



# Design and Operation of the Level 1 Track Trigger Monitoring Code

**Michael P. Cooke**  
Rice University

**Kyle J. Stevenson**  
University of Indiana

## **Abstract**

The D-Zero Level 1 Central Track Trigger ( L1 CTT ) is designed to reconstruct axial tracks during each Tevatron bunch crossing at D-Zero. These tracks are then used to form basic and/or terms which can be used by the D-Zero Trigger Framework for event selection. Online monitoring code was developed by the authors to access information that is produced during various stages by the CTT electronics boards.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Review of the Central Track Trigger Design</b>	<b>1</b>
<b>3</b>	<b>Design of the Monitoring Code</b>	<b>1</b>
<b>4</b>	<b>How to Use the CTT Examine</b>	<b>3</b>
4.1	Starting the CTT Examine . . . . .	3
4.2	CTT Examine Run Control Parameters . . . . .	3
<b>5</b>	<b>Information Displayed by the Monitoring Code</b>	<b>4</b>
5.1	Graphical User Interface and Histogram Organization . . . . .	4
5.2	Comparison Histograms . . . . .	6
5.3	General Information Plots . . . . .	9
5.4	Expert Information Plots . . . . .	13
<b>6</b>	<b>Using the Monitoring Code to Identify and Diagnose Problems</b>	<b>16</b>
<b>7</b>	<b>Details of the CTT Examine Code</b>	<b>16</b>
7.1	Run Control Parameter (RCP) Files . . . . .	18
7.2	Unpacking the Data . . . . .	22
7.3	Significant Event System . . . . .	25
7.4	Booking and Filling ROOT Histograms . . . . .	26
7.5	Socket Communication in ROOT . . . . .	27
7.6	ROOT Macros and Displaying Histograms . . . . .	29
7.7	Graphical User Interface in Python . . . . .	31
<b>8</b>	<b>Conclusion</b>	<b>34</b>

# 1 Introduction

The design and operation of the L1 CTT online monitoring code ( the package is called *l1CTT\_examine* ) is discussed in this note. The information provided by the online monitoring code is of importance in both the commissioning and operation of the track trigger.

## 2 Review of the Central Track Trigger Design

The Central Track Trigger is comprised of a series of field programmable gate array's (FPGA's), whose main function is to perform a rapid series of calculations (aimed at basic axial track reconstruction) in the time needed to arrive at a trigger decision (this is essentially the time between Tevatron bunch crossings, which is 396 ns, at level 1 decision time). The series of electronics boards, which contain these logic gates, is shown in Figure 1. The acronyms for these electronics boards and their basic functions are described below.

- DFEA (*Digital Front End Axial*)  
The DFEA boards each contain two daughtercards [1], each daughtercard containing 5 FPGA's and serving a  $4.5^\circ$  axial sector. The function of the logic algorithms defined in these cards is to compare hits from the fiber tracker detector with pre-selected combinations of hits which are deemed to form good axial tracks. In addition axial tracks are matched to CPS clusters in the logic in order to trigger on electrons and CPS clusters are recorded in order to indentify photons. Each DFEA board in the l1 CTT chain has two output LVDS (low voltage differential signal) cables which form the inputs to the CTOC boards (defined below).
- CTOC (*Central Track Octant*)  
The CTOC boards takes as inputs the tracks formed in the DFEA daughtercards and sum the tracks on an octant level. The level 1 outputs of the CTOC boards are tracks and CPS hit counts at the octant level and these are then sent to the CTTT board via LVDS cables for further logic operations. The inputs to the CTOC's that were recieved from the DFEA LVDS cables are then output to level 3 from the CTOC boards via g-links cables. Hence there are a total of 8 g-link outputs for this portion of the chain. Each one contains the information transferred
- CTTT (*Central Track Trigger Terms*)  
The CTTT card forms the basic and/or terms which are sent to the level 1 CFT/CPS (Central Fiber Tracker/ Cental Pre-Shower [2] ) trigger manager. These form the basis for level 1 specific triggers used to decide whether or not events are selected for level 2.

## 3 Design of the Monitoring Code

The monitoring code was designed to run with ROOT and to display graphs relevant to the operation of the l1 CTT in real-time. In order to help diagnose problems the code

## DZERO Central Track Trigger

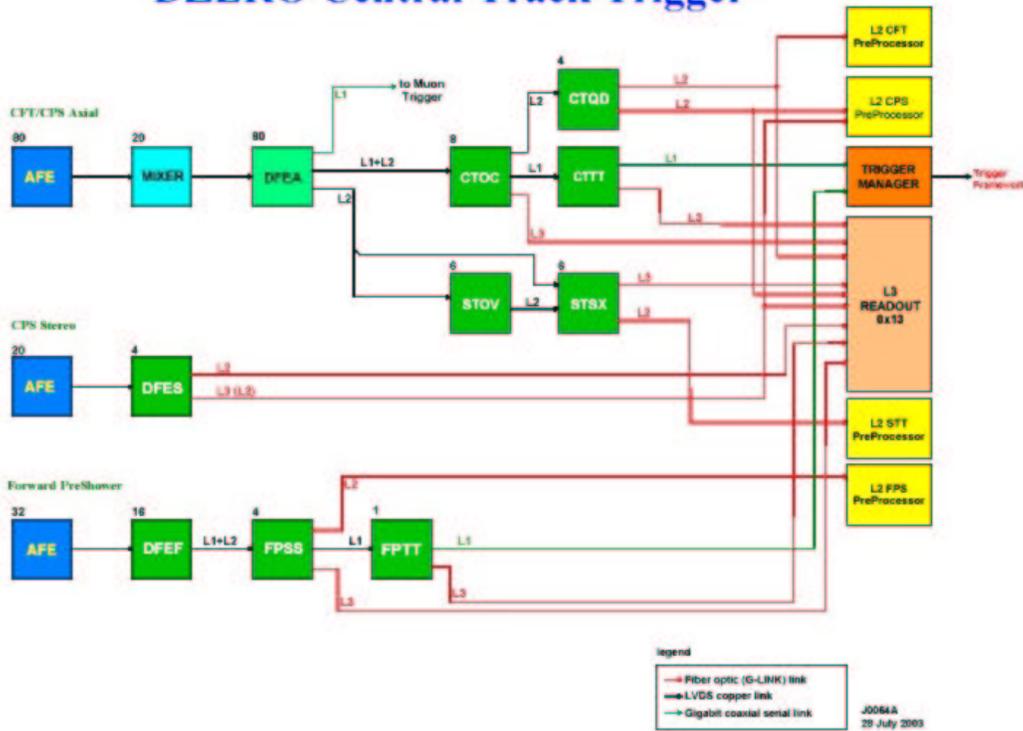


Figure 1: The various electronic boards that make up the level 1 CTT chain and the various cables connecting them. The information that is used by the monitoring code is the information that is transferred through g-link cables from readout crate 0x13 to level-3, where the information is then saved to tape.

runs the level 1 track trigger simulator [3], the output of which can then be compared to the actual response of the trigger electronics. Plots that overlay the online data with the output of the trigger simulator are the “standard” plots used by CFT shifters to monitor the status of the CTT.

The examine program itself is a compiled C++ executable that runs within the D-Zero framework and makes use of RCP files to specify run-time options. It is stored in the Concurrent Versions System (CVS) as package “l1CTT\_examine”. Both online data, streamed from the distributor as the D-Zero detector is taking data during a run, and offline data, read from a “.raw” file, can be used as inputs to the CTT Examine by adjusting the RCP files. For each event sent through the examine, the routine will first fill a set of histograms based on crate x13 (crate 19 in decimal) information, which includes copies of many of the data frames sent between boards in the CTT chain. After the first set of “data” histograms are filled, the trigger simulator package is called and a second full set of histograms are filled based on simulated CTT chain outputs using either SVX or VSVX input. Either of these sets of histograms are available, through socket communication that is built into the ROOT libraries, to the ROOT macros that display the final histograms on screen.

A Graphical User Interface (GUI, see Figure 2, pg 6), written in the scripting language Python, is the user’s front end to the CTT Examine. Through this GUI users can run ROOT macros (which are syntactically based on C++) which communicate with the main examine program and display sets of related histograms. Links to important information online are also provided through the GUI. The ROOT macros that the GUI calls will query the examine program, through the computer’s ports, asking for copies of histograms for display. After receiving them from the examine, the macros can normalize histograms, color them, and overlay or stack them in order to produce the most comprehensible plots possible for the shifter. Histograms are displayed in sets related to one another in order to enhance the clarity of the information displayed. By viewing related plots together, it’s possible to confine the possible causes of errors to a small set with reasonable certainty.

## 4 How to Use the CTT Examine

### 4.1 Starting the CTT Examine

Like other examines, the CTT Examine is started with a “start\_daq” command. To monitor the CTT as part of a control room shift, log into “d0o135” as user “d0cft” and issue these commands:

```
> ssh d0o135 -l d0cft
...
[d0cft@d0o135 ~]\$ setup d0online
[d0cft@d0o135 ~]\$ start_daq ctt_examine vsvx_global
```

During a zero-bias run, replace “vsvx\_global” with “vsvx\_zero” in order to get the proper data stream from the distributor. These commands will start the examine code and bring up the GUI window. Many commands can be used in place of the “vsvx\_global” option. These commands are described in Table 1. The examine code takes a minute or two to start processing events, so macros will not display histograms for some time after these commands are issued. During a run, histograms can be reset by pressing the “Reset” button in the GUI. In order to stop the examine, the GUI window must be closed (using the button labeled “Ex” next to the word “Exit”) and the examine process must be stopped (usually accomplished by “Ctrl+C”).

While it is possible to feed the trigger simulator SVX data, the most accurate agreement between online data and simulated data will occur with the VSVX data since it very closely emulates the input to the CTT hardware.

### 4.2 CTT Examine Run Control Parameters

Various aspects of the CTT Examine’s operation are controlled by Run Control Parameter (RCP) files. These RCPs can be found in the “l1CTT\_examine/rcp” directory. The RCPs are described briefly in Table 2 and a more thorough discussion can be found in Section 7.1. Most notable of these RCPs are “cft\_trk\_data.rcp”, which specifies the track sector equation files that the trigger simulator (tsim) loads and whether it uses SVX or VSVX information as input, and “tsim\_l1ft\_data\_vsvx.rcp” (or ...\_svx.rcp), which contains

Argument	CTT Examine Response
vsvx_global	start in online mode x13 data and trigsim data VSVX
vsvx_any	start in online mode x13 data and trigsim data VSVX
vsvx_zero	zero bias online mode x13 data and trigsim data VSVX
vsvx_x25	l1ctt_x25 tick 25 online mode x13 data and trigsim data VSVX
svx_global	start in online mode x13 data and trigsim data SVX
svx_any	start in online mode x13 data and trigsim data SVX
svx_zero	start in online mode x13 data and trigsim data SVX
vsvx_off	start in offline mode x13 data and trigsim data VSVX
svx_off	start in offline mode x13 data and trigsim data SVX

Table 1: Command line arguments for “start\_daq ctt\_examine”

the parameter that will tell the trigger simulator to either output a single one of or rotate between sending DFEBoardL1, DFEBoardL2 and DFEBoardL2CPSA information. Many other RCPs are required to remain static in order for the CTT Examine to operate, so it is not advisable to adjust other RCP files without serious consideration.

Note that the RCP file “l1CTT\_examine.rcp” contains many options that adjust the operation of the CTT Examine code. The “graph\_level” parameter controls the general number of histograms booked and filled by the Examine. This parameter can take the values “0” through “4”, and higher values include the operations of lower numbers. Setting this parameter to “0” will turn off all but the Level 3 Header information histograms, while all other information will be ignored. This would make the CTT Examine create fewer histograms and run much faster, though if the trigger simulator is turned on processing events will be slow due to it. Currently, setting “graph\_level” to “1” acts exactly like setting it to “0”. If it is set to “2”, DFEA→CTOC histograms will be created and their information processed. When set to “3”, all Level 1 cards, now including CTOC→CTTT information, will have histograms booked and information filled. At level “4”, the CTT Examine will also book histograms and process information based on Level 2 cards.

