National Superconducting Cyclotron Laboratory

R00TLe

Nothing Works, Everything is Broken^{TM}

Version 1.0

Sam Lipschutz (DDAS)
Shumpei Noji (S800)

?/?/?

Contents

1 Introduction

1.1 What is R00TLe?

R00TLe is a software package designed to analyze experimental data from the S800 spectrometer and the Low Energy Neutron Detector Array (LENDA) at the NSCL. Specifically it is designed for data taken with the digital data acquisition system (DDAS) for LENDA under **NSCL Readout version 10.02**. It's main features include the following:

1. The ability to unpack evt files containing S800 and Pixie-16 DDAS events into raw ROOT trees
2. Preform preliminary processing on the data and produce calibrated ROOT trees
3. Implement a library of different trace processing routines necessary for the digital system

1.2 Acknowledgments

This code has not been developed by me alone. There have been contributions from a variety of different sources that need acknowledgment. Thanks is given to Kathrin Wimmer (CMU) and Eric Lunderberg (NSCL) for the parts of R00TLe adopted from grROOT. Further thanks is given to Shumpei Noji (NSCL) who wrote much of the S800 analysis provided by R00TLe.

1.3 What is up With the Spelling of R00TLe?

R00TLe is a high information density name. It combines ROOT (the data analysis package from CERN), the S800, and LENDA into one beautifully conceived title. Further it is an expandable name. Future experiments combining the S800, LENDA and Gretina can easily be analyzed by grR00TLe. Credit is given to Chris Sullivan for first espousing the construct that is ROOTLe.

1.4 Getting R00TLe

R00TLe is maintained in a git repository. Git is a popular software version control system. The code can be obtained from github.com where the repository is backed up. For example from the NSCL fishtank machines type:

```
git clone https://github.com/soam5515/R00TLe.git
```

This will create a directory called R00TLe that contains all of the R00TLe package. In the directory you should see the following directories:

```
bin lib prm scripts shell skelton src users
```

1.5 Compiling and Installing R00TLe

To compile R00TLe go in to the "src" directory and type:

```
make -j4
```

You should see:

Make in S800
Compiling S800.cc...
Compiling ROOT dictionary S800Dictionary.cc...
Compiling S800Event.cc...

...

R00TLe all has been built successfully

Once it has been compiled successfully you can configure your environment and finalize instillation by typing:

```
make install
```

Enter the full path to where the evtfiles will be and where you would like R00TLe to put the ROOT files. The install script will add lines to your ~/.bashrc file to set up all of the different paths and what not need for R00TLe to work. After running make install, source your bashrc:

```
source ~/.bashrc
```

Note you will only need to run make once (unless you change something in the source) and you will only need to run make install once (unless you uninstall).

1.6 Un-installing R00TLe

"Uninstalling" R00TLe simply will remove the environment settings that installing added to your bashrc. This would be useful if you wanted to put R00TLe into a different directory (installing adds a R00TLeInstall environment variable). Simply go to the source directory and run:

```
make uninstall
```

1.7 Important Things About how Everything is Set Up

When you install R00TLe there are many features in the code that look for the enviorment variables that were defined at instillation. So if you try and put another copy of R00TLe somewhere don't install it. Remember that the code will use what ever settings

are in your ~/.bashrc for where stuff is located.

Also R00TLe does not use $LD\_LIBRARY\_PATH$ for finding and linking libraries at run time. Everything is defined at compile time and set relative to the directory R00TLe is being installed in.

