CARMA Memorandum Series #47

# Revised CARMA Correlator Software Modules

Kevin P. Rauch (UMD), Tom Costa and Dave Hawkins (Caltech)

August 15, 2008

## ABSTRACT

This document discusses the correlator software components that will differ between the revised and COBRA boards.

*Change Record*

| Revision | Date | Author | Sections/Pages Affected |
|---|---|---|---|
| | Remarks | | |
| 0.1 | 2007-August-22 | KPR | |
| | First draft. | | |
| 0.2 | 2007-August-27 | KPR | |
| | Added updates from DWH. | | |
| 0.3 | 2007-November-12 | DWH | |
| | Added an overview of the COBRA software design. Asked Tom to add comments on the areas he worked on. | | |
| 0.4 | 2007-November-21 | KPR | |
| | Added updates from TC. | | |
| 0.5 | 2008-March-25 | KPR | |
| | Updated division of data processing tasks occurring on fast and slow timescales. | | |
| 0.6 | 2008-August-15 | KPR | |
| | Assigned official CARMA memo number. | | |

## 1. Overview

The correlator software chain can be divided into the following major components:

1. **FPGA VHDL code:** The software from which the FPGA download configurations are produced; fully determines data FPGA processing capabilities and system controller functionality.
2. **Board PPC code:** The software managing communication between the band x86 host and a particular digitizer or correlator board.
3. **Band x86 code:** The software implementing the interface between a single band and the high-level CARMA software which interacts with it.

This document is concerned with the plan and implementation status of the latter two categories. As both the on-board PPC CPUs and the crate x86 host will run Linux, there is significant overlap between the two codebases. Each is discussed separately below, with common elements noted. Use of the PPC running Linux allows several tasks formerly relegated to the x86 host to be transferred to the on-board CPU.

## 2. Embedded PPC Code

The PPC code can be grouped into three basic functions: control, data processing, and monitoring. Direct interaction between the three components is very limited; however, to permit convenient data exchange between them where it does exist, the base implementation plan is for each function to have a dedicated thread within a single board server process. The three main threads may delegate work to additional threads as appropriate.

### 2.1. Control Functionality

- **Set Bandwidth Mode**
  This procedure switches the correlator from one bandwidth mode to another, which entails the following steps:
  - *Modify the ‘change bandwidth’ monitor system flag.*
    This informs the system that a bandwidth change is in progress.
  - *Instruct delay processing thread to suspend operation.*
    This informs the thread to stop communicating with the FPGAs.
  - *Instruct the data processing thread to suspend operation.*
    This informs the thread to stop processing data. It still needs to send data to the x86 host every 500 ms; however, it marks that data as empty. The monitor system contains the reason for the empty data.

  - *Cancel all pending DMA transactions to or from the FPGAs.*
  - *Download new FPGA configurations to the data FPGAs.*
    Once all PPC related communications to the FPGAs has ceased, the FPGAs are reconfigured. If possible all FPGA configurations will be loaded into local RAM after the PPC boots, to minimize delay when switching modes. Memory requirements for this are TBD, and depend on the number of sample requantization options offered to observers. (The VHDL codebase supports 2-, 3-, or 4-bit cross-correlations, but each generates a unique download configuration.) Configurations can be retrieved from an NFS-mounted filesystem if RAM usage is excessive. Implementation of this step requires rework of the current code (how much??)
  - *Verify proper download of the new FPGA configurations.*
    After downloading, FPGA configuration checks are run; the most basic being a check that the configuration version numbers and FPGA compatibility flags are correct (to check that the correct FPGA configuration has been loaded to a specific FPGA). Once the checks pass, the inter-FPGA data buses can be enabled, and correlation processing can restart.
  - *Reset/enable the data FPGAs.*
  - *Reload appropriate delay and phase corrections.*
    The control thread performing delay processing is reactivated.
  - *Reactivate the data processing thread.*
  - *Modify the 'change bandwidth' monitor system flag.*
    The flag value is changed to indicate that a bandwidth change has completed.

  The sequence must ensure that any processed data has delays correctly applied. Data should be marked as bad if it arrives prior to the timestamp of the new delay corrections; e.g., when the delay correction thread is restarted, and sends delay information to the FPGAs, it is only after the next complete 500ms frame that the data can be considered to be the first valid data. Any incomplete frame data must be discarded, otherwise it will cause apparent phase and amplitude glitches in the data.

  The clock/data alignment required for COBRA boards after a bandwidth change will no longer be necessary; it will be guaranteed through the use of a synchronous 1 pps tick routed individually to each FPGA.