Other parameters in this RCP include “debug1”, which sets the level (from 0 = low to 10 = high) of debug information written out to the screen as the CTT Examine runs, “online”, a boolean variable which determines whether the examine will try to connect to ROOT macros via socket communication or write all of the histograms out to a ROOT file, and “root\_file”, which defines the name of that output ROOT file.

## 5 Information Displayed by the Monitoring Code

### 5.1 Graphical User Interface and Histogram Organization

The information displayed by the monitoring code is arranged into several layers (see Section 4.2 about “graph\_level”). These layers of information are displayed as separate button groups in the CTT Examine Graphical User Interface (GUI, see Figure 2). The CTT Examine Graphical User Interface (GUI) includes buttons for each set of histograms that can be displayed. Information, or “inf”, buttons are also included to describe what data many of these histograms are based upon. All ROOT macros called by the GUI are

<b>RCP File</b>	<b>Summary of Parameters Contained</b>
ReadEvent.rcp	Specify “.raw” files to read in offline mode
ReadEventDaq_l1ctt_x25.rcp, ReadEventDaq_zero_bias.rcp	Modified read_event_daq RCPs that initiate the “daq_test” or “zero_bias” streams from the distributor
cft_trk_data.rcp	Specify tsim track sector files and choose SVX or VSVX input
cft_trk_data_svx.rcp	Copy of cft_trk_data intended for use with SVX input
l1CTT_examine.rcp	Sets various input and output parameters for CTT Examine
l1l2_collector.rcp	Modified tsim_l1l2 RCP that eliminates the “L2 send to L3” iogen objects
run_dsvx_offline.rcp, run_dsvx_online.rcp, run_dsvx_online_all.rcp, run_dsvx_online_any.rcp, run_dsvx_online_zero.rcp, run_dsvsx_offline.rcp, run_dsvsx_online_all.rcp, run_dsvsx_online_any.rcp, run_dsvsx_online_x25.rcp, run_dsvsx_online_zero.rcp	Base RCP files that specify the other RCP files necessary to run the examine in various modes. These RCPs are called on the command line with the examine executable by the various startup scripts (which follow the same naming convention). The “start_daq” command for initiating the examine will call the proper script, and thus the proper RCP file, based on the argument used in the command line (see Table 1).
tsim.rcp, tsim_svx.rcp, tsim_vsvx.rcp	Modified tsim_l1ft RCPs that specify the proper “tsim_l1ft_data” RCP files
tsim_l1ft_data_svx.rcp, tsim_l1ft_data_vsvx.rcp	Modified tsim_l1ft RCPs for use in SVX and VSVX input modes; contains the switch for choosing one of or cycling between DFEBoardL1, DFEBoardL2 and DFEBoardL2CPSA information

Table 2: Brief summary of RCP files found in CTT Examine. A thorough discussion of these files takes place in Section 7.1.

tracked and their processes are killed when either the “GCL” button is pressed or the GUI is closed. A drop-down box on the right allows experts quick access to any macro without a limit to the number of spawned windows, all of which can be cleared normally.

The most basic information in the Examine is recieved at the VRB level [4]; the information that is displayed shows the number of bytes flowing into the VRB’s from the various electronics components of the CTT chain that send information to crate x13 (which contains the VRB’s). This information is useful for tracking problems with the physical readout of crate x13. The GUI contains buttons for viewing this information just to the right of the central 80-sector tracking image, though often this information is not important to someone on shift.

The next layer of information is the output of the DFEA cards to the Level 1 Trigger hardware chain, or DFEA→CTOC data. This data is displayed by buttons on the left of the GUI labeled “DFEA L1”. The most important monitoring histograms are displayed

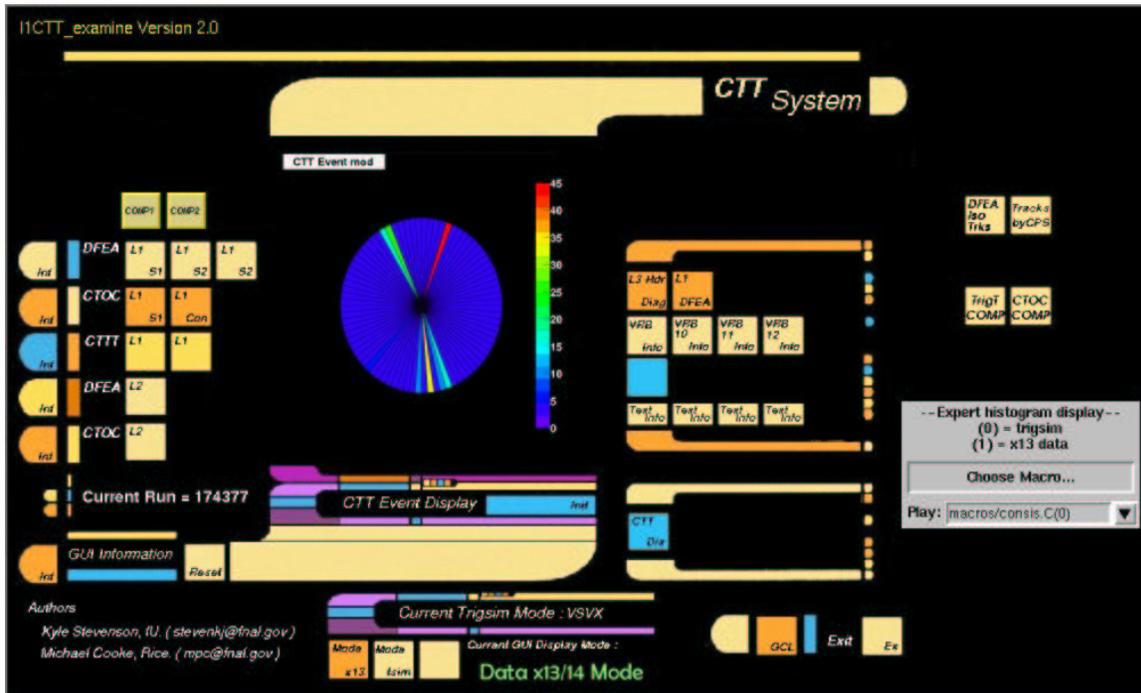


Figure 2: The CTT Examine Graphical User Interface (GUI) includes buttons for each set of histograms that can be displayed.

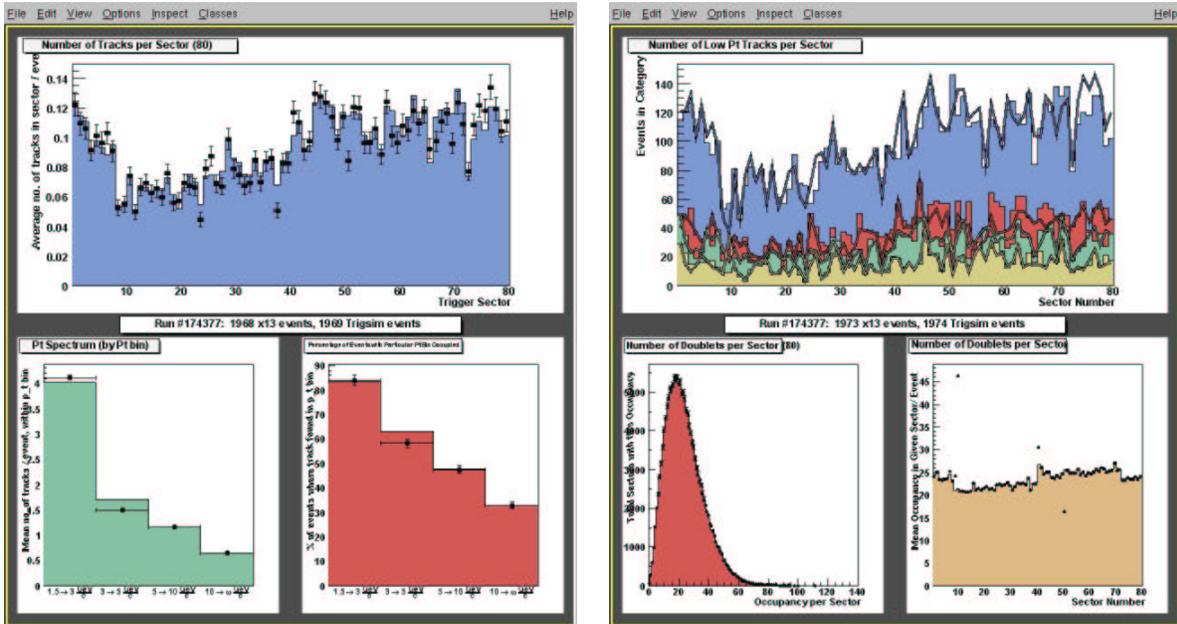
by the buttons just above this section of the GUI labeled “COMP1” and “COMP2”, which display an overlay of DFEA L1 data from the trigger hardware and from simulated trigger output. These histograms are discussed in Section 5.2. The dots on these plots represent the data from crate x13 while the colored histograms are simulated trigger output. If there are problems in the CTT chain, these two graphs will quickly make it evident.

Following the DFEA L1 output layer is one containing the outputs from the rest of the L1 CTT hardware chain. Buttons for plots containing these data sets follow just below the DFEA L1 row in the GUI. These plots are described thoroughly in Section 5.3.

Finally, Level 2 information from the hardware is analyzed in the CTT Examine. This layer of information is listed in the “DFEA L2” and “CTOC L2” plots. This histograms will be among the last to be implemented. Currently, only “DFEA L2” has any histograms associated with it, and those are only filled by the trigger simulator.

## 5.2 Comparison Histograms

The most significant plots that the CTT Examine produces are the “comparison plots,” where data from the CTT hardware chain (received through crate x13) is displayed on the same plot with data from the trigger simulator (tsim). In these plots, shown in Figure 3, the colored histograms represent data from tsim while the data points with error bars, or colored lines in the case of the “COMP2” plot, represent data from crate x13. The error bars on crate x13 data are simply the square ROOT of the number of events in that data bin, intended mainly to give a visual cue to the number of events currently in the data set. All histogram sets contain a field that display the run number of the input data and a count of the number of events that have contributed to filling the displayed histograms.



“COMP1” (dfea\_C.C)

“COMP2” (dfea2\_C.C)

Figure 3: “Comparison Plots” will overlay data from Crate x13 as points on colored histograms crated from the trigger simulator. The error bars on x13 data are the square ROOT of the number of events in that bin.

Most significant of the comparison plots are the “COMP1” and “COMP2” histogram sets, which display various aspects of the L1 DFEA data. Viewing this plots and recording them in the online logbook is part of the regular duties of a CFT shifter. Note that these two plots have two counters: one for the number of crate x13 events read and one for the number of trigger simulator events read. It is normal for these counters to be within one of each other; a drastic difference between these numbers usually means that many crate x13 events do not contain 80 DFEA card outputs.

The “COMP1” set of plots, drawn by the ROOT macro “dfea\_C.C”, displays three histograms (see Figure 3). The upper plot (in blue), shows the average number of tracks found per event for each of 80 trigger sectors in the CTT. Because 8 layers of axial hits are required in the CFT to create a track in the CTT, problems with even a single layer of fibers in the CFT can cause the number of tracks found in the affected sectors to drop dramatically. This distribution should be flat across all sectors during optimal running conditions. During a zero-bias run, tracks will rarely be created and this graph will be a series of a very few, randomly scattered, spikes.

The lower left plot in “COMP1” (in green) shows the average number of tracks created for each of the four level 1 pt bins per event. Normally, this graph shows highest occupancy in the lowest pt bin and occupancy drops off with increasing pt. The lower right plot (in red) shows closely related information, displaying the chance (in percentage) of finding any tracks in a given level 1 pt bin during an event. This plot also tends to fall off in value with increasing pt.

