

COOR

scott snyder

May 3, 1999

Contents

1	Notational Conventions	1
2	Overview	2
3	Summary of Requirements	4
4	Detector Model	5
5	Protocols	9
5.1	COOR-Client Communication	9
5.2	COOR-Downloader Communication	13
5.2.1	Common Protocol	14
5.2.2	EPICS	18
5.2.3	Level 1	18
5.2.4	Level 3	20
5.2.5	Data Logging	20
6	Run Transitions	21
A	State Diagrams	21

1 Notational Conventions

Text in *this font* refers to features of COOR which are planned, but not yet implemented.

2 Overview

The COOR program is responsible for coordinating changes in state of the software and hardware components comprising the data acquisition system. Clients wishing to use pieces of the system must send requests to the COOR, which will ensure that the request does not conflict with those of other clients. COOR will then communicate with the rest of the system to put it in the state requested by the client.

Requests to begin and end runs are also considered state changes which should go through the COOR. In response to such a request, it will step the other components of the system through the proper sequence of actions. These requests may come not only directly from clients, but may also be generated by the data logging and alarm systems to automatically stop or pause a run due to an error condition.

Figure 1 illustrates the main communication paths to and from COOR. At the top of the figure are the *clients*, which make requests of COOR. These will most commonly be instances of the *TAKER* program, which is the primary user interface for controlling data taking. However, there will also be clients to display status information about COOR, and to provide expert-level control of COOR itself. At the bottom of the figure are the *targets*, with which COOR communicates in order to effect changes in the system. At present the set of targets includes the Level 1 and 2 triggers (through the L1TCC), the Level 3 trigger (through the L3 supervisor), the EPICS system (though a translator program), and the data logging system.

COOR also talks to the alarm system (illustrated on the left), not only to report its own errors, but also to report major state changes, such as the beginnings and ends of runs. The alarm system will then make these state change notifications available to any other components of the system which want them.

Note the following:

- COOR is not in the data path. It is responsible for setting up and terminating runs, but does not directly participate in them.
- The flow of commands is largely one-way, from the clients, through COOR, and finally down to the targets. The exception is that COOR may send a notification back to a client if its run has been asynchronously stopped or paused. Note in particular that this implies that if the alarm system or one of the targets wants to change the run state — such as

