdout. If the code is not found, it prints a message that the code is unknown. (This would happen if the cemsg utility were out of date, for example).

The output string is in three parts: (1) severity, (2) error name, and (3) error text. The severity is one of the strings Info, Warning, Error, or Fatal Error. The error name starts with a leading S_ followed by the short name of the facility causing the error message, followed by another underscore, followed by an abbreviation of the error. The exception to this are the two error names S_SUCCESS and S_FAILURE which do not indicate a facility name.

Example

```
% cemsg -2140209130
Error: S_DA_NOPARAM Parameter does not exist
```

This error was generated by the DA facility (Data Acquisition). Facility names are as follows:

CAMAC	CAMAC I/O package
DA	Data Acquisition / Run Control
EVFILE	Event file I/O
EVLIVE	Live event I/O (inter-process communications)

cnaf

The *cnaf* utility allows some of the features of the CAMAC I/O library to be used interactively. In particular, single functions to a CAMAC register may be performed.

Syntax

```
% cnaf target [c n a f [data]]
```

Arguments

target = hostname of the CAMAC server

c = crate number

n = slot number

a = address in slot

f = function code

data = data for write functions (16-23)

Description

If all arguments are specified, the operation is performed and the status is printed onto stdout. Q=0 is a normal successful operation, Q=1 means no Q response, Q=3 means no Q and no X response. In the following examples, user input is in italics:

```
% cnaf myvme 1 3 0 16 1234
q:0
% cnaf myvme 1 3 0 0
dec:1234    hex:4d2    q:0
```

If only the target name is specified, the server is contacted and *cnaf* reads the c, n, a, f, and data arguments repetitively from stdin, performing each operation and reporting return status on stdout. Input is terminated by end of input (control-D).

```
% cnaf myvme
c n a f [d]: 1 3 0 16 2222
q:0
c n a f [d]: 1 3 0 0
dec:1234    hex:4d2    q:0
c n a f [d]: ^D
%
```

coda_activate

In order to simplify starting processes which contain Run Control network components, a script has been provided which invokes *rsh* on the proper host, as specified in the rcNetwork file. This command is normally used only in the rcNetwork file.

Syntax

```
% coda_activate [options]
```

Options

-l[og] <file>	Log filename. Default is none, except for the console logger, where the default is coda_console.log.
-nol[og]	Force no log file.
-p[rogram] <p>	The program file to activate. This overrides the default which is derived automatically from the component type.
-f[ile] <file>	Synonym for -program.
-o[ption] <opt>	An option to be passed onto the program. Multiple such options may be specified, each being preceded by -option.

Description:

When used without options, coda_activate activates the default program corresponding to the component (e.g. ROC, EB) on the node specified in the rcNetwork entry. The options described above may be used to override the defaults.

Note that the location of any log files will depend on whether the corresponding component is activated on the same node as RunControl, or remotely on another node, *unless* the log filename is fully specified (e.g. /usr/users/me/ana.log). In the case where the component is activated on the same node as RunControl, the logfile will be located in the directory from which RunControl was activated. If the component is activated remotely, the logfile will be located in the home directory corresponding to the current account.

Example

The following is a sample rcNetwork file showing the use of this command:

```
! File:-
! rcNetwork: Network Configuration file.
!
! A Host of $NODE implies the same node as RunControl.
!-

!Name Num Type Host BootScript
!---- --- ---- ---- ----------

ROC1 1 ROC dadev -l roc.log
MYEB 0 EB myhost $CODA/bin/coda_activate -p ~/test/myebana
MYANA 0 ANA myhost $CODA/bin/coda_activate -p ~/test/myebana
```

codaf77

In order to simplify compiling and linking fortran applications with the CODA libraries, a script has been provided which executes the compiler and links to all CODA libraries and to the most recent CERN program libraries.

Syntax

```
% codaf77 [f77_options] [loader_options] file(s)
```

Any options may be specified which are valid for the compiler and linker, including additional object modules, fortran files, and library files.

Description

This script simply invokes f77 on the host, passing all arguments through to the compiler, and appending on the CODA libraries and the CERN libraries (packlib and kernlib). It may be copied from $CODA/bin and customized as necessary.

Example

The following example compiles a program *drift.F* which references files included from the sub-directory *include*, and links to the library *drift.a* and the X11 system library:

```
% codaf77 -I./include -lX11 drift.F drift.a
```

makelist

makelist is a utility for compiling CODA readout language.

Syntax

```
% makelist file[.crl] version
```

The name of the file to be compiled must end with “.crl”.

Description

This script first passes the source through an interpreter (called *ccrl*) which converts the readout list into c. Next, the GNU compiler is invoked to compile the c readout list. Currently, the macro expanded c source is left behind in the file “file.c”.

The version argument specifies the VxWorks version number the file is to be compiled under. The two currently supported versions are 5.0 and 5.1.

Example

The following example compiles the readout list fastbus.crl to produce the 2 files fastbus.c and fastbus.o:

```
% makelist fastbus 5.0
```

vxmon

The vxmon utility gives access to several of the system monitoring routines of the VxWorks kernel (operating system used in the VME and FASTBUS systems).

Syntax

```
% vxmon
```

(Select a node and a display type from pull down menus.)

Description

This X-windows based utility can display information about all tasks on any of the front end systems in CODA, both readout controllers and slow controls servers. When vxmon first starts up it reads the file rcNetwork in the directory pointed to by the environment variable RCDATABASE to obtain a list of all nodes in the experiment. This list is used to build a pull down list of nodes (see below).

The menu bar at the top of the window contains 4 buttons: Control, Display, Node, and Application.

The Control button brings up 2 menu choices: Set Interval and Quit. The Set Interval menu item brings up a slider to vary the update rate from 0.5 seconds to 10 seconds.

The Display button brings up 4 menu choices: Status, CPU Load, Stack, and None.

- The Status display shows, among other things, the priority and current status (READY, PEND, SUSPEND) of each task. Any task in the SUSPEND state is probably hung -- please submit a problem report.
- The CPU Load display shows what percent of the CPU time is being consumed by each task, with one column giving figures since the monitor started and the other giving figures from the last second of operation. The last few lines of this display also show time spent in the kernel and executing interrupt code (both data acquisition and ethernet I/O).
- The Stack display shows the stack size of each task, the largest stack the task has ever used, and the amount of the stack never used (Margin). If a task is ever observed to have a margin of less than 100 bytes, please submit a problem report.

The Node button brings up a list of known nodes to select for display, as well as an option for entering a node name in a popup window.

The Applications button brings up 2 choices: Shell and Reboot. The Shell application creates a terminal window and attempts to log in to the current node (this may require entering a username and password). This option will not work on systems configured without a shell task and the rlogin daemon. The Reboot application forces a soft reboot of the currently selected node (after prompting for verification).