The “COMP2” set of histograms, drawn by the ROOT macro “dfea2\_C.C” (see Figure 3), compliment the information shown in “COMP1”. The upper plot of “COMP2” shows the total number of tracks found in a given pt bin for each of the 80 trigger sec-

tors. Here, the pt bins are color coded: blue represents the lowest pt bin, 1.5→3 GeV/c; red denotes the second pt bin, 3→5 GeV/c; green indicates events in the third pt bin, 5→10 GeV/c; and yellow events are in the highets pt bin, 10 GeV/c and beyond. Color coded lines represent the crate x13 information in this plot and error bars are omitted to increase readability. The sum of all four of the pt bins in a given sector, divided by the total number of events read by the examine, is the upper, blue, plot in “COMP1”.

The lower left plot (in red) of “COMP2” shows the distribution of doublet occupancy for all the sectors. The number of doublets fired from each sector is entered into this histogram, causing 80 entries per event, and this total doublet distribution is accumulated. Crate x13 information will match the trigger simulator very closely in the chart if VSVX information is used as input to the simulator; SVX information fed into the simulator tends to cause the simulator to have a higher average number of doublets fired in a sector than crate x13 output indicates, and the overlay will not match closely.

Finally, the lower right plot (in yellow) found in “COMP2” shows the average doublet occupancy per event of each of the 80 trigger sectors. The average doublet count here should correlate with the peak of the doublet occupancy distribution graph next to this chart. A flat distribution across all trigger sectors is expected here.

Two other comparison plots (see Figure 4) are available but are not as directly useful to CFT shifters in determining how well the CTT hardware chain is performing during a data run. These comparison plots, “CTOC Comparison” and “Trigger Term Comparison,” are created by the “ctoc1\_C.C” and “trigbits\_C.C” ROOT macros, respectively.

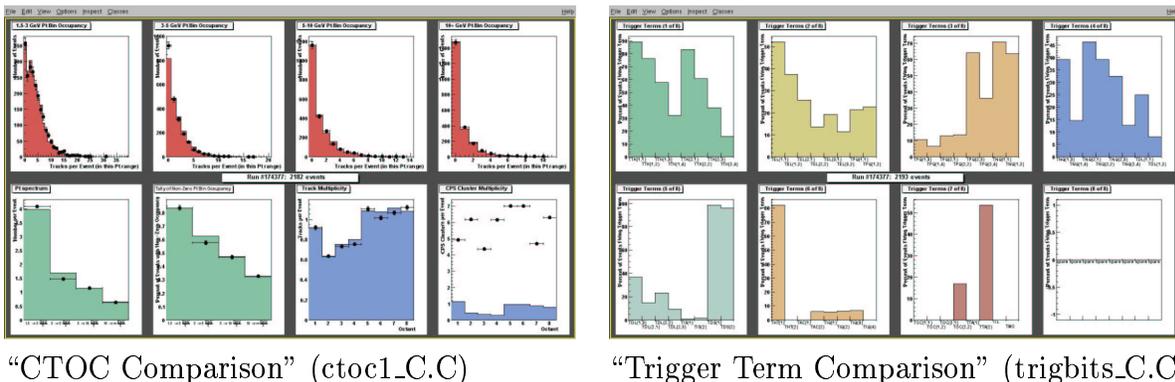


Figure 4: “Comparison Plots” will overlay data from Crate x13 as points on colored histograms crated from the trigger simulator. The error bars on x13 data are the square ROOT of the number of events in that bin.

In the top four plots (all in red) of the “CTOC Comparison” plot, the total number of tracks in a given level 1 pt bin are summed, across all octants, and that number is accumulated in the corresponding histogram. There is a histogram for each of the four level 1 pt bins, and each histogram displays the x-axis (the total number of tracks found in an event for that pt bin) only out to the highest filled bin.

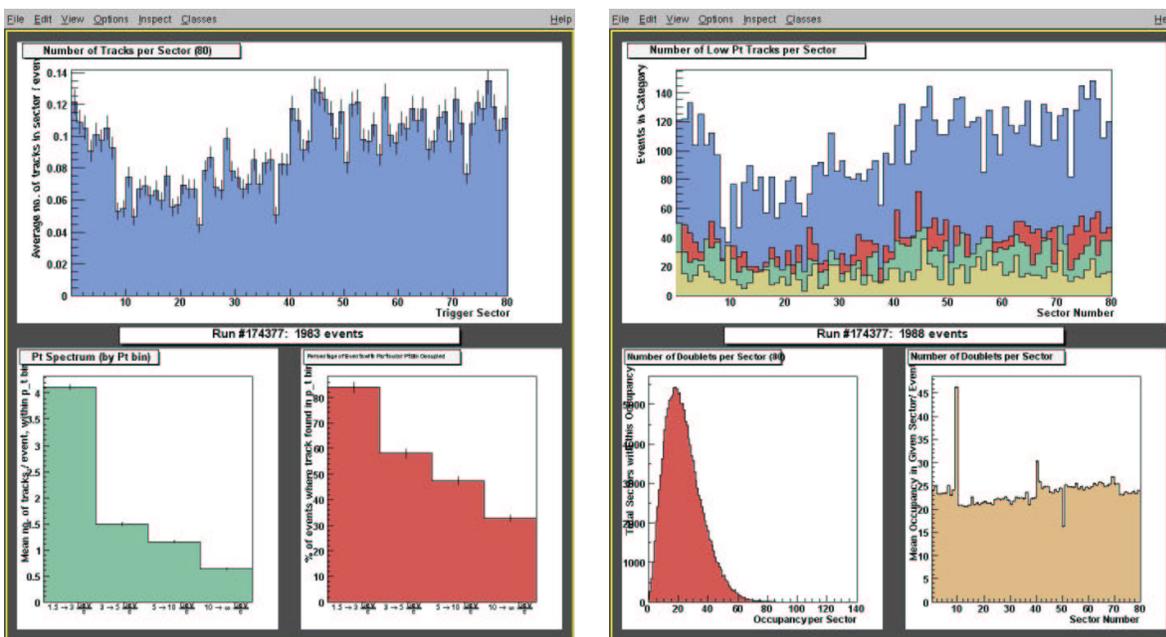
The lower left plots two plots (in green) display the same information as the lower plots of “COMP1”, though now they represent data from the CTOC cards and not L1 DFEA information. The third plot in the bottom row (in blue), labeled “Track Multiplicity,” shows the average number of tracks found in an octant per event, with a seperate bin for each octant card. The lower rightmost plot (in blue), labeled “CPS Cluster Multiplicity,”

displays the number of Central Preshower Clusters found in each octant, with a separate bin for each octant.

Last in the set of comparison plots, the “Trigger Term Comparison” diagrams show that percentage of events that satisfy the conditions for each of the CTT trigger terms. As trigger terms are implemented in the CTT hardware chain, crate x13 data points will appear on these charts. The trigger simulator does not currently have all of the CTT trigger terms programmed into it as well, so some terms may remain at “0%” for some time. Additionally, because the CTT Examine does not run on every event, and in fact gets a datastream from the distributor containing a level 3 record, these rates *will not* match the And/Or term rates in the Data Acquisition monitoring programs which determine rates based on the number of events occurring in the detector.

### 5.3 General Information Plots

All of the CTT Examine histogram sets in this section can display either crate x13 information or trigger simulator data, but not both simultaneously like the comparison plots described in Section 5.2. A pair of buttons in the lower central portion of the GUI (see Figure 2), labeled “Mode x13” and “Mode tsim”, control which information source is used to create the histograms displayed. The current selection, either “Data x13/14 Mode” or “Trigsim VSVX Mode” (or “SVX” if that was used as tsim input), is displayed in green to the right of this button panel. Note that only by using the “Expert Histogram Display” portion of the GUI can you display the same plot in both tsim and crate x13 modes; clicking the same plot button a second time, even if the tsim/crate x13 mode button has been pressed in between, will clear the first plot displayed and then display the second.



“DFEA→CTOC Links : Set 1” (dfea.C)

“DFEA→CTOC Links : Set 2” (dfea2.C)

Figure 5: These plots, similar to “COMP1” and “COMP2”, display DFEA L1 information from either crate x13 output or the trigger simulator output.

“DFEA→CTOC Links : Set 1” and “Set 2” were the basis of “COMP1” and “COMP2”. See Section 5.2, for details on the information contained in these sets of histograms. The most notable difference between these plots and the comparison plots is that these histograms display error bars proportional to the square ROOT of the number of events that have accumulated in each bin of the histogram, much the same as the error bars for crate x13 data points in the comparison plots were calculated. Similarly, “CTOC→CTTT Information” (see Figure 6) was the basis of the “CTOC Comparison” plot discussed in Section 5.2.

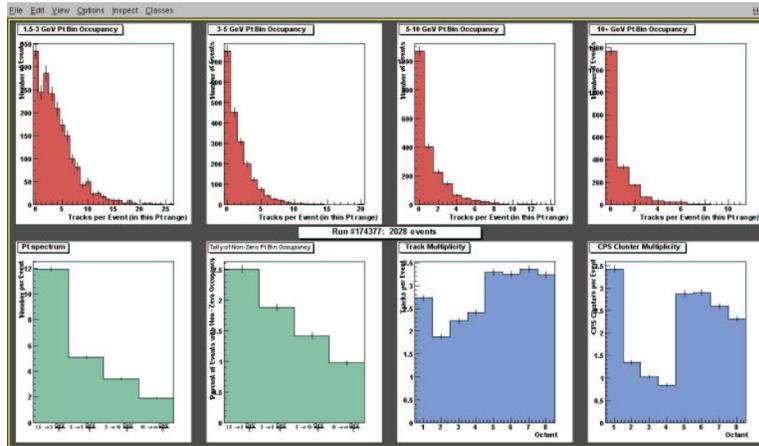
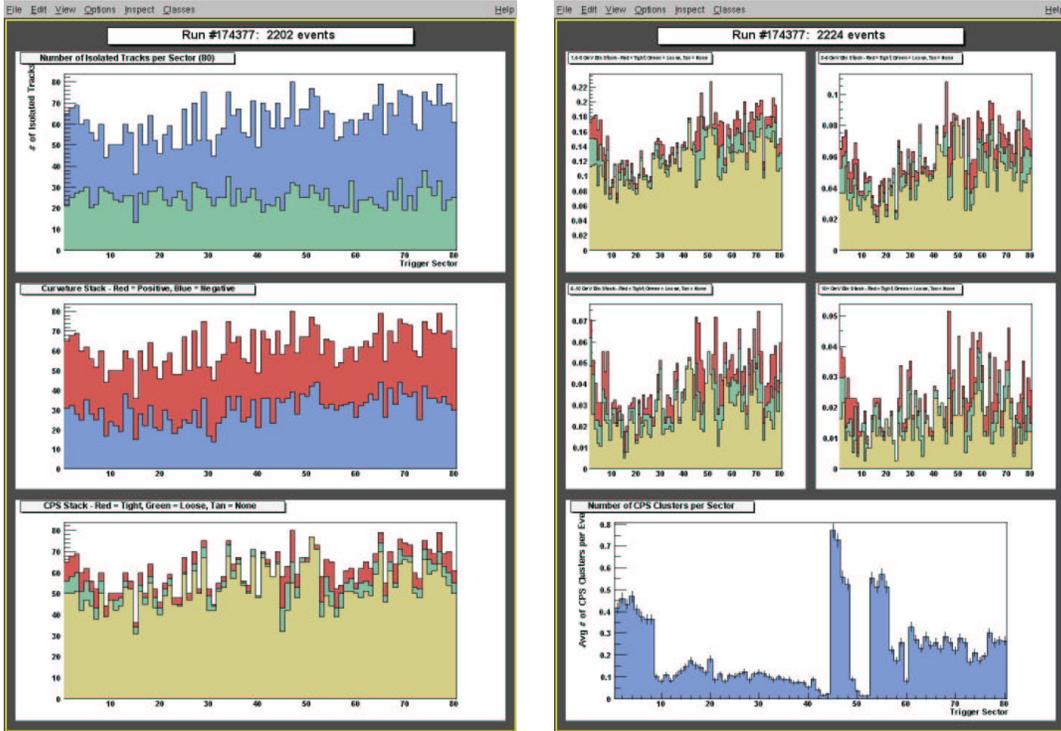


Figure 6: “CTOC→CTTT Information” (ctoc1.C)

“Isolated Track Information (DFEA)” and “DFEA Tracks by CPS Association” (see Figure 7) are both based on L1 DFEA information, much like the “DFEA→CTOC Links” (or “COMP1/2”) plots. All of the plots in both of these histograms show information binned by trigger sector.

