Version

1.0

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

3-Aug-2012

12000 Jefferson Ave
Newport News, VA 23606
Phone 757.269.5130 • Fax 757.269.6248

Table of Contents

1. Introduction 4
1.1. Input / Output 5
1.2. Modules 5
1.3. Configuring 6
1.4. Monitoring, and Controlling 6

2. Data Input and Output 7
2.1. Transports 7
2.1.1. FIFOs 7
2.1.2. Files 7
2.1.3. cMsg 8
2.1.4. ET 8
2.2. Channels 10
2.2.1. FIFOs 10
2.2.2. Files 10
2.2.3. cMsg 11
2.2.4. ET 12

3. Modules 14
3.1. Config File 14
3.2. Event Building 15
3.3. Event Recording 17
3.4. Disentangling 17
3.5. ROC Simulating 17

4. Running an EMU with Run Control 19
4.1. Config File Final Form 19
4.2. Creating EMUs 20
4.3. Run Control & EMUs 20
4.4. Configuring EMUs 21
4.4.1. With Command Line 21

4.4.2. With Run Control *21*
4.5. Running Multiple EMUs *21*
5. Running an EMU with the Debug GUI **22**
6. Developer's Details **25**
6.1. Data Flow *26*
6.2. ET Channels *26*
6.2.1. Output *26*
6.2.2. Input *27*
A. CODA Types **28**

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 a specific ET system, cMsg server, or 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.

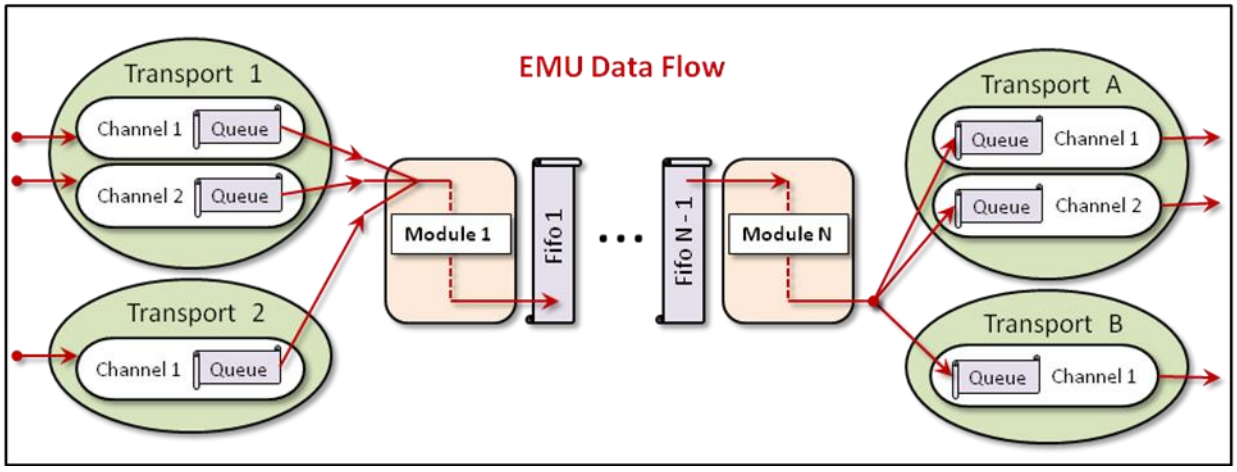
The transport handling code is abstracted from the rest of the EMU in such a way as to be able to add other means of I/O. To document the complete interface to accomplish this is too much work for a task best left to the DAQ group itself. Thus, if a user needs to interface with another type of I/O, contact 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 user-written Java code that take evio events from an input fifo and place evio events on an output fifo. 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.

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 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 fifo. At this point either the next module gets, processes, and passes it on, or it goes 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 extraordinarily 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 extraordinarily 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 3 different means of communication to the outside world: ET systems, cMsg messages, 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 3 different types of transports to the outside world that may be used: cMsg, 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

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).

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

2.1.4. 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			
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".