Status Display

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	_excTask	3fccd0	0	PEND	1cab8	3fcc3c	d0003	0
tLogTask	_logTask	3fb788	0	PEND	1cab8	3fb6f0	0	0
tNetTask	_netTask	3f7568	50	READY	4d79e	3f7500	0	0
tPortmapd	_portmapd	3f0e64	100	PEND	59ada	3f0d30	16	0
vxServ	_vxserv	3be560	100	PEND	59ada	3be464	36	0
scServ	_main	3df960	150	PEND	59ada	3df844	1c0001	0
caServ	_main	3c0dd0	150	PEND	59ada	3c0cb4	d0003	0

CPU Load Display

NAME	ENTRY	TID	PRI	total % (ticks)	delta % (ticks)
tExcTask	_excTask	3fccd0	0	0% (2)	0% (2)
tLogTask	_logTask	3fb788	0	0% (0)	0% (0)
tNetTask	_netTask	3f7568	50	0% (74838)	0% (408)
tPortmapd	_portmapd	3f0e64	100	0% (46)	0% (0)
vxServ	_vxserv	3be560	100	0% (15465)	1% (3100)
scServ	_main	3df960	150	0% (5)	0% (0)
caServ	_main	3c0dd0	150	0% (0)	0% (0)
KERNEL				0% (1310)	0% (60)
INTERRUPT				0% (1500)	0% (44)
IDLE				98% (8150509)	98% (192662)
TOTAL				98% (8243783)	99% (196326)

Stack Display

NAME	ENTRY	TID	SIZE	CUR	HIGH	MARGIN
tExcTask	_excTask	3fccd0	2988	148	652	2336
tLogTask	_logTask	3fb788	4988	152	220	4768
tNetTask	_netTask	3f7568	9528	104	736	8792
tPortmapd	_portmapd	3f0e64	4528	308	2616	1912
vxServ	_vxserv	3be560	19212	252	3764	15448
scServ	_main	3df960	9208	284	1500	7708
caServ	_main	3c0dd0	9208	284	1500	7708
INTERRUPT			8000	0	312	7688

APPENDIX A
A

Run Control Configuration File Formats

Run Control is configured through 8 files:

- rcNetwork -- definitions of each distributed component
- rcRunTypes -- definitions of names of types of runs
- <runType>.config -- configuration strings for components in this type of run
- <runType>.options -- Run Control options for this type of run
- rcRunNumber -- file containing the number of the last started run
- rcDefaults -- file containing default settings for Run Control (optional)
- rcExperiment -- file containing an Experiment name (optional)
- rcPriority -- file containing service names and associated priority numbers (optional)

A.1 rcNetwork

The network definition file contains 1 line per distributed component in the following format:

```
name number class IP [command]
```

where “name” is an arbitrary alphanumeric name, ”number” is a unique integer number identifying the named component, “class” is one of the 7 supported classes (see below), “IP” is an internet hostname or address (e.g. mynode.cebaf.gov or 129.57.99.99), and “command” is an optional csh command used to start the component if it is not currently active. An example of such a command is coda_activate, discussed in Chapter 5. Comment lines are those with a leading exclamation mark (!), and are ignored.

TABLE 5 Run Control Classes

Class Name	Title	Description
TS	Trigger Supervisor	TS control ROC (optional).
ROC	ReadOut Controller	Read event fragments from input boards and forward them to the event builder (one or more).
EB	Event Builder	Assemble fragments into events (one required).
ANA	Analysis Program	Analyze events (one required).
ER	Event Recorder	Write analyzed events to the output file (optional, at most one).
LOG	Console Logger	Accept console messages from other components and optionally display them in a

		window and/or write them to a log file (one required).
UC	User Component	Arbitrary user component (optional, more than one allowed).

If the EB and LOG components are not found in rcNetwork, they are automatically added to the system with a component name the same as the class name, and an IP address the same as Run Control's (for the LOG component) or the analysis program (for the EB component). If the EB is listed in rcNetwork, it must have the same IP address as the analysis program. A minimal rcNetwork file would look something like:

```
fastbus 1 ROC 129.57.99.99
crunchEB 1 EB localhost /usr/users/me/myana/a.out
crunchANA 1 ANA localhost /usr/users/me/myana/a.out
```

This generates a system with one front end crate named fastbus. An event builder named "crunchEB" on the local host and an analysis program named "crunchANA" are started by running the file a.out in the specified directory, and a console log running on the local host will also be started. This system is not using a trigger supervisor, nor does it include an event recording process (but may write output events from the analysis program to disk).

It is actually possible to define more than one EB, ANA, ER, or LOG, provided only one of each is referenced in any <runType>.config file. Multiple Services should be identified by their name and number

A.2 rcRunTypes

The run types file contains a list of run type names and run type numbers. For each run type, user's may specify a different set of configuration information. Each line of the file is either a comment line (leading "!") or defines a run type with the following format:

```
name number
```

where "name" is an alphanumeric string, and "number" is a decimal integer. The run type name is used to locate configuration information for runs of that type, and the number is propagated to all components during the download change of state. Example file:

```
physics 1
calibration 2
test 3
```

If the rcRunTypes file is missing, Run Control uses a run type of "default", number 0.

A.3 <runType>.config

For each entry in rcRunTypes, there is a corresponding ".config" file (e.g. physics.config). Each line in the file is either a comment line (!) or defines a component to be used for runs of this type. Format:

```
name configurationString
```

where "name" matches a name defined in rcNetwork, and "configurationString" is an ASCII string whose meaning is specific to each class (see table Configuration String

Usage on page A-3). This string may contain embedded spaces, and is taken to start at the first non-whitespace after the name and continue to the end of the line or to an exclamation mark. It is sent to the component as part of the “download” command.

TABLE 6 Configuration String Usage

Class Name	Configuration String Usage
UC, ANA	User defined
TS, ROC	Name of file containing user’s compiled readout code
EB	Ignored
ER	name of event output file[1]
LOG	name of console log file, if desired

A.4 <runType>.options

Each run type has a run options file with an postfix of “.options” (e.g. physics.options). This file contains values for various options in the Run Control program. Each non-comment line has the form:

```
name value
```

or in the case of scripts,

```
transitionName [priority] scriptName
```

The only non-script option currently supported is “runNumber”, with the following values and meanings:

increment => increment and save the run number in rcRunNumber (default)

noincrement => fetch the run number from rcRunNumber (file is not changed)

Example:

```
runNumber increment
```

For user script execution, the “transitionName” can take on any of the following: download, prestart, go, pause, and end. The script’s execution priority is optional and defaults to 29 if no value is given. The “scriptName” should include the path specifying its location.

Script Examples:

```
prestart $RCDATABASE/rcScripts/getHVstatus
go 10 $RCDATABASE/rcScripts/getruninfo
```

If the option file is missing, all options take their default values, and no script components are created.

1. This component is not implemented in CODA 1.4

A.5 rcRunNumber

This file contains the most recently used run number as a decimal ASCII integer. The run number is incremented at the start of a run if the option “runNumber” is set to “increment” (the default). If the rcRunNumber file is missing, it is created when the first run is started with a run number of 1. The run number can optionally be set to any value through the options menu in RunControl.