2 The Code In Detail

2.1 What is in the Main Directory

In the main directory you should see the following:

1. prm: (Parameters) Place to put input parameter files and run by run correction files
2. scripts: ROOT macros that could be useful
3. shell: Bash scripts for installing/uninstalling and other usefull wrappers on thing
4. skeleton: Directory containing generic histogram dumper that can be built off of for analysis
5. users: Directory containing space to run analysis codes
6. lib: Directory containing all of the shared object libraries built in the code
7. bin: Directory containing all the executable programs of R00TLe
8. src: Directory containing all the source code of R00TLe

2.2 The Bin Directory

The bin directory contains the main executables for R00TLe. It contains the following programs:

1. Evt2Cal: Take a raw evt data file and build a ROOT tree with calibrated events
2. Evt2Raw: Take a raw evt data fiel and build a ROOT tree with raw events
3. Evt2Scalers: Extract the scaler information stored in the EvtFile
4. Raw2Cal: Convert a raw ROOT tree to a calibrated ROOT tree
5. Cal2Cal: Rebuild a calibrated ROOT tree with new calibration parameters (a work in progress)

Each of these programs makes a step through the data processing pipeline. The general phases of data processing for normal data (not scalers) are as follows:

1. Raw binary data file (.evt file)
2. Raw ROOT tree file containing DDASEvents and S800 events (.root file)
3. Calibrate ROOT tree file containing LendaEvents and S800calc events (.root file)

See section 3 for more information about how the raw and calibrated ROOT tress are set up.

The general command syntax for running the main programs is as follows:

1. Evt2Cal InputEvtFile OutputROOTFile [TheRunNumber]
2. Evt2Raw InputEvtFile OutputROOTFile
3. Raw2Cal InputROOTFile OutputROOTFile [TheRunNumber]

Where the [TheRunNumber] is an optional argument. Note if you provide a run number then it will look for a run specific corrections file and map file for the DDAS building. If you give 400 as the run number it will look in "prm" directory for files called Corrections400.txt and MapFile400.txt. If it can find them it will use those run specific files automatically. If it can't find them it will use the default files: Corrections.txt and MapFile.txt. If you do not provide the run number it will use the default files.

There are convenient bash scripts that will call the main programs for you. See section 2.3 on the "shell" directory.

2.3 The Shell Directory

The shell directory contains a variety of convenient bash scripts. The important ones are:

1. Install.sh: Called when you type make install from the src directory
2. Uinstall.sh: Called when you type make uninstall from the src directroy
3. BuildData.sh: Called from a R00TLe user directory it will build all segments of data with a given run number. If there are already RAW ROOT files built it will call Raw2Cal and make Calibrated ROOT files. If there are only evt files it will call evt2Call and make calibrated ROOT files. It requires that the input files have the format: run−#### − ##.evt or run−#### − ##-RAW.root.
4. BuildRaw.sh: Called from a R00TLe user directory it will build all segments of data with a given run number into raw ROOT files. It requires evt files with format run−#### − ##.evt.
5. R00TLeLogon.sh: Used for logging into a R00TLe user. See section 4 on R00TLe user directories for more information

6. GetEvtInfo.sh: Script that will print the title information stored in the evt files for a given run

2.4 The Skeleton directory

The skeleton directory contains a standard start up code that is copied over when a new R00TLe user is made. It has a skeleton histograming program that will loop over calibrated ROOT trees. It also has the standard ROOT startup scripts that give R00TLe is trader marked splash screen. See the R00TLe Users section 4 for details.

3 Description of ROOT data structures

The following two sections describe the data structures that are stored in the raw and calibrated ROOT files. That is to say in each ROOT files there is a ROOT tree that holds a C++ class. These classes are what I mean by data structures. For instance the raw ROOT tress contain DDASEvent and S800 classes.

3.1 Raw ROOT files

The raw root tree has data structures in it that are simply reorganizations of the data in the .evt file. On the DDAS side DDASEvents are made. The DDASEvent class is a data structure from Sean's group. It is simply a list of ddaschannel objects (also from Sean's group). The ddaschannel objects contain the raw DDAS information: timestamp, energy, slot number, channel number, the trace etc... See figure 3.1 below.