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. 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 FIFIO 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.
endian		out	Endianness of outgoing data.	Case indep. "little" for little endian. Defaults to big endian.
capacity		both	Size of queue for incoming or outgoing evio events.	Any positive integer, ignores other values. Defaults to 40.
blockNumCheck		in	Require incoming evio block numbers to be sequential.	Yes, no, true, false, on, off (case insensitive). Default is no.
fileName		both	Name of file to write or read.	A valid file name. May contain the wildcard ***%d*** which will be replaced by the run number. 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 file.	Valid directory. Not used by default.
prefix		out	Prefix in generated file name.	Any string. Defaults to blank string.

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 6-digit number appended to the end of its name to differentiate it from the previous one. This number starts at 000000 and increases by 1 for each subsequent file and is called the ***fileCount***.

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<fileCount> where session refers to its session name received from runcontrol. If ***prefix*** is defined, then it writes to <prefix>_<run#>.dat<fileCount> instead. 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 occurrences of ***%d*** are replaced by the run number. The second is that all environmental variables in fileName of the form $(ABC) will have the corresponding env variable values substituted in place. If a env variable cannot be found, a blank string is substituted. 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). If split is defined, it adds <fileCount> (initially 000000) to the end of the file name.

2.2.3. cMsg

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.
endian		out	Endianness of outgoing data.	"little" for little endian (case insensitive). Defaults to big endian.
capacity		both	Size of queue for evio events.	Any positive integer, ignores other values. Defaults to 40.
blockNumCheck		in	Require incoming evio block numbers to be sequential.	Yes, no, true, false, on, off (case insensitive). Default is no.
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.

2.2.4. 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"
            wthreads="2" 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"  ithreads="3"  idFilter="on"  capacity="10"
            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.
endian		out	Endianness of outgoing data.	Case indep. "little" for little endian. Defaults to big endian.
capacity		both	Size of queue for incoming or outgoing evio events.	Any positive integer, ignores other values. Defaults to 40.
blockNumCheck		in	Require incoming evio block numbers to be sequential.	Yes, no, true, false, on, off (case insensitive). Default is no.

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 with the given value of **id.**	Case indep. "on", everything else is off. Default is off.
ithreads		in	Number of input threads, each takes & parses ET events, puts them back into the ET system, and puts parsed evio banks in the queue.	Defaults to 3. Max of 10.
othreads		out	Number of output threads, each takes a bank from the queue, puts it into a new ET event and puts that into the ET system.	Defaults to 1. Max of 10.
wthreads		out	Number of writing threads in a thread pool for each output thread. Each writes an evio bank's contents into an ET buffer.	Defaults to 2. Max of 20.

Section 3

3. Modules

Modules are the heart of an EMU. While most users will want a single module to make the EMU perform a single task, it is possible to string multiple modules together in a single EMU. And while modules can be written and used by anyone, this chapter only covers the modules written and supported by the DAQ group.

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 runcontrol 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 an: 1) event builder, 2) event recorder, 3) data disentangler, and 4) simulated ROC.