- **Delay and Phase Corrections**
  These corrections include antenna-based whole (ns) and fractional (sub-ns) sample delays, and phase offset corrections. Implementation of this functionality is radically different compared to COBRA boards, as all corrections are now applied continuously in the digitizer data FPGAs. The supporting (digitizer) PPC code must perform these tasks every 500 ms:

  - *Generate delay and phase correction tables for the next 500 ms.*
  - *Generate sub-ns filter coefficient sets for each delay.*
  - *Encode the filter data and ns delays into FPGA-required format.*

  - *Download the encoded data to the corresponding digitizer FPGAs.*
    Two data FPGAs (one per antenna) receive a small table of phase offsets, one offset for each 15.625 ms integration; the other two FPGAs receive larger tables combining ns delay and sub-ns delay filter information, again one for each integration. Transfer to FGPA RAM involves scheduling DMA transfers of pre-defined memory buffers to memory-mapped locations.

  The delay and phase corrections are tightly coupled, and hence must always be updated together. Activities affecting delay and/or phase include bandwidth changes, observing frequency (second LO) changes, Doppler tracking, and noise source control. Delay values are interpolated from a triplet of values received from the host; these are updated every 20 s.

- **Set Noise Source**
  This procedure turns the noise source on/off. At the board level it merely instructs the CPU to ignore/enable the astronomical delay corrections. This in turn requires a call to the delay and phase correction update routine.

  Phase flattening on the noise source can be implemented in two ways:

  - *aligning digitizer clocks to remove the slope*
  - *using a reference noise source integration*

  In the latter scheme, a noise source integration is used to determine a slope, and all subsequent noise integrations have the opposite slope and offset applied. The COBRA scheme uses the clocks to remove the phase slope, however, there remains a band-to-band offset. The offset is static so can be removed after-the-fact. The digitizer FPGA delay/offset registers do not need to be used. Regardless of which scheme is used, there is a sequencing protocol.

  When turning the noise source on:

  - *Noise source on command is received.*
    This is asynchronous. When the noise on command is received, it has to be assumed that any integration data in the last frame could be corrupted, i.e., contain some fraction of the radio source and some fraction on the noise source, so the 500ms data packet received from the FPGAs in the same 500ms as the command was received is discarded, and the 500ms frame sent to the x86 is marked as empty.
  - *Set FPGA delay/offset values to zero for the next 500 ms.*
  - *Send 500ms noise integrations to the host CPU.*

  When turning the noise source off, the above sequence is repeated, this time setting things up for the radio source. Any data that has potentially mixed radio source and noise source data needs to be discarded. The correlator can only make a discard decision based on the time it receives a noise on/off command. The data could still be corrupted, eg. if the downconverter receives the command in the next 0.5s frame after the correlator, the data will still be bad. It is up to the high-level control system to filter data at that level.

- **Set Walsh Table**
  This procedure provides a way to reset the current Walsh sequence and/or reload a new table. This is implemented in COBRA; however, the host implementation is not exposed to the CARMA control system as an API call. The `CorrelatorBandServer` in the current system uses the set Walsh command to send the Walsh sequences to the DSPs. Without that information the DSPs can not demodulate the data. There are two Walsh sequences:

  – *The 180-degree sequence sent to the digitizer FPGAs.*
    This sequence removes DC offsets on a 1/1024 ms timescale.

  – *The 90-degree sequence sent to the correlator FPGAs.*
    This sequence supports sideband separation (16/1024 ms timescale).

  The API could be modified to reflect the two separate Walsh tables. The COBRA system uses the antenna-based Walsh table to create baseline-based Walsh tables to send to the baseline processing code.

- **Set Digitizer Thresholds**
  This procedure sets the digitizer quantization thresholds to maximize cross-correlation signal-to-noise. The algorithm we will use has not been worked out. There is limited analog gain control on the ADCs, and the FPGAs will most likely be responsible for renormalizing the input samples so that the input distribution is zero-centered and optimally scaled. However, the available analog controls should be fully utilized to avoid losing significant bits to digital rescaling. The digitizer FPGAs produce quantization state histograms each integration that can be used by the PPC to compute the required renormalization constants, which can be written back to the FPGAs. In practice this will be done only on initial start-up of the correlator.

- **Digitizer Clock Control**
  The phase flattening algorithm (controlled by the x86 host) requires a way to slew the digitizer clocks to remove static delay offsets between antennas. This procedure provides the requisite control hook. Implementation is unique to the new hardware.