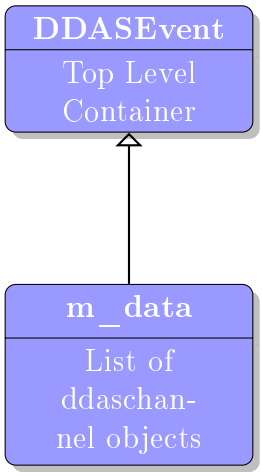

Figure 3.1: Structure of the DDASEvent Class

On the S800 side the data is stored in S800 objects (from grROOT). It contains uncalibrated S800 data. It does not have things like CRDC xy positions. See figure 3.2

below. Each S800 object has 6 different major sub objects that hold the information from the different parts of the S800. For instance to get the first scintillator information one would do:

```
myS800->GetScintillator(0)->GetDE_up()
```

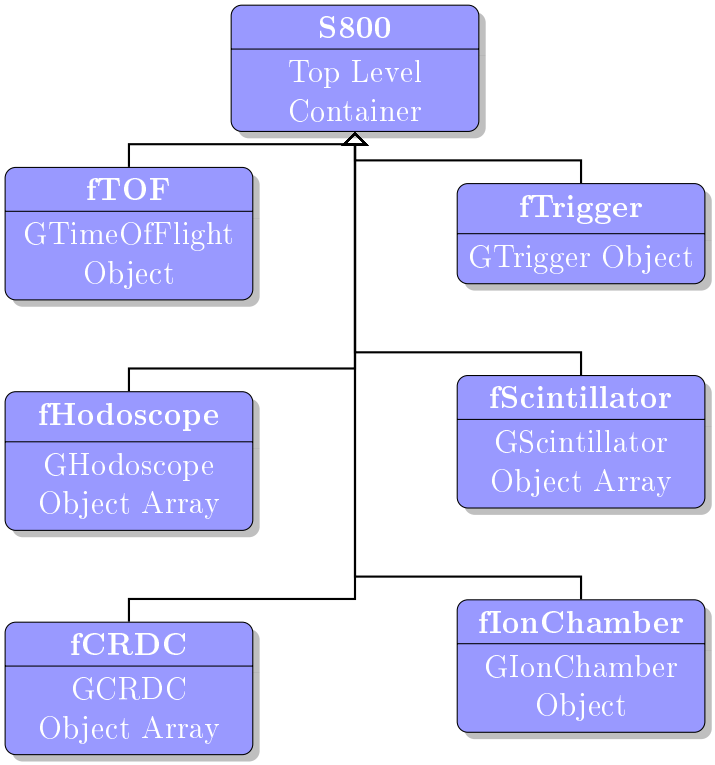

Figure 3.2: S800 Class Structure

3.2 Calibrated ROOT Files

The calibrated root trees are more complicated. On the S800 side it will be stored as S800calc objects (also from grROOT) where you have provided calibration files to make calibrated energies, times and CRDC xy positions. See figure 3.3 below. The structure is similar to the raw S800 objects except that the names are slightly different and they have the calibrated numbers. For example to get information from the first scintillator:

```
myS800Calc->GetSCINT(0)->GetDEdown()
```

On the DDAS side it will also make calibrated energies and offset corrected times in the form of LendaEvent objects. Further, it will take a cable map file and associate together the different channels that fired into bars. It will also preform the wave form analysis for the different off-line timing algorithms and store each of them in the event. See