All of the plots in “Isolated Track Information (DFEA)” are constructed from information contained in the isolated track field in the L1 DFEA information. The upper plot (blue and green) shows the total number of isolated tracks found in blue and the subset of that total, shown in green, that satisfy the “low occupancy” constraint (bit 00 in the isolated track field for L1 DFEA data frames). The middle histogram (red and blue) displays all isolated tracks color coded by their curvature. Here, the blue portion of the bar represents the portion of tracks with negative curvature and the red portion denotes those with positive curvature. Note that the sum of these two adds up to the total number of isolated tracks found. The lower histogram (red, green and tan) color codes isolated tracks by the association they have with CPS clusters. The portion of each filled trigger sector bin that is associated with no CPS cluster is colored tan. Red denotes isolated tracks associated with tight CPS clusters, and green indicates the portion of isolated tracks with loose CPS association. Once again, the colors are shown as a portion of the whole bar, whose height is the total number of isolated tracks in each trigger sector bin.

“DFEA Tracks by CPS Association” displays a total of five histograms (see Figure 7), all once again binned by trigger sector. Each of the upper four (red, green and tan) display all tracks that fall into a single level 1 pt bin, one histogram for each pt bin, and are proportionately color coded based on CPS cluster association. The upper left



“Isolated Track Information (DFEA)”  
(isotrk.C)

“DFEA Tracks by CPS Association”  
(tracks\_by\_cps.C)

Figure 7: These plots show L1 DFEA information. Isolated tracks are defined to be alone in their sector with no tracks in immediately neighboring sectors. Standard DFEA tracks can have tight, loose or no CPS cluster association.

plot displays the  $1.5 \rightarrow 3$  GeV/c tracks, upper right the  $3 \rightarrow 5$  GeV/c, lower left the  $5 \rightarrow 10$  GeV/c, and lower right the 10 GeV/c and higher transverse momentum tracks. The color coding is much the same as in the “Isolated Track Information (DFEA)” plot, with tan representing a track with no CPS cluster associated with it, red a tight CPS cluster association, and green a loose CPS cluster association. The lower plot (blue) of this set of histograms displays the average number of CPS clusters found in a given trigger sector per event.

The information plots shown in “Level 2 DFEA Histograms” are driven by the differences between the data DFEA cards send to the Level 1 CTT chian compared to the data sent to Level 2 CTT hardware. For the Level 2 portion of the CTT, DFEA cards send more specific information as input to higher hardware cards along the chain. This information includes a finer binning of  $p_t$ , specific doublet number within a sector that a track instersects in the outermost axial layer, and the center fiber of an associated CPS cluster, if one exists. There are twenty extended level 2  $p_t$  bin, four sub-bins in each of the highest level 1  $p_t$  bins, and eight sub-bins to the lowest level 1  $p_t$  bin. The highest eight level 2  $p_t$  bins, for tracks above 5 GeV/c, are based on ranges of possible  $p_t$ , while the lower 12  $p_t$  bins are based on the “fiber offset.” This term refers to the number of fibers between where a straight line (infinite momentum) track would cross the innermost axial layer of the CFT, assuming it exits at the same detector phi of the outer axial layer hit, and where the actual track crosses the first axial layer.

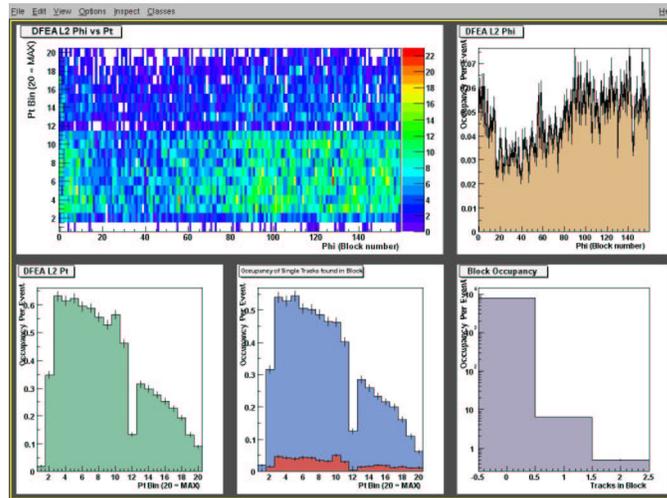


Figure 8: “Level 2 DFEA Histograms” (level2.C)

Also notable is that at this level tracks are grouped into “blocks,” which are half of one sector, making 160 blocks in the detector. The algorithms used to find tracks for the CTT can find only up to two tracks in any level 2 pt bin for a given block of the detector. The two tracks found would be the closet tracks to either end of the block (the “position” of the track determined by where it exited the outermost axial layer), so a third track in the “middle” of a block would not be found. In general within the CTT chain higher pt tracks are given precedence over lower pt tracks, so if a long list of tracks must be truncated to fit a data frame that will be sent along the CTT hardware chain, highest pt tracks are entered first.

“Level 2 DFEA Histograms” (see Figure 8) displays track information derived from data fields that DFEA cards send along the Level 2 CTT hardware chain. The upper left and largest diagram in this set of plots is a two dimensional histogram displaying the total number of tracks found in every level 2 pt bin within each block of the detector. This information is displayed as a color map, with the color axis key on the right side of the histogram. Many events must be processed by the examine to accumulate this plot to a reasonable degree. The upper right plot (tan) is the average number of tracks found per event for each of the 160 blocks. This distribution should be flat, though statistical fluctuations will likely cause great variance until a large number of events have accumulated.

The lower left plot (green) of “Level 2 DFEA Histograms” shows the average number of tracks found in each level 2 pt bin, where bin “1” represents tracks with the lowest momentum and bin “20” representing tracks with pt higher than 40 GeV/c. The central lower plot (blue and red) displays similar information as the last histogram, but here events with a single track found in a block are displayed in blue and events with two tracks found in the same block are superimposed in red. (Note that twice the red portion plus the blue portion will equal the lower left plot in green.) This gives an indication of the level 2 pt bins likely to lose tracks due to blocks that have three or more tracks in an event. Similarly, the final plot (violet) in the lower right corner can help determine if many tracks are lost due to counting only the first two tracks found in a block. For each event, this histogram accumulates a separate bin in the cases that 0, 1 or 2 tracks are

found in a particular level 2 pt bin and block combination. The average number of times this occurs each event is displayed.

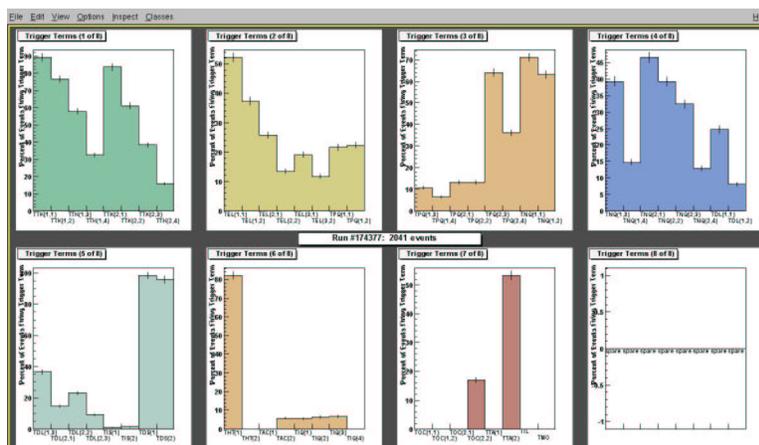


Figure 9: “Level 1 Track Trigger Terms” (trigbits.C)

Finally, the “Level 1 Track Trigger Terms” (Figure 9) histogram set displays the CTT trigger terms fired for each event processed by the CTT Examine. This plot is the basis of the comparison plot “Trigger Term Comparison,” as described in Section 5.2.

## 5.4 Expert Information Plots

CTT Examine plots described in this section show very specific information mainly related to the passing of data to and through various CTT hardware chain cards. Data field header checks, information consistency, and transmission error bits are common features of these plots. Hardware experts for the CTT can identify hardware problems quickly with this information, but control room shifters have no need to regularly check these plots during normal detector operation.

The “DFEA Parity Errors” (see Figure 10) set of histograms displays information based on the horizontal and vertical parity bits in the DFEA→CTOC data frames. The average total number of vertical and horizontal parity bits found in an event are displayed in the upper left and upper right histograms, respectively. The lower left and lower right show an accumulation of the frames in which vertical and horizontal parity errors, respectively, occur per event.

A set of histograms labeled “Data Consistency: DFEA→CTOC vs. CTOC→CTTT” (see Figure 11) show results of a direct comparison between information contained in the ten DFEA→CTOC that contribute to a CTOC→CTTT data frame for all eight CTOC cards. Each CTOC data field (see [5], which describes all Level 1 CTT data frame protocols) is checked against the ten DFEA cards that contribute information to it. The error rate for any data field is listed as a percentage for each octant card in the large left two dimensional plot, color coded for easy reference. A few extra checks are performed, beyond the CTOC data fields alone, namely a check for matching numbers of Isolated Tracks within the CTOC card data fields and the ten DFEA card’s information.

The upper right plot, colored grey, shows an accumulation of the total number of data parity errors in an event on a logarithmic scale. Both horizontal and vertical parity errors

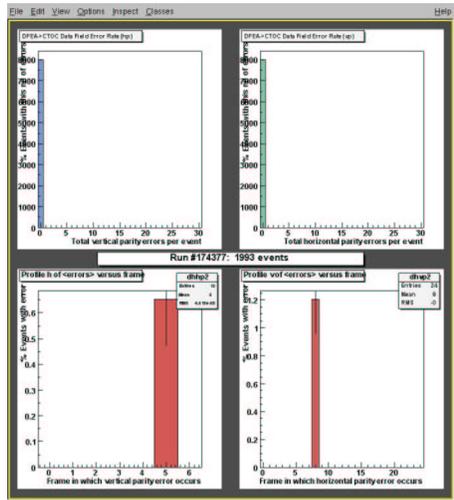


Figure 10: “DFEA Parity Errors” (parity.C)

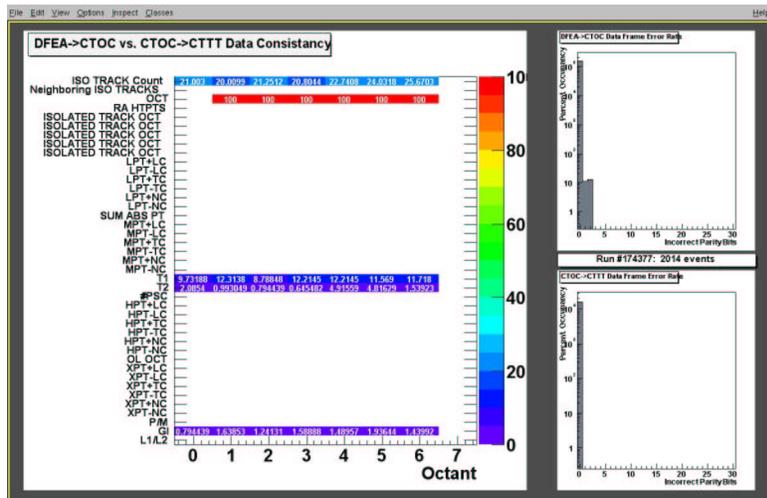
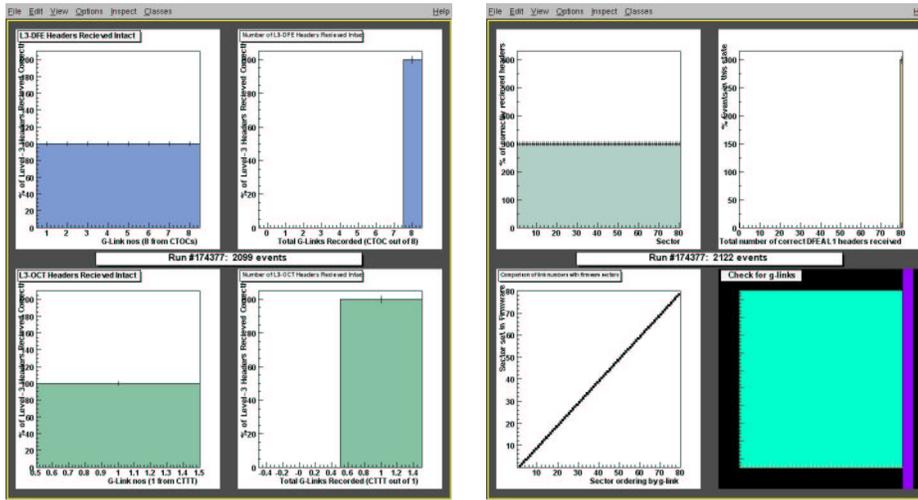


Figure 11: “Data Consistency: DFEA→CTOC vs. CTOC→CTTT” (consis.C)

are included in this total. The lower right plot shows a similar parity error check on the data frames sent from the CTOC card.