## 2.2. Signal Processing

Raw correlation data from the FPGAs will be transferred to DDR RAM via DMA; hence the PPC data processing code will read the data from local memory. Processing occurs on two timescales, the ‘fast’ 15.625 ms integration time and the ‘slow’ 500 ms accumulation time. Correlation data flows through these steps on the fast timescale:

- **DMA transfer of FPGA lag RAM to local (kernel driver) DDR RAM.**

- **Copy driver data to user space and collate into a Band object.**

- **Accumulate data into proper phase bin (in-phase or quadrature).**

The DMA transfer operates as a low-level kernel task, independent of the data processing. The remaining tasks are similar to current COBRA processing, except that lags are accumulated instead of spectra.

This processing occurs on the slow timescale:

- **Conversion of integer data to floating-point format.**
- **Renormalize data to remove bias and scale.**
- **FFT the data to obtain spectra.**
- **Sideband separate the spectra.**
- **Package spectra into a form suitable for consumption by x86 host.**
- **Transfer the encoded data to the host.**

Renormalization depends on the multiplication scheme, which varies with the bit-width of the (2-, 3-, or 4-bit) cross-correlation samples. The FFTW package, which can be configured to produce optimal transforms for vectors of a specific length, will be used to Fourier transform the data. The number of lags produced by the FPGA configurations has also been tuned to permit the use of a fast transform, without the need for zero-padding. Aside from the increased resolution, this process is no different than for COBRA.

The code to package and transfer data can be largely (completely?) reused from COBRA, and shared between the PPC code and x86 code. In the current scheme, transfer of data places it into the ACE domain; prior processing can be performed sans the ACE framework.

## 2.3. Monitoring

The monitoring thread is tasked with gathering monitor point information from the various on-board devices, and transferring collected data to the x86 host every 500 ms. The following modifications to the existing COBRA code are needed:

- **Implement data collection from new low-level hardware devices.**
- **Separate monitor points into common and hardware-specific ones.**
- **Modify raw data structures to accommodate new monitor points.**
- **Support conversion of new data structures to monitor packet format.**

Note that these changes propagate even into high-level CARMA code; addition to and reorganization of existing MPML is needed, as well as updates to the x86 host code which aggregates monitor data from all boards in a band.

## 3. Host x86 Code

The major changes compared to COBRA are:

- **Flatten Phase Routine**
  The current procedure suffers from an inability to write into the monitor stream (to update flattening status and set control sequence number). The COBRA flattening algorithm also needs to be reworked to support the revised hardware (what control API is needed from the PPC?)
- **Support new monitor point organization.**
- *What else??*

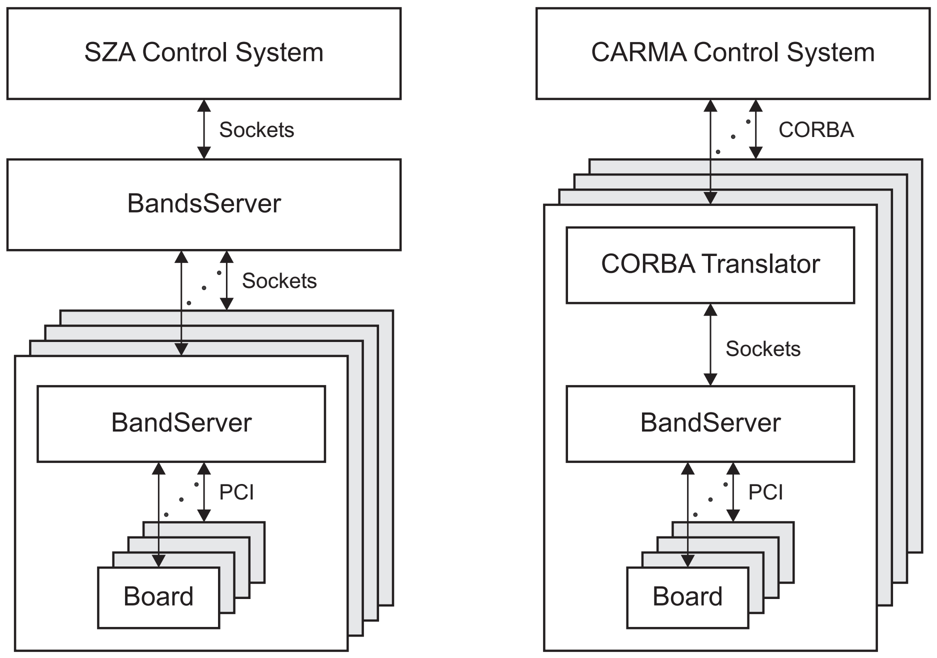
## 4. FPGA Data Format

In 2006 we created a document that contained the data format requirements for the data from the FPGAs. The description of the data should be added to this document. The objective of the Linux data processing is as follows:

- **Hardware is responsible for meeting real-time deadlines.**
  The PowerPC DMA controller will be configured by the control process to transfer the 15.625ms data into a bounded kernel buffer (a linked list of buffers, probably no more than a second long).