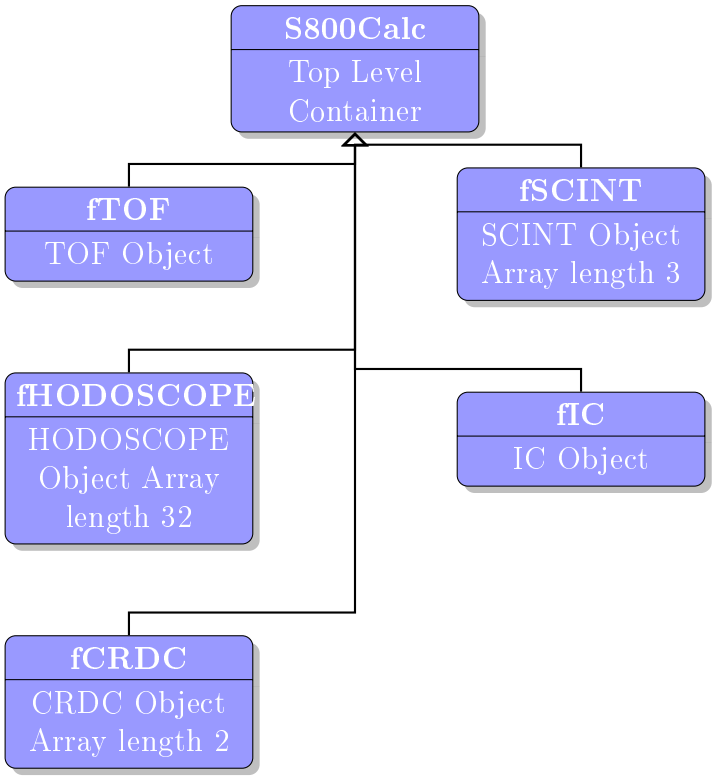

Figure 3.3: S800Calc Class Structure

figure 3.4 below. Each LendaEvent has a variable sized array (std::vector) of LendaBar objects, which contain two variable sized arrays for top and bottom firings. The top and bottom arrays contain LendaChannel objects. The LendaChannel objects contain the raw information for a DDAS channel. The object scintillators are treated differently from the bar channels. They are stored in a variable sized array of LendaChannels in the LendaEvent called TheObjectScintillators. If there was an event with 1 bar you can access the energy of the top channel (from pulse integration) with:

```
myLendaEvent->Bars[0].Tops[0].GetEnergy()
```

If the event had one bar with 2 top channel firings (a pile up event) the second firing could be accessed with:

```
myLendaEvent->Bars[0].Tops[1].GetEnergy()
```

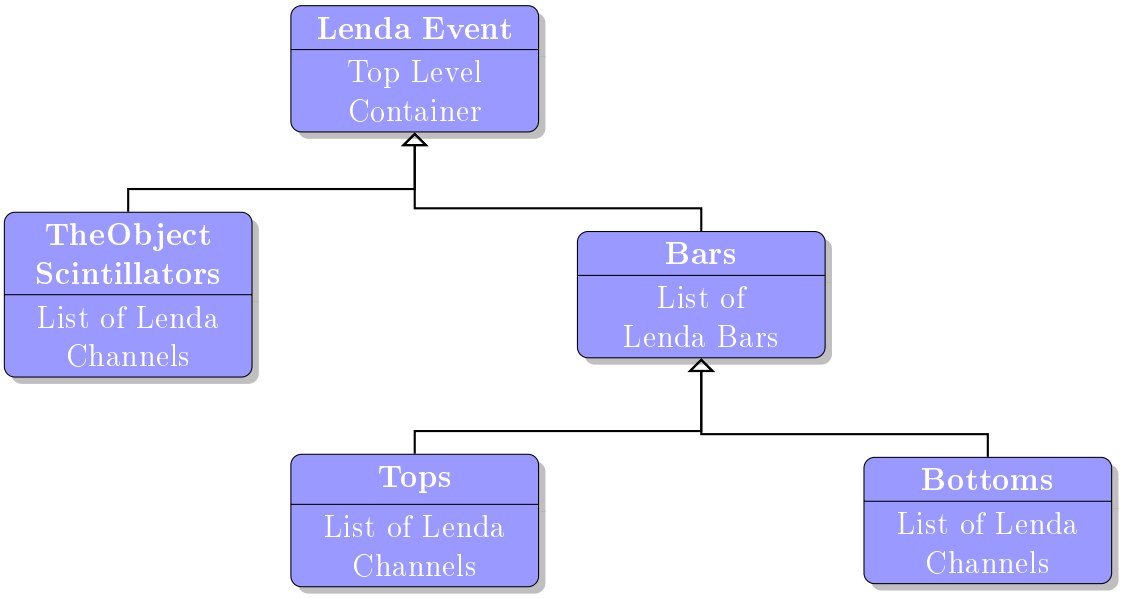

Figure 3.4: LendaEvent Class Structure

4 R00TLe Users

Making a "user" in R00TLe simply means having a directory in the R00TLeInstall/users/ directory. This is done by using the R00TLeLogon.sh script. The first time you logon as a new user use:

```
R00TLeLogon.sh -f AUserName
```