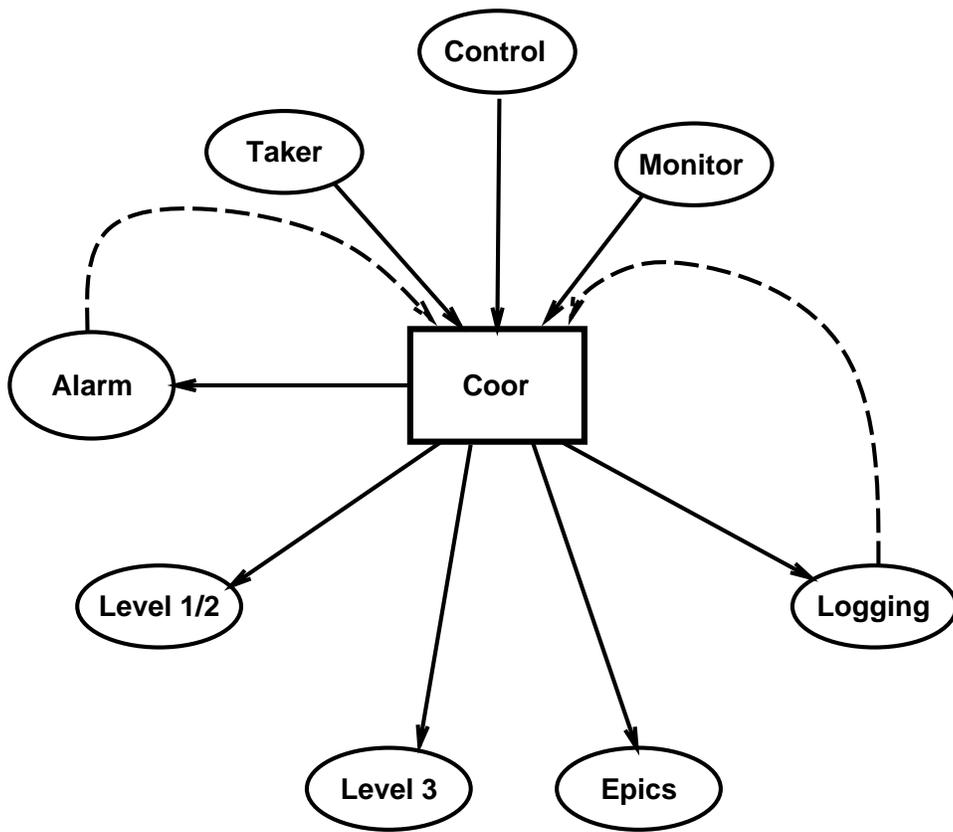


Figure 1: Primary communication paths to and from COOR.

pausing physics runs on a fatal alarm or stopping a run after a fixed number of events has been recorded — that component should make a separate client connection back to COOR to send the command. (This is illustrated by the dashed lines in Figure 1.)

- Some of the targets, such as Level 3 or the data logging system, are actually a collection of cooperating processes. For these targets, there should be a single process with which COOR communicates; this process is then responsible for distributing the commands from COOR to the rest of the subsystem. This strategy should help to insulate COOR from the internal subsystem details and make it easier to isolate the various pieces for testing and debugging.

3 Summary of Requirements

- COOR receives requests from clients to use pieces of the data acquisition system. If such a request is compatible with other concurrent users of the system, COOR communicates with the other system components to put them in the desired state.
- The subsystems which COOR should be able to configure include:
 - The Level 1, Level 2, and Level 3 triggers.
 - The data logging system.
 - The digitizing crates and VBDs.

Other components may be added as the need arises, or they may be controlled through other pathways. For example, in run I, calibration pulsers were controlled through COOR, while the high voltage, low voltage, and clock control subsystems were not.

- COOR receives requests from clients to begin and end runs. It again ensures that the request is compatible with other users and, if so, communicates with the triggers and the data logging system to start data flowing.
- COOR receives requests from clients to pause and resume ongoing runs. Pausing a run means disabling the Level 1 trigger bits used by that

run. This halts the flow of data in such a way that it can be rapidly restarted.

- COOR may also receive requests to pause runs from the alarm system (due to the detection of an error which may compromise the quality of the data) or from the data logging system (if a limit was set on the number of events to be collected in a run).
- COOR should send notification of significant state changes (such as runs starting or ending) to the alarm system for distribution to any other interested parties.
- COOR should arrange to record in an appropriate database the information it has about each recorded run taken. (This may not actually be done by COOR, but instead by some other process with which it communicates.)
- COOR should be able to supply information to clients and monitoring programs concerning the current state of the detector components which it manages.
- Client programs should be provided for taking data runs and displaying the current status.

4 Detector Model

This section briefly summarizes the concepts COOR uses to model the current state of the system.

All objects used to model the system state are instances of a class deriving from `Object_Base.Object`. Each object has an *object name*. Objects are associated with a *registry*, deriving from `Object_Base.Object_Registry`. The `Object_Base.Object` class provides hooks for tracking all changes to the object contents. This is used for logging, monitoring, and to allow for undoing changes after discovering an error in a configuration.

Object names are conventionally divided into two parts: a *class tag*, and the name itself, written like *tag:name*. The class tag identifies the type of the object. Note that it is possible for two objects with different class tags to be implemented by the same Python class.

All objects which can be allocated derive from `Ownership.Ownable`, and those which can allocate other objects derive from `Ownership.Owner`.