  The configuration of the driver is TBD. Basically the driver needs to know how many lags to read, and from where. Its possible that this information can be encoded in an FPGA register that is changed when the FPGAs are reconfigured. But a driver call is also an option. Driver calls to cancel pending DMA transactions prior to an FPGA reconfiguration are required anyway.

  *Bottom line:* the driver will ensure that 15.625ms data packets are transferred to DDR memory before the next 15.625ms integration is ready.
- **Linux is responsible for maintaining data throughput.**
  In other words, it needs to process 500 ms of data every 500 ms, but is subject to no other scheduling deadlines. If during processing it misses a 15.625ms deadline, the extra data will be buffered in the driver. However, this potential for missing a deadline means that the Linux process can not just ’read the time’ to determine which phase bin in which to accumulate the data; the data packet needs to contain an indication of time (e.g., a sequence counter), or phase-switch state, so that the data packet is self-contained. A timestamp is probably the most useful indicator, as the Linux process can use it to determine whether the process has missed a critical deadline (e.g. a 500ms boundary), and that the data has to be discarded. This timestamp would need to exist in the data for each FPGA, or be read from a system controller register and pre-pended to the data packet as part of the DMA to DDR memory.



(a) (a)

Fig. 1.— COBRA software hierarchy for (a) SZA and (b) CARMA.

These issues were discussed when we were developing the data processing packet format, and the packet should already contain this information.

## 5. COBRA review

### 5.1. Software hierarchy

This section contains comments on the COBRA control system design, comments on areas Tom was working, and comments on areas that require work for the COBRA hardware, and the new CARMA hardware.

Figure 1 shows the software hierarchy for the SZA and CARMA. Both control systems use a common code-base. Figure 1(a) shows how the SZA control system receives correlation results;

- The SZA control system uses a bands client class to connect to the bands server running on `szanet`
- The bands server application contains a server class, and multiple band client classes. The server receives data over each of the band client connections, combines the data into a bands data object, and serves that to bands clients. The SZA control system is typically the only bands client. A bands client

file writer class and application exists that can be used to write bands data into a file for analysis in MATLAB. There are 16 bands in the SZA system.

- The band server application contains a server class, and multiple board classes. The server receives data from the board devices, combines the data into a band object, and serves that the band clients. There are 6 boards in an SZA band.

Figure 1(b) shows how the CARMA control system receives correlation results;

- The CARMA control system communicates to each band directly using the CORBA protocol.
- Each CPU that runs a band server also runs a CORBA translator application. The translator application implements the CORBA correlator interface using the sockets-based interface provided by the band client interface.
- The band server application contains a server class, and multiple board classes. The server receives data from the board devices, combines the data into a band object, and serves that the band clients. There are 19 boards in a CARMA band.

## 5.2. Hardware initialization

The correlator hardware needs to be initialized after power-up. Initialization consists of;

- Loading the OS
- Start the board application
- Reference signals checks; 1pps, phase-switch reference, external clock
- Clock setup (phase-lock to external clock)
- Setting digitizer thresholds
- Clock alignment

The hardware can check the references and lock its on-board clocks without any communication to the control system. However, it can not set RF levels or flatten clock phase until the control system has informed the boards that the power-levels are correct (for setting thresholds), and that the noise source is on (for aligning clocks). Hence, the board application should only perform basic initial checks, start generating monitor data, and wait for control system commands before progressing further.

The current COBRA code does not operate in this manner. The hardware still requires manual setup. Operations such as booting the DSPs, and setting the time, take longer than the timeout required by the CARMA control system, so these operations are performed manually so that when the IMR starts the application, these steps are skipped as the DSPs are alive and the time is set.

The COBRA board setup code requires work to get it to operate reliably, and then work to add the functionality as control system commands.