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 src="modules.jar" usr_src="user_modules.jar">
   <EbModule class="EventBuilding" id="1" threads="3" statistics="on">
          <inchannel  id="1"  name="Roc1"  transp="InET" stationName="stat1"
                      position="1"  idFilter="on" capacity="20" chunk="5"/>
          <inchannel  id="2"  name="Roc2"  transp="InET" stationName="stat2"
                      position="2" idFilter="on" capacity="20" chunk="5" />
          <outchannel id="1"  name="EbOut" transp="OutET" capacity="400"
                      group="1" chunk="200" />
   </EbModule>
</modules>
```

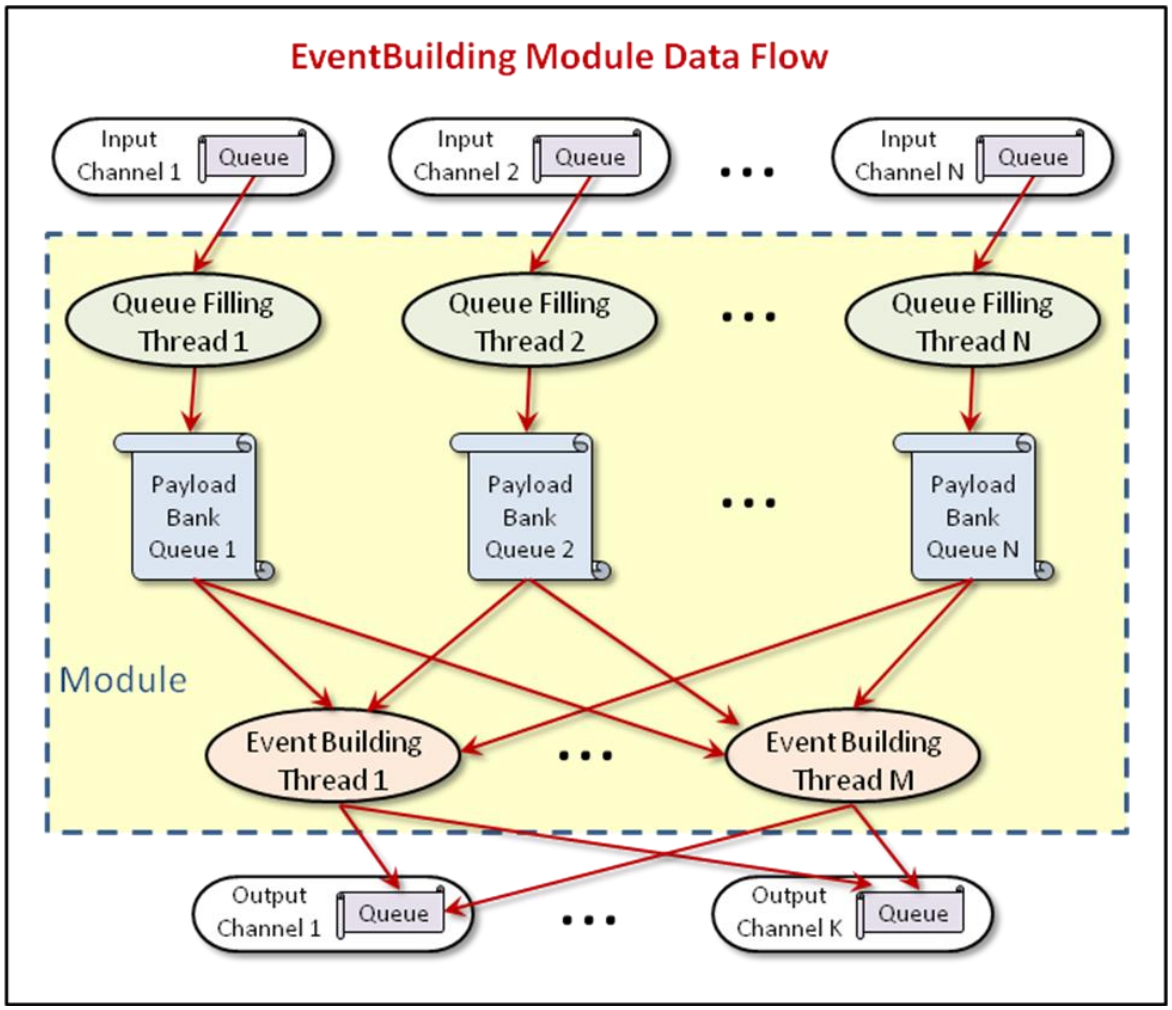
The modules element has 2 attributes that can be set, ***src*** and ***usr_src***. The src attribute gives the name of the jar file containing all the standard, CODA-supplied modules. If not given it defaults to modules.jar. The usr_src attribute gives the name of the jar file containing any other user-supplied modules. If not given, Java looks in all the standard

places for the needed classes. These jar files must be in the user's CLASSPATH enviromental variable in order for Java to find them.

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 primary event builder (PEB) which is the second level of a 2-stage event building. It can also function as a stage event builder (SEB) 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 blue dotted line in the figure above represents the event building module. It creates a single queue-filling thread for each input data channel. This thread takes events from a channel and checks to see if they contain valid ROC raw, physics, control or user evio format data. If so they are placed in an internal payload bank queue for use in actual building. Other banks are ignored.

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, at this point a single event from each payload queue will be needed to build a complete event.

This is accomplished by the event building threads - the number of which is determined by the config file and defaults to 3. Each thread will grab an event from each payload queue, build them into a single event and place that built event on an output channel's queue. If more than one output channel exists, the events are placed in round-robin fashion.

Any user events are not built but simply passed on to the first output channel. Any control events must appear on each payload queue 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 details of the event building are all contained in the event building threads, some of which 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" which is the Java class name.
threads		Number of event building threads.	Positive integer between 1 and 10 inclusive. Defaults to 3. Negative values set it to 1, and 10 is max.
statistics		If multiple modules exist, may statistics from this module be reported to runcontrol as representative of the whole EMU?	Yes, true, on (case insensitive) if representative. Default is off.
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.
swap		Swap data, if necessary, to big endian by assuming data is 32 bit int array.	No, false, off (case insensitive) to not swap. Default is on.
runData		Include run number and run type in built trigger bank.	Yes, in, true, on (case insensitive) to include data. Default is off.

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.

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. The only parameter the user can tweak is the number of threads which grab events from the input channel and send them to the output channels.

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 "EventRecording" which is the Java class name.
threads		Number of event recording threads.	Integer between 1 and 10 inclusive. Defaults to 1. Negative values get set to 1, over 10 values get set to 10.
statistics		If multiple modules exist, may statistics from this module be reported to runcontrol as representative of the whole EMU?	Yes, true, on (case insensitive) if representative. Default is off.

3.4. Disentangling

This is only an ***example*** of how to write a data disentangling module. The main problem is that the format of the data being disentangled affects how the it is done. This example assumes a certain data format and shows how the procedure is done. It is hoped that having the user copy this example and modifying it to suit the actual data format, that a functional disentangling module can be produced which will be useful to a specific experiment.

3.5. ROC Simulating