All objects which represent things which must be configured derive from `Generic_Item.Item`. Such objects have a set of attributes which define their downloadable state. These are ordinary Python attributes of the object, but they follow a special naming convention. Downloadable attributes which may be changed at any time should start with `'d_'`. There are also 'immutable' attributes, which start with `'i_'`. These can be set when the item is first allocated, but not afterwards. These are usually used to represent structural relationships between items.

Two copies of the attributes are maintained. The actual Python attributes of the item represent COOR's idea of the current state of the item. If the value of an attribute is `None`, that means the current state is unknown. Each item also maintains a *requested value* for each attribute. This is what has requested by the owner of the item, and is what COOR will try to make the current values reflect by means of a download.

Each item can be in one of four states:

- **UNKNOWN** — The current state of the attributes in the external system is unknown.
- **VALID** — The current state of the attributes in the external system matches what has been requested.
- **DOWNLOADING** — There is a download operation in progress on this item, to make the state of the attributes in the external system match what has been requested.
- **DOWNLOADING_INVALID** — There is a download operation in progress, but while it was in progress, this item has been invalidated. (Probably due to a lost connection to the download target.) When the download completes, the state of this item will become **UNKNOWN**.

When which has been owned is deallocated, the list of requested attribute values is cleared and the item state is changed to **UNKNOWN**. Also, the state of an owned object can be changed to **UNKNOWN** either by an explicit invalidation request or due to a broken connection to the download target.

For most item types, the set of attributes used is defined by the Python class representing that item. For most EPICS devices, however, a more generic

representation is used, implemented by the class `Devices.Device`. Each device has a *device type* associated with it (class `Devices.Devtype`); the device type defines the set of attributes which the device holds. The device types are defined during COOR initialization.

Some items are *generic*, in that there is a set of identical items. Such items are usually implemented using the `Generic_Item.Numbered_Item` class. In this case, the name of the item is simply a number. Each distinct set of such objects must thus have a distinct class tag.

The registry into which the objects are collected is implemented by the class `Config.DO_Config`. This class has high-level methods for allocating the various item types.

The following table gives the presently defined class tags, the Python classes used to implement them, and a brief description of each.

<code>conn</code>	<code>Config.Connection_Status_Reporter_Object</code> Used to send status information about the downloader connections to monitoring programs.
<code>client</code>	<code>Clients.Client</code> Represents an external program requesting services of COOR. This includes instances of <code>TAKER</code> , as well as monitoring and control programs. This class derives from <code>Ownership.Owner</code> , so these objects can own others.
<code>logclient</code>	<code>Logger.Logger_Client</code> The data logging system needs to be told some information about each client which could start a run, such as whether or not it has recording turned on (see Sec. 5.2.5). These objects have the attributes containing that information. Note that these objects are not permanent; they are created when a client loads a configuration, and destroyed when they are deallocated.
<code>l1eg</code>	<code>Level1.L1eg</code> A Level 1 exposure group.
<code>l1specterm</code>	<code>Level1.L1specterm</code> A Level 1 specific (named) and/or term. Also called direct-in terms.
<code>l1muomgr0</code>	<code>Level1.L1trigmgr</code>

l1sftmgr0	Level1.L1trigmgr Level 1 and/or terms from trigger manager cards.
l1emetsum	Level1.L1ct_Thresh
l1hdetsum	Level1.L1ct_Thresh
l1misspt	Level1.L1ct_Thresh
l1totetsum	Level1.L1ct_Thresh Level 1 calorimeter trigger global energy threshold and/or terms.
l1emetrefset	Level1.L1ct_Refset
l1hdvetorefset	Level1.L1ct_Refset
l1ltrefset	Level1.L1ct_Refset
l1totetrefset	Level1.L1ct_Refset Level 1 calorimeter trigger reference sets.
l1emcount0	Level1.L1ct_Count
l1emcount1	Level1.L1ct_Count
l1emcount2	Level1.L1ct_Count
l1emcount3	Level1.L1ct_Count
l1totcount0	Level1.L1ct_Count
l1totcount1	Level1.L1ct_Count
l1totcount2	Level1.L1ct_Count
l1totcount3	Level1.L1ct_Count Level 1 calorimeter trigger count threshold and/or terms.
l1ltcount0	Level1.L1ct_LT_Count
l1ltcount1	Level1.L1ct_LT_Count
l1ltcount2	Level1.L1ct_LT_Count
l1ltcount3	Level1.L1ct_LT_Count
l1ltcount4	Level1.L1ct_LT_Count
l1ltcount5	Level1.L1ct_LT_Count
l1ltcount6	Level1.L1ct_LT_Count
l1ltcount7	Level1.L1ct_LT_Count Level 1 calorimeter trigger large tile count thresh- old and/or terms.
l1bit	Level1.L1bit Level 1 specific trigger bit.
l3l1shad	Level3.L3l1shad