The implementation of the setup code consists of some board functionality, and some band functionality. For example, the phase-flattening routine requires information from a band (the phases on all baselines), so it needs to be part of the band server. The commands to move clocks are then issued to a board, i.e., the boards are not really aware they are aligning clocks, they just respond to commands to adjust the clock phase.

### *5.2.1. Low-level setup programs and usage*

- **/usr/local/carmaTools/cobra/CorrelatorReset**
  - Will reset boards.
  - Takes a `-v` for verbose output.
  - `-all` will reset all boards that can be found.
  - `-slot <s>` will reset slot s.
- **/usr/local/carmaTools/cobra/digCorlWideband**
  - Will attempt to set up boards using low level direct manipulation.
  - Tries to set up the thresholds.
  - Takes a `-v` for verbose output.
  - `-all` will set up all boards that can be found.
  - `-all-dig` will set up all digitizer boards that can be found.
  - `-slot <s>` will set up slot s.
  - For correlator boards it back tracks using the config file info to manipulate the digitizer boards that feed that correlator in addition to manipulating the correlator board.
- **Usage order if everything works**
  Note that many of the command lines below appear short because I have taught several of these program to recognize the host name for the machine they are running on and default many of the usual CORBA program arguments if they are not explicitly given on the command line. The set of hosts that are "known" includes ovrolab2, slcor1, slcor2, slcor3.
  - Make sure the noise is on and psysPreset is okay.
  - Make sure the `CorrelatorBandServer` and `CorrelatorBandTimeChecker` processes are **NOT** running.

- If needed, reset all the boards:
  `/usr/local/carmaTools/cobra/CorrelatorReset -v --all`
- Try to get the thresholds and other low level bits set up:
  `/usr/local/carmaTools/cobra/digCorlWideband -v --all`
- Get the DSPs running, FPGAs programmed, and DSP time set up:
  `/usr/local/carmaTools/cobra/CorrelatorBandServer -v`
- Wait for the preceding to get to the steady state where it is publishing data every half second.
- Control-C this app and start it up again in non-verbose mode. This step is not required but it makes watching its messages easier.
- Start up the `CorrelatorBandServer` process (but *not* the time checker).
- Double check that noise is on and the power level is okay.
- Start up RTD to watch various MPs.
- Start up CDV to watch the 0.5 second data coming off the crate.
- If on the RTS and needed then use `~dwh/bin/host_comms` to match up the coarse/fine delays for the 2 inputs on the last digitizer card (the unconnected DigB input settings must match those of DigA).
- Run the flattener:
  `/usr/local/carmaTools/cobra/CorrelatorBandMonitorChecker -v --flatten`
- Wait until the flattener has decided it is done flattening or just isn't quite finishing but things look good enough in CDV.
- Again if on the RTS and needed then use `~dwh/bin/host_comms` to match up the coarse/fine delays for the 2 inputs on the last digitizer card. The flattener is supposed to maintain this state but either something else breaks it or the code has a flaw because they seem to not end up matched all the time. This should be investigated and fixed.
- If you want you can flatten progressively tighter by running the flattener again and again, reducing its flatten criteria via the `-flatten-cutoff=PS` command line argument
- Start up the `CorrelatorBandTimeChecker` process.
- Use a sac session, RTD, and CDV to double check that the band is working and can swing correctly from 500 MHz mode into 62 MHz mode and back to 500 MHz. Repeat the test moving from 500 Mhz to a narrowband mode and back to 500 Mhz for the 31, 8, and 2 MHz bandwidth modes.
- Done.

### 5.3. Board communications

In the COBRA system, access to the digitizer and correlator boards is provided by a device driver which exposes the boards as multiple devices; download, control, data, monitor, stdio. The control, data, and monitor threads within the server processes open specific device descriptors. The separate device descriptors are required so that the different data types are demultiplexed at the driver level; where the interrupts due to the different data types can be handled.

The proposal for the new CARMA boards is to create a virtual network over the PCI bus. The sockets API would the be used to interface to the boards. The different data types from the board would be demultiplexed by the TCP/IP stack, i.e., each data type would be a different socket connection. This implementation was considered for the COBRA boards, however, the DSPs were too resource constrained, and a TCP/IP stack was not available for the RTOS. In the CARMA design, both the board and the host CPU run Linux, so passing ethernet packets over the PCI backplane between the Linux network drivers is possible. This reduces the board-to-host Linux driver complexity, and allows the use of networking tools for debugging and monitoring communications.

