Version

2.6

JEFFERSON LAB

Data Acquisition Group

EMU User's Guide

JEFFERSON LAB DATA ACQUISITION GROUP

EMU User's Guide

the author

Carl Timmer
timmer@jlab.org

24-Mar-2016

Table of Contents

Section

1

1. Introduction

Prior to CODA (CEBAF online data acquisition) version 3, CODA's data-handling software components were self-contained, independent software entities. These components included the Readout Controller (ROC) which ran on embedded computers using the realtime operating system vxWorks. Its task was to read the data-producing hardware modules, package the data and send it to the next component. Next in line was the Event Builder (EB) which took the data from all of the ROCs and made one EVIO event out of it. Finally there was the Event Recorder (ER) which took the nicely packaged data and wrote it to a file.

Each component's communications had to be carefully coordinated with the other components and each was also individually responsible to communicate with run control and respond to its commands. As you can imagine, much of the code was redundant between the ROC, EB, and ER.

With the development of the ET system, which was used in CODA version 2 to transport data from the EB to users and to the ER, it was a small jump to use it between the ROC and the EB as well. The additional availability of the cMsg message-passing software package made it another tiny hop to replace all run control communication code with calls to cMsg. Between these 2 pieces of software, all the interprocess communication needs were met and all the data transfer software was abstracted out of the CODA components.

While CODA version 3 is built on its ability to use ET and cMsg to do all the "talking", another area of abstracting functionality involves the EB and ER. Both are very similar in functionality in that they both read data, do something to the data, write the data, and respond to run control commands. Right away it's obvious that the reading in, writing out, and run control parts can be identical between the EB and ER. It's also a fairly simple matter to take the middle part (doing something with the data) and make that a plug-in. This is the fundamental structure of the EMU. It's a framework to ease development by taking out all the identical CODA component pieces and programming them once for all. It allows selection of standard inputs and outputs and it accepts run control commands. All the user must do is write the plug-in to handle the data and respond to the incoming commands.

1.1. Input / Output

The EMU is designed to read and write evio format data. It may accept such data by 3 different means or ***transports***: through the ET system, in cMsg messages, and from files. A single transport deals with either an ET system, cMsg's cMsg domain, direct sockets in cMsg's emu domain, or a file. Multiple transports can be defined and used in a single EMU. Each of these transports can have multiple ***channels*** in a single EMU as well. Each channel is a single connection to an ET system or cMsg server, or opening of a file.

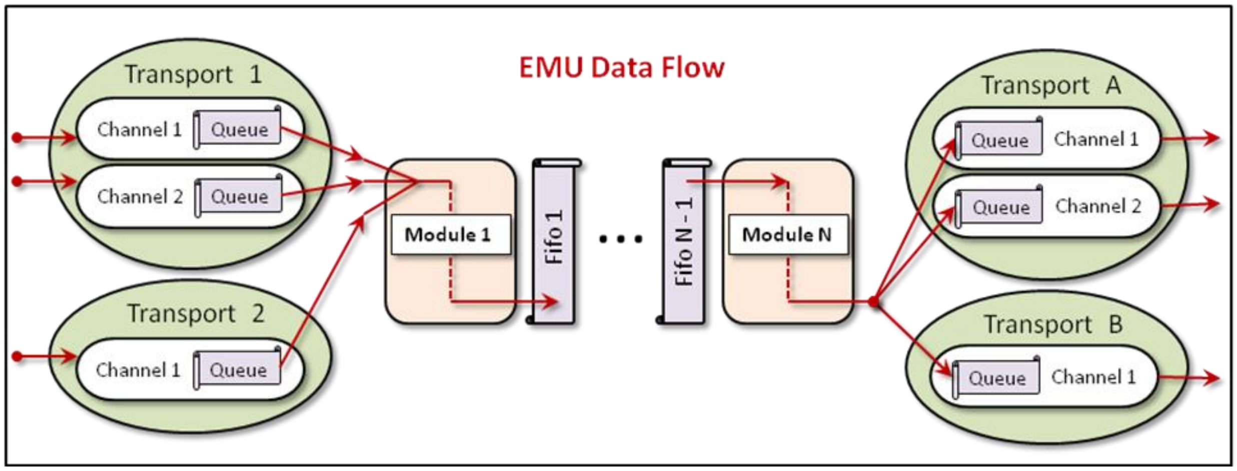
The transport handling code is abstracted from the rest of the EMU in such a way as to be able to add other means of I/O. To document the complete interface to accomplish this is too much work for a task best left to the DAQ group itself. Thus, if a user needs to interface with another type of I/O, contact the DAQ group to make arrangements for adding the necessary code.

The EMU expects ET events or buffers to contain evio data - simple enough. In cMsg, it expects the byte array to contain the evio data (in Java, msg.getByteArray()) - equally simple. Files are self-explanatory.

1.2. Modules

Between the input and output channels are ***modules***. These are Java classes that take evio events from an input channel and place evio events on an output channel. There can be more than one module to a single EMU. The modules are ordered sequentially so that the data flows in one path through the EMU. Fifos are I/O channels used between modules.

The data flow looks like the following diagram. An input channel receives data through an ET event, cMsg message or file. It parses the incoming data into evio events. Each evio event received is placed into its queue. That queue along with all the other input channel queues are inputs to the first module. That module takes each evio event, processes it, and then places it in an output channel or fifo. At this point either the next module gets, processes, and passes it on, or it goes to a queue of one of the output channels. The channel then gets, processes, and sends it somewhere else.



1.3. Configuring

Each EMU can be configured through an XML configuration file with specific elements and attributes (details are contained in later chapters). They can be complicated to configure since there are such a large number of parameters one can tweak. A large part of the complication is due to interprocess communication.

1.4. Monitoring, and Controlling

The EMU can be visually monitored and controlled by means of a built-in debugging GUI. This GUI is useful when, for example, the user is testing a module and no run control platform is operating, since it can send the run control commands to the EMU.

Section 2

2. Data Input and Output

EMU's can be complicated to configure since there are such a large number of parameters one can set. Most of the complication is due to interprocess communication. To review, EMUs may use 4 different means of communication to the outside world: ET systems, cMsg messages in cMsg domain, direct sockets in cMsg emu domain, and files. The data being sent to, through, and out of the EMU is in EVIO version 4 format. Whether stored in files, in buffers, or sent through sockets, the EVIO format is identical in each case. The transport objects in the EMU read and write only in this format.

The input data is read and parsed, breaking it up into individual EVIO events. These events are what are placed in the internal queues and FIFOs seen in the last chapter's diagram. Thus modules deal only with evio events directly. Similarly, the output transport objects take the events from their queues and repackage them to be sent in proper EVIO format.

So how does one go about the business of specifying all this data movement? Each EMU is configured through an XML file with 2 major sections. Parts of the I/O config come in both sections. One major section specifies data transport mechanisms while the other specifies the modules. Under each module definition come individual data channels from one of the defined transports. The following will walk the user through creating the I/O portion of these config files.

2.1. Transports

One major part of a config file is contained in the single XML element, ***transports***, usually in the first lines. There are 4 different types of transports to the outside world that may be used: cMsg, emu, ET, and files. And there is 1 type of transport between modules, FIFOs. Each of these is configured quite differently. The xml element defining a transport uses the name ***server***.

2.1.1. FIFOs

The FIFO is a type of transport that is built into the EMU and needs no specification in transports section of the config file.

2.1.2. Files

Specifying the parameters to define a file transport requires only 2 things: 1) the user-given ***name*** and 2) the ***class*** - which is always and exactly "File" (case sensitive).

```
<transports>
   <server name="myFile" class="File" />
</transports>
```

2.1.3. cMsg messages in cMsg domain

Specifying the parameters to define a cMsg transport requires 3 things: 1) the user-given ***name*** which will also be used as the cMsg client name and therefore must be unique to the cMsg server**,** 2) the cMsg ***udl,*** and 3) the ***class*** which is always and exactly "Cmsg" (case sensitive). The udl may be set to "**platform**" in which case the udl is internally set to the cMsg server inside the run control platform being used and the namespace is set to "CODA".

```
<transports>
   <server name="mycMsg" udl="cMsg://host:port/cMsg/nspace" class="Cmsg" />
   <server name="yourcMsg" udl="platform" class="Cmsg" />
</transports>
```

2.1.4. Sockets in cMsg emu domain

Specifying the parameters to define a cMsg, emu domain transport when **receiving** data requires 3 things: 1) the user-given ***name*** which will also be referred to by any channels that use it**,** 2) the TCP ***port*** number used for communication**,** and 3) the ***class*** which is always and exactly "Emu" (case sensitive). Note that the xml element name is **client** since it is receiving data even though it acts, in fact, as a TCP server awaiting connections from DAQ components upstream. A unique port is required for each component in the DAQ system that uses this protocol. If a port is not specified, it defaults to 46100.

```
<transports>
   <client name="emuSocket1" port="46101" class="Emu" />
</transports>
```

When **sending** data with this transport, the port does not need to be set (since that is set in the channel) and the xml element name is **server** since it is sending data.

```
<transports>
   <server name="emuSocket1″ class="Emu" />
</transports>
```

To go into a little more detail, a downstream component, say an EB, will start such a server in the transport object's download transition which behaves somewhat like an rc multicast server. When it starts up, it listens for multicasts, but only accepts packets from upstream components in the configuration which must connect to it, say Roc1. (Thus multiple sessions can coexist with servers of the same port & EXPID if they serve different components). It sends a packet back to the upstream components (Roc1) with enough information so that they can make a TCP connection. It stays this way until prestart when the Roc1's output channel will connect to the TCP server and send data over that socket. See the section below on the cMsg emu domain channel for more info.

2.1.5. ET

The EMU uses the ET system in 2 different ways. In the first, the EMU "owns" an ET system. It creates the ET system, uses it, and removes it when finished. The second method is to simply connect to an existing system, use it, then disconnect when finished.