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 is simulated FADC250 entangled data of about 150 bytes per event plus extra header and trailer words. The size of these extra words depend on the number of electronics modules in a crate and the number of events entangled and end up being about 40 bytes/event if there are 100 events and 10 modules.

There are some parameters the user can control in the following table. But perhaps the ***end*** attribute needs a bit of explanation. In order for any simulation to work properly, all ROCs must have sent the same number of events when the runcontrol commands to end

or reset are given. The only way to insure this is to tell each ROC exactly how many events to create before stopping production, hence the end attribute - a poor man's trigger.

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 15gets set to 15.
detectorId		Id of detector producing data in data block bank.	Integer >= 0. Defaults to 111. Negative values set it to 0.
numRecords		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.
SEMode		Use single event mode - one event per data block. Takes precedence over ***numRecords***.	Yes, true, on (case insensitive) if representative. Default is off. Sets ***numRecords*** to 1.
end		Number of events to generate before quitting data generating threads. Used for stopping multiple simulated ROCs together without a trigger supervisor.	Integer >= 0. Defaults to 0 which means no limit. Negative values set it to 0.

Section 4

4. Running an EMU with Run Control

This is generally why you run ***from*** an emu, the only flightless and carnivorous bird on the planet. You definitely want run control on this guy - an END button might come in handy. Thankfully, running an emu is much safer. Seriously, the running of an EMU can range from a straightforward process to something of a nightmare. It's a good thing this manual is here to keep the reader straight. Much of the complexity arises from the multithreaded nature of the Java Virtual Machine (JVM) and the EMU which can run in a JVM simultaneously.

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 runcontrol, 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:

Argument	Required	Function
-h or -help		Print help.
-Dname		Name of CODA component to create.
-Dtype		Type of CODA component to create. See Appendix A.
-Dconfig		Name of configuration file to use.
-Dlconfig		Name of local config file which contains static info for debug gui.
-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. Used to generate default ET system name (/tmp/<expid>_<emuName>). Used to construct default UDL (rc://multicast/<expid>) to connect to cMsg server for receiving runcontrol commands and sending cmlog messages ***if cmsgUDL not defined***.
-Dsession		Set the experimental session. Used to generate output file names. Runcontrol, 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 runcontrol.
-Duser.name		Arg passed to EMU objects which set user's name (defaults to expid, then session)

4.3. Run Control & EMUs

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. Configuring EMUs

There are 2 ways to configure EMUs. It can be done with user-created config files given on the command line or with those created by the ***jcedit*** program and passed to components by run control during the ***configure*** transition. The second way is the one endorsed by the DAQ group. See the jcedit manual for more details.

4.4.1. With Command Line

Do something like the following to use a command line given config file:

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

The ***name*** & ***type*** arguments tell the what its name is and what type of CODA component it is. And the ***config*** argument gives the name of the file with the configuration info. Be warned that if the name and the file are both specified, then the file's component name must be identical to the one given on the command line. It's certainly safer to specify the config file and forget about simultaneously specifying the name. The same goes for the type. If it's specified in both places, they must agree.

4.4.2. With Run Control

When using run control to configure an EMU, 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.

Undoubtedly the reader will want to know what happens if a command line config file is also specified. The config sent by run control at the configure transition simply supersedes the one given on the command line.

4.5. Running Multiple EMUs

It is possible, for whatever reason, to run multiple EMUs in a single JVM. This, however, can only be done from the command line simply because a single config file is only allowed to specify a single component. To do this the names, types, and config files 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"
     -Dconfig=../Roc.xml,../Eb.xml,../Er.xml org/jlab/coda/emu/EmuFactory
```