Level 3 header information is examined in the “L3 DFE/OCT Header Checks” (see Figure 12, left) histogram set. The upper plots, in blue, represent the headers from the eight CTOC cards. The left plot shows each of the eight CTOC headers separately, listing the percentage of time they have been successfully received. This plot should be a flat distribution at 100%. The right plot shows how many G-Links were recorded for an event, each possibility displayed as a percentage of the total number of events. This plot should show that all eight G-Links were successfully read for every event. The lower two plots, in green, show the same statistics but for the CTTT headers. Only one CTTT header is sent per event.

The second histogram set of header checks is “L1 DFE Header & G-Link Checks” (see Figure 12, right) which looks at each of the 80 DFEA cards. The upper left plot,



“L3 DFE/OCT Header Checks”  
(diag1.C)

“L1 DFE Header & G-Link Checks”  
(diag2.C)

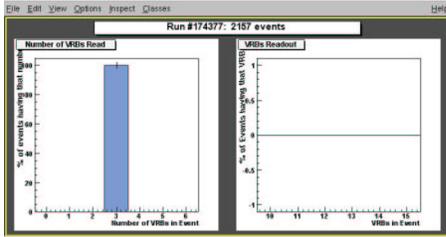
Figure 12: diag1 diag2

in green, shows the percentage of time that each of the 80 DFEA headers are received correctly. This should be a flat plot at 100%. The upper right plot, in yellow, shows an accumulation of the number of DFEA headers received correctly in an event. Since all 80 DFEA cards are expected to report during each event, only the 80th bin should be filled here. The lower left plot compares the G-Link numbers with the firmware sector number, and should be a diagonal line showing a direct correlation between the numberings.

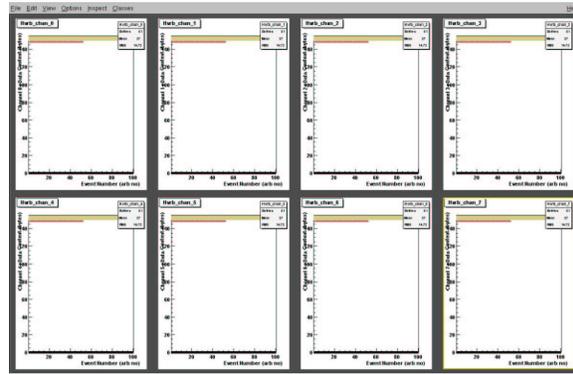
The final sets of expert oriented histograms are oriented toward displaying the response of the Virtual Readout Buffers (VRBs) that provide information to crate x13. Four of these VRB based histogram sets are available, one general plot of the VRBs that are being read out and three specific plots of each VRB’s channels.

“General VRB Info” (see Figure 12, upper left) displays two plots. On the left, the number of VRBs read out in an Event are accumulated and shown as a percentage of the total. Three VRBs are used to provide CTT information to crate x13, so this plot should only accumulate in bin three. The right plot displays the VRB numbers that have reported to crate x13. Currently, VRBs 10, 11 and 12 provide information and should appear for every event.

Three histogram sets, “VRB x13-10 Channel Content”, “VRB x13-11 Channel Content”, and “VRB x13-12 Channel Content” (see Figure 12, upper right, lower left, lower right, respectively), display the data content of each VRB’s eight channels across eight histograms (one for each channel). These plots all show points, in red, that represent the data content of a single event. The x-axis binning shows up to 100 consecutive events and the graph resets each time the event count reaches a multiple of 100. A tan band highlights the “expected region” of this plot (at 150 bytes for any given channel). The red points should lie within the tan band under normal conditions. Note that only the first two channels carry any information for VRB 12.



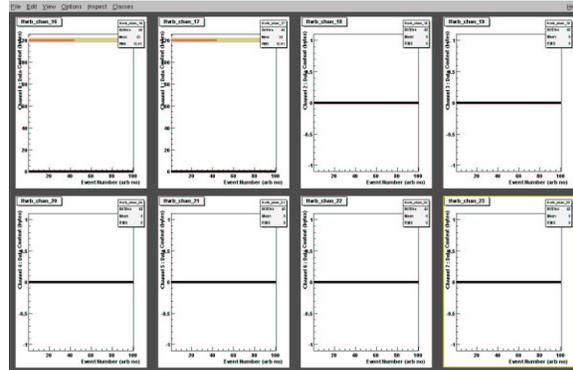
“General VRB Info” (vrbl.C)



“VRB x13-10 Channel Content” (vrbl0.C)



“VRB x13-11 Channel Content” (vrbl1.C)



“VRB x13-12 Channel Content” (vrbl2.C)

Figure 13: vrb plots

## 6 Using the Monitoring Code to Identify and Diagnose Problems

Add stuff here ...

*Is enough said about “Diagnosing Problems” in the plot explanations above?*

## 7 Details of the CTT Examine Code

The CTT Examine code is stored in the Concurrent Version System (CVS) under the package name l1CTT\_examine. Currently the examine runs in D0 code release p13.06.01. These sample commands will create a new release area for p13.06.01 and then download a copy of the CTT Examine code:

```
nylarathotep-clued0:~/work/ctt-examine> setup D0RunII p13.06.01
**** setup D0RunII p13.06.01
nylarathotep-clued0:~/work/ctt-examine> newrel p13.06.01 relarea
Creating a test release "relarea" in the directory
/home/mpc/work/ctt-examine
```

```
nylarathotep-clued0:~/work/ctt-examine> cd relarea/
nylarathotep-clued0:~/work/ctt-examine/relarea> setup d0cvs
nylarathotep-clued0:~/work/ctt-examine/relarea> kinit mpc
Password for mpc@FNAL.GOV:
nylarathotep-clued0:~/work/ctt-examine/relarea> addpkg -h l1CTT_examine
```

A fresh copy of the CTT Examine code must be compiled before it can be run. In order to compile, go to the new release area (called “relarea” in the example above) and type these commands:

```
nylarathotep-clued0:~/work/ctt-examine/relarea> setup D0RunII p13.06.01
**** setup D0RunII p13.06.01
nylarathotep-clued0:~/work/ctt-examine/relarea> d0setwa
nylarathotep-clued0:~/work/ctt-examine/relarea> gmake clean
nylarathotep-clued0:~/work/ctt-examine/relarea> gmake all
```

Recompiling the CTT Examine needs only the “gmake clean” and “gmake all” commands run from the new release area directory. Note that the commands “gmake l1CTT\_examine.clean” and “gmake l1CTT\_examine.all” may be used once a full “gmake all” has successfully completed in the new release area (this can help keep compile time low if l1CTT\_examine is not the only package added to the release area).

On an online machine, a typical shifter would use a “start\_daq” command to run the examine. A copy in a personal directory (or a development copy) must be started using scripts in the “relarea/l1CTT\_examine/bin/” directory. Various scripts exist depending how the examine program should acquire and process data. These choices include running the trigger simulator on SVX or VSVX data and running the examine on “.raw” data files or on events from the event distributor during a live run (this latter function only available on online cluster machines).

Startup Script	TrigSim Input	CTT Examine Input
run_dsvx_offline.sh	SVX	.raw file(s) (defined in ReadEvent.rcp)
run_dsvsx_offline.sh	VSVX	.raw file(s) (defined in ReadEvent.rcp)
run_dsvx_online_all.sh	SVX	Online distributor “all” stream
run_dsvx_online_any.sh	SVX	Online distributor “any” stream
run_dsvx_online_zero.sh	SVX	Online distributor “zero_bias” stream
run_dsvsx_online_all.sh	VSVX	Online distributor “all” stream
run_dsvsx_online_any.sh	VSVX	Online distributor “any” stream
run_dsvsx_online_zero.sh	VSVX	Online distributor “zero_bias” stream
run_dsvsx_online_x25.sh	VSVX	Online “daq_test” stream

Table 3: Startup scripts for CTT Examine

Each of these scripts calls a different RCP file as it starts up the examine. The script names listed on Table 3 correlate directly to the similarly named RCP files in Table 2, pg 5. Note that RCP files (including “ReadEvent.rcp” as mentioned in Table 3) are described in Section 7.1. Since RCP settings drastically adjust the way that the CTT Examine operates, it is very good practice to double check associated RCP files before

running one of the startup scripts. Also remember that this scripts must be executed by calling “./run\_dvsvx\_offline” while in the “relarea/l1CTT\_examine/bin/” directory (don’t leave out the “dot-slash” at the beginning).

## 7.1 Run Control Parameter (RCP) Files

The Run Control Parameter (RCP) files used to adjust the run-time functionality of the CTT Examine are discussed here. Refer to Table 2, pg 5, for short descriptions of all of these files. Note that the RCP files “l1CTT\_examine.rcp” and “ReadEvent.rcp” are the most commonly adjusted and define things such as Significant Event System (the online alarm system) settings and the lists of “.raw” files that are used as input for the examine when run in “offline” mode. Also notable are “cft\_trk\_data.rcp”, which defines the location of the sector equation files for the trigger simulator, and “tsim\_l1ft\_data\_vsvx.rcp”, which will tell the trigger simulator to either output a single one of or rotate between sending DFEBoardL1, DFEBoardL2 and DFEBoardL2CPSA information.

**ReadEvent.rcp** specifies the “.raw” files that will be used as input when the CTT Examine is run in “offline” mode. The “InputFile” parameter defines these files, which can simply be listed consecutively as in this example:

```
string InputFile=(
    // these files all exist on clued0 & most likely not forever
    "/work/nylarathotep-clued0/mpc/ctt-examine/rawdata/all_0000174377_003.raw"
    "/work/nylarathotep-clued0/mpc/ctt-examine/rawdata/all_0000174377_007.raw"
    "/work/nylarathotep-clued0/mpc/ctt-examine/rawdata/all_0000174377_012.raw"
)
```

**ReadEventDaq\_l1ctt\_x25.rcp** is called by “run\_dvsvx\_online\_x25.rcp” and initiates the “daq\_test” distributor stream for the CTT Examine.

**ReadEventDaq\_zero\_bias.rcp** is called by “run\_dsvx\_online\_zero.rcp” and “run\_dvsvx\_online\_zero.rcp”, and initiates the “zero\_bias” distributor stream for the CTT Examine.