The xml element defining an ET transport uses the name ***server***. The following is a table containing each of its xml attributes (case sensitive), whether it's required, its meaning, and its acceptable values:

Attribute	Required	Function	Allowed Value(s)
All ET systems			
name	✓	User given name of transport	Any string.
class	✓	Java class for transport object	The exact string, "Et".
etName		ET system name	Et file name. Defaults to /tmp/<EXPID>_<EMU name>, where EXPID is an environmental variable or given on command line
create		Does EMU create the ET system?	Case indep. "true", "on", or "yes". Anything else is false. Default = false.
port		TCP port.	Integer > 1023 and < 65536. Defaults to 11111.
uPort		UDP port.	Integer > 1023 and < 65536. Defaults to broadcasting port, 11111.
wait		How many seconds to wait for an ET system when connecting	Integer >= 0 seconds. Defaults to 0.
mAddr		Multicast address	Any valid multicast address. Needed only if **method** = "mcast" or "cast".
When creating ET systems			
type		Create java-based or C-based ET sys.	The case indep. string “java” for java-based, else C-based.
eventNum	✓	Number of events.	Integer > 0.
eventSize	✓	Size of event data in bytes.	Integer > 0.
groups		Number of groups to divide events into.	Integer > 0 and <= **eventNum**. Defaults to 1.
revBuf		TCP receive buffer byte size.	Integer >= 0. Value of 0 gives operating system default. Defaults to 0.
sendBuf		TCP send buffer byte size.	Integer >= 0. Value of 0 gives operating system default. Defaults to 0.
noDelay		TCP NODELAY value	Case indep. "true", "on", or "yes". Anything else is false. Default = false.
When connecting to existing ET systems			
method		Method to connect with.	Case indep "direct" for direct to server, "mcast" for multicasting, "bcast" for broadcasting, "cast" for both.

host		Host running ET system.	Case indep "anywhere", "remote", "local" denotes where on network to look for ET. Or name of host. Defaults to "anywhere".
bAddr		Broadcast address.	Any valid broadcast address. Needed only if **method** = "bcast" or "cast".
subnet		Preferred IP address / subnet to use for ET communication if possible.	May be preferred IP address of local host which is used to for ET I/O. May also be broadcast address of subnet preferred for communication which gets changed to host IP address on that subnet if possible.

Now for a few examples. The following is an example of a config file entry that can be used to have the EMU create and control an ET system:

```
<transports>
   <server name="myEt"  class="Et"  etName="/tmp/Eb1" create="true"
          port="54321"  uPort="12345"  eventNum="1000"  eventSize="2100000"
          groups="2" wait="10" recvBuf="3000000" sendBuf="130000"/>
</transports>
```

Notice that ***method*** and ***host*** are not defined. The EMU will automatically attach to created ET systems using a direct connection to a local host since the TCP server port is known and the host must be local by definition.

When not creating an ET system, omit the "created" attribute. In such cases, the attributes of ***eventNum***, ***eventSize***, ***groups***, ***revBuf***, and ***sendBuf*** are all ignored. Next is the simplest possible entry for attaching to an existing system since it uses all the defaults:

```
<transports>
   <server name="myEt"  class="Et />
</transports>
```

In this case the EMU needs to be run with the "-Dexpid=..." command line argument or the environmental variable EXPID needs to be defined (command line takes precedence). The reason is that if not given, the ET system name is automatically generated as /tmp/<expid>_<emu name>.

To make a direct connection to an existing ET system on a known ***host*** and ***port***, do something like:

```
<transports>
   <server name="myET"  class="Et" etName="/tmp/myET"  host="myHost"
           method="direct" port="12376"/>
</transports>
```

In such cases it is meaningless to specify attributes like the number and size of events, since for an existing system they are already defined and the EMU will just ignore them.

2.2. Channels

For each transport, multiple channels can be created. These are designated in the config files as ***inchannel*** or ***outchannel*** xml elements depending on which way the data flows. These elements are associated with and defined under the elements for each module. Where exactly they belong in the file will be seen later, but for now we'll only look at the individual channels.

2.2.1. FIFOs

Since FIFOs are only used between modules, they are not allowed to be specified as input to the first module and an exception will be thrown if that's the case. Although not prohibited, placing a FIFO after the last module guarantees that the data will not flow through the EMU once that FIFO is full.

In order to keep the data flow from getting ridiculously complex, if a module has a fifo as its output channel, then it may only have **ONE** output channel. Likewise, if a module has a fifo as its input channel, then it may only have **ONE** input channel.

For modules to use FIFOs to pass data between them, one does something like this:

```
<modules src="modules.jar" usr_src="user_modules.jar">
    <InModule class="someClass1">
        <inchannel id="1" name="a" transp="myFile" fileName="a" />
        <outchannel name="F1" transp="Fifo"/>
    </InModule>
    <MidModule class="someClass2">
        <inchannel  name="F1" transp="Fifo"/>
        <outchannel name="F2" transp="Fifo"/>
    </MidModule>
    <OutModule class="someClass4">
        <inchannel name="F2" transp="Fifo"/>
        <outchannel id="2" name="b" transp="myFile" fileName="b" />
    </OutModule>
</modules>
```

Simply name a FIFO by using the ***name*** attribute of a channel and specify the ***transp*** attribute as being exactly "Fifo". Make one module's output channel fifo the same as the next module's input channel fifo and data will flow from one to the other.

2.2.2. Files

For a module to define a particular file as an output one does this:

```
<outchannel id="0" name="me" transp="myFile" fileName="$(DIR)/a%d"
            split="10000" />
```

While an input file (seldom used) looks like:

```
<inchannel id="0" name="me" transp="myFile" fileName="abc" />
```

Here is a list of all the file channel attributes and what they do:

Attribute	Required	In/Out	Function	Allowed Value(s)
id		both	User given id of channel.	Any integer. Defaults to 0.
name	✓	both	User given name of channel.	Any string.

transp	✓	both	Name of the transport to use.	An already defined transport's name.
ringSize		in	Ring buffer capacity for parsed evio events from file.	Power of 2. Defaults to 4096. Min is 128.
fileName		both	Name of file to write or read.	A valid file name. May contain wildcards. For details see below. Defaults to the automatically generated name.
split		out	Number of bytes at which to create & start writing to another file.	Any non-negative integer. Defaults to 0 which means do not split the file. Negative values are ignored.
dir		out	Directory in which to write.	Valid directory. Not used by default.
dictionary		both	Name of file containing evio format xml dictionary.	A valid file name.

The most complicated part of the file channel is the default values of and the automatic generating of the file name. Let's start with splitting the file. In order to prevent output files from getting too large, they can be split into smaller ones. This happens when the user specifies a positive value for the ***split*** attribute which specifies the maximum file size in bytes before it is closed and the next one created. A negative or zero value means no splitting takes place. When files are split, each created file has a different integer number appended to or placed somewhere in its name to differentiate it from the previous one. This number starts at 0 and increases by 1 for each subsequent file and is called the ***file count***.

The rules for automatic naming of files are built into evio and are as follows. The base file name may contain up to 2, C-style integer format specifiers using “d” and “x” (such as %03d, or %x). If more than 2 are found, an exception will be thrown. If no "0" precedes any integer between the "%" and the "d" or "x" of the format specifier, it will be added automatically in order to avoid spaces in the generated name. The first occurrence will be substituted with the given run number. If the file is being split, the second will be substituted with the file count. If 2 specifiers exist and the file is not being split, no substitutions are made. If no specifier for the file count exists, it is tacked onto the end of the file name after an added dot.

The base file name may contain characters of the form “$(ENV_VAR)” which will be substituted with the value of the associated environmental variable or a blank string if none is found. It may also contain occurrences of the string "%s" which will be substituted with the value of the run type or nothing if the run type is null.

If no fileName is given, for input files it defaults to reading from codaDataFile.evio. For output files, it defaults to writing to <session>_<run#>.dat<file_count> where session refers to its session name received from run control. If ***dir*** is defined then it writes to a file of the generated name in that directory and it reads codaDataFile.evio from that directory as well.

If, on the other hand, fileName is given, then the first thing that happens is that all substitutions mentioned above are made. For input files, it reads from that file (in the directory dir if given). For output files, it writes to that file (in the directory dir if given).

2.2.3. cMsg – cMsg domain

For a module to define a particular cMsg server as an output one does this:

```
<outchannel id="0" name="me" transp="mycMsg subject="a" type="b" />
```

While an input cMsg subscription looks like:

```
<inchannel id="0" name="me" transp="mycMsg" subject="sub" type="*" />
```

Here is a list of all the cMsg channel attributes and what they do:

Attribute	Required	In/Out	Function	Allowed Value(s)
id		both	User given id of channel.	Any integer. Defaults to 0.
name	✓	both	User given name of channel.	Any string.
transp	✓	both	Name of the transport to use.	An already defined transport's name.
ringSize		in	Ring buffer capacity for parsed evio events from cMsg.	Power of 2. Defaults to 4096. Min is 128.
single		out	If true, each evio event sent out in single cMsg msg, else events marshalled into 1 msg.	Case indep. "true", "on", or "yes". Anything else is false. Default = false
subject		both	cMsg subscription subject for inchannel. cMsg message subject for outchannel.	Valid cMsg subject. Subscription subject may contain wildcard chars.
type		both	cMsg subscription type for inchannel. cMsg message type for outchannel.	Valid cMsg type. Subscription type may contain wildcard chars.
wthreads		out	Number of cMsg message buffer filling threads	Positive int which defaults to 1. Max is 10.

A note on the input channel’s subject and type. In the cMsg domain, the wildcard characters “*”, “?”, and “#” are allowed in subscriptions’ subject and type, where “*” matches any number of characters, “?” matches a single character, and “#” matches one or no positive integer. Also wildcard constructs like {i>5 | i=4} can be used to match a range of positive integers which meet the conditions in the parentheses. The logic symbols >, <, =, |, & are allowed along with the letter i, any positive integers, and spaces. As an example, a subscription to the subject abc{i>22 & i<26} will match a message with the subjects abc23, abc24, and abc25.