A.6 rcDefaults

The rcDefaults file has a format similar to the options file above, in which each line specifies the name of a variable and its initial value. The following table lists all options, their default values, and a brief description.

TABLE 7

Name	Default	Description
buttonFeedback	true	Determines whether the various buttons on the Run Control command panels show feedback describing their actions when the mouse cursor enters them. The default is true.
online	true	Determines whether Run Control is online. A value of false will result in state transitions being performed without any communication with the components described in the rcNetwork file. This is useful for diagnostics. The default is true.
rpcUpdate	true	Determines whether updating of variables from components in the rcNetwork file occurs. The default is true.
rpctimeout	3	Determines the time in seconds the Run Control rpc communication will wait for a reply. The rpc request is issued 3 times. The timeout refers to a single request.
verboseReporting	true	Determines how verbose the status messages in the scrolling status region are. The default is true (verbose).

A.7 rcExperiment

The rcExperiment file contains a single string identifying a name for the Experiment. This name will be displayed on the RunControl front panel. While the string may be arbitrarily long, if the user uses a very long string it may be cropped in the RunControl display.

A.8 rcPriority

The rcPriority file can optionally be used to either redefine standard class component execution priorities (NOT encouraged) or define priorities for user components (UC). The syntax of the file is as follows:

```
class priority
```

where “class” can take on the values LOG, ER, ANA, EB, ROC, TS, and UC. The default priorities for all the standard classes are,

```
LOG         27
ER          23
ANA         19
EB          15
ROC         11
TS          -27
```

The purpose of priorities in the starting and stopping of runs is to give RunControl a defined order of communication to all the components. RunControl will not issue transition requests to one class before all the “higher priority” classes have successfully completed their transitions. For example, in starting a run RunControl communicates with the larger number classes first like ANA then EB then ROC and finally TS. However, when ending the run RunControl reverses the order of communication: TS, ROC, EB, and then ANA. Hence the TS component is the last to become active but the first to deactivate.

The user may define priorities for USER_SCRIPT components via the <runType>.options file. Hence, if he wished to have a script execute during the go transition after all the classes have activated except TS, then he could define it to be executed with a priority level 10.

APPENDIX B

CODA 1.4 Support for the Trigger Supervisor

B

The Trigger Supervisor is configured by a compiled readout list whose name is specified by an entry in the <runType>.config file (see Appendix A). The *coda_ts* executable must be running on the VME CPU in the crate housing the trigger supervisor module (this just a specialized ROC). TS programming is done the prestart routine and it is made Live or Not Live in the go, pause and end transitions.

Most necessary programming of the Trigger Supervisor can be done through writing to 4 different registers: the csr, timer, trig, and roc. In addition, there are 4096 memory locations that can hold ROC codes (accessed by ts_memory[i]) corresponding to all possible permutations of the 12 trigger input states. For more details on TS programming the user is referred to the CEBAF Trigger Supervisor Users Guide.

Hardware necessary for use of the Trigger Supervisor are the custom built Trigger Interface Cards for both the FSCC as well as for VME, and a 34 pin (17 pair) cable connecting each interface card to the Trigger Supervisor (via daisychain). The user should be aware that each interface card has a settable ROC number (0-7) which is independent of the ROC number defined in the CODA rcNetwork file. In the Trigger Supervisor readout list below the ROC number referred to in the ts->roc register is the one set on the interface card.

For example, a CODA readout list file for a system with 2 ROC's connected on separate cables (branches), and a Level 1 trigger enabled, might look something like this:

```
! CODA Readout list for VME control of Trigger Supervisor
! David Abbott, CEBAF 1994

vme readout

TS_ADDR = hex f0ed0000
MEM_ADDR = hex f0ed4000
MASK_01 = hex 0000ffff

begin download
! Define Pointers to TS control registers and Memory
   tsmem = (long *)MEM_ADDR;
   ts = (struct vme_ts *)TS_ADDR;
   log inform "Download Executed\n"
end download

begin prestart
   variable jj, addr, mem_value

   log inform "Entering Prestart\n"
```

```
%%
   ts->csr = 0x8000; /*reset*/
   ts->csr = 0x0200; /*ROC Lock mode on all BRANCHES*/
   ts->trig = 0x1FFF;/*Enable Trig inputs in non-strobe mode*/

/* The user should set bits in this register coressponding the
the ROCS that are being communicated with on each trigger */
/* Enable ROC 3 on BRANCH 1 and ROC0 on BRANCH 3*/
   ts->roc = 0x00010004;

   ts->timer[1] = 0x05;/* Level 2 Timer */
   ts->timer[2] = 0x05;/* Level 3 Timer */
   ts->timer[3] = 375; /* FrontEnd Busy timer 40ns/count */
   ts->csr = 0x0080; /* Enable Timer */

/*  construct memory data  ---  in the following model, all
trigger patterns that form the memory address are assigned to
trigger class 1.  For those trigger patterns with a single hit,
the ROC code is set to be the trigger input number.  Otherwise,
the ROC code is set to 0xE.  All LEVEL 1 ACCEPT signals are
asserted for every pattern. */

   ts_memory[0] = 0;
/* assign data to all memory addresses */
   for(addr=1;addr<=4095;addr++)
         ts_memory[addr] = 0xFFE3;
/* fix ROC code for single hit patterns */
   jj = 0;
   for(addr=1;addr<=4095;addr=2*addr)
   {
      jj++;
      ts_memory[addr] = 0xFF03 + (0x10)*jj;
   }
/* load and test memory */
   for(addr=0;addr<=4095;addr++)
   {
      tsmem[addr] = ts_memory[addr];
      mem_value = tsmem[addr];
      if(ts_memory[addr] != (MASK_01 & mem_value))
      printf("***** TS memory error %x %x\n",
                ts_memory[addr],MASK_01&mem_value);
   }
%%
   log inform "Prestart Executed\n"
end prestart

begin end
! Disable Triggers
   ts->csr = 0x210000;
   log inform "End Executed\n"
```

```
end end

begin pause
! Disable Triggers
   ts->csr = 0x210000;
   log inform "Pause Executed\n"
end pause

begin go
   log inform "Entering Go\n"
! Enable Triggers
   ts->csr = 0x21;
end go

begin trigger
end trigger

begin done
end done

begin status
end status
```

APPENDIX C Readout Controller Configuration File (Language Summary)

C

This appendix gives a summary of the data acquisition statements used to build readout lists for the supported FASTBUS and VME/CAMAC ROC's (readout controllers), and the format of the file containing these lists. The file is compiled with the *makelist* utility, and the name of the compiled file is passed to the ROC by an entry in the <runType>.-config file (see Appendix A).