	Level 3 ‘shadow’ of Level 1 information. There is one of these objects for each <code>l1bit</code> object. Some information associated with a Level 1 trigger bit needs to be sent to Level 3 as well (such as the crate readout list). The attributes of this object keep track of that information.
<code>l3bit</code>	<code>Level3.L3bit</code> Level 3 trigger bit.
<code>crate</code>	<code>Devices.Device</code> A digitization crate; a geographic section.
<code>dev</code>	<code>Devices.Device</code> A generic EPICS device.

5 Protocols

This section summarizes the protocols used in the communication between COOR and the various processes with which it communicates.

5.1 COOR-Client Communication

All communication between COOR and the clients is by way of DOME string messages. The usual scenario is for the client to send a command to COOR, then wait for a response. Commands should not be overlapped (except for `abort`, as noted below). There are also several responses which COOR may send asynchronously; these include `TEXT`, `UPDATE_MODATTR`, `UPDATE_NEWOBJ`, `UPDATE_DELOBJ`, and `CMND`, and are discussed further below.

Except as noted below, COOR will respond to each command with a message starting with either `DONE`, `FAIL`, or `ABORTED`. In each case, there may be further data following the keyword, as noted for the individual commands below. `DONE` means that COOR has made the state change requested by the command, `FAIL` or `ABORTED` means that it has not. (Note that this is not necessarily the same as whether the command completed successfully or not.) `ABORTED` is sent after a download has been aborted, either explicitly by the user sending an `abort` command, or automatically, following a timeout.

Commands which initiate a download may take considerable time to complete. To confirm that the download has started, COOR will first reply with

WAIT. This will then be followed by one of DONE, FAIL, or ABORTED, once the download is complete.

At any time, COOR may send a TEXT message to the client. This will contain additional text after the keyword which should, in most cases, be displayed to the user. *In a couple instances, TAKER parses the TEXT commands returned from a dump. We probably should use a different reply type for asynchronous broadcast messages, to avoid race conditions where they could get lost (or screw up the parsing).*

If the run state is changed by COOR, it will send a CMND message to the client in order to notify it of the change. The messages of this type presently defined are:

- CMND `pause` — COOR has paused the client’s run.
- CMND `stop` — COOR has stopped the client’s run.

Following are all the client commands which COOR recognizes. Note that the set of these that are allowed depends on what COOR thinks the current state of the client is — see the state diagrams for details.

- `abort` — Abort a download in progress. It should be sent after the WAIT reply, but before the final reply. Note that COOR will not generate a DONE or FAIL response for this command — when the download completes, the `abort` request is also considered ended.
- `broadcast text` — Send *text* to all clients as a TEXT message. The message will be prefixed with ‘-->’.
- `coor_force_pause` — Used internally to implement the `force_pause` command. This command should not be sent by clients.
- `coor_force_stop` — Used internally to implement the `force_stop` command. This command should not be sent by clients.
- `dump pattern` — Dump COOR’s internal configuration state to the client. The argument *pattern* is a regular expression; information about all objects matching *pattern* will be included in the dump. If *pattern* is omitted, information for all objects is dumped.