**cft\_trk\_data.rcp** specifies the trigger simulator track sector equation files loaded by the simulator as the CTT Examine program initializes. The choice between using SVX or VSVX data as trigger simulator input is also made in this RCP file and all startup scripts that use VSVX input use this RCP file. The equation files are defined in the “Veqn\_file” string:

```
// these are made Graham Wilson, use latest geometry (as of 6/30/03).
// -----
// FOR USE ONLINE!!!
// -----
string Veqn_file=("/online/examines/sectorfiles/links/sector01.trk",
"/online/examines/sectorfiles/links/sector02.trk",
. . .
"/online/examines/sectorfiles/links/sector80.trk")
```

The choice between SVX and VSVX is made with the boolean variable “vsvx” (though this setting should not be adjusted in this RCP since “cft\_trk\_data\_svx.rcp” exists for this explicit purpose):

```
// setting this false uses SVX rather than SIFT (KJS)
// discriminator values
bool vsvx = true
```

**cft\_trk\_data\_svx.rcp** is similar to “cft\_trk\_data.rcp”, but has selected to use SVX input instead of VSVX input. All startup scripts that specify SVX data as trigger simulator input use this RCP file. See the discussion of “cft\_trk\_data.rcp” for more details.

**l1CTT\_examine.rcp** defines all parameters specific to the “l1CTT\_examine” package. It contains options to adjust the number of histograms that the CTT Examine creates and fills, the amount of text output the examine produces, and options for enabling socket communication, output to ROOT files, and alarm signals through the Significant Event system. The “graph\_level” flag loosely sets the number of histograms booked by the examine by specifying a “cutoff” level of information. All histograms below the cutoff level are booked and filled as usual, while those above the cutoff are not created and the data used to fill these histograms are not processed. The lowest level turns off all of the histograms. The first histograms turned on are those that carry information from the Level 3 headers attached to all of the examine’s inputs. Next, the Level 1 DFEA→CTOC information is analyzed and histogrammed. Following that, all of the histograms dealing with Level 1 trigger information are booked and filled. The highest fill level includes all of the Level 2 central tracking trigger information. Comments in the RCP file sum up this information for easy reference, and the typical setting for this parameter is the highest setting available:

```
// Graph level  0: nothing          1: L3-Header    2: + DFEA->CTOC
//              3: All level-1     4: + level-2
int graph_level = 4
```

Text output from the examine is controlled in a similar manner through the “debug1” flag. The lowest setting only allows examine initialization messages, an event counter, and alarm messages to the output window. The next highest setting allows “ERROR” signals through, which represent problems that the examine should not see during a “GOOD” CTT run. Next follows a setting that adds “WARNING” messages, then a setting adding “NOTICE” messages to the text output. Further along, this flag allows Level 3 header information, Level 1 DFEA information, then Level 1 CTOC information at various steps, in that order, until the highest setting allows all text produced by the CTT Examine through to the output window. Once again, comments in the RCP file sum up this information as a quick reference:

```

// This is a debug level flag : - 0->low through 10->high
// 0: init messages, event count, ALARM messages
// 1: + ERROR    2: + WARNING
// 3: + NOTICE  6: + Header info  7: + L1DFE   8: + L1CTOC
// 10: All messages
int debug1 = 2

```

The “online” boolean variable chooses between two major modes of operation. If set to “true”, the CTT Examine will create server sockets through ROOT that allow ROOT macros to receive histograms and display them onscreen, live, as the examine operates. If set to “false”, the examine will not create the server sockets but instead will write out a “.root” file when it finishes running (though, since it will only “finish” running when it’s reading a “.raw” file, this parameter should only be set to “false” when running an “offline” startup script and feeding the examine files as input).

```

// Use of code: online mode or offline? (ie. monitoring or root file)
bool online = true

```

Parameters for the Significant Event System (the alarm system used on the online cluster) specify whether a connection to the SE system should occur at all, and, if so, what port on what server the examine should seek a connection with. The port and server name should not normally be adjusted, but it is a good idea to set the “alarm” parameter to “false” when working offline or on a development version of the code.

```

// Significant Event server information
// alarm = false will turn off SE communication
bool alarm = false
int se_port = 52155
string se_server = d0ol20.fnal.gov

```

**l1l2\_collector.rcp** eliminates the “L2 send to L3” iogen objects from the trigger simulator and is called by “tsim.rcp”, “tsim\_svx.rcp”, and “tsim\_vsvx.rcp”.

**run\_dsvx\_offline.rcp** is called in the “run\_dsvx\_offline.sh” startup script and subsequently calls the proper RCP files for running the CTT Examine on “.raw” data files using SVX information as trigger simulator input. This file should not be adjusted under normal circumstances.

All of the “run\_” RCP files also include a string that defines the ordering of the D0 Framework packages used. Since the CTT Examine runs once on unpacked data from the distributor and once on simulated trigger data, the “exam” substring occurs twice:

```

string Packages="read epicscoor l1coor runConfigMgr unpack exam tsim exam"

```

**run\_dsvx\_online.rcp** is a skeleton for creating other RCP files and is not directly called by any CTT Examine startup script.

**run\_dsvx\_online\_all.rcp** is called in the “**run\_dsvx\_online\_all.sh**” startup script and subsequently calls the proper RCP files for running the CTT Examine on events from the standard distributor stream, using SVX information as trigger simulator input. This file should not be adjusted under normal circumstances.

**run\_dsvx\_online\_any.rcp** is called in the “**run\_dsvx\_online\_any.sh**” startup script and subsequently calls the proper RCP files for running the CTT Examine on events from the distributor stream “anything”, using SVX information as trigger simulator input. This file should not be adjusted under normal circumstances.

**run\_dsvx\_online\_zero.rcp** is called in the “**run\_dsvx\_online\_zero.sh**” startup script and subsequently calls the proper RCP files for running the CTT Examine on events from the distributor stream “zero\_bias”, using SVX information as trigger simulator input. This file should not be adjusted under normal circumstances.

**run\_dsvsx\_offline.rcp** is called in the “**run\_dsvsx\_offline.sh**” startup script and subsequently calls the proper RCP files for running the CTT Examine on “.raw” data files using VSVX information as trigger simulator input. This file should not be adjusted under normal circumstances.

**run\_dsvsx\_online\_all.rcp** is called in the “**run\_dsvsx\_online\_all.sh**” startup script and subsequently calls the proper RCP files for running the CTT Examine on events from the standard distributor stream, using VSVX information as trigger simulator input. This file should not be adjusted under normal circumstances.

**run\_dsvsx\_online\_any.rcp** is called in the “**run\_dsvsx\_online\_any.sh**” startup script and subsequently calls the proper RCP files for running the CTT Examine on events from the distributor stream “anything”, using VSVX information as trigger simulator input. This file should not be adjusted under normal circumstances.

**run\_dsvsx\_online\_zero.rcp** is called in the “**run\_dsvsx\_online\_zero.sh**” startup script and subsequently calls the proper RCP files for running the CTT Examine on events from the distributor stream “zero\_bias”, using VSVX information as trigger simulator input. This file should not be adjusted under normal circumstances.

**run\_dsvsx\_online\_x25.rcp** is called in the “**run\_dsvsx\_online\_x25.sh**” startup script and subsequently calls the proper RCP files for running the CTT Examine on events from the “daq\_test” stream, using VSVX information as trigger simulator input. This file should not be adjusted under normal circumstances.

**tsim.rcp** is a skeleton for creating other RCP files and is not directly called by any CTT Examine RCP files.

**tsim\_svx.rcp** is called by all “**run\_dsvx\_**” RCP files in order to properly set certain trigger simulator parameters. This file should not be adjusted under normal circumstances.

**tsim\_vsvx.rcp** is called by all “run\_dvsvx\_” RCP files in order to properly set certain trigger simulator parameters. This file should not be adjusted under normal circumstances.

**tsim\_l1ft\_data\_svx.rcp** is called by “tsim\_svx.rcp” in order to properly set certain trigger simulator parameters. This file should not be adjusted under normal circumstances.

**tsim\_l1ft\_data\_vsvx.rcp** is called by “tsim\_vsvx.rcp” in order to properly set certain trigger simulator parameters. This file should not be adjusted under normal circumstances.

## 7.2 Unpacking the Data

The CTT Examine receives raw information from either a “.raw” file or from the online distributor, but performs its actual analysis on “unpacked” information. The unpacking process transforms data from a bitstream, such as what is sent between actual hardware elements within the central tracking trigger framework, into a set of C++ objects that contain decoded data. The trigger simulator returns data that mimics the output from these unpackers, making it simple within the CTT Examine code to switch between looking at live data and output from the simulator. The data objects defined by the unpacker can be found in the “l2io” package, and often each piece of hardware has a specific unpacker object that unpacks its specific set of data.

While the unpacker itself for the data used by the CTT Examine resides in the “l1l2unpacker” package, python scripts generate the C++ code that define the data that is unpacked, and the python scripts use “iogen” files as their input. This makes the process of adjusting how the unpackers create data objects easy at the expense of making some of the actual C++ code extremely obscure. The iogen files themselves dictate exactly how the bits in a bitstream are turned into C++ objects.

For example, consider the Level 1 DFEA information. This information is contained in “l2io” objects, namely those constructed by “l2io/l1ft.iogen”. A look at the iogen file reveals some of the workings of this process:

```
OBJECT DFEHeaderL1 VERSION 1
    bitfield[32] {
        0:2  Level1L2;
        3:7  DataType;
        8:11 lsbSect_inOct_Addr;
        12:18 zNull12;
        19:21 msbOctAddr;
        22:23 Pass_Mark;
        24   TPP;
        25:27 ctrl_bit;
    };
END_OBJECT
```

```
OBJECT DFESectorL1 VERSION 1
    bitfield[32] {
```

```

    0:1  zNull1;
    2:4  numNEG_hgPT_noPS;
    5:7  numPOS_hgPT_noPS;
    8:9  zNull2;
   10:12 numNEG_hgPT_loPS_T;
   13:15 numPOS_hgPT_loPS_T;
   16:17 zNull21;
   18:20 numNEG_hgPT_loPS_L;
   21:23 numPOS_hgPT_loPS_L;
    24   TPP1;
   25:27 ctrl_bit1;
};
bitfield[32] {
    0:1  zNull20;
. . .
   25:27 ctrl_bit5;
};
END_OBJECT

OBJECT DFETrailerL1 VERSION 1
    bitfield[32] {
        0:23 Long_parity;
        24   zNull;
        25:27 ctrl_bit;
    };
END_OBJECT

OBJECT DFEBoardL1 VERSION 1
    DFEHeaderL1(1)  l1header;
    DFESectorL1(1)  l1data;
    DFETrailerL1(1) l1trailer;
END_OBJECT

```

First to note is that the leading numbers specify which bits within the bitfield (zero being the “least significant bit”) are used to create a data object. Protocols between the hardware specify exactly what this correlation should be, and the inclusive set of bits specified should translate directly to information fields. The names that follow the bitfield numbers are the names of the C++ data objects that will be created. Notice that the objects are organized into a hierarchy where larger objects, such as “DFEBoardL1”, are created by piecing together smaller objects, such as “DFEHeaderL1”, “DFESectorL1”, and “DFETrailerL1” (which will be called “l1header”, “l1data”, and “l1trailer”, respectively, in the C++ object they create).

The unpacker headers must be included in the C++ code in order to use the unpacker’s objects in the examine. The “l2base” class contains some of the structure that “l2io” uses, so parts of it must be included as well. Thus, these statements are included in the “l1CTT\_examine.cpp” file:

```
#include "l2base/L2.hpp"
```

```

#include "l2base/io.hpp"
#include "l1l2unpacker/interface.hpp"
#include "l1l2unpacker/ModuleUnpacker.hpp"
// Declaration of Data & IO Headers in Use (see l2iogen objects in l2io)
#include DATAHEADER(DFEBoardL1)
#include IOHEADER(DFEBoardL1)

```

The data will not be unpacked without first declaring the source of the bitstreams, so these statements are also a necessity (but note that they should only ever be called once during the program's execution!):

```

declareUnpacker(new CFTModuleUnpacker(19,0));
declareUnpacker(new CFTModuleUnpacker(19,1));
declareUnpacker(new CFTModuleUnpacker(19,2));

```

The arguments for the “CFTModuleUnpacker” are the crate number (in decimal notation) and VRBs where the data resides. The VRBs are numbered in the order they are “found” in the crate, not by slot number directly. This numbering will skip any VRBs that are not read out, which can cause problems. It's important to make sure that currently disabled VRBs are not effecting the unpacking process which will happen if a previously active VRB that occurs “before” a VRB containing pertinent information is disabled.