C.1 File Format

Each ROC configuration file is composed of 1 or more sections of code to be executed upon receipt of a corresponding event, either a hardware trigger or a change-of-state command from Run Control. In addition there may be a section of declarations and/or definitions at the top of the file, for example to define constants, global variables, and compiler options. Each section other than the declaration/definition section starts with a "begin section-name" and ends with "end section-name":

```
! comments start with exclamation points
fastbus              ! set compiler for fastbus readout
slots = 5            ! constant definition
variable i           ! global variable declaration
begin usercode       ! begin section for user specific routine
%%                   /*imbed section of code using double % */
 ...
%%                   /* c comments also allowed anywhere */
end usercode         ! end must have matching name
begin prestart       ! hardware initialization
 ...
usercode();          !single line c code requires semicolon
end prestart
```

C.2 Compiler Flags

The first non-comment lines of code select what type of readout hardware is being used, and whether the readout will be triggered by interrupt or by polling. If polling is to be used (interrupt is the default), the following must be the first non-comment line:

```
polling                     ! (this line is optional)
```

Next, there must be a line containing one of the key statements for FASTBUS, CAMAC, or VME to enable support for the corresponding hardware:

```
camac readout               ! uses camac standard routines
vme readout                 ! camac and vme may both be used
fastbus readout             ! FSCC fastbus routines included
```

If the fastbus readout option is chosen there are several additional flags which can be set enabling different options. These are,

```
inline fastbus          ! Inlines all FB routines for faster
                          execution
nocheck fastbus         ! Turns off error checking. Speeds up
                          execution, but should be used only
                          when user is confident that FB
                          readout is operating properly
ts control              ! Should be used when the FSCC is
                          being triggered by the Trigger
                          Supervisor
parallel link           ! Redirects data flow through the
                          FSCC output port into a VME memory
                          module. (requires correct hardware)
```

C.3 Code Sections

There are 3 types of code sections: readout (trigger) lists, state transition command lists, and user command lists.

The hardware trigger information is conveyed to the ROC's on the Trigger Supervisor ROC cable, and is a 4 bit code (values 0-15). This code is generated via memory lookup in the Readout Code MLU of the Trigger Supervisor, with the trigger inputs as an address. When the ROC receives the trigger from the TS a list named "trigger" will be executed and a local variable called "trig_type" will be set with the value of the trigger information from the TS (if ts control is not specified then the default value of trig_type=1). At the end of each trigger, an additional list is executed to re-enable interrupts, clear lams, etc. (interrupts should not be re-enabled in the trigger list). This list is labelled "done":

```
begin trigger
 ...                    ! read data
end trigger
begin done              ! (executed at end of event)
 ...                    ! re-enable lam or trigger
end done
```

In addition to specifying instructions for hardware triggers, the following state transitions may also have ROC instructions associated with them: *download*, *prestart*, *go*, *pause*, and *end* (with identical readout section names: download, etc.):

```
begin prestart
 ...                    ! initialize hardware
end prestart
```

Finally the user may define hist own list that may be called (or spawned as a task in VxWorks) in any of the transition lists. These routines are created as `void` so no values may be returned:

```
begin userCode
 ...                    ! user specific (can be called from
 ...                    ! other lists)
end userCode
```

List statements may either be English-like readout statements defined in the next section, or may be any valid c expressions (the file is first passed to a CODA readout language interpreter (ccrl), and then to the c pre-processor and compiler). Individual lines of c code must end with a semicolon. Large sections of c code may be imbedded by placing %% prior to and at the end of the code section.

The readout language is designed to be complete enough for most experimenters, and hides many of the board specific implementation details.

C.4 Language Elements

CODA Readout Language (crl) statements include flow control, arithmetic operations, and hardware I/O statements. Each of the statements recognized by the CODA pre-processor begins with a keyword, and may have additional keywords or expressions following.

In the statements that follow, optional elements are shown in [], and alternative choices are shown in [] separated by |.

Variables and Expressions

Four byte integers, with case sensitive names of up to 31 characters, may be declared either at the top of the file (global variable) or within a section (local variable).

```
variable xxx,yyy,zzz
```

Constants may be declared at the top of the file by giving a name followed by an equals sign followed by a value:

```
NSLOTS = 6
```

Expressions may be built up from variables and arithmetic operators: * / + - ( ). Logical expressions may use the conventional logical operators < > == != <= >= or may use English equivalents:

```
is less than
is greater than
is equal to
is not equal to
is less than or equal to
is greater than or equal to
```

Logical expressions may be combined using parentheses and the operators *and or*:

```
(xxx is greater than 7) and (yyy is 8)
```

Arithmetic Statements

Constructs exist for clearing, incrementing, and decrementing a variable, as well as assigning an expression to a variable:

```
clear xxx
increment xxx
decrement xxx
xxx = expression
```

Flow Control

There are 4 flow control constructs: *begin...end*, *while...end while, if...else if...else...end if*, and *select on...case...end select:*

```
begin section-name
 statement(s)
end section-name

while logical-expression
 statement(s)
 ...
 break                    ! alternative way to exit loop
 ...
end while

if logical-expression
 statement(s)
else if logical-expression
 statement(s)
else
 statement(s)
end if
```

The *else* expressions are optional; there may be as many *else if* clauses as desired.

```
select on expression
case constant1
 statement(s)
case constant2
 statement(s)
...
default
 statement(s)
end select
```

No explicit *break* statement is required in a case clause: flow does not fall from one case into another, but rather terminates at the next case or end statement.

Hardware I/O

There are 3 basic hardware operations: read from a hardware module, write to a module, and write to the output data stream. The read from a module also has a variations that allows reading into a variable or directly into the output data stream. In addition, there is a clear crate statement which performs the appropriate operation for that crate.

Output, either explicitly done with the output statement, or implicitly done by a hardware read operation, is assumed to be in units of 4 byte integers. Each time a code section is called, it produces a single bank of 4 byte integers. The bank header (including bank length) is inserted automatically.

Generic I/O

The crate clear or reset operation performs a CAMAC crate clear or Z or a FASTBUS reset. The crate number is ignored for FASTBUS, and defaults to 0 for CAMAC.

```
clear crate [number]
reset crate [number]
```

The output statement transfers a single integer variable into the output stream:

```
output yyy
```

FASTBUS I/O

Reading and writing FASTBUS modules is a 4 step operation: (1) address the module, (2) select which register in the module to read or write (secondary address), (3) transfer 1 or more words (4 byte) of data, and (4) release the module. Modules have a unique geographical address (slot number, used most often), and may have one or more logical addresses (used is special applications). In addition, they have 2 internal address spaces *data* and *control*. Control space is typically where control registers are found for setting board options, and data space is typically used to read event data.

```
fastbus
 ...
address [geographic] [data] slot-number
address [geographic] control slot-number
address logical [data] laddr
address logical control laddr
secondary address saddr
```

The FASTBUS spec allows for addressing of multiple modules at the same time (called broadcast addressing). The syntax for this is

```
broadcast address [geographic] control broadcast_addr
```

where `broadcast_addr` is a module or function specific number. Common examples would be the All Local Module Address 1 or the Sparse Data Scan 9. Refer to specific FASTBUS module manuals on support for broadcast addressing.