The information returned from COOR will be in a message starting with the keyword DUMP. The rest of the message will consist of a string

which, when passed through the Python reader, will produce a Python dictionary containing the dump data.

The DUMP message will then be immediately followed by a DONE message.

- **force_invalidate** *pattern* — Invalidate all items which match the regular expression *pattern*. (If *pattern* is omitted, all items are considered to match.) Only items which are owned by some client are invalidated. This command differs from the **invalidate** command in that it does not require that the client issuing the command own the items being invalidated.
- **force_pause** *runlist* — Force the runs given by *runlist* to pause. *Runlist* should be a space-separated list of run numbers. If it is omitted, all runs in progress will be paused. *When configuration type keywords are implemented, this should pause only those runs which have been declared as respecting pauses.* The intention is that this message is sent by the alarm server to COOR when there is a fatal alarm.
- **force_stop** *runlist* — Stop the runs given by *runlist*, which should be a space-separated list of run numbers. If it is omitted, all runs in progress will be stopped. The intention is that this message is sent by the data logging system to stop a run.
- **free** — Free all resources held by this client.
- **info** *report-type* — Get back a formatted report of some aspects of the current state. This report will be sent back as TEXT messages (followed by a DONE message); it is intended to be human-readable. The report types available are:
 - **clients** — Print information about all clients presently connected to COOR.
 - **downloaders** — Print information about the status of COOR's connections to the download targets.
 - **11bits** — Print information about all presently defined Level 1 trigger bits.
 - **local_11bits** — Print information about the Level 1 trigger bits owned by this client.

- **invalidate** *pattern* — Invalidate all items which match the regular expression *pattern* and are owned by this client. (If *pattern* is omitted, all items are considered to match.)
- **load** *configname* — Load a new configuration, named *configname*. The present implementation looks for a file ‘`configs/configname.py`’ and loads it. This file should define a function called *configname*, which takes the clientstat object as an argument and returns the download list. *The details of this are subject to change.*
- **modify** *name* — Modify the currently downloaded configuration, based on the commands in *name*. The present implementation looks for a file ‘`configs/name.py`’ and loads it. This file should define a function called *name*, which takes the clientstat object as an argument and returns the download list. *The details of this are subject to change.*
- **pause** — Pause the client’s run.
- **prescale** *bitnumber prescale ...* — Change the prescale for Level 1 trigger bit *bitnumber* to *prescale*. Multiple *bitnumber–prescale* pairs may be given in the command. All trigger bits being modified must be owned by this client.
- **reconnect** — If the connections to any of the download targets has been lost, this command will attempt to reestablish them. Note that this is done implicitly before each download. (In fact, this command is implemented by merely queuing an empty download.)
- **recording** *state* — Set the recording state for this client. The *state* argument should be either ‘on’ or ‘off’.
- **reenable** *bitnumber* — Tell the trigger framework to reenable the Level 1 trigger bit *bitnumber*. This only has an effect if the bit was configured in auto-disable mode. The bit must be owned by this client.
- **resume** — Resume the client’s run after a pause.
- **revalidate** — If any of the items owned by this client are marked as invalid, try to do the required downloads to make them valid again. This is done automatically before starting a run.

- **start** — Start a new run. The run number of the new run will be returned as an argument in the `DONE` message.
- **stop** — Stop the client's run.
- **update *pattern*** — Request asynchronous updates for changes in COOR's configuration database for all objects matching the regular expression *pattern*. If *pattern* is omitted, all objects are considered to match.

The messages sent back by COOR when updates occur are of one of three forms:

- `UPDATE_MODATTR objname attrs` — Some attributes of *objname* have changed. *Attrs* is a string which when passed through the Python reader will yield a Python dictionary containing the modified attributes.
- `UPDATE_NEWOBJ objname attrs` — The object *objname* has been added to the configuration. *Attrs* is a string which when passed through the Python reader will yield a Python dictionary containing the attributes of the new object.
- `UPDATE_DELOBJ objname` — The object *objname* has been removed from the configuration.