Afterwards simply

```
R00TLeLogon.sh AUserName
```

Logging on as a user will do several things. Firstly, it will bring you to an independent working area where you can do analysis without affecting other users. Further, in your directory it will generate startup files for ROOT so that your sessions have access to all the R00TLe libraries. In particular a .rootrc file and a rootlogon.C file will be generated. These will setup your ROOT sessions (only when you execute ROOT from your user's directory) so that the S800 and LENDA libraries are preloaded. It will also make symbolic links to the directories containing the evt and ROOT files that were specified at install time. This way you can access the files built with R00TLe easily with:

```
root rootfiles/run-0001-00.root
```

The other main thing that the R00TLeLogon.sh script does is copy over the skeleton histogram-er into your directory (the first time it is executed). It consists of three directories:

1. macros: A local directory for the user to put ROOT macros in
2. src: The directory where the Analoop program is (the histogram dumper)
3. histograms: The directory where Analoop puts your histogram files

To analyze a run (that has already been built into a calibrated tree) you can simply do the following from a ROOT session:

```
.x main.C(TheRunNumber)
```

main.C is a ROOT macro (it is copied over into the macros directory at login) that will load the R00T file for the given run, then compile your Analoop program and then run Analoop over the file. See the example section 5.2 for more details.

5 Examples

5.1 Accessing information in the Interpreter

In this example the syntax for making a histogram from within the ROOT interpreter is shown. It will use calibrated ROOT file run-0399-00.root, which was one of the test beam runs. First log in to your R00TLe directory:

```
R00TLeLogon.sh sam
```

Then open the root file with:

```
<pike:sam >root rootfiles/run-0398-00.root
```

To list the contents of the ROOT file type:

```
.ls
```

You should see something like the following:

```
TFile**         rootfiles/run-0398-00.root
 TFile*         rootfiles/run-0398-00.root
  KEY: TTree    caltree;1       S800 and DDAS calibrated events
  KEY: R00TLeSettings   TheSettings;1   R00TLe's Settings
```

The file contains a ROOT tree named caltree and a settings object called TheSettings. Histograms can be drawn from the tree with the draw command. It has the syntax:

```
caltree->Draw("variable selection","cuts","graphics options");
```

5.1.1 DDAS Side

So to draw the energy spectrum for the top PMT from one of the LendaChannels you would have the following:

```
caltree->Draw("lendaevent.Bars[0].Tops[0].GetEnergy()",
"lendaeavent.NumBars==1&&lendaevent.Bars[0].SimpleEventBit&&
lendaevent.Bars[0].BarId==2")
```