2.2.4. cMsg – emu domain

For a module to define a particular emu socket output channel one does something like:

```
<outchannel id="0" name="me" transp="myEmu timeout=”5” port=”46123”
   maxBuf=”256000” />
```

While an emu socket input channel looks like:

```
<inchannel id="0" name="me" transp="myEmu" />
```

Here is a list of all the emu domain channel attributes and what they do:

Attribute	Required	In/Out	Function	Allowed Value(s)
id		both	User given id of channel.	Any integer. Defaults to 0.
name	✓	both	User given name of channel.	Any string.
transp	✓	both	Name of the transport to use.	An already defined transport's name.
ringSize		in	Ring buffer capacity for parsed evio events from cMsg.	Power of 2. Defaults to 4096. Min is 128.
single		out	If true, each evio event sent out immediately over socket, else events marshalled before sending.	Case indep. "true", "on", or "yes". Anything else is false. Default = false
recvBuf		in	TCP socket receive buffer byte size.	Integer >= 0. Value of 0 gives operating system default. Defaults to 0.
sendBuf		out	TCP socket send buffer byte size.	Integer >= 0. Value of 0 gives operating system default. Defaults to 0.
noDelay		out	TCP NODELAY value.	Case indep. "true", "on", or "yes". Anything else is false. Default = false.
subnet		out	If is a local IP address, it is used for outgoing traffic. If subnet address, local IP address on that subnet is chosen, if any.	Dotted-decimal format IP address.
maxBuf		out	Internal buffer size for sending.	Max size in bytes of a single send and defaults to 2.1MB.
port		out	UDP socket to multicast to when finding emu server..	Integer > 1023 and < 65536. Defaults to 46100.
timeout		out	Time in seconds to wait for connecting to emu server	Any non-negative integer. Defaults to 3.
direct		out	If true, use direct instead of normal Java ByteBuffers.	Case indep. "true", "on", or "yes". Anything else is false. Default = false

In download, the emu domain transport object will have created a server for the downstream component, say an EB. In prestart the upstream component, say a ROC, module will create its output channel by making a TCP connection to this server. The UDL used to make the connection is:

```
emu://port/expid/compName?codaId=id&timeout=timeout&bufSize=maxBuf&tcpSend=sendBuf
      &subnet=subnet&noDelay
```

where “port” is the TCP port to use when creating the socket, “compName” is the name of the destination CODA component, “timeout” is the time to wait in seconds for connecting to emu server and defaults to 3 seconds, “bufSize” is the max size in bytes of a single send and defaults to 2.1MB, “tcpSend” is the TCP send buffer size in bytes, “subnet” is the preferred subnet (broadcast) IP address used to connect to server, and “noDelay” is the TCP no-delay parameter turned on.

2.2.5. ET

For a module, for example an EMU-based ROC, sending output to a particular ET system:

```
<outchannel id="1" name="EB" transp="myET" capacity="24" group="1" chunk="6"
            recvBuf="350000" sendBuf="350000" noDelay="on" />
```

While input from an ET system, for an EB for example, looks like:

```
<inchannel  id="1" name="Roc1"  transp="myEt" stationName="stat1"
            position="1" idFilter="on" chunk="1"/>
```

Here is a list of all the ET channel attributes and what they do:

Attribute	Required	In/Out	Function	Allowed Value(s)
id		both	User given id of channel.	Any integer. Defaults to 0.
name	✓	both	User given name of channel.	Any string.
transp	✓	both	Name of the transport to use.	An already defined transport's name.
ringSize		in	Ring buffer capacity for parsed evio events from ET.	Power of 2. Defaults to 4096. Min is 128.
single		out	If true, each evio event sent out in single ET buffer, else evio events marshalled into 1 ET buffer.	Case indep. "true", "on", or "yes". Anything else is false. Default = false
revBuf		both	ET consumer's TCP receive buffer byte size.	Integer >= 0. Value of 0 gives operating system default. Defaults to 0.
sendBuf		both	ET consumer's TCP send buffer byte size.	Integer >= 0. Value of 0 gives operating system default. Defaults to 0.
noDelay		both	ET consumer's TCP NODELAY value.	Case indep. "true", "on", or "yes". Anything else is false. Default = false.
group		out	Group from which to obtain new (unused) events.	Integer > 0. Defaults to 1.
chunk		both	How many events to get at once (in an array).	Integer > 0. Defaults to 100.
stationName		in	Name of station to attach to and to create if non-existing.	Any string with < 48 characters. If inchannel, defaults to **station**<**id**>. If outchannel, GRAND_CENTRAL is used.
position		in	Set position of existing or created station.	Integer > 0. Defaults to 1. GRAND_CENTRAL has reserved position of 0.
idFilter		in	Station only accepts events from component with this coda id**.**	Case indep. "on", everything else is off. Default is off.
controlFilter		in	Allow only control events into station. Overrides idFilter.	Case indep. "on", everything else is off. Default is off.

Section 3

3. Modules

Modules are the heart of an EMU and perform all the data processing tasks. While modules can theoretically be written by anyone, in practice it is not a simple thing to do and is best left to the Jefferson Lab DAQ group. This chapter covers the modules written and supported by the DAQ group. Although it is possible to string multiple modules together in a single EMU, this capability is currently unneeded. Each EMU used in the DAQ system contains a single module.

Modules are implemented as objects created from dynamically loaded Java classes. One benefit of this design is that module behavior can be changed in fully operational CODA DAQ systems without stopping to recompile and restart the software, facilitating quick software development. During the download transition, all existing modules are removed and new ones are created. Thus all the user needs to do is to modify any module code, compile it, then use run control to issue a download command. The newly modified module will be used automatically.

The JLAB DAQ group provides the modules necessary for creating a functional data acquisition system. These include modules to implement: 1) an event builder, 2) an event recorder, 3) a simulated ROC, 4) a simulated trigger supervisor, and 5) a farm controller.

3.1. Config File

Undoubtedly the reader already knows that an EMU's configuration file has 2 sections. The previous chapter dealt with the first section on transports. This chapter deals with the second section on modules. The xml element used is not surprisingly named ***modules***. It is used to configure all the modules used in a single EMU. The following is an example from a config file for an event builder.

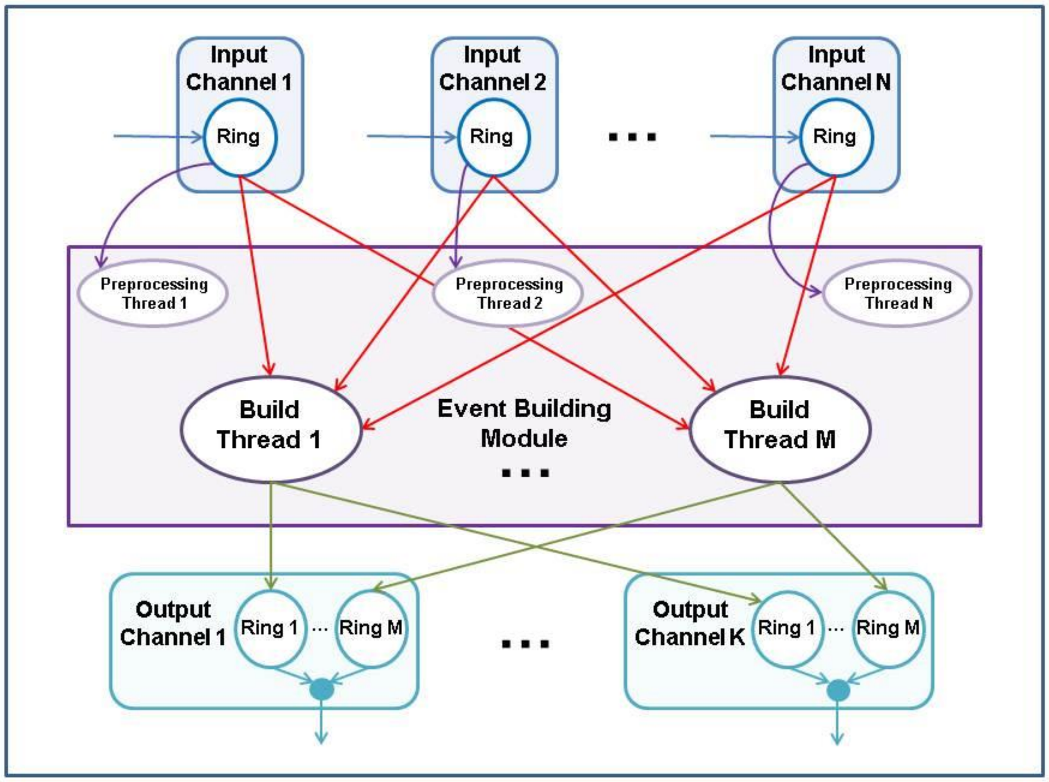
```
<modules>
  <EbModule class="EventBuilding" id="2" timeStats="off" runData="false"
                tsCheck="true" tsSlop="2" sparsify="false" >

    <inchannel  id="0" name="Roc1" transp="inET " idFilter="on" />
    <inchannel  id="2" name="Roc2" transp="inET " idFilter="on" />
    <outchannel id="4" name="Er1"  transp="out_ET " group="1" chunk="5 />
  </EbModule>
</modules>
```

The classes used to implement the supported modules are all included the emu jar file, emu-2.6.jar. This jar file must be in the user's CLASSPATH environmental variable in order for Java to find them. If a user-supplied class is being used instead, its full name must be specified in the class attribute in the xml element defining the module (***EbModule*** in the example above). The jar file which contains it must also be in the user's CLASSPATH.

Under the single "modules" element are the elements for each of the modules to be loaded. Modules' element names (EbModule in the above example) are used only for readability and may be set by the user to any string. Attributes for each module are what determines its behavior and depend entirely upon the module itself. Currently, all DAQ-provided modules only have ***inchannel*** and ***outchannel*** subelements to determine its input and output data channels.

3.2. Event Building

The event building module implements all the functionality of an event builder. This module can function as a data concentrator (DC) which is the first level builder of a 2-stage event building. It can function as a stage event builder (SEB) which is the second level of a 2-stage event building. It also can function as a primary event builder (PEB) which is a single, stand-alone event builder that does a complete build. A rough outline of this module's internal structure is given in the figure below.