This will run 3 EMUs in 1 JVM assuming that the config files can be found and there are no contradictions between the names & types in the files compared to those on the command line. Again, it is safer to leave out the command line specification of names & types and to use only the files or vice versa.

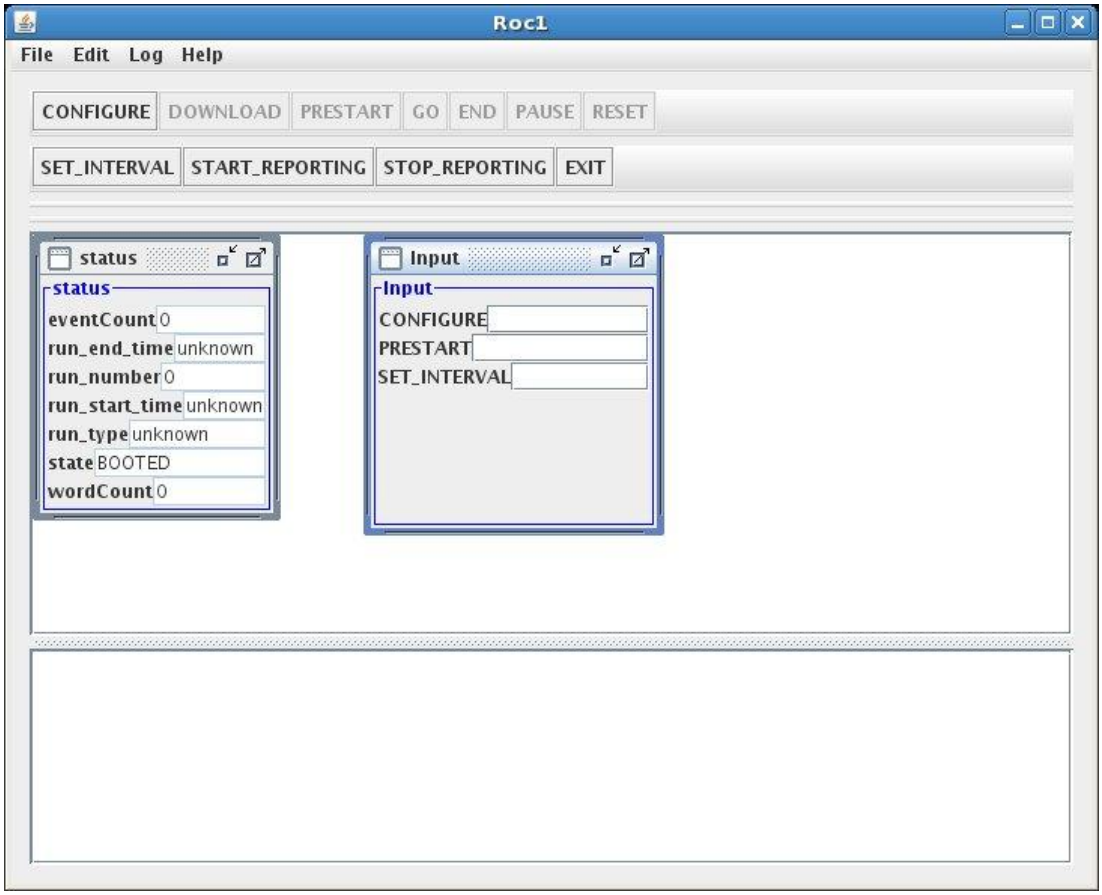
Section 5

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
```

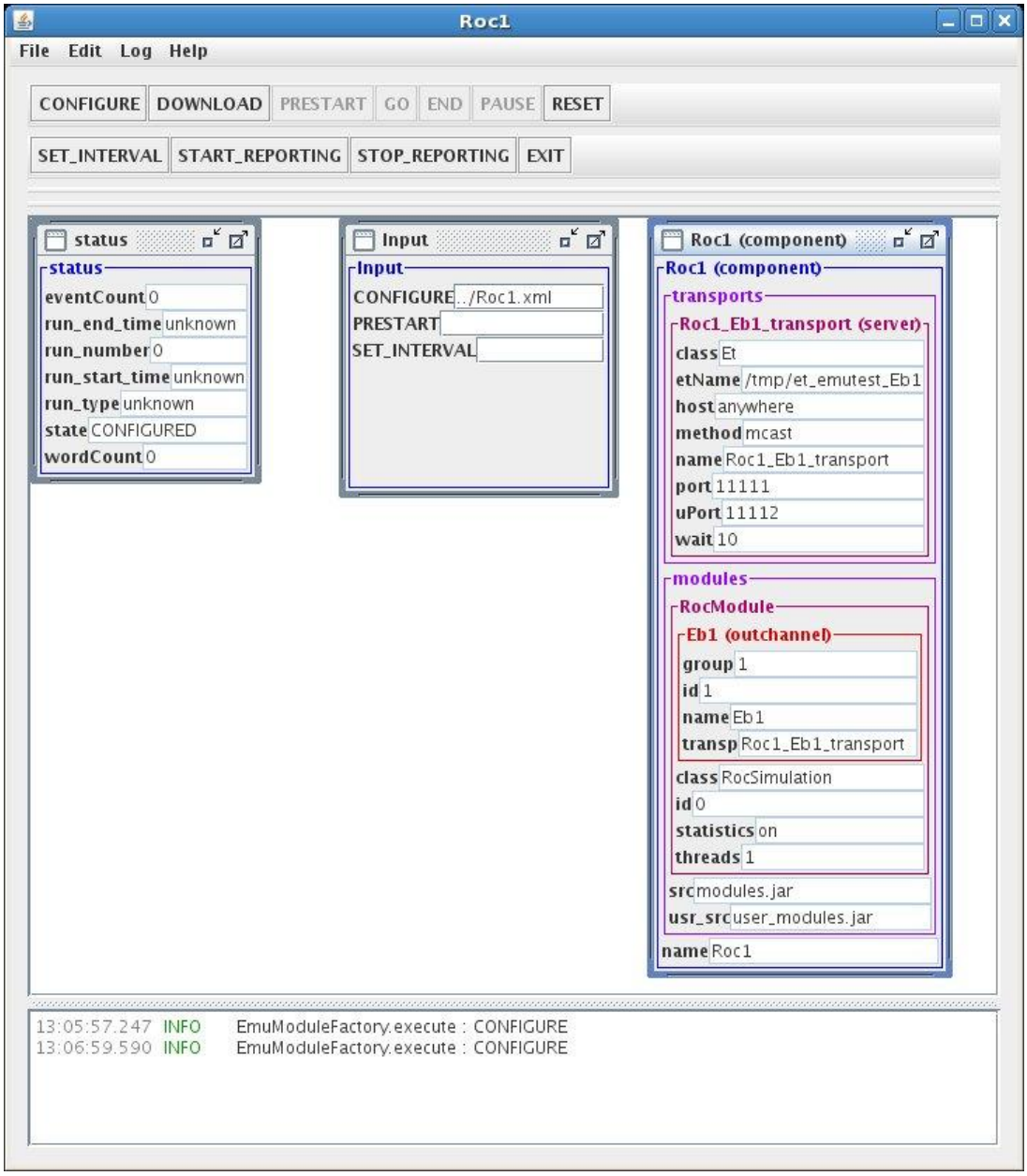
This gui allows the user to send locally generated run control commands to the EMU and see its output. It looks like:



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:



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 maintained for receiving and responding to run control commands as well as for sending cmlog messages. Generally the cMsg 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>?multicastTO=<mtimeout>&connectTO=<ctimeout>
```