Note: I like my software overly long and wordy

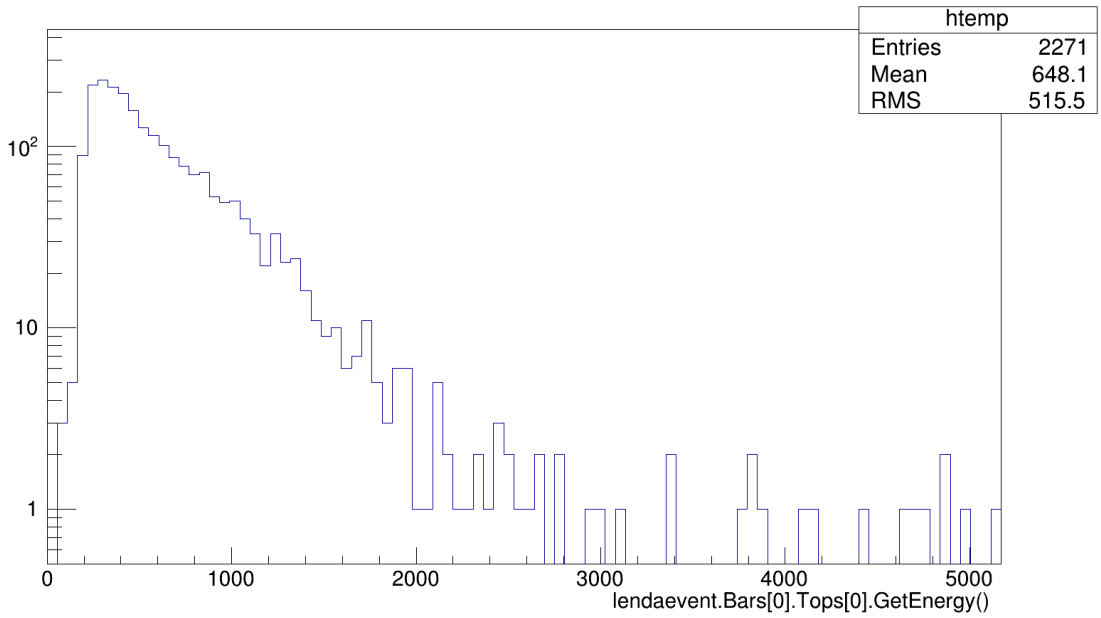

Figure 5.1: What the histogram should look like

Figure 5.1 shows what this draw command should look like in from run 398.
To break down this command the first argument:

```
"lendaevent.Bars[0].Tops[0].GetEnergy()"
```

calls for the first top firing in the first bar in the event and gets the energy. The second argument has three different cuts to the data. First:

```
lendaeavent.NumBars==1
```

only plot from events with one bar. Second:

```
lendaevent.Bars[0].SimpleEventBit
```

requires that the first bar in the event (and only bar due to condition one) is a "simple event". A Simple Event in this context means that there was no crazy pile up. IE the event had exactly one top and one bottom. The third condition:

```
lendaevent.Bars[0].BarId==2
```

selects which of the 48 bars you are interested in. During the building of the calibrated root file each bar is given an unique bar id. In this case we are selecting number 2. If you don't remember which bar is what, you can ask the settings object (which has saved all the information from the building process). In this case BarId 2 corresponds to SL03.

```
root [2] TheSettings->GetBarName(2)
(class TString)"SL03"
```

5.1.2 S800 Side

Making histograms of s800 data is done in a similar fashion. As an example we will plot the sum energy from the IC (also from run 398).

```
caltree->Draw("s800calc.GetIC().GetSum()")
```

Here we are making no cuts so there is not second argument. The result of this plot can be seen in figure 5.2. Plots containing both S800 and DDAS data can be combined in a draw command.

```
caltree->Draw("s800calc.GetIC().GetSum():lendaevent.Bars[0].Tops[0].GetEnergy()",
"lendaeavent.NumBars==1&&lendaevent.Bars[0].SimpleEventBit&&
lendaevent.Bars[0].BarId==2")
```

Here we have the S800 ion chamber sum plotted against the energy in the top PMT of Lenda Bar 2. It has the same cuts as the above Lenda examples.

5.1.3 Complicated Draw Commands

Below are some more examples of ever more complicated draw commands with explanations on what they mean.

Below will plot the 2-D histogram of center of gravity vs top-bottom time difference in bar 2. It has the same restrictions on multiplicity as the above example.

```
caltree->Draw("lendaevent.Bars[0].GetCOG():lendaevent.Bars[0].GetDt()",
"lendaevent.NumBars==1&&lendaevent.Bars[0].SimpleEventBit&&
lendaevent.Bars[0].BarId==2")
```