Everything inside the center purple box in the figure above represents the event building module. Rings are fast circular buffers which take the place of queues used in previous versions of this software package. This module creates one preprocessing thread for each input data channel. This thread does some bookkeeping for ROC raw, physics, control and user evio format data. It also places user events directly into the first output channel since they don't need to be built.

If this module is used to build ROC raw events, each ROC will have its own input channel. If it's used as a second-level event builder, each first-level EB will have its own input channel. In any case, a single event from each input channel will be needed to build a complete event.

This is accomplished by the building threads - the number of which is determined by the config file and defaults to 1. Each thread will grab an event from each channel's ring, build them into a single event and place that built event on an output channel's ring.

If this module is a DC with multiple SEBs for output channels, then seb chunking is implemented in which a fixed number of sequential or contiguous events are all sent to one SEB before switching to the next in round-robin form. By default this chunk is set to 1. For details of seb chunking go here. If on the other hand this is not such a DC, then if more than one output channel exists, the events are placed in output channels in round-robin fashion.

User events are not built but simply passed on to the first output channel. Any control events must appear on each input ring in the same position. If not, an exception is thrown. If so, the control event is passed along to all output channels. If no output channels are defined in the config file and no additional modules follow it, this module discards all built events.

The user should be aware that this module can act as an event recorder in addition to its ability to build. Simply set its output channel to be a file and the deed is done.

Some of the details of the event building can be controlled through the configuration file. Following is a list of the module's attributes that can be set:

Attribute	Required	Function	Allowed Value(s)
id		CODA id of event builder	Any non-negative integer. Defaults to 0.
class	✓	Name of Java class defining this module.	Must be exactly "EventBuilding".
threads		Number of event building threads.	Any positive integer. Defaults to 1.
endian		Endianness of output data.	Set to case indep. "little" for little endian, else big. Default is big.
sebChunk		If DC, set number of sequential evio events to sent to each SEB before switching to next.	Any positive integer. Defaults to 1. Only used if DC and multiple output channels (SEBs).
repStats		Does this module's stats accurately represent the whole EMU?	Case indep. "false", "off", or "no". Anything else is true. Default = true.
timeStats		Make histogram of time to build single events (in nsec). Printed in console during END transition.	Yes, true, on (case insensitive) to make histogram. Default is off since it kills performance.

ringCount		Number of internal, reusable ByteBuffers in which to place built events.	Must be power of 2 since implemented with fast ring. Default 128, min 32.
tsCheck		Check consistency of timestamps. Throw exception if inconsistent.	No, false, off (case insensitive) to not check. Default is on.
tsSlop		Maximum allowed differences (slop) in timestamps in ticks.	Any positive integer, ignores other values. Defaults to 2.
sparsify		If on, do not include roc-specific segments in trigger bank.	Yes, true, on (case insensitive) to sparsify. Default is off.
runData		Include run number and run type in built trigger bank.	Yes, in, true, on (case insensitive) to include data. Default is off.

3.3. Event Recording

One benefit of abstracting out the data communication from the modules is that the event recorder becomes trivial to implement. All the real work is done in the I/O transports and channels. The event recording module simply funnels all input events into all of the output channels. Each incoming event goes to all output channels. Note, however, any events arriving prior to the prestart event are thrown away. The only exception to this is that any “first events” arriving before prestart are stored and written after prestart. Previously the only parameter the user could tweak was the number of threads which grab events from the input channel and send them to the output channels. Now, however, it is set permanently to 1.

Following is a list of the module's attributes that can be set:

Attribute	Required	Function	Allowed Value(s)
id		CODA id of event recorder	Any non-negative integer. Defaults to 0.
class	✓	Name of Java class defining this module.	Must be "EventRecording" which is the Java class name.
threads		Number of event recording threads.	Currently set to 1.
repStats		Does this module’s stats accurately represent the whole EMU?	Case indep. "false", "off", or "no". Anything else is true. Default = true.

3.4. ROC Simulation

This was written merely for testing purposes. In cases when an actual data-producing ROC is unavailable, this module will provide an EMU-based ROC which generates ROC raw records. Having such a ROC allows testing of CODA components downstream such as event builders and event recorders.

The data in each record/event are all zeros except the very first word which is the event number.

In order for any simulation to work properly, all ROCs must have sent the same number of events when the run control commands to end or reset are given. This is done by having ROCs sync with each other through a simulated Trigger Supervisor every 100k events through sending cMsg messages - a poor man's trigger. For more detail, go here. ROC emus are automatically set to sync by means of a TriggerSupervisor emu. If you wish to run without the simulated TS, add

```
sync=”off”
```

to the ROC module’s config file or else your ROCs will sit around all day waiting to communicate with the TS.

There are some parameters the user can control in the following table:

Attribute	Required	Function	Allowed Value(s)
id		CODA id of ROC	Any non-negative integer. Defaults to 0.
class	✓	Name of Java class defining this module.	Must be "RocSimulation" which is the Java class name.
threads		Number of data generating threads.	Integer between 1 and 20 inclusive. Defaults to 5. Values < 1 get set to 1, over 20 get set to 20.
triggerType		Trigger type from trigger supervisor.	Integer between 0 and 15 inclusive. Defaults to 15. Negative values set it to 0, over 15 gets set to 15.
detectorId		Id of detector producing data in data block bank.	Integer >= 0. Defaults to 111. Negative values set it to 0.
blockSize		Number of events in one (entangled) data block.	Integer between 1 and 255 inclusive. Defaults to 1. Values < 1 set it to 1, over 255gets set to 255.
eventSize		Number of bytes in a single event	Positive integer. Defaults to 40, min = 1
syncCount		Number of writes of a single, entangled block of evio events, before syncing with other simulated ROCs.	Positive integer. Defaults to 100k, min is 10.
sync		Do we sync this with other simulated ROCs?	Case indep. "false", "off", or "no". Anything else is true. Default = true.

3.5. *Trigger Supervisor Simulation*

This was written merely for testing purposes. In cases when actual data-producing ROCs are unavailable, this module will provide a trigger for EMU-based ROCs. Having such a system allows testing of CODA components downstream such as event builders and event recorders.

In order for any simulation to work properly, all ROCs must have sent the same number of events when the run control commands to end or reset are given. This is done by having ROCs sync with each other through this simulated Trigger Supervisor every 100k events by sending cMsg messages - a poor man's trigger. For more detail, go here.

The TS config file must contain a list of ROCs to synchronize as attributes as in the example below:

```
<modules>
    <TsModule class="TsSimulation" r1="Roc1" r2="Roc7" />
</modules>
```

Starting with r1 and going up sequentially in number, each ROC name is identified. These will be the ROCs that will be synchronized to each other. The following are parameters that can be set:

Attribute	Required	Function	Allowed Value(s)
id		CODA id of trigger supervisor	Any non-negative integer. Defaults to 0.
class	✓	Name of Java class defining this module.	Must be "TsSimulation" which is the Java class name.
rN	✓	Identify all ROCs to sync.	"r" followed by a sequential number (starting with 1) equals ROC's name. Use as many times as there are ROCs (e.g. r1="Roc1" r2="Roc7" r3="myRoc").

3.6. *Farm Controller*

This is a simple emu designed to work with an ET system for input and to pass events right through much as an ER. The farm controller is set up to consume only control events from its input channels (hopefully only one channel) and pass it through to all output channels (also hopefully only one). The ET input channel needs to have the attribute:

```
controlFilter="on"
```

in order to accept only control events. The physics events are all handled by the farm nodes.

Attribute	Required	Function	Allowed Value(s)
id		CODA id of farm controller.	Any non-negative integer. Defaults to 0.
class	✓	Name of Java class defining this module.	Must be "FarmController" which is the Java class name.

Section
4

4. Running an EMU with Run Control

Figure 4.1 A running emu

Emus run pretty fast. When they reach their top speed we say they're smokin'.

Figure 4.2 A smokin' emu

The running of an EMU is very easy. Configuring it with ***jcedit*** may be challenging, but actually running it is simple. Although several can be run in a JVM simultaneously, in CODA only 1 emu is ever run in a single JVM

4.1. Config File Final Form

Each configuration file may specify a single EMU to be run in a single JVM. In outline form it looks like:

```
<?xml version="1.0"?>
<component name="EB" type="SEB" >
   <transports>
         ...
   </transports>
   <modules>
         ...
   </modules>
</component>
```

The previous 2 chapters explain in some detail the nature of the xml entries under transports and modules. The ***component*** element has 2 attributes, ***name,*** which must be present and unique in run control, and also ***type*** which is the CODA type and is optional (see Appendix A for values allowed).

4.2. Creating EMUs

EMUs are created using the ***EmuFactory*** class which does the work of sorting through command line arguments, reading and parsing config files, and starting up all the EMUs that are indicated in these arguments. To run it execute,

```
java  org.jlab.coda.emu.EmuFactory
```

where the following options are allowed and the "-D" options need to precede the class name:

Argument	Required	Function
-h or -help		Print help.
-Dname		Name of CODA component to create. Can be list of names separated by ",", ":", or ";" for multiple components.
-Dtype		Type of CODA component to create. See Appendix A. Can be list of types corresponding to list of names separated by ",", ":", or ";" for multiple components.
-DcmsgUDL	✓	cMsg UDL used to connect to the cMsg server the AFECS platform is running.
-Dexpid		Set the experimental id, overriding environmental variable EXPID in EMU. If not set on cmd line or in env.var., default to "unknown". Used to generate default ET system name (/tmp/<expid>_<emuName>). Used to construct default UDL (rc://multicast/<expid>) to connect to platform cMsg server ***if cmsgUDL not defined***.
-Dsession		Set the experimental session. Used to generate output file names. Run control, if being used, sets the session, so this is useful only when using the EMU with the debug gui.
-DDebugUI		Start up a debug gui to run EMU without run control.
-Duser.name		Arg passed to EMU objects which set user's for use in error messages.