Once the addressing has been set up, any number of words may be transferred (depending on the application). The read statement transfers a single word, and the block read statement transfers a variable length block of data (generally 1 event's worth of data).

```
read                     ! transfer 1 word to output
read into <xxx>          ! transfer 1 word to variable
write yyy                ! write yyy to current address
block read               ! transfer block to output
fast block read          ! FSCC specific (faster)
broadcast read into <xxx> ! used after a broadcast address
```

Finally, after data transfer the module(s) should be released. The two forms for address/bus release are,

```
release
broadcast release
```

CAMAC I/O

CAMAC has a different addressing scheme in which a register in a module is addressed by a combination of branch number (b), crate number (c), slot number (n), internal address (a), and function code (f). The function code generally distinguishes between read, write, and control functions, but may also be used to select between group 0 and group 1 data space (most modules only support group 0). NOTE: The current implementation of CODA only supports branch b=0.

```
camac
 ...
read b,c,n,a,f              ! transfer 1 word to output
read b,c,n,a,f into <xxx  > ! transfer 1 word to variable
write yyy into b,c,n,a,f    ! write yyy to module
control b,c,n,a,f           ! execute control function
```

CAMAC only supports a single trigger (currently) through a CAMAC lam:

```
link trigger lam b,c,n,a
```

The above statement specifies which lam to poll in polling mode, or which lam to expect as an interrupt in interrupt mode (default).

VME I/O

There is limited support for addressing and readout of VME modules in crl. Most module access must be done through imbedded c code. However, structures for memory maps of several commonly used modules at CEBAF have been added to aid the user in addressing these modules (See Table 8). These structures are defined when the user specifies `vme readout` at the top of his readout list code. (See the example trigger supervisor readout list in Appendix B.)

TABLE 8

Module	Description	Structure	Pointer
Trigger Supervisor	Control registers	vme_ts	*ts
	Memory	ts_memory[4096]	*tsmem
TS Interface card	Control registers	vme_tir	*tir[2]
Lecroy 1190	Dual ported memory	vme_dpm	*dpm, *dpml
Lecroy 1151	Scaler	vme_scal	*vscal[32]

The user should be aware of address modifiers and data transfer modes supported by their particular slave modules. The default kernel for the MV162 and MV167 boards used at CEBAF have 4 address spaces defined:

```
A16/D16      0xffff0000 - 0xffffffff
A24/D16      0xf0000000 - 0xf0ffffff
A24/D32      0xe0000000 - 0xe0ffffff
A32/D32       sysMemTop - 0xdfffffff
```

For example the Lecroy 1190 Memory requires certain registers to read and written to with single word transfers (A24/D16) while the memory can be read via longword transfers (A24/D32), hence the definition of two pointers (*dpm, *dpml) which can be defined using the appropriate address modifiers (0xf0xxxxxx and 0xe0xxxxxx).

Utility Statements

Arbitrary messages may be sent to the console task, tagged with a severity. This routine should be used with caution inside event readout lists as they may over run the logger's ability to keep up (thereby losing messages and degrading performance). The format of this call is similar to a c printf statement wherein the message string includes embedded format descriptors. For each format descriptor, the next unused argument is fetched and formatted according to the descriptor.

```
log [ inform | warn | alarm ] "quoted string",args,...
```

For example:

```
log warn "counter value is %d",counter
```

%d converts a decimal integer, %x produces hex output; other formats may be found in any c manual.

C.5 Example File

The following is a listing of the configuration of a readout controller which reads out a single Lecroy 1881 ADC. Triggering is provided by the Trigger supervisor. In the trigger routine, a broadcast address (Sparse Data Scan) is made to determine if the module has valid data in its buffer. The fastbus routines will be inlined providing approximately 50% faster execution of the trigger routine.

```
! Example Fastbus readout code for a single Lecroy 1881 ADC
! FSCC is being triggered by the Trigger Supervisor
! David Abbott, CEBAF 1994

fastbus readout
inline fastbus
ts control

ADCSLOT = 16
SCANMASK = hex 00010000

begin download
  log inform "User Download\n"
end download

begin prestart
  variable adcid

  reset crate 1

! Reset, clear ADC
  address geographic control ADCSLOT
  write hex 40000000
```

```
  release

! Program for no sparsification, Gate from FP
  address geographic control ADCSLOT
  write hex 00000104
  secondary address 1
  write hex 00000080
  release

! READ ADC ID
  address geographic control ADCSLOT
  read into adcid
  log inform “ADC ID = H8”, adcid
  release

  log inform “User Prestart Executed\n”
end prestart

begin end
  log inform “User End Executed\n”
end end

begin pause
  log inform “User Pause Executed\n”
end pause

begin go
  log inform “User Go Executed\n”
end go

begin trigger
   variable datascan, ii

! loop until ADC is completed buffering

   broadcast address geographic control 9
   broadcast read into datascan
   broadcast release
   ii = 0
   while ((datascan is not equal to SCANMASK) and (ii < 5))
     broadcast address geographic control 9
     broadcast read into datascan
     broadcast release
     increment ii
   end while

if ii is less than 5 then
! Load next event
  address geographic control ADCSLOT
  write hex 400
```

```
  release

! Read out ADC
  address geographic data ADCSLOT
  block read
  release
else
! Output my own header into the data stream
  output hex da0100ff
  output datascan
end if

end trigger

begin done
end done
```

APPENDIX D The CODA/EPICS Interface

D

With the introduction of EPICS as the slow controls system for the accelerator and increased interest for using EPICS for physics slow controls, there is in development an interprocess communication mechanism to allow CODA data acquisition and EPICS slow controls to interact. This communication will be facilitated by a single, user configurable process running on a UNIX workstation (called *ceimon*).

D.1 EPICS Requirements

The EPICS software must provide two EPICS database records into which *ceimon* may write values.

- **The CODA Status Record**: A single Analog Output record will contain the state (i.e. Active) of the CODA System.
- **The CODA Alarm Record**: A single Binary Output record will be used for triggering a signal to the EPICS Alarm Handler software indicating that the data acquisition run has halted for some reason.

D.2 CODA Requirements

The CODA software must provide a mechanism for *ceimon* to insert EPICS database record data into the event stream and hooks for *ceimon* to halt a CODA data acquisition run. The event insertion mechanism is accomplished with `daInsertevent()` which is part of the CODA RPC library. The mechanism to halt a data acquisition run is accomplished with a command line utility that can send various events to the RunControl software. For example, to halt a run one might type:

```
unix> rccommand -c end
```

D.3 CEIMON Requirements

Ceimon is responsible for six actions.

- **Self Configuration**: *ceimon* should configure itself with the three configuration files. The names of these files should be a default value or perhaps pointed to with a UNIX environment variable.
- **Generate DAQ Start Flag**: *ceimon* will retrieve and analyze a the value fields of specific database records contained in one file and generate a flag if CODA DAQ may continue or not.
- **Insert EPICS Data into CODA Event Stream**: *ceimon* will periodically read value fields from EPICS database records listed in a second file and call the CODA library routine daInsertEvent().