The unpacker, when running on the selected VRBs, will fill “l2io” objects with the information it finds. A look at the l2io object list is recommended since there are many and finding the object that contains pertinent information is not always trivial. The object headers are listed in “l2io/l2io/online” and the source code is found in “l2io/src/online/”. For the case of Level 1 DFEA information, “DFEBoardL1Data.hpp” and “DFEBoardL1Data.hpp” hold the pertinent information, namely this object contains the information as described in the “l1ft.iogen” file as described above. In order to access this information in the examine program, a vector of these objects must be created, where each item in the vector contains the information from one board:

```

const vector<const DFEBoardL1Data*> *l1ftdfe=extract<DFEBoardL1Data>();
int nmod = l1ftdfe->size();
for(int i=0; i<nmod; i++) {
    cout << (*l1ftdfe)[i]->l1data().numNEG_hgPT_noPS() << endl;
}

```

Notice that in order to access the data field containing the number of negatively signed highest pt bin tracks not associated with a preshower cluster (the data “numNEG\_hgPT\_noPS” represents as per the protocol), the “l1data” member of the “DFEBoardL1Data” must first be accessed. All of the data fields listed in the iogen file may be accessed in a similar manner, accounting for the internal organization between the “l1header”, “l1data”, and “l1trailer” members within the overall class.

## 7.3 Significant Event System

The Significant Event System allows communication with the online alarm server. A few simple commands can send alarm signals that will appear on the alarm displays in the control room. Currently, the “SigEvtSys” libraries are maintained by Geoff Savage and attempts at communicating with the Significant Event server for the online systems should not be made without his approval.

The “SigEvtSys” libraries include all the tools necessary for creating and sending alarm signals. Specifically, “SE\_Message” objects are made that contain all of the alarm information and these objects are sent to the alarm server via the “SEClient” object. The headers for these two classes must be included in order to use their functionality. In the CTT Examine, both the header and “.cpp” files include alarm system objects because the message and client objects are declared in the header file, to ensure their permanence, while they are instantiated and used in the .cpp file.

In “l1CTT\_examine.hpp”:

```
// Includes for SIGNIFICANT EVENT SYSTEM (alarms)
#include "SigEvtSys/SEClient.hpp"
#include "SigEvtSys/SE_Message.hpp"
. . .
SE_Binary_Alarm_Message *SE_L1DFEA_missing;
SESender *SE_sender;
```

In “l1CTT\_examine.cpp”:

```
// Includes for SIGNIFICANT EVENT SYSTEM (alarms)
#include "SigEvtSys/SEClient.hpp"
#include "SigEvtSys/SE_Message.hpp"
. . .
SE_sender = new SENDER(se_port, se_server, "CTT_Examine");
SE_L1DFEA_missing = new SE_Binary_Alarm_Message("CTT_EXAMINE_L1DFEA/MSNG",
    SE_Message::TRANS_NONE, SE_Message::MAJOR, 75);
```

A Run Control Parameter (RCP) file is used to supply the alarm server name and port number to the CTT Examine, specifically “l1CTT\_examine.rcp”. This RCP file also includes a boolean variable, “alarm” that defines whether or not the examine communicates with the alarm server. When the examine determines that an alarm condition exists, this is a typical block of code used to inform the Significant Event server:

```
if (alarm) SE_sender->send(SE_L1DFEA_missing);
else cout << "(ALARM signal suppressed by RCP)" << endl;
```

Many different alarm signals, all defined by their correlated “SE\_Message” subtype, can be set using the same “SESender” object. Therefore, adding more alarms is simply a matter of defining more message strings and sending them to the alarm server through the same sender when a given alarm condition is satisfied.

## 7.4 Booking and Filling ROOT Histograms

ROOT histograms are used to store the data that the CTT Examine tracks and displays as it runs. These histograms are declared in the “11CTT\_examine.hpp” file, so that they are permanent components of the examine program and do not go out of scope when a routine ends. The declarations look like this:

```
TH1F *HTrksPSec;  
TH1F *OctNumInBin[4];  
TH2F *PhiVsPtDFEA;
```

ROOT has many different options available for histogramming, including multidimensional histograms, such as “TH2F” above, a two dimensional histogram, and it is also possible to define arrays of histograms as also appears above. Arrays of histograms can make it easier to separate histogrammed information into separate sets for octants or similar subsets, allowing them to be easily indexed within a “for” loop. The final letter in the variable type for ROOT, the “F” in above examples, dictates the length of the bitfield used to store information for each bin. Four options are available: “C” creates histograms with one byte per channel, with maximum bin content of 255; “S” creates histograms with one short per channel, with maximum bin content of 65,535; “F” creates histograms with one float per channel, yielding 7 digit precision; “D” creates histograms with one double per channel, yielding 14 digit precision.

The histograms are instantiated in the “.cpp” file. This gives them a name used to track them within the ROOT framework, a title that is displayed when the histogram is plotted, defines the number of bins and the range those bins cover:

```
HTrksPSec = new TH1F("HTrksPSec","Number of Tracks per Sector (80)",80,0.01,80.01);  
OctNumInBin[0] = new TH1F("OctNumInBin1","1.5-3 GeV Pt Bin Occupancy",42,-0.5,41.5);  
OctNumInBin[1] = new TH1F("OctNumInBin2","3-5 GeV Pt Bin Occupancy",42,-0.5,41.5);  
OctNumInBin[2] = new TH1F("OctNumInBin3","5-10 GeV Pt Bin Occupancy",42,-0.5,41.5);  
OctNumInBin[3] = new TH1F("OctNumInBin4","10+ GeV Pt Bin Occupancy",42,-0.5,41.5);  
PhiVsPtDFEA = new TH2F("PhiVsPtDFEA","DFEA L2 Phi vs Pt",160,-0.5,159.5,20,0.5,20.5);
```

The ROOT name, the first field in the instantiation, must be unique, as shown in the example above with the “OctNumInBin” histograms. This posed a unique problem for the CTT Examine since it runs twice: once on unpacked data from the distributor and a second time on data provided by the trigger simulator. Thus, the actual code in the CTT Examine looks slightly different, with separate names for the distributor based data and the simulated data. A boolean variable, “tsimkf”, tracks whether the examine is looking at simulated data or unpacked data and histograms are booked accordingly:

```
if (tsimkf) OctNumInBin[0] = new TH1F("OctNumInBin1", . . .  
if (!tsimkf) OctNumInBin[0] = new TH1F("dOctNumInBin1", . . .
```

Filling histograms in ROOT simply involves calling the “Fill()” command on a histogram. This function is overloaded, supporting multiple methods for calling it. For one dimensional histograms, where “x” is the abscissa (or “namex” is the name associated with the proper bin) and “w” is the weight to add to the bin at that position (the default for w being “1”), these calls are supported:

```

Int_t Fill(Axis_t x)
Int_t Fill(Axis_t x, Stat_t w)
Int_t Fill(const char *namex, Stat_t w)

```

Multidimensional histograms fill in much the same way, only adding fields for the extra dimensions directly after the field for “x”. Examples of fill calls in the CTT Examine code appear below:

```

HTrksPSec->Fill(secnt,xadd);
for(int k=0;k<4;++k) OctNumInBin[k]->Fill(xoadd2[k]);
PhiVsPtDFEA->Fill(phi/22,(20-ptbin));

```

ROOT is very well documented. User manuals are available at the ROOT webpage, <http://root.cern.ch/>, as are many examples of more complicated uses of ROOT. The source code is also available online and each ROOT class may be examined individually to determine the functions available within it. This is a highly recommended course of action for those working regularly in ROOT.

## 7.5 Socket Communication in ROOT

The ROOT libraries include built in functions for communicating between different programs that use ROOT. The communicating programs do not even need to run on the same machine. One program can act as the server, creating an open port that other programs, the clients, can connect to for information exchange. Specifically, the CTT Examine C++ program acts as a server for histograms while the ROOT based macros that actually display histograms on the screen are clients that connect to the examine’s server. In order to find and pass the proper histograms to the clients, the examine program also adds all histograms to a “TList”. In the “l1CTT\_examine.hpp” file, the required libraires to setup a ROOT server are included and the objects required are declared:

```

#include <TServerSocket.h>
#include <TSocket.h>
#include <TSeqCollection.h>
. . .
TServerSocket *serversocket;
TSocket *testsocket;
TList *HistList;

```

All of these objects are instantiated in “l1CTT\_examine.cpp”. Also, more ROOT libraries are included for use during socket communication. The CTT Examine uses “TMessage” objects to send information between clients and servers such as the name of histograms requested by the clients. All histograms are added to the “TList” object as they are created, as well. Recall that “tsimkf” is a boolean variable representing whether the currently running examine program is looking at unpacked data from the on-line distributor or simulated data from SVX or VSVX information. The boolean “online” represents whether the examine program will write a ROOT file as output or will attempt to connect to ROOT macros for live display. The “testsocket” was created in conjunction with setting the “kNoBlock” flag on the server port. When the flag is set, the socket

will not wait for an incoming communication before continuing execution of the rest of the program (which is essential to running an examine program), but will talk to connections when they exist. The “testsocket” should have a “NULL” value, but remains a good test in case the libraries change and choose a different value to return when no client is waiting at the end of the open server socket.

```

#include <TServerSocket.h>
#include <TSocket.h>
#include <TSeqCollection.h>
#include <TString.h>
#include <TMessage.h>
. . .
if (online)
{ // Open communication socket
  cout << "Opening communications socket" << endl;
  if (tsimkf) serversocket = new TServerSocket(9090, kTRUE);
  if (!tsimkf) serversocket = new TServerSocket(9091, kTRUE);
  serversocket->SetOption(kNoBlock,1);
  testsocket = serversocket->Accept();
  cout << "Creating Histogram List" << endl;
  HistList = new TList();
}
. . .
if (online) HistList->Add(PhiVsPtDFEA);
if (online) for (int i=0;i<8;i++) HistList->Add(HTrigTerm[i]);
. . .
void l1CTT_examine::SocketCommunication()
{
  TSocket *s = serversocket->Accept();
  if(s!=NULL && s!=testsocket && s->IsValid()) // check for good socket
  {
    // mess_in is the input message object that recieves information
    // mess_out is the output message object that sends information
    TMessage *mess_in;
    TMessage mess_out(kMESS_OBJECT);

    // TSocket::Recv(TMessage) returns length of message in bytes
    // (can be 0 if other side of connection is closed) or -1 in
    // case of error or -4 in case a non-blocking socket would block
    while(s->Recv(mess_in) > 0)
    {
      if (mess_in->What() == kMESS_STRING)
      {
char str[64];
mess_in->ReadString(str, 64);
if (!strcmp(str, "MODE")){
  if (!tsimkf )

```

```

        {s->Send("DATA");}
    else
        {s->Send("TSIM");}
} // if "MODE"
if (!strcmp(str, "RAWDUMP")) {RawDump(); return;}
if (!strcmp(str, "RAWDUMP2")) {RawDump2(); return;}
if (!strcmp(str, "RESET1")) {ResetHistos(1); return;}
if (!strcmp(str, "RESET2")) {ResetHistos(2); return;}
if (!strcmp(str, "RESET3")) {ResetHistos(3); return;}
if (!strcmp(str, "THANK YOU")){
    delete mess_in;
    s->Close("force"); delete s;
    return;
} // if "THANK YOU"
// TList::FindObject(char*) returns 0 if string not found
TObject *out = HistList->FindObject(str);
if (out)
    {
        mess_out.Reset();
        mess_out.WriteObject(out);
        s->Send(mess_out);
    } else s->Send("String Not Found");
        } else cout << "Socket input not a string..." << endl;
    } // if good socket
} // while getting messages
}

```

The communication process used by the CTT Examine is initiated by a client. A ROOT macro that displays histograms attempts to connect to the server, a process that will wait for a response from the server (since “kNoBlock” has not been set for the client). Once the server opens the socket for communication, the client will send a string to the server containing the ROOT name string of the histogram requested. The client then will receive this histogram from the server whereupon the server will continue to wait for information from the client. In this way, the client can download many histograms in succession without the server port closing, and since the examine program only looks for potential clients once each event it analyzes, it would be a very slow process otherwise. This comes at the expense of requiring the client to send a closure string, in the CTT Examine “THANK YOU”, before the server will close the socket and continue analyzing events.