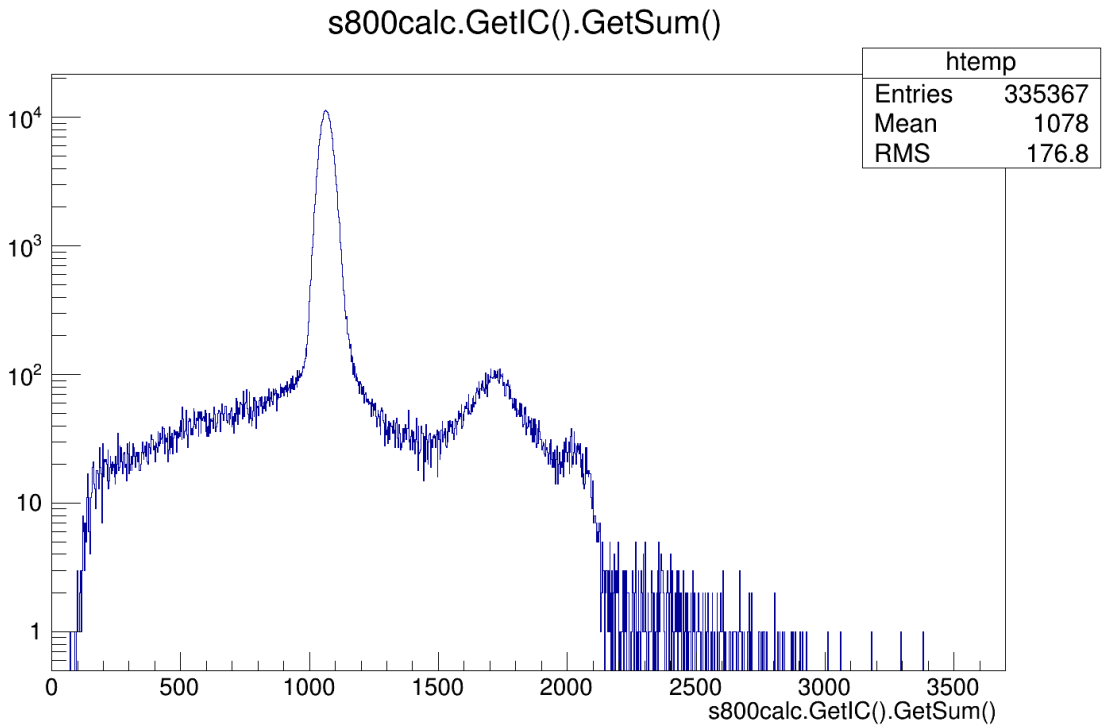

Figure 5.2: S800 IC Sum example

Below will plot the 1-D histogram of the top channels time of flight. IE the timestamp of the top PMT channel minus the timestamp of the top PMTs reference time (as defined in the map file). Like Above it still has the same multplicity cuts in it and still selects bar ID equal to 2.

```
caltree->Draw("lendaevent.Bars[0].GetTopTOF()",
"lendaevent.NumBars==1&&lendaevent.Bars[0].SimpleEventBit&&
lendaevent.Bars[0].BarId==2")
```

Below will produce the same 1-D histogram as above. The time of flight for the top PMT channel. For some quantities the events store redundant information. The LendaBar class will calculate and store the time of flight, but the raw information is still stored in the underlying LendaChannels. Most of the quantities that LendaBar calculates are for convenience (like TOF, average energy, average time...).

```
caltree->Draw("lendaevent.Bars[0].Tops[0].GetTime()-
lendaevent.Bars[0].Tops[0].GetReferenceTime()",
"lendaevent.NumBars==1&&lendaevent.Bars[0].SimpleEventBit&&
lendaevent.Bars[0].BarId==2")
```

5.2 Using the Skeleton Histogram Dumper

The generic histogram dumper, that is copied to each user directory when it is created, consists of three files: main.C, Analoop.cc and Analoop.h.

5.2.1 main.C

Main.C is a wrapper script to make calling running the Analoop program simpler. It can be called in the following ways:

1. .x main.C : when no arguments are given it will load the files that are hard coded into the script. When it is done dumping those files it will save them in the histograms directory as temp.root.

2. .x main.C(RunNum) : when a run number argument is given it will look for files called "rootfiles/run-RunNum-??.root" if it finds them it will run Analoop and save the result in the histograms directory as HistogramsFromRun####.root

3. .x main.C(RunNum,AFileName) : when two arguments are given you can specify an output file name. Instead of saving the histograms as HistogramsFromRun####.root it will save them as the given file name.

5.2.2 Analoop

The Analoop program is the bit of code that actually making the histograms. Technically it defines a "TSelector" object, which a ROOT class designed to loop over ROOT trees and make histograms. Doing it this way makes the syntax slightly more cumbersome but allows for the use of PROOF (the parallel ROOT facility