Development of the CARMA board software will initially use a PowerPC development board. Communication with the development board is via ethernet and sockets code. The objective for the new CARMA board software would be to have the band server running on an x86 host receive board data using sockets code to communicate to the development board. The CARMA board prototypes will have front-panel ethernet, so the same sockets code will be used to test them. The ethernet interface has lower performance and less determinism than a PCI interface, so when complete bands of CARMA hardware are deployed the code would use the virtual network over PCI interface.

The current COBRA code does not communicate to boards using sockets. However, the bands server does communication to band servers using the appropriate code. The bands server client connection code can be used as the basis to develop the band server client connection code.

### 5.4. Control command requirements

The COBRA code uses a single thread to implement server commands. This enforces serialization of commands to the hardware, i.e., since the hardware is busy, it can not handle another command. Unfortunately this implementation does not work well with the CORBA control system 30s command timeout; long running commands exceed the timeout.

The COBRA control server code requires some rework. For example, the control server could be re-written to use two threads; a server thread, and a worker thread that performs the commands. The server thread can provide an immediate response to a long running command that the command has been accepted, and pass it off to the worker thread. The server thread would then reject further commands until the long running command completes. The control system is supposed to wait for a sequence number in the case of a long running command, so rejection of control commands until the sequence number is returned in the monitor stream is acceptable.

Basically the COBRA commands are implemented using synchronous I/O (the command returns when complete), whereas asynchronous I/O is desired (the command returns immediately, and the sequence number indicates when the command is complete).

The requirement for Asynchronous I/O for the band and board operations ultimately depends on how long each function takes. For example, if it takes 2s to load the FPGAs on a board, and there are 15 boards in a band. A band server implemented using a single thread to communicate with the boards will take 30s to complete, however, a server that uses a thread per board will require 2s to complete. In that case, using asynchronous I/O does not help the responsiveness of a band.

The current COBRA band server uses a single thread to deal with each board, so board operations are dealt with in sequence rather than in parallel. This made sense for some operations are the operations share a common resource; the PCI bus.

## 5.5. CORBA translator asynchronous I/O

The `CorrelatorCarmaServer` supports making commands asynchronous internally. Incoming DO commands are placed onto a queue and a separate thread actually dispatches them to the COBRA code in the order they were queued. The queuer side will queue the request and then wait a short time (1 second?) to see if the command can complete fast. If not then it just returns from the DO call and trusts that the command will complete at some point. On the processor side CORBA commands are given up to 30 seconds to complete before they time out and log the error. For commands with a sequence number (bandwidth change is the only one implemented at present) then the sequence number and success boolean info is posted by the processor to a shared piece of memory when the call errors out, times out, or completes successfully. This shared piece of memory is read by the monitor publisher thread when each COBRA monitor packet is received and the info is used to set the values of the CARMA monitor points for sequence numbers and success state.

Something that could be explored is coalescing consecutive compatible command requests in the queue. This would allow things like combining consecutive delay sample commands that are queued behind a bandwidth change command. If this is to be explored then I would recommend doing it on the processor end of the queue. That is, the processor would keep popping things off the queue and combining them for as long as a compatible command is available at the top of the queue. It would stop and actually issue to the COBRA code when no more combinable commands are at the top of the queue. However, this whole scheme is probably not needed if either the multiple consecutive command overhead is not hurting us or if the upstream clients (i.e. the interferometry engine) can be taught to not issue such command sequences in the first place.

At present, what seems to happen is a bandwidth change command followed immediately by a downconverter settings command that stalls in the queue behind bandwidth change and then some number of delay sample commands which also stall waiting for the BW change and DC settings command. It should be examined if the DC settings command should be fused to the BW change to form a single command that set the sequence number only when both pieces have been completed.

### 5.6. ACE logging redirect

The `CorrelatorBandServer`, `CorrelatorCarmaServer`, and `CorrelatorBandTimeChecker` processes can all be told to redirect ACE log messages to the CARMA logs. This is done by setting `redirectAceLogging=true` on the command line. This in turn causes the code to call the installAceLoggingBackend() function defined in carma/correlator/obsRecord2/aceUtils.cc This will install a logging backend hook class instance into the ACE logging system. At present the instance will throw away all ACE logging events that are below `ACE_WARN` level and log the remaining logging event into the CARMA logging system at `WARN` level with a prefix that tells you it was an ACE logging event and the ACE logging level that was used. I don't log at CARMA `ERROR` level even for ACE `ERROR` level because too many things that aren't actually errors get logged at this level in the COBRA code. This should be easy to clean up and the policies in aceUtils.cc are pretty obvious and easy to adjust if and when it is needed.