Only the pattern from the last `update` command is remembered.

Probably need a `dump_update` combo, to avoid possible lossage of updates between the two commands.

- **username *name*** — Set a username for this client, for use in status displays. Note: COOR does not send a reply for this command. *Should this be changed, for consistency?*

5.2 COOR-Downloader Communication

There is a common protocol for communication between COOR and the targets. Within the context of this protocol are various target-specific commands. We first describe the common protocol, then summarize the target-specific parts.

5.2.1 Common Protocol

The design of the common downloading protocol was the result of several considerations:

- The protocol should be usable for all the download targets.
- It should allow the target to process requests concurrently. This implies that COOR should be able to send multiple download requests to the target without receiving a reply, and that the replies may be sent back to COOR in a different order than that in which the commands were received.
- It should allow the target to process requests in a “batched” fashion — to queue up all the commands for a particular configuration request, and only start processing them once all commands have been received. This implies that there must be some way to mark the end of a configuration request.
- The protocol should be easy to test, debug, and extend.

The resulting protocol has the following characteristics:

- DOME is used for the underlying transport. One command or acknowledgment is sent per DOME message. All messages are of type `String_Message`. Where it makes sense, targets should not be case-sensitive.
- Commands are always sent one way: from COOR to the target. As discussed in Sec. 2, a target that needs to make asynchronous requests should explicitly open an additional command channel to COOR.
- Except as noted below, every command should result in an acknowledgment from the target back to COOR. In some cases, the order of acknowledgment may not be the same as the order in which the commands were issued. In order to keep straight the correspondence between commands and acknowledgments, each command has a “command id” which is sent with the command and returned with the acknowledgment. Targets should not assume anything about the format of this id, other than that it consists of printable characters, contains no whitespace, and is no longer than 32 characters. *Do we want to make*

stronger guarantees here? E.g., that it's a monotonically increasing number?

- Except as explicitly, noted, commands can be *batched*. A batch is implicitly started by the first batched command received. It is ended by the special command 'configure'. When `configure` is received, the batched commands should describe a consistent configuration. Once the `configure` command is sent, no additional commands will be sent (except for `abort`, and, if the connection breaks, `init`) until every command in the batch (including the `configure` command) has been acknowledged.

The target can start processing commands at any time. It can process them as they are received, or it can queue them up and process them all once the `configure` command has been received. Acknowledgments can be sent before the `configure` command arrives. Except for the `configure` command (which must be acknowledged last) *and for commands within a block*, commands may be acknowledged in any order.

Note that it is possible for COOR to send a `configure` command with no preceding commands. Targets should simply acknowledge and ignore these requests. *But perhaps the sending of these should be suppressed?*

Commands which are not batched are called *immediate*. They will not be followed by a `configure` command, and in most cases, no other commands will follow (except for `abort` and possibly `init`) until the command is acknowledged.

The general format of a command sent by COOR to the targets is as follows:

command-id command [args...]

The target should reply to the command with a message of the form

command-id status [text]

The *status* should be one of the strings 'ok', 'bad', or 'more'. The optional *text* is either status information being returned from the command (if *status* is 'ok') or an error message (if *status* is 'bad'). If *text* would take more than one line, each line should be sent separately, in order. For all except the last

line, *status* should be ‘more’. For the last line, *status* should be the final value (either ‘ok’ or ‘bad’).

If *status* is ‘ok’ and the command was not requesting any information, then *text* may be blank. If *status* is ‘bad’, *text* should contain a brief error message.

There is a set of common commands which should be recognized by all targets. They might not have to do anything for some of them, but they should be able to recognize and acknowledge them:

- **configure** — As discussed above. A simple target which processes all commands as they are received can ignore these messages.
- **abort** — This command is somewhat special. It is sent to the targets by COOR when a download has been aborted. When this command is received, the target may discard any commands which it has queued, but has not yet processed. If there are no queued commands, the abort request should be ignored. (In particular, should *not* undo any commands which it has already reported as completing successfully.) The target need not respond to the **abort** command.