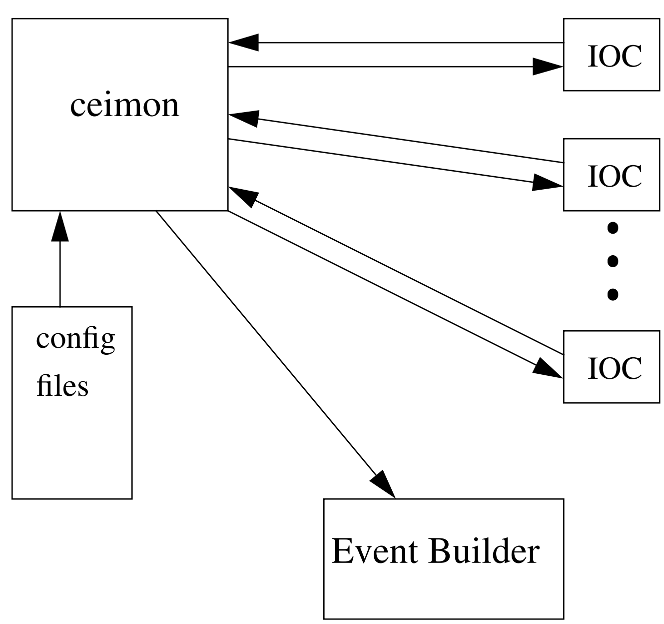
- **Halt CODA DAQ Run**: *ceimon* will use the CODA RunControl command line utility to send a halt signal to the RunControl software.
- **Notify CODA User of Run Halt**: *ceimon* will notify the CODA user that it has stopped the run due to a slow controls signal range error via a pop-up window as well a to the console logger.
- **Notify EPICS of Run Halt**: *ceimon* will write a "halted" value the EPICS CODA Status Record and it will write a value to the EPICS CODA Alarm Record indicating an alarm state.

D.4 CEIMON Configuration files

There are three configuration files.

- **The Start File**: A file containing slow controls signals and valid ranges for *ceimon* to check before CODA RunControl starts a data acquisition run.
- **The Extra File**: A file containing slow controls signals to be inserted into the CODA event stream not found in the Start file.
- **The All File**: A file containing all slow controls signals whose values are periodically inserted into the CODA event stream.

FIGURE 5 CODA/EPICS Interface



APPENDIX E
E

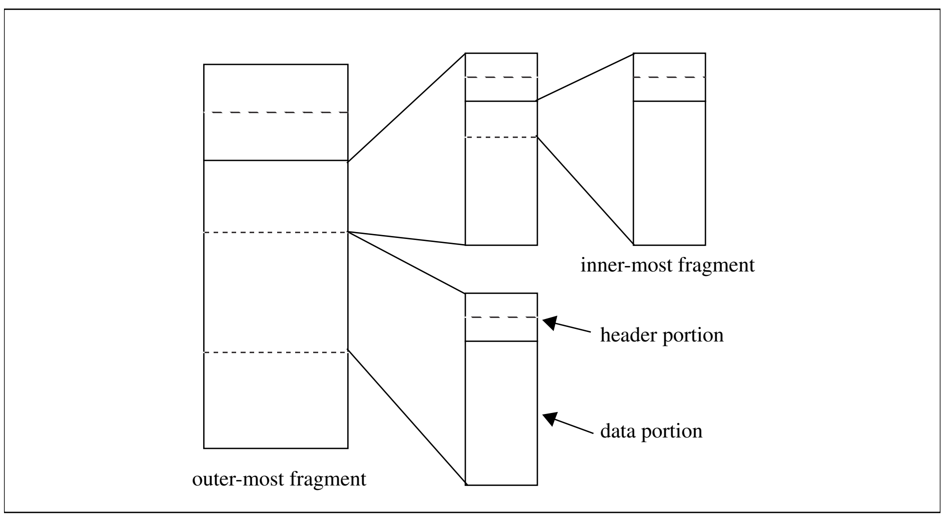
CEBAF Common Event Format

The CEBAF Common Event Format was designed to meet the following requirements:

- a partitioned format in which each fragment contains a header portion and a data portion.
- a recursive format to support complex structures (a tree). That is, fragments may contain other fragments within their data fields.
- a length field in each fragment header.
- a "type" field in each fragment header giving the data type of the data for that fragment. This allows for data conversion from one machine architecture to another, and marks each fragment as a branch or leaf node in the tree.
- a "tag" field in each header to indicate the source or purpose of the data contained in that fragment.
- a small fragment header.

This structure is shown in the following figure:

FIGURE 6 Event with fragment depth of three.



The last requirement above (low overhead) makes it difficult to derive a single header format. In some situations it may be desirable to have a header with a lot of identifying information and a large length field. In other situations, only minimal length and tagging information is needed.

As a result of these conflicting requirements, 3 header formats of differing sizes will be used. The corresponding fragments, in decreasing order of header size and functionality, will be referred to as "banks", "segments", and "packets".

E.1 Event Format

Banks

Following the nomenclature of other high energy physics formats, the first and largest constituent will be referred to as a “bank”. The bank header will consist of 2 longwords in the following format:

FIGURE 7 Bank Header Format

length		
tag	data_type	num
31 16	15 8	7 0

The bank “length” is the length of the fragment in longwords, excluding the length word (i.e. length is the number of words to follow); “tag” is an integer identifier which may be used as an index into an optional dictionary of bank and segment names and titles (described later); “data_type” is an integer giving the type of data in the data portion of this bank, and “num” is an integer which may be used as a bank number or may be used to encode additional information about the bank.

Each event (at its outermost level) will be a single bank in which the bank length is the event length; for this reason, the minimum overhead per event is 8 bytes. The tag and num fields together can be used to specify the type of event. For example, physics events will have a tag of 0xCEBA and a num field equal to the readout code number.

Segments

Applications requiring complex structure with less overhead per fragment can use a smaller fragment type called a “segment”. The segment header will consist of 1 longword in the following format:

FIGURE 8 Segment Header Format

tag	data_type	length
31 24	23 16	15 0

The major differences between bank and segment headers are the maximum fragment length allowed, and the number of available tags; there is also no “num” field in a segment header, so tags need to be unique. The segment length is still given in longwords (unsigned, up to 1/4 Mbyte, and excluding the header word), and the tag field (unsigned) takes on the values 0-255 vs. 0-65535.

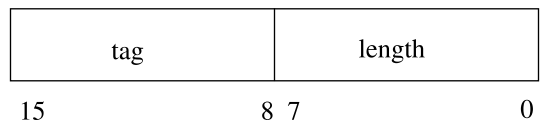
Padding of a bank or segment may be necessary to bring it up to a multiple of 4 bytes; it is up to the application to determine the actual end of the data (it should be obvious from context).

Note that the bank and segment headers must be treated as longwords for the purposes of converting between big and little endian machines.

Packets

For data structures with even less overhead per fragment, a third fragment type, the "packet", may be used. The packet header will consist of a single 16 bit integer as shown in the following format:

FIGURE 9 Packet Header Format



Packets contain data items with widths of no more than 16 bits, and the length field is in units of 16 bit words. Packets cannot be recursive as there is no "data_type" field to indicate when the recursion should stop. They are intended to encode small arrays tagged by a small number (0-255). Unlike banks and segments, packets may start on odd word boundaries.

Data Types

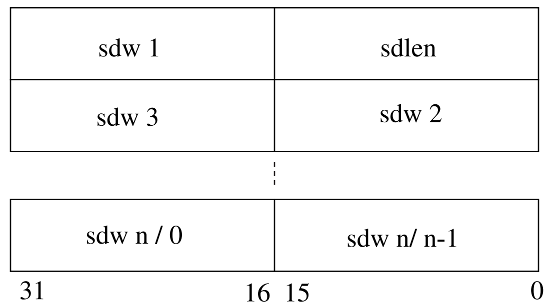
The following is a preliminary list of defined data types (in hex):

0 = unknown
1 = long (32 bit) integer
2 = IEEE floating point
3 = null terminated ASCII string
4 = 16 bit signed integer
5 = 16 bit unsigned integer
6 = 8 bit signed integer
7 = 8 bit unsigned integer
8 = double precision IEEE floating point
9 = VAX floating point
A = VAX double precision floating point
F = repeating structure
10 = bank
20 = segment
30 = packet with data_type = 0
33 = packet with data_type = 3
34 = packet with data_type = 4
35 = packet with data_type = 5
36 = packet with data_type = 6
37 = packet with data_type = 7

Complex Structures

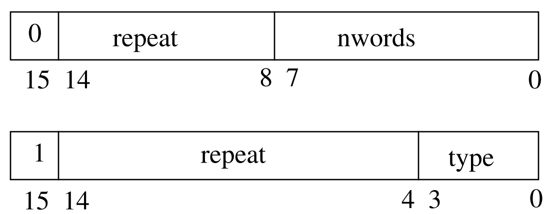
In applications requiring mixed data types (integer particle ID and real particle energy, for example), banks or segments may contain "repeating structures". In these banks (segments), the first few words of data are the structure description, and the remaining data words are data items of that structure. The structure description has the following form:

FIGURE 10 Structure Description Format



where “sdlen” gives the length of the structure description in longwords, “sdw i” is a 16 bit structure description word in one of the following formats:

FIGURE 11 Structure Description Word Format



The basic idea is to encode a format descriptor like (4I,4F,5(1I,1F),1F). The first form (high bit zero) is used to encode an open parenthesis: “repeat” gives the repeat count of the parenthesized expression, and “nwords” gives the number of following descriptor words until the parenthesis closes. The second form (high bit set) is used to encode a repeated field of data type given by the low 4 bits. Only types 1 through 8 are currently allowed. Since the entire structure description must be a whole number of longwords, it may be necessary to pad the structure with a null descriptor, which is ignored. A repeat count of zero is not allowed; an implicit repeat is performed on the whole structure until the end of the bank or segment is reached.

Another condition to keep in mind when designing structures is to make them multiples of the largest sized data item because some architectures can only access data items on natural boundaries (i.e. longwords on 4 byte boundaries, short words on 2 byte boundaries, doubles on 8 byte boundaries).

One application of structures is as an alternative to packets. For example, consider the problem of encoding the left and right TDC hits of a wire along with the wire number, where the data is 16 bits. Using packets would require 3 words per wire, (using the tag field to give the wire number) but would be limited to 256 wire numbers. Alternatively, a structure of the form (3I), in which the first word is the wire number and the next 2 words are the data, allows 65536 wire numbers and still only uses 3 words per wire hit (plus 2 words of structure definition).

E.2 Physical Record Format

Events (logical records) will be packed into fixed sized physical records. Each physical record will have an 8 longword header as follows:

BLKSIZ blocksize (in longwords)

BLKNUM block number

HDLEN header length (offset to data)

START offset to first start of logical record in block

END number of valid words (header + data) in block

VER header version number (=1 in CODA 1.0)

(reserved, =0)

(reserved, =0)

All lengths and offsets are in units of longwords, and offsets are relative to the start of the block (i.e. offset=8 points to the start of the data area). Blocksizes must be multiples of 256 longwords, and no greater than 32768 longwords (i.e. 1 to 128 Kbytes). With this convention, the "blocksize" word should only have non-zero data in bits 8-15, and can be used to detect any byte swapping problem (big/little endian).

Note that if a logical record spans 3 physical records, the middle physical record will have START=0.

E.3 Name Dictionary

The name dictionary is a simple ASCII file, with at most one name entry per line. Curly brackets { } will be used to indicate the beginning and end of a set of names of sub-fragments.

Each definition line will contain the following:

- index / tag value in hex
- fragment name (ASCII)
- fragment title (ASCII)

The index value for a bank must be in the range 0000 to FFFF and for a segment it must be in the range 00 to FF. The fragment name must contain only alphanumeric characters, and names are case insensitive. The title may contain any printable character. The three fields are delimited by any amount of white space (space or tab), and the title starts at the first non-space character after the name and continues to the end of line or to a matching close curly bracket.

Comments may be embedded anywhere in the file (including the middle of titles) using C style delimiters:

```
/*comment */
```

If the first non-comment, non-white space character on a line is the left curly bracket, all following definitions up to the close curly bracket are for substructures of the previous name. For example:

```
/* This is a sample name dictionary */
1   aname   now is the time /* I hope */ for all
2   another
  {1  abc   good men
   2  def   to come to
  }
3  lastname (sic)
  {99 abc "handle"}
```

Defines the outermost names *aname*, *another*, and *lastname*, where the fragment *another* has 2 defined sub-fragments named *abc* and *def*, and *lastname* has one defined sub-fragment named *abc*. By convention, sub-fragments may be uniquely referred to by giving all parent names in order separated by periods. In this example, *another.abc* is a fragment with a title of *good men*, and *lastname.abc* has a title of *"handle"*.

APPENDIX F
F

CODA Event Bank Definitions

(See Appendix E for a discussion of the bank structure used below.)

F.1 Standard Physics Event

A standard physics event will consist of an outermost bank with a data type of “bank”, a tag equal to the event type, and the “num” field containing the hex constant 0xCC (used to verify that this is an event header word):

event_length		
event_type	0x10	0xCC
data bank		
data bank		
...		

The first data bank is created by the event builder (see below), and other banks come from the readout controllers. Event types produced by CODA data acquisition front ends range from 0-15, corresponding to the event readout code.

F.2 Event ID Bank

The event builder creates an event ID bank in the following format:

length = 4		
tag = 0xC000	dtype = 1	num = 0
event number		
event classification		
status summary		

The event number is a counter of the events within a run, starting at 1 with each new run. The event classification word initially holds the 4 bit trigger code; analysis programs would use other bits to indicate the presence of application specific features in the event. The status summary initially holds readout status, one bit per readout controller, and is later replaced with analysis status information.

F.3 Readout Controller Data Banks

Data from each readout controller will be contained in separate banks, with the tag equal to the ROC number (0-31) and the datatype = 1 (longwords). The num field for these banks will contain the low 8 bits of the event counter on the corresponding ROC, and will be used by the event builder to verify that the fragments are properly synchronized.

fragment_length			
(code & status)	ROC#	1	counter
data words			

Prior to event building, the high 11 bits of the tag field will be used to pass the readout code number and status information from the ROC's to the event builder. These bits are cleared by the event builder leaving a tag from 0-31.

F.4 Run Control and Sync Events

For each of the state transitions *prestart, go, pause,* and *end,* and for each synchronization event, each ROC generates an event fragment containing information about the transition. The event builder waits until it receives the events from all participating

ROC's and then forwards only one of the events up the analysis chain. The format for these events is as follows:

event_length		
event_type	1	0xCC
data words		

The first data word in each of these events is a 32 bit integer containing the time in seconds since Jan 1, 1970 GMT (this follows the Unix convention for time formats). Additional data words specific to the event type follow (see below).

The following event types are defined:

Event Type	Definition
16	Sync event
17	PreStart event
18	Go event
19	Pause event
20	End event

F.5 Sync Event

Sync events are generated automatically by the trigger supervisor, or upon operator command. Sync events have the following format:

event_length = 5		
event_type = 16	1	0xCC
time		
number of events since last sync		
number of events in run		
status		

Time is the number of seconds since Jan 1, 1970 GMT. Status is a bit mask with one bit set for each ROC which encountered an error in checking the readout synchronization.

F.6 PreStart Event

PreStart events are generated during the transition from the downloaded state to the pre-started state. PreStart events have the following format:

event_length = 4		
event_type = 17	1	0xCC
time		
run number		
run type		

Time is the number of seconds since Jan 1, 1970 GMT. The run type is a 32 bit number from the rcRunTypes file (Appendix A).

F.7 Go Event

Go events are generated for each Start or Resume command. Go events have the following format:

event_length = 4		
event_type = 18	1	0xCC
time		
(reserved)		
number of events in run thus far		

Time is the number of seconds since Jan 1, 1970 GMT. The number of events in the run will be 0 for the first “go” (i.e. the initial Start command), and will be non-zero for each resume command following a pause.

F.8 Pause Event

Pause events are generated for each transition from the active state to the paused state. Pause events have the following format:

event_length = 4		
event_type = 19	1	0xCC
time		
(reserved)		
number of events in run thus far		

Time is the number of seconds since Jan 1, 1970 GMT.

F.9 End Event

End events are generated each time a run is ended. End events have the following format:

event_length = 4		
event_type = 20	1	0xCC
time		
(reserved)		
number of events in run		

Time is the number of seconds since Jan 1, 1970 GMT.

Index

A
analysis program
  linking 3-14

B
banks E-2
  event ID F-2
  readout controller F-2
blocksize E-5
buttonFeedback 3-7, A-4

C
CAMAC 2-2, 4-2, 4-6
caopen 4-3
cccc 4-2
cccd 4-2
ccci 4-2
cccz 4-2
cclc 4-2
cclm 4-2
cdlam 4-2
cdreg 4-2
cedump 3-15
  name dictionary E-5
cefdmp 3-15, 5-4
ceimon D-1
ceMsg 4-9
cePmsg 4-9
cfga 4-2
cfmad 4-2
cfsa 4-2
cfubc 4-2
cfubl 4-2
cfubr 4-2
coda_activate 3-3, 5-8, A-1
components A-1
configuration string 2-6
configuring CODA 2-5
csga 4-3
csmad 4-3
cssa 4-2
csubc 4-3
csubl 4-3
csubr 4-3
ctcd 4-2
ctci 4-2
ctgl 4-2
ctlm 4-2
ctstat 4-3

D
daCopyEvent 4-20
daCopyNext 4-20
daCopyPoll 4-20
daCopyRegister 4-20
daCopyUnregister 4-20
daInsertEvent 4-20
daReadInt 4-20
data flow 2-1
data types E-3
dump, see cefdmp

E
end F-5
errors 4-9
evClose 4-11
Event Builder 2-3
event number F-2
Event Recorder 2-3
events
  banks E-2
  dump utility 3-15
  end F-5
  format E-1
  go F-4
  I/O routines 4-11
  inserting 4-20
  packets E-3
  pause F-5
  physical records E-5
  physics F-1
  prestart F-4
  segments E-2
  spying 4-20
  structures E-4
  synchronization F-3
evIoctl 4-11
evOpen 4-11
evRead 4-11
evWrite 4-11

F
FASTBUS 2-2, C-5
FASTBUS I/O 4-6
fb_init_1 4-6
fpac 4-6
fpad 4-6
fparb 4-6
fparel 4-6
fpr 4-6
fprb 4-6
fprel 4-6
fpsar 4-6
fpsaw 4-6
fpw 4-6
fpwb 4-6
FSCC 2-2

H
HBOOK 4-14
  ntuples 4-14
HBOOK1 4-14
HBOOK2 4-14
HBOOKN 4-14
HFILL 4-14
histograms 4-14
HLIMAP 4-14
HLIMIT 4-14

M
makelist 3-12

N
name dictionary E-5
ntuples 4-14

O
online 3-7, A-4

P
packets E-3

pause F-5
PAW 2-3
prestart F-4

R
RCDEFAULTS 3-6, 3-7
rcDefaults A-4
rcExperiment A-4
rcNetwork 2-5, 3-2
rcPriority A-4
rcRunNumber A-4
rcRunTypes 2-5, 3-3, A-2
rcService 4-17
readout controller 2-2
readout controllers
  configuration 3-11, C-1
readout language C-1
ROC, see Readout Controllers
rpcUpdate 3-7, A-4
Run Control
  architecture 2-4
  components 3-2
  configuration 2-5
  network definitions 3-2
  starting 3-1
  state machine 2-5, 3-5
  user interface 2-6
<runType>.config 3-4, A-2
<runType>.options 3-4, A-3

S
scReporting 3-6, 3-7, A-4
segments E-2
severity 4-9
spying 4-20
structures E-4
synchronization 3-10
  event format F-3

T
trigger
  multi-level 2-2
  Trigger Supervisor 2-1
Trigger Supervisor 3-7
  configuration 3-7, B-1
  parameters 3-8, B-1
  synchronization 3-10

U
usrDownload 4-17
usrEnd 4-17
usrEvent 4-17
usrGo 4-17
usrPause 4-17
usrPrestart 4-17

V
verboseReporting 3-7, A-4
VME 2-2
VXI 2-2

X
xcefdmp 5-4