4.3. Platform connection

In order to place an EMU under run control, it must be able to connect to the desired AFECS platform. This is done using the ***cmsgUDL*** option to specify the run control domain cMsg server that's running inside the platform. That sounds complicated, but in practice the user only needs to type:

```
java -DcmsgUDL="rc://multicast/<expid>" org.jlab.coda.emu.EmuFactory
```

where **<*expid*>** is replaced by the user's run control experiment id. This enables the necessary communication. If the EMU is run but the platform is down, the EMU will continue to try to connect until the platform comes up and it succeeds.

4.4. Running a single EMU

The only way to configure an EMU is by using the ***jcedit*** program to create config files which are then passed to components by run control during the ***configure*** transition. However, the user must provide the EMU's name on the command line, otherwise there is no way for run control to know which configuration to send it. In addition, it is also necessary to specify the component type on the command line. Although it is possible to specify the type in the config file, jcedit currently does not do it. Do something like the following to use run a single emu:

```
java -Dname=Eb -Dtype=SEB -DcmsgUDL="rc://multicast/myExpid"
     org.jlab.coda.emu.EmuFactory
```

4.5. Running Multiple EMUs

It is possible, for whatever reason, to run multiple EMUs in a single JVM. To do this the names and types are specified as single strings with the individual components separated by either colons, semicolons, or commas but ***not*** white space. Here's an example:

```
java -Dname=Roc,Eb,Er -Dtype=ROC,PEB,ER -DcmsgUDL="rc://multicast/myExpid"
     org.jlab.coda.emu.EmuFactory
```

This will run 3 EMUs in 1 JVM – a fake ROC, an EB and an ER.

Section

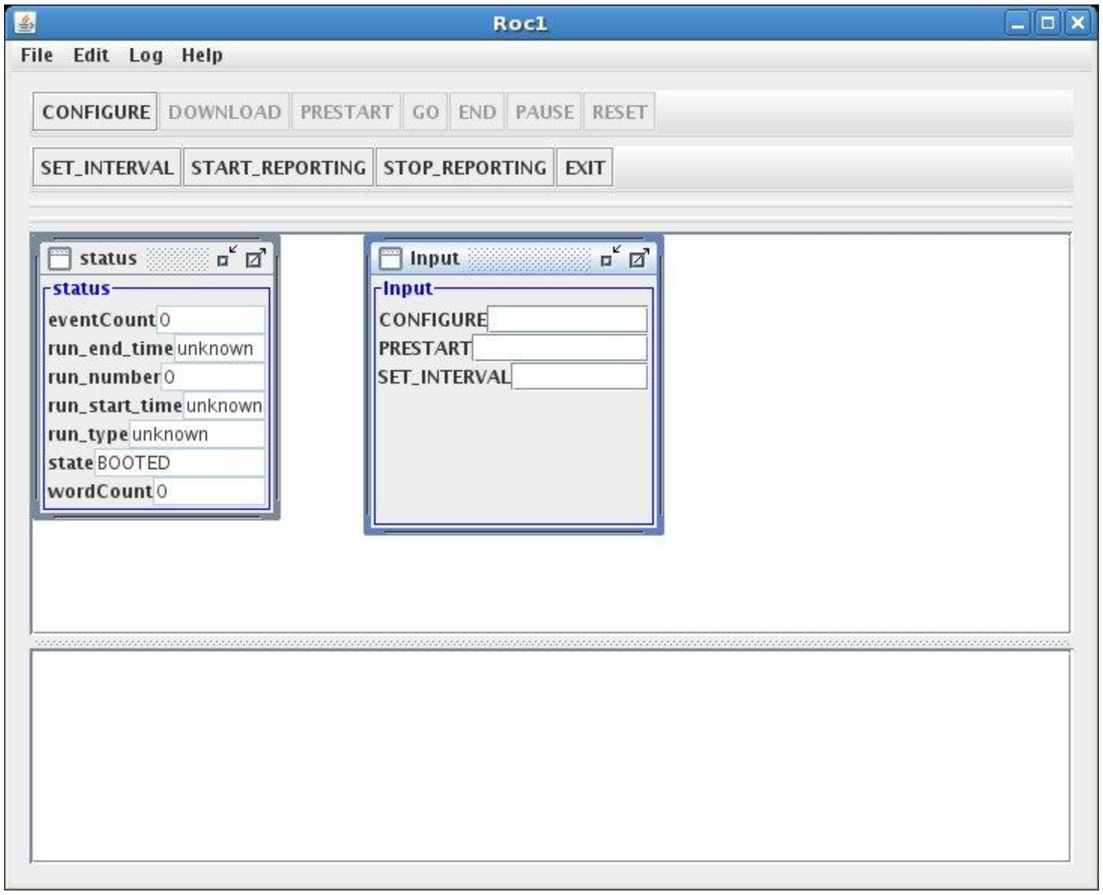
5

5. Running an EMU with the Debug GUI

EMUs can be run standalone - without run control - which may not appear to make much sense on the surface of things since they are designed to respond to run control commands. However, there is a gui used for debugging which is part of the EMU and can be run by specifying ***-DDebugUI*** on the command line:

```
java -Dname=Roc1 -DDebugUI org/jlab/coda/emu/EmuFactory
```

This gui allows the user to send locally generated run control commands to the EMU and see its output. It looks like:



status

eventCount	0
run_end_time	unknown
run_number	0
run_start_time	unknown
run_type	unknown
state	BOOTED
wordCount	0

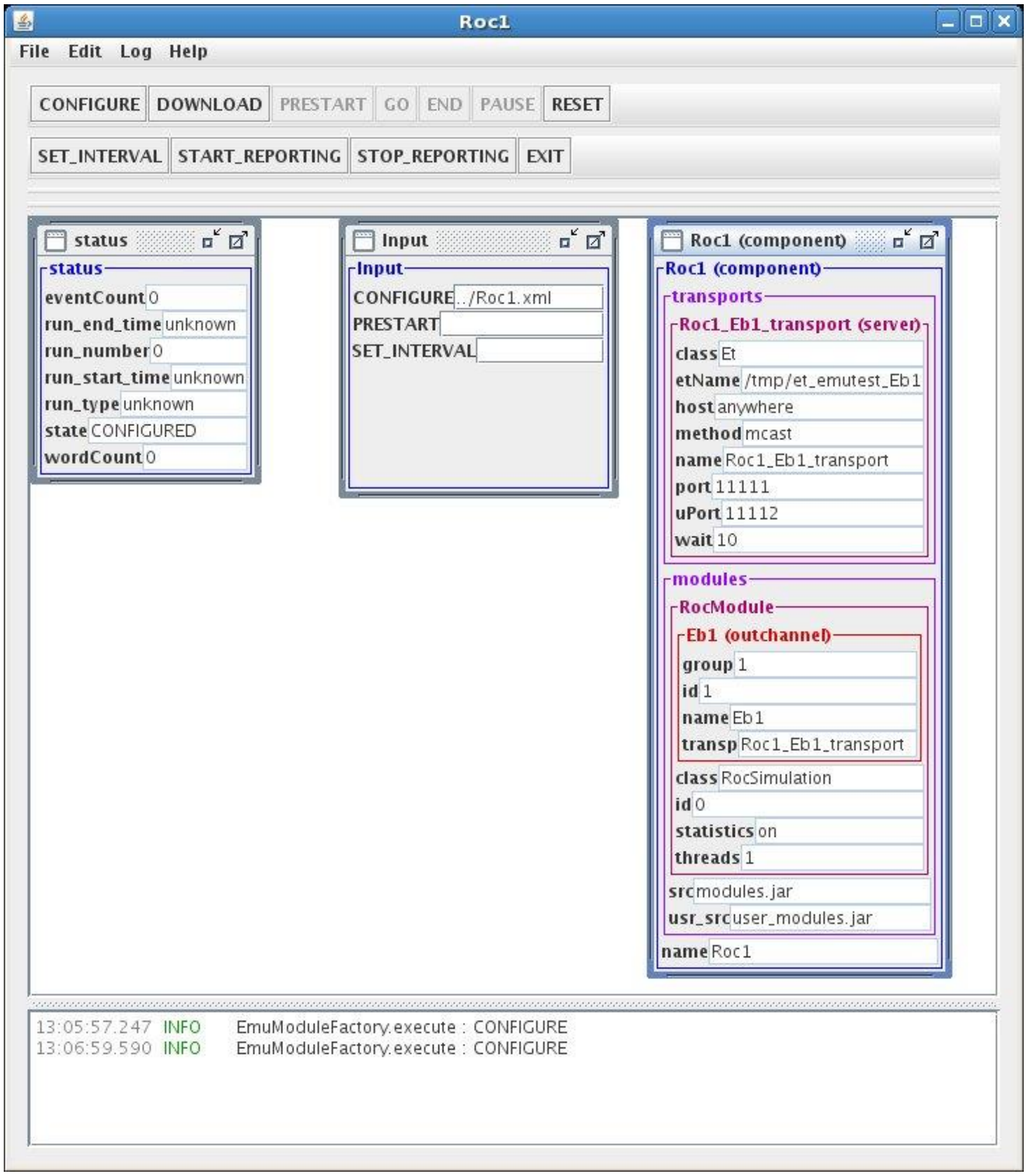
Input

CONFIGURE	
PRESTART	
SET_INTERVAL	

The top panel of the application has a number of run control transition-inducing buttons along with a few other run control commands. In the bottom panel is a listing of all the EMU-produced debugging, info, warning, and error messages.

In the middle, the status window keeps track of a few stats and the input window allows user input into the EMU. Type the desired config file name in the CONFIGURE input widget and hit the CONFIGURE button to get it to load that file. The input to PRESTART is the run number. And the input to the SET_INTERVAL sets the time interval between status messages in seconds.