A simple target which processes all commands as they are received can ignore these messages.

- **init** — This is an immediate command. The target should immediately end all ongoing DAQ, release resources, and restore all programming to the default state.

It should not be necessary to reboot nodes, reload FPGAs, etc. in response to this command. The assumption being made is that the target system is still sane, but is in an unknown state.

This is sent by COOR on startup, and whenever a connection to a target has been broken and reestablished.

- **start_run** *runno spectrigs* — This is an immediate command. Run number *runno* is starting. A successful reply to this command implies that the target is now ready for that run to start. *Spectrigs* is a list of Level 1 specific triggers participating in the run. It is a space-separated list of integers. (*This may get abbreviated using a notation like FIRST:LAST to specify a range.*)

At the time this message is issued, all the triggers associated with the run in question will be disabled, so data for that run will not be flowing yet.

- `stop_run runno` — This is an immediate command. Run number *runno* is stopping. A successful reply to this command implies that the target is now ready for that run to stop.

At the time this message is issued, all the triggers associated with the run in question will be disabled. *But there is presently nothing to synchronize flushing of any buffered data. Is this needed?*

- `begin_block`
- `end_block` — *These are not really commands, per se, and need not be acknowledged. Commands which occur between `begin_block` and `end_block` must be processed in the order in which they were sent.*

Only batched commands may appear within a block, and `configure` may not appear within a block.

Not all order dependencies will be protected within a block. In general, if it doesn't make sense to reorder the commands they won't be put into a block. (This will probably only be used for EPICS downloads.)

- `pause`
- `resume` — *Pause and resume processing events. Systems not in the readout chain won't have to do anything for these commands.*
- `begin_store storenum`
- `end_store storenum` — *Note that a store is beginning or ending.*

Note one exception to the above protocol: all messages sent by COOR to the data logging system will have the string 'COOR ' prefixed to them.

The following sections summarize the target-specific commands for each target. Note that these are not intended to be complete summaries of the commands which the targets can accept; rather, they document the subset of those commands which the present implementation of COOR will actually send.

5.2.2 EPICS

Every downloadable EPICS device has an object name, which consists of a *class-tag:name* pair. (See Sec. 4.) The downloadable state of a device consists of a set of named attributes, each of which has some value.

The command to request a download consists of the device name (without the class tag) (*maybe we should preserve the class tag?*) followed by a list of attribute name-value pairs:

name attr value ...

Any *value* which contains embedded whitespace should be enclosed in single quotes.

Perhaps a keyword should be added before all download requests?

5.2.3 Level 1

The Level 1 target is presently used to configure both the Level 1 trigger framework and the Level 1 calorimeter trigger.

Here are the commands used to configure the Level 1 framework:

- L1FW_Expo_Group *egnumber*
[And_Or_List *termstring*]
[Get_Sect_List *geosect-string*]

Configure L1 exposure group *egnumber*. The list of associated and/or terms is given by *termstring*. This is a space-separated list of integers; a dash before an integer indicates that that particular term is to be vetoed. The list of geographical sections to read out is given by *geosect-string*. This is a space-separated list of integers, except that a range of consecutive integers from *first* to *last* inclusive may be written using the notation '*first:last*'.

- L1FW_spec_trig *bitnumber*
[deallocate]
[prescale *prescale*]
[L1_Qualifier *l1qualifiers*]
[Obey_FE_Busy]
[Auto_Disable]
[Re_Enable]

[coor_enable]
[expo_group egnumber]
[And_Or_List termlist]

Configure L1 specific trigger *bitnumber*. If *bitnumber* starts with a dash, any boolean options mentioned in the command are to be turned *off*; otherwise, they are to be turned *on*.