The client side of the communication process is detailed in Section 7.6 below.

## 7.6 ROOT Macros and Displaying Histograms

The ROOT macros used by the CTT Examine program are responsible for collecting histograms from the C++ program, that constantly runs and fills histograms with data from events, and displaying those histograms on screen. Each macro is designed around a related set of histograms. Examples of ROOT macro output and explanations of the

contents of the CTT Examine histograms are contained in Section 5. These macros can be found in the “relarea/l1CTT\_examine/bin/macros” directory.

Many of the macros are written so that they can access either the unpacked distributor data or the simulated trigger data from the examine. Some macros download both sets and overlay the data for a live comparison. As described above in Section 7.5, the ROOT macros are expected to ask for histograms by their ROOT name string successively then send “THANK YOU” to the examine program in order to close the communication. This “handshaking” process allows the examine program to continue running as the ROOT macro normalizes and displays the downloaded histograms.

Because ROOT macros are fairly straightforward, and many examples exist even just within the CTT Examine itself, only excerpts from the code for the “dfea\_C.C” macro will be discussed. This macro downloads both “live” and “simulated” data and overlays their plots. The simulated data is plotted as a histogram while the live data from the distributor appears as data points with error bars. An example of this macro’s output appears in Figure 3, pg7.

```

TH1F *DFEA_L1_Count = 0;
TH1F *dDFEA_L1_Count = 0;
TH1F *HTrksPSec = 0;
while(1) {
    TSocket *s1 = new TSocket("localhost", 9090);
    TMessage *mess_in;

    s1->Send("DFEA_L1_Count");
    s1->Recv(mess_in);
    if (mess_in){
        if (mess_in->What() == kMESS_OBJECT)
            DFEA_L1_Count = (TH1F *)mess_in->ReadObject(mess_in->GetClass());
        else {
            cout << "***Unexpected message***" << endl;
            if (mess_in->What() == kMESS_STRING)
                { char errstr[64]; mess_in->ReadString(errstr, 64);
                  printf("%s\n", errstr); }
        }
    }
}
. . .
s1->Send("THANK YOU");
s1->Close("force");
delete s1;

TSocket *s2 = new TSocket("localhost", 9091);

s2->Send("dDFEA_L1_Count");
s2->Recv(mess_in);
if (mess_in){
    if (mess_in->What() == kMESS_OBJECT)
        dDFEA_L1_Count = (TH1F *)mess_in->ReadObject(mess_in->GetClass());
}

```

```

else {
    cout << "***Unexpected message***" << endl;
    if (mess_in->What() == kMESS_STRING)
        { char errstr[64]; mess_in->ReadString(errstr, 64);
          printf("%s\n", errstr); }
    }
}
. . .
s2->Send("THANK YOU");
s2->Close("force");
delete s2;
delete mess_in;
. . .
if (DFEA_L1_Count && dDFEA_L1_Count && HTrksPSec && . . .) {
    HTrksPSec->SetFillColor(38); HTrksPSec->SetXTitle("Trigger Sector");
    HTrksPSec->SetYTitle("Average no. of tracks in sector / event");
    dHTrksPSec->SetXTitle("Trigger Sector");
    dHTrksPSec->SetYTitle("Average no. of tracks in sector / event");
    entries1 = DFEA_L1_Count->GetEntries();
    entries2 = dDFEA_L1_Count->GetEntries();
    if (!gROOT->IsBatch() )
        {
pad1->cd();
HTrksPSec->Sumw2();
if (entries1 > 0) HTrksPSec->Scale(1.0/(float)entries1);
dHTrksPSec->Sumw2();
if (entries2 > 0) dHTrksPSec->Scale(1.0/(float)entries2);
dHTrksPSec->SetMarkerStyle(20);
dHTrksPSec->SetMarkerColor(1);
dHTrksPSec->SetMarkerSize(0.7);
if(dHTrksPSec->GetMaximum() > HTrksPSec->GetMaximum()) {
    dHTrksPSec->DrawCopy("e1");
    HTrksPSec->DrawCopy("SAME,hist");
} else HTrksPSec->DrawCopy("hist");
dHTrksPSec->DrawCopy("SAME,e1");
. . .
        } // if IsBatch
    } // if histograms were downloaded
gSystem->ProcessEvents();
gSystem->Sleep(500);
} // while loop

```

## 7.7 Graphical User Interface in Python

A python script is used to create the Graphical User Interface (GUI) for the CTT Examine. The script can be found at “relarea/l1CTT\_examine/bin/ctt4.py”, and it uses many images from the “relarea/l1CTT\_examine/bin/images” directory. The script itself is

quite long and most of it simply defines the many buttons available in the GUI, though there are some interesting features. The GUI will launch the various ROOT macros by running operating system commands, then track and kill the processes via their process IDs. A combo box that displays all ROOT macros is also available, the advantage of this being that new macro names can be added to the combo box's list of strings quicker than a complete new button can be added to the python script. An automatically updating image at the center of the GUI also exists, though it updates only in conjunction with a ROOT macro that changes that central image. See Figure 2, pg 6, for an example of what the CTT Examine's GUI looks like.

Most buttons on the GUI were created following the same strategy. A variable was created to track the process ID of the root macro the button was intended to run. Then, the button itself was defined, creating a clickable image for the users. A short routine was created for each button that checks the process ID, kills that macro's old process if needed, then starts the ROOT macro, recording the new process ID. This short excerpt includes all of these steps for a button in the python script, plus a short routine, called when the script itself is exited, that cleans up rogue processes the GUI may have started.

```

from Tkinter import * # all the basic gui stuff
from tkMessageBox import showinfo
import Pmw,sys
import string,re,os,sys,signal,atexit # useful python modules
import thread,threading
import commands # useful for running commands

class ButtonFrame(Frame):
    ## PID storage initialization
    dfea_pid = 0
    . . .
    self.but1 = PhotoImage(file="images/but1.gif")
    # create buttons
    self.dfea1 = Button(root, image=self.but1, relief="flat", borderwidth=0,
highlightthickness=0,highlightbackground="black",highlightcolor="black",
background="black",activebackground="black")
    self.dfea1.config(command=lambda func=dfeapush1,arg=self: func(arg))
    self.dfea1.place(x=101,y=204)
    self.balloon=Pmw.Balloon(self)
    self.balloon.bind(self.dfea1,'DFEA->CTOC Graphs (at L1 time): L1CTT Tracks')
    . . .
def dfeapush2(instance):
    # BUTTON-PUSH FUNCTION DEF.
    if bf.dfea_pid:
        os.kill(bf.dfea_pid,signal.SIG_IGN)
        if (instance.y=="0"):
            bf.dfea_pid = os.spawnlp(os.P_NOWAIT, 'root.exe', '',
'macros/dfea2.C(0)',os.path.abspath('.'))
        if (instance.y=="1"):
            bf.dfea_pid = os.spawnlp(os.P_NOWAIT, 'root.exe', '',
'macros/dfea2.C(1)',os.path.abspath('.'))
    else:
        if (instance.y=="0"):
            bf.dfea_pid = os.spawnlp(os.P_NOWAIT, 'root.exe', '',
'macros/dfea2.C(0)',os.path.abspath('.'))
        if (instance.y=="1"):
            bf.dfea_pid = os.spawnlp(os.P_NOWAIT, 'root.exe', '',
'macros/dfea2.C(1)',os.path.abspath('.'))

```

```

. . .
def Cleanup():                                # CLEANUP FUNCTION
    if bf.dfea_pid: os.kill(bf.dfea_pid,signal.SIG_IGN)
    bf.dfea_pid = 0
# atexit will ensure Cleanup is called upon exiting the program (mpc)
atexit.register(Cleanup)
. . .
if __name__ == '__main__':                    # MAIN LOOP
    usage = "ctt4.py <yvalue> <zvalue>"
    inputs = sys.argv[1:]
    if len(inputs) != 2:
        print "Bad input args"
        print usage
        sys.exit(1)
    # Create the basic Tk gui instance
    root = Tk()
    root.title("CTT Examine")
    # Add one of our objects
    bf = ButtonFrame(root,inputs[0],inputs[1])
    # Pack the ButtonFrame
    bf.pack(side=TOP)
    # run the gui
    bf.Update()
    infobutton = showinfo("CTT Examine: Information",
        "The examine takes a few minutes to start recording data. "
        "During this time, no windows you open will display histograms "
        "(even blank ones). Once you see the examine scroll through events "
        "histograms will be available.")
    root.mainloop()

```

In creating the GUI for the CTT Examine, most of the difficulty was in determining a method for running ROOT macros and in tracking those macros so the total number of processes running on the examine's computer did not get too cumbersome. The above code is an example of the initial methodology for completing this set of tasks, but in the meantime a more elegant solution has presented itself. The combo-box, used as a quickly updateable macro launcher for CTT experts, uses a list of process IDs to track the macros it has launched. Details from this solution are reproduced below:

```

### TO ADD MACROS TO THE DROP DOWN LIST:
### just add them to this list! Make sure that they include a (0)
### or (1) option as necessary
macrolist = ("macros/consis.C(0)", "macros/consis.C(1)", . . . )
. . .
#####
### Drop-down list definition ###
#####
self.macroframe = Frame(root, width=180, height = 200)
self.messageframe = Label(self.macroframe,
    text="--Expert histogram display--\n(0) = trigsim\n(1) = x13 data",
    width=24)
self.messageframe.pack(side=TOP)
self.macroboxchoice = Label(self.macroframe, text='Choose Macro...',
    relief='sunken', padx=5, pady=5, width=24)
self.macroboxchoice.pack(expand=1,fill='both',padx=5,pady=5)
self.macrobox = Pmw.ComboBox(self.macroframe, label_text='Run:',

```

```

        labelpos='wn', listbox_width=24, dropdown=1,
        selectioncommand=chooseEntry, scrolledlist_items = self.macrolist)
self.macrobox.pack(fill=BOTH,expand=1,padx=2,pady=2)
self.macrobox.selectitem(self.macrolist[0])
self.macroframe.place(x=790,y=350)
. . .
#####
### Drop-down list macro display function definition ###
#####
def chooseEntry(entry):
    bf.macroboxchoice.configure(text='Displaying ' + entry)
    bf.drop_down_pid_list.append(os.spawnlp(os.P_NOWAIT, 'root.exe', '', entry,
        os.path.abspath('.')'))
. . .
def Cleanup():
    ##### This should kill all processes started by the drop-down macro
    ##### list, in the process resetting the list to zero-length
    while(len(bf.drop_down_pid_list)):
        os.kill(bf.drop_down_pid_list.pop(),signal.SIG_IGN)

```

## 8 Conclusion

See CTT Examine. See CTT Examine run. Run, CTT Examine, run!  
 (What the heck is *supposed* to go here, anyway?)

## References

- [1] The CDF Collaboration, *Phys. Rev. D* **58** (1998) 920.
- [2] The CDF Collaboration, *Phys. Rev. D* **57** (1993) 5382.
- [3] The ALEPH Collaboration, *Zeit. Phys.* **C71** (1996) 31.
- [4] Ask Kyle what he intended to reference here...(vrb)
- [5] Levan Babukhadia *D0 Track and Preshower Trigger Level 1 Trigger Terms and Data Transfer Protocols V07-00* (2002)