Once the configure button is pressed and the configuration is loaded, a new window appears:


Roc1
File Edit Log Help
CONFIGURE DOWNLOAD PRESTART GO END PAUSE RESET
SET_INTERVAL START_REPORTING STOP_REPORTING EXIT
status
Input
Roc1 (component)

status

eventCount	0
run_end_time	unknown
run_number	0
run_start_time	unknown
run_type	unknown
state	CONFIGURED
wordCount	0

Input

CONFIGURE	../Roc1.xml
PRESTART	
SET_INTERVAL	

Roc1 (component)

transports

Roc1_Eb1_transport (server)

class	Et
etName	/tmp/et_emutest_Eb1
host	anywhere
method	mcast
name	Roc1_Eb1_transport
port	11111
uPort	11112
wait	10

modules

RocModule

Eb1 (outchannel)

group	1
id	1
name	Eb1
transp	Roc1_Eb1_transport

class	RocSimulation
id	0
statistics	on
threads	1

src	modules.jar
usr_src	user_modules.jar
name	Roc1

```
13:05:57.247  INFO    EmuModuleFactory.execute : CONFIGURE
13:06:59.590  INFO    EmuModuleFactory.execute : CONFIGURE
```

The new window simply shows what is in the loaded configuration. In this case there is one ET transport with its name, host, connection method, ports, etc. specified. The module to be loaded is also specified - the ROC simulation module. Under modules, its output channel, the ET system, can also be seen. Notice the EMU's messages in the bottom panel.

If running multiple EMUs in 1 JVM, there will be one gui per component.

Section
6

6. Developer's Details

There are many details concerning the internals of the EMU which are of no interest to the general user. This chapter contains some such detail more as a reminder to the EMU developer. Although comments are spread through the code itself, it's always nice to have more coherent notes on the software that serve to jog the memories that lose much of the finer points over time.

6.1. cMsg Run Control Connection

There's a cMsg connection in the rc domain maintained for receiving and responding to run control commands as well as for sending dalogmsg messages. Generally the rc multicast server being connected to resides in the AFECS platform. The general form of the UDL used to connect to it is:

```
rc://<host>:<port>/<expid>?connectTO=<timeout>&ip=<address>
```

where:

- ***host*** is required and may also be "multicast", "localhost", or in dotted decimal form
- ***port*** is optional with a default of 45200
- ***expid*** is the run control experiment id being used for the experiment currently underway
- ***timeout*** is the time to wait in seconds before connect returns a timeout while waiting for the rc server (in AFECS platform) to send a special (TCP) concluding connect message. Defaults to 30 seconds.
- ***address*** is the IP address in dot-decimal format which the rc server or agent must use when connecting to this rc client. This is sent to the rc server along with its corresponding subnet address. If this is not set, then the rc client sends the rc server all of its IP addresses and subnet addresses.

This UDL may be specified by the user when starting up the emu by giving the following flag to the JVM:

```
-DcmsgUDL=<udl>
```

This is done for the user in the provided emu startup scripts with the value:

```
-DcmsgUDL=rc://multicast/$EXPID
```

which is also the UDL used if not given when starting up an EMU directly.

6.2. Data Flow

In some sense the EMU is like a mini DAQ system in one program. To refresh your memory a traditional DAQ originates data in a ROC, it flows through the EB and eventually finishes with the ER storing it somewhere. To properly shut down such a system, run control first informs the ROC to quit sending data. After which the EB is told to end, and then finally the ER is told to do the same. Each component, however, must wait for run control's end (or reset) event to come through before finally ending.

Similarly, the data flow through an EMU starts with the input channel. It goes through the modules in order and then through the output channel. When an EMU is shut down, the input channels/transports must be the first to end, followed by the modules in order and finally the output channels/transports. All parts must wait for the end (or reset) event to come through before finally ending.

6.3. ET Channels

When the ET system is used for data transport, it is usually done for performance purposes. The ET channel software is, therefore, multithreaded in a way to squeeze every last bit of speed out of it. Next is a brief description of both the input and output ET communication channels.

6.3.1. Output

Why not start with the most difficult part first? The ET output channel uses the interior class, DataOutputHelper, to do all the work. It uses a fast ring buffer (see the next chapter) to do the coordination between several threads.

There is a single thread which gets new events from ET and places each into one of the container entries of the ring. The DataOutputHelper's main thread then claims an entry from the ring, grabs as many evio events coming from the module that will fit into that ET event, and stores them in that entry as well. Note that control and user events have their own ET events. Then 2 concurrent writing threads each claim different entries and write the evio events into the ET buffers. Finally, the putter thread puts the full ET events back into the ET system.

If one of the evio events is the END event, all threads shut themselves down as it passes through. A reset command just stops all these threads.

6.3.2. Input

ET input is a little simpler than the output. The input channel uses the interior class DataInputHelper to do all the work with only 1 thread to do everything. This apparently works well because in actual use, the input ET systems are always on the local host and so there's little I/O cost.

The input thread gets an array of ET events to begin with. Each ET event is, in turn, copied into a ByteBuffer from a fast, ring-based supply of reusable buffers. Because each parsed evio event contains a reference to the buffer it was parsed from, the copy is necessary to prevent problems when the ET event is put back and eventually reused – thereby changing the underlying buffer's data. Each data buffer is parsed into evio events which are put into the channel's ring. The ring will be used as input to the first module. When all the ET events are parsed, they are put back into the (local) ET system.

If an END event appears, it is placed on the ring, all ET events are returned and the thread exits.

6.4. cMsg Channels, cMsg domain

6.4.1. Output

The output channel uses the interior class DataInputHelper to do all the work with 1 thread to do the dispatching and 2 threads, by default, to do all the writing. The number of writing threads can be set in the configuration by setting the wthreads attribute in the outchannel element. The dispatching thread grabs evio events from the module and stores them in ArrayLists (one for each writer) until either the count exceeds 1000 or the total memory exceeds 256KB in which case the lists are passed off to the write threads (control and user events get their own cMsg message). While the dispatching thread waits, the write threads will write the evio events into cMsg messages' byte arrays. When done, the dispatching thread will send the messages. This is not the fastest way to do things but that's not important if using cMsg to begin with.

6.4.2. Input

cMsg input is a little simpler than the output. The input channel has a subscription to a particular subject and type. The subject and type can be set as attributes in the configuration's inchannel element. If not explicitly set, the subject defaults to the name (attribute) of the inchannel and the type defaults to "data". Each time a message arrives, the subscription's callback is run and it parses the message's byte array and places the resulting banks on the channel's ring. The ring will be used as input to the first module.

If an END event appears, it is placed on the ring. The callback does nothing else, but the emu unsubscribes and then disconnects when the END or RESET commands reach the channel and the transport object.

6.5. cMsg Channels, emu domain

6.5.1. Output

The output is handled by a single DataInputHelper thread that grabs evio events from the module and writes them into a single internal buffer which is written to the socket when it becomes full or it's a user or control event.

6.5.2. Input

The input is handled by a single thread which reads a socket. The first int is the command, the second the size, and then comes the evio file format data. A ByteBuffer is obtained from a fast, ring-based supply and the data is read into it. The data is parsed and evio events are placed into the ring. This just loops until the command is that the data is at an end, then the thread exits. Very simple compared to other channel types.

6.6. SEB Chunking

The situation may arise in which there are multiple SEBs in a single configuration. In this case, the DCs must be capable of directing its output so that each SEB gets a chunk of contiguous events in turn. The logic is a little tricky so it's outlined here so looking at the code will hopefully not be too confusing. It's easiest to start by looking at the figure below:

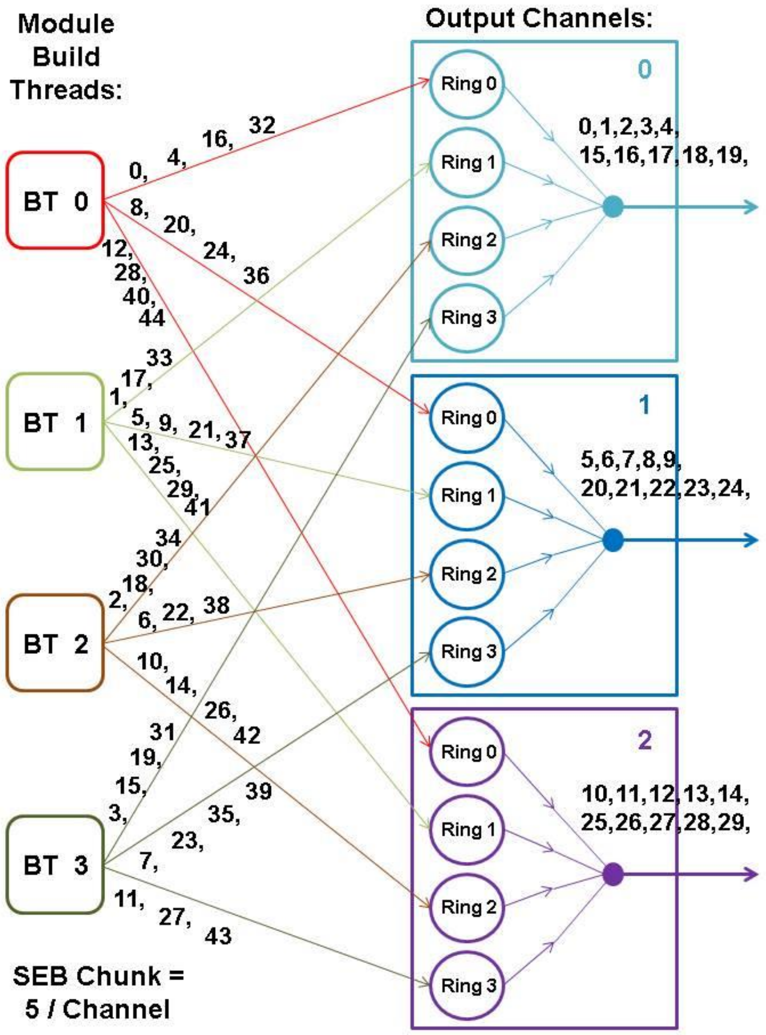


Figure 6.1 SEB chunking

The figure shows an example of the DC having 4 build threads. The SEB chunk is set to 5 meaning that 5 contiguous events are sent to each SEB in turn. There are 3 SEBs and therefore 3 output channels with each channel going to a single SEB. The way the EB modules work is that each successive build thread takes the next event to build with thread 0 taking the first. For example, thread 0 builds events 0, 4, 8, 12, 16, etc. Thread 1 builds events 1, 5, 9, 13, 17, etc. Thread 2 builds events 2, 6, 10, 14, 18, etc. And finally, thread 3 builds 3, 7, 11, 15, 19, etc.