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
- ***mtimeout*** is the time to wait in seconds before connect returns a timeout when the rc multicast server (in AFECS platform) does not answer
- ***ctimeout*** 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

With no timeouts and the default port, most users will use a UDL like the following:

```
rc://multicast/myExpid
```

If the UDL is not given when starting up an EMU, it will attempt to use the following if the EXPID environmental variable exists:

```
rc://multicast/<EXPID>
```

if EXPID is not defined, it will try:

```
cmsg://localhost/cmsg/test
```

If the connection cannot be established, the EMU will retry every 5 seconds forever and ever.

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, which extends Thread, to do all the work. The config file's output channel's ***othreads*** attribute determines the number of these threads that are running.

Each one of these threads does the following. There is a single getter thread which gets new events from ET. Once an array of new events arrives, this getter immediately runs again to get another batch simultaneously with the filling and putting of these events.

Once new events are available, an evio bank is grabbed from the channel's queue (placed there by the last module). Starting with the first new event, a list is created and the bank is added to it. Another new event is taken when there are enough banks to fill up the previous event or the type (e.g. roc raw, physics, partial physics, disentangled physics, user or control) of the bank changes. In this way, each empty new ET event has a list of banks of the same type associated with it.

Once all the new events have everything they can hold in their associated lists, then each event and its list is given to a writer thread from a thread pool. The size of this thread pool is determined by the config file's output channel's ***wthreads*** attribute. The writer thread takes the banks and writes them into the ET event's buffer. When all the write threads are done, the array of ET events is put back into the ET system.

If one of the ET events put back contained the run control end event, the channel shuts down all its threads and exits. (Note, by this time the end event has been passed on like it should be). Likewise, if a reset command was received sometime during this process, it does the same thing.

By default there is 1 data output helper thread, 1 getter & 2 write threads per output helper for a total of 4 threads in a single ET output channel.

6.3.2. Input

ET input is a little simpler than the output. The input channel uses the interior class DataInputHelper and by default 3 such threads are started although that can be set through the config file's input channel's ***ithreads*** attribute.

Each data input thread gets an array of ET events to begin with. Each event is, in turn, parsed into evio banks which are put in a list. Once all the banks are extracted from a single event, the complete list is placed in the channel's queue. That queue will be used as input to the first module. When all the ET events are parsed, they are put back into the ET system.

If an end event appears, it is placed on the list and passed on along with everything else, but after that all parsing stops (it should be the last bank to come through anyway). ET events are returned and the thread exits.

By default there are only 3 threads, all of the same type.

6.4. cMsg Channels

6.4.1. Output

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 config file'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 queue. That queue will be used as input to the first module.

If an end event appears, it is placed on the queue along with everything else. The callback does nothing else, but the close() is called when the end command reaches the cMsg transport object and it unsubscribes.

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