The keywords in the command have the following meanings:

- *deallocate* — Reset the bit to its default configuration.
- *prescale prescale* — Set the bit’s prescale to *prescale*.
- *L1_Qualifier l1qualifiers* — Set the bit’s L1 qualifier mask to *l1qualifiers*, which should be a space-separated list of integers.
- *Obey_FE_Busy* — Turn on/off whether or not the bit is disabled on front-end-busy.
- *Auto_Disable* — Turn on/off auto-disable (one-shot) mode.
- *Re_Enable* — Reenable an auto-disable’d bit.
- *coor_enable* — Enable/disable this trigger bit.
- *expo_group egnumber* — Set the exposure group associated with this trigger bit to *egnumber*, which should be an integer.
- *And_Or_List termlist* — Set the list of and/or terms for this bit to *termlist*. This is a space-separated list of integers; a dash before an integer indicates that that particular term is to be vetoed.

Here are the commands used to configure the Level 1 calorimeter trigger:

- *L1CT_Energy_Threshold type Comparator number Value thresh*
Set a threshold for comparator *number* (an integer) of type *type* to *value* (a floating point number). The possibilities for *type* are ‘EM_Et’, ‘HD_Et’, ‘TOT_Et’, and ‘Miss_Pt’.
- *L1CT_Ref_Set type number contents*
Set the reference set *number* of type *type* to the string *contents*. (This string is passed through COOR uninterpreted. *Maybe we should be smarter than that?*) The possibilities for *type* are ‘EM_Et_Ref_Set’, ‘HD_Veto_Ref_Set’, ‘TOT_Et_Ref_Set’, and ‘Large_Tile_Ref_Set’.

- `L1CT_Count_Threshold` *type* `Ref_Set` *refset* `Comparator` *number*
Value *value*

Set the count threshold *number* (an integer) of type *type* associated with reference set *refset* (an integer) of that type to *value* (an integer). The possibilities for *type* are: ‘EM_Et_Towers’ and ‘Tot_Et_Towers’.

5.2.4 Level 3

Here are the commands COOR sends to configure Level 3:

- `clear_trigger` *number*
 Forget about L3 bit number *number*.
- `define_trigger` *number* *l1bit*
 Define L3 bit number *number*, associated with the L1 bit *l1bit*.
- `crate_list` *l1bit* *geosect-list*

Note that the list of crates read out by L1 bit *l1bit* is *geosect-list*. This is a space-separated list of integers, except that a range of consecutive integers from *first* to *last* inclusive may be written using the notation ‘*first:last*’.

5.2.5 Data Logging

Since data logging configuration information is associated with particular clients of COOR, we must have some way of identifying these clients to the logging system. For this purpose, each client needing logger configuration is assigned a small integer “client number.” The commands sent are as follows:

- `clear_client` *client-number*
 Delete all configuration information for client *client-number*. The number may then be reused in subsequent configuration messages.
- `set_client` *client-number* [`recording on`] [`recording off`]
 [*configname* *configname*]

Change configuration information for client *client-number*. *Configname* gives the name of the configuration which this client has loaded. The strings ‘`recording on`’ and ‘`recording off`’ toggle recording on and off for this client.

- `runinfo` *client-number runnumber*

Declare that run number *runnumber* is being started by client *client-number*. This is an immediate command, and is sent just before the `start_run` command.

6 Run Transitions

Here we summarize the actions taken when starting and stopping runs.

Starting a run:

- Do a revalidation step. If this fails, the run cannot start.
- Send a `start_run` message to all targets. *The alarm system should probably also be notified here.* Wait until all have responded. If any report an error, the run cannot start.
- Tell the L1 framework to enable the trigger bits for this run.

Stopping a run:

- Tell the L1 framework to disable the trigger bits for this run.
- Send a `stop_run` message to all targets. *The alarm system should probably also be notified here. At present, there is nothing to guarantee that all events have been flushed through the system.*

Presently, there are no explicit pause requests sent to the framework. Is this needed?

A State Diagrams