The communication channels use fast circular (or ring) buffers instead of queues to minimize latency with each output channel having 1 ring per build thread. This is done to speed things up by eliminating any contention for ring usage – each has only a single supplier. The channel then reads from each ring in turn in order to send events on their way.

So then, what is the algorithm used by each build thread to decide which channel to send a particular event to?

```
channel_number = event_group_# % #_channels
```

where the event group number is defined to be the order (0 first) of the particular seb-chunk containing the event of interest. The event group number is found by:

```
event_group_# = event_# / seb-chunk
```

Make the substitution and get

```
channel_number = (event_# / seb-chunk) % #_channels
```

or in this case

```
channel_number = (event_# / 5) % 3
```

Note that all quantities are integers and normal integer arithmetic is assumed. So, for example, event # 17 should go to channel (17 / 5) % 3 = 3 % 3 = 0. Event 17 goes to channel 0 which can be confirmed by looking at the figure. Likewise event 29 goes to (29 / 5) % 3 = 5 % 3 = 2 which is also consistent with the figure.

What about the internal workings of a single channel? Since there are multiple build threads (and therefore rings), which event is read next and on which ring does it show up? The following is the algorithm used to determine that. It depends on the seb chunk, the channel number, the number of output channels, and the number of build threads.

```
// The output channel we’re using right now (0 = first)
channel_number = 1
chunk_counter = seb-chunk

// First event to be read on this channel
next_event = channel_number * seb-chunk
// First ring to read from
ring_number = next_event % output_channel_count
// First read
read_from(ring_number)

while {
    // If true, reading seb-chunk contiguous events
    if (--chunk_counter > 0) {
        next_event++
        ring_number = next_event % output_channel_count
    }
    // After reading seb-chunk contiguous events, hop over
    // 2*seb-chunk that will be read by the other channels
    else {
        chunk_counter = seb-chunk
        next_event_# += seb-chunk * (output_channel_count – 1) + 1
```

```
            ring_number = next_event % output_channel_count
        }

        // Read from ring
        read_from(ring_number)
    }
```

If you take the time to check out this algorithm with actual numbers, it will produce the pattern in the figure. To find it in the actual code, it's contained in the ***setNextEventAndRing*** method of the ***DataChannelAdapter*** class.

6.7. Simulated ROCs and TS

In cases when it's necessary to test a DAQ system but actual data-producing ROCs are and trigger supervisors are unavailable, EMU-based ROCs and TS's can be used. In order for such a simulation to work properly, all ROCs must have produced the same number of events when the run control commands to END or RESET are given, otherwise the EB will throw an exception. This is done by having ROCs sync with each other through the simulated Trigger Supervisor after every fixed number of events (100k by default) by sending cMsg messages - a poor man's trigger.

6.7.1. Trigger Supervisor

The TS is given a list of expected ROCs to sync in its configuration file. Using the platform's cMsg server (cMsg domain), it subscribes during prestart in the "RocSync" namespace to:

```
subject = "syncFromRoc",   type = "*"
```

This subscription's callback receives messages from the ROCs. From each message it gets the ROC's name from the "type" and whether that ROC has received the END command from run control or not from the "user int" (1 if END received, else 0). If all expected ROCs send such a message, then the callback sends a message back to all ROCs with:

```
subject = "sync",   type = "ROC"
```

If all ROCs have indicated that they have received the END command, then this message is sent with a user int of 1 meaning "stop sending events", else it's sent with a user int of 0 meaning "send the next batch of events". It then waits for another round of messages from the ROCs.

6.7.2. ROCs

In prestart, each simulated ROC connects to the platform's cMsg server and subscribes in the "RocSync" namespace to:

```
subject = "sync",   type = "ROC"
```

Once the GO command is received from run control, the ROC will start writing events. Once it has reached the predetermined limit (100k by default), it will stop and send a message to:

```
      subject = “syncFromRoc”,   type = “<roc name>”
```

where <roc name> is its actual name. If it has received the END command from run control, the message’s “user int” is 1, else it’s 0. Then it waits for an incoming message from the TS. The TS responds when all ROCs have finished their batch of events and sent their message. If that TS message contains a “user int” of 1, then the ROC quits producing events and ends, otherwise it writes another batch of events.

The END command actually gets blocked and waits until the writing thread gets the TS message to end things. At that point the writing thread allows the END command to proceed.

It is probable that each ROC receives the END command at a different point in its production of events. Yet this system allows all ROCs to end after having produced the exact same number of events. Say for example, that in a 2 ROC configuration, the first ROC got an END after 295k events. It stopped after 300k, sent its message to the TS and waited for a response. Roc2, on the other hand, stopped at 300k but had not yet gotten an END. It also sent its message but indicated that it had no END and then waited for the TS response. The TS got the first ROC’s message, but was waiting until the second also reported. When the TS had received both messages, one had not yet received an END so the TS told both to write another round of events. The second ROC eventually got an END but at event 310k. Thus when both stopped producing at 400k, the TS received their messages and realized that both had gotten the END and told them to end their event production. In this way all ROCs end up having produced the identical number of events and the EB will end nicely.

6.8. Evio Events Per ET Buffer

Another largely hidden piece of controlling the data flow is instituted to avoid mismatches in the number of evio events coming from each ROC to the EB. ROCs with large amounts of data will send fewer evio events in an ET buffer than will those with little data. What we don’t want is for ROCs with only a little data to wait until its 2MB ET buffer is full before sending while in the meantime the EB has already received ET events from ROCs with big events. This keeps the EB waiting when it could be building.

To facilitate faster event building, all ROCs receive feedback from either the SEB or PEB (which ever is being used) in the form of cMsg messages, telling them the maximum number of evio events to fit into a single ET buffer before sending. The trick is to pick this number dynamically based on current DAQ conditions which will ensure good data flow.

Currently it works as follows. The SEB or PEB emu connects to the platform’s cMsg server in the cMsg domain in the namespace “M” and creates a subscription to:

```
    subject = <emu name>,   type = “*”
```

The associated callback receives messages from the ET input channels containing the number of evio events in each ET buffer or M. This callback finds the lowest and highest M values from all reporting channels in order to send that number to the ROCs on the other end of the input channels. It also sends the highest safe value of M (a value that will allow the ROCs to operate without using the timeout to send data) which is calculated to be 2 x lowest-M. This feedback message is sent to the ROCs only if the highest safe value changes and no more than once every 2 seconds. Furthermore, once the feedback is sent, all M values return to their initial values and are recalculated. This keeps the feedback up to date within 2 seconds.

The ET input channels meanwhile report their current M value by publishing a cMsg message to:

```
subject = <emu name>,   type = "M"
```

with the user int set to M. They report no more than every ½ second in order to keep from overloading the cMsg server.

The feedback messages sent to the ROC are sent to the platform's cMsg server in the namespace <expid> with:

```
subject = <emu name>,   type = "eventsPerBuffer"
```

The highest safe M is sent as the user int while the low M and high M are sent as integer payload items called "lowM" and "highM" respectively.

Section
7

7. Fast Ring Buffers

Strictly speaking this chapter is really a part of the previous one about developer's details, but the fast ring buffers which are used throughout the emu deserve their own chapter. The folks at LMAX wrote a software package implementing fast ring buffers called the "**Disruptor**" which we'll be taking a close look at. For the moment, however, look at the figure in section 1.2 which shows queues between input channels and modules. A channel or module grabs an event off a queue, processes it, and places it on another queue. In the EB, for instance, there were 3 queues that each event spent time in. Originally, the queues used in the emu were of the type built into the Java language, like the ArrayBlockingQueue. Using this class was convenient and easy, but when the performance of the emu was profiled, 40% of the time the emu was putting stuff on and taking stuff off of queues – not good. What is it about queues that are such a problem? Why are they so slow?

7.1. Locks are Bad

Let's look at an important component of queues – locks. Locks provide a thread-safe way to read and write common data, but are very expensive to use. When there is contention for a lock, the operating system must arbitrate resulting in context switching and suspended program threads waiting for the lock. While in control, the OS may decide to do other tasks during which the original context may lose cached data and instructions.

To demonstrate these effects the programmers at LMAX called a function which incremented a counter in a loop 500 million times under different conditions.

Method	Time (ms)
One thread	300
One thread with lock	10,000
Two threads with lock	224,000
One thread with CAS	5,700
Two threads with CAS	30,000
One thread with volatile write	4,700

Once locks are used, even uncontested, performance is 30x worse. With contention, it's 750x worse! Instead of using locks, atomic CAS or Compare-And-Swap operations can be used if the item to be updated is a single word. It’s a much better approach since it does not require kernel arbitration and a context switch, but the processor must still lock its instruction pipeline to ensure atomicity and use a memory barrier to make the changes visible to other threads. The downside is that using only CAS operations and memory barriers to protect data beyond a simple counter is extremely difficult. Less costly in Java is the reading or writing of a volatile field which uses read and write memory barriers respectively.

7.2. Cache Lines

Let’s look at one more subject of importance in queue performance – cache-lines. Hardware doesn’t move memory around in bytes or words but in cache-lines that in linux are 64 bytes. What this means is that if two variables are in the same cache-line, and they are written to by different threads, they result in the same problem of write contention as if they were the same variable. This is known as “false sharing”. For minimal contention and therefore best performance, it is necessary that independent, but concurrently written, variables do not share the same cache-line.

7.3. The Trouble with Queues

The bounded queues originally used in the emu have write contention at the head, tail, and size variables. If there is more than one producer (the case for EB build threads writing to output channels), the tail pointer will be the point of contention as more than one thread wants to write to it. If there’s more than one consumer (the case for EB build threads reading from input channels), then the head pointer is contended since this is not just a read but also a write as the pointer is updated when the element is consumed.

In actual use, queues are almost always full or empty since producers and consumers never run at the same rate. This translates into high levels of contention as the head and tail are the same pointer. And to top things off, the head, tail and size are often in the same cache-line resulting in false sharing. Okay, there’s one more thing. In Java the queues are a constant source of garbage. Objects are created, stored on the queue, removed and eventually discarded – nothing is recycled. If the queue is linked-list based, objects representing the nodes of the list need to be created and eventually reclaimed.

7.4. Disruptor Design

The LMAX disruptor is designed to address all the issues just mentioned. First off the memory usage is much better. All memory for the ring buffer is pre-allocated on startup. The ring is populated with an array of permanent objects or entries (therefore not garbage collected) that can contain data of interest. Allocating memory in this way allows traversal of the entries to be done in a very cache-friendly manner.

Since there is neither head nor tail, both consumers and producers track their own place, or sequence, as they go around the ring. Producers claim the next slot in sequence when claiming an entry in the ring. Since the emu only uses rings with one producer, this sequence of the next available slot is a simple, uncontested counter. Once a sequence value is claimed, its corresponding entry in the ring is now available to be written to by the claiming producer. When the producer is done with the entry, it can commit the changes by updating a separate counter which represents the cursor on the ring for the latest entry available to consumers. This it can do by using a memory barrier which is done by a volatile write in Java – no lock, no CAS. For better performance, the cursors are padded in such a way that false sharing never occurs.

Consumers, on the other hand, wait for a sequence to become available by using a volatile read of the cursor. Various strategies can be used to wait, but the fastest and the one used in the emu is a busy spin loop check of the cursor. Consumers each contain their own sequence which they update as they process entries from the ring. These sequences allow the producer to track consumers in order to prevent the ring from wrapping, and they also allow other consumers to coordinate work on the same entry.

So we see that the disruptor ring buffers use no locks or CAS, minimize contention, and are cache-friendly. All concurrency is achieved using memory barriers.

An added benefit from this design is that when consumers are waiting for the next available sequence, the sequence actually read may be several entries beyond the next one. Thus only one cursor read is necessary for the processing of several entries. This type of batching increases throughput while reducing and smoothing latency at the same time.

7.5. Disruptor Use in the EB

As an example of ring buffer use, consider the figure below which represents the ring buffer of one of the event builder's input channels.

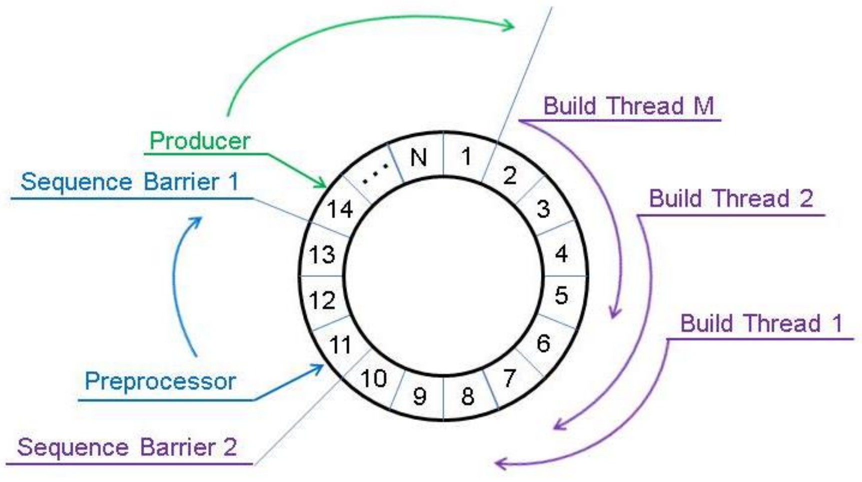


Figure 7.1 EB Input Channel Ring Buffer

In this case, the producer is an input channel. It starts with entry 1 and keeps placing evio events into the entries as data flows in and eventually ends up working on entry/event 14. When done with event 13, it updates its cursor saying that event 13 is available for consumers. That cursor is represented by "sequence barrier 1".

The preprocessing thread is the first consumer. It reads the coda id of the sending coda component from the data and sees if it matches what is expected from the configuration (among other things). It starts consuming events as they become available and is currently working on event 11. When done with that it will read the value of sequence barrier 1 which is 13 and will be able to move to 12 and after that to 13 without reading the cursor again, and it will also update its sequence seen as "sequence barrier 2" in the figure. Just as the first barrier indicates when the producer is finished with an event, the second does that same thing to indicate that the preprocessor is finished with its event. All the build thread consumers are programmed to wait on the second barrier.

Each build thread takes one event from each input channel in order to build an event. Build thread 1 take event 1, build thread 2 takes event 2, etc. etc. Thus, in this case, build thread 1 will grab event 1, skip events 2 thru M, and go to event 1+M for its second and so on.

In the figure, the input channel cannot produce beyond event 1 (for the second time) since the build threads have yet to release any more.

Prior to using a single ring buffer, the input channel placed parsed evio events onto a queue which the preprocessing thread then used as its input. The preprocessor then placed its output on another queue which the build threads used as their input. All the build threads would contend for the second queue's lock in order to read it. Thus 2 queues were replaced by only one ring buffer.

7.6. General Ring Buffer Use in the Emu

Ring buffers are used in several places throughout the emu:

- Each input channel has one ring
- Each output channel has one ring for each of the preceding module's event-processing threads
- Internal operation of the ET output channel
- The ByteBufferSupply, used to provide reusable ByteBuffers, has one ring and is used by
    - EB build threads
    - Emu input channel
    - ET input channel
    - Simulated ROC

All input channels place their parsed evio events into 1 ring buffer. All output channels have the same number of ring buffers as the last module has event-processing threads (build threads in the EB). Thus each event-processing thread has its own ring buffer in each output channel – one producer per ring. This design eliminates multiple producers and contested writes. One can see this clearly in figure 6.1.

In addition to using rings to store events passing through each emu, rings are also used to create a very fast supply of ByteBuffer objects for use in building events. Although one could create a new ByteBuffer for each built event, it would eventually have to be garbage collected resulting in a tremendous strain on the JVM. By using a ring buffer, a fixed number of ByteBuffers could be pre-allocated and would never have to be garbage collected. This usage of the ring buffer is encapsulated in 2 classes with the following being actual code to create and use such a ring.

```
// Create a supply of ByteBuffers
int ringSize = 1024; // number of buffers
int bufferSize = 256000; // bytes per buffer
ByteBufferSupply supply = new ByteBufferSupply(ringSize, bufferSize);

// Use the supply & double a single buffer size
ByteBufferItem item = supply.get();
item.ensureCapacity(2*buffSize);
ByteBuffer buf = item.getBuffer();

// Release the buffer and allow for its reuse
supply.release(item);
```

Although the actual classes contain more complexity, they can be quite simple to use. The ring contains ByteBufferItem objects each of which contain a single ByteBuffer. In this way, the contained ByteBuffer objects can be replaced easily if more memory is needed.

One complication is that one ET event contains data consisting of several evio events. The input channel will grab a ByteBuffer from the supply and copy the incoming ET data into it. When the evio reader splits these into individual evio events and places them into separate entries of the input channel's ring buffer, they still all contain references to the ByteBuffer from the supply. When the module is finished with an evio event entry, it also needs to release its hold on its containing ByteBuffer. Furthermore, this needs to be done for all evio events taken from that one ByteBuffer for it to be reused in the supply.

7.7. Ring Buffer Example Code

The code to create a ring is:

```
 // Number of ring entries must be power of 2
int ringSize = 1024;

// Define factory to produce objects contained in the ring
final private class RingItemFactory implements EventFactory<RingItem> {
    final public RingItem newInstance() {
        return new RingItem();
    }
}

// Create the ring buffer using a very fast busy-spin waiting strategy
RingBuffer<RingItem> rb = createSingleProducer(new RingItemFactory(),
                              ringSize, new YieldingWaitStrategy);
```

The code to produce data for a ring is:

```
while(true) {
    // Get the next available sequence or ring slot (may block)
    long sequence = rb.next();

    // Get the actual entry at that sequence
```

```
        RingItem entry = rb.get(sequence);

        // Fill entry with data …

        // Release the entry back to the ring buffer for consumers
        rb.publish(sequence);
    }
```

A note on publishing the sequence, it not only releases its corresponding entry to all consumers that follow, but it also releases all previous entries. For example, if a producer does several "rb.next()" and "rb.get()" calls without publishing, publishing the last sequence will also release all prior ones.

The code to consume data from a ring is:

```
    // Object allowing us to wait for producer to be finished
    SequenceBarrier barrier = rb.newBarrier();
    // Object allowing us to tell others where we are
    Sequence sequence = new Sequence(Sequencer.INITIAL_CURSOR_VALUE);
    // If this is the last consumer before the producer,
    // in other words, no other consumers are depending on us going first,
    // then add the following line.
    rb.addGatingSequence(sequence);
    // Keep track of our place in the ring
    long nextSequence = sequence.get() + 1;

    while(true) {
        // Get the next available sequence or ring slot (may block).
        // May be > the sequence asked for (nextSequence).
        long availableSequence = barrier.waitFor(nextSequence);

        // While we have access to available entries …
        while(nextSequence <= availableSequence) {
            // Get the actual entry at the next sequence
            RingItem entry = rb.get(nextSequence);

            // Read entry data …

            // Go to the next ring entry
            nextSequence++;
        }
        // Release all used entries back to the ring
        // buffer for producer or dependent consumers
        sequence.set(availableSequence);
    }
```

The code is rather straight forward. Although not shown here, additional barriers can be added so that the ring understands that certain consumers depend on other consumers going first.

Appendix
A

A. CODA Types

The following is a list of CODA DAQ component types, their default run control priority levels and their descriptions.

CODA Type	Default Priority	Description
TS	1000	Trigger Supervisor
CDS	910	CODA Data Source (simulated, EMU-based ROC)
ROC	900	Readout Controller
DC	800	Data Concentrator (first level event builder)
SEB	700	Secondary Event Builder (2nd level event builder used with DCs)
PEB	600	Primary Event Builder (one and only one event builder)
ANA	500	Analysis Application
ER	400	Event Recorder
SLC	200	Slow Control Component
USR	100	User Component
EMU		Event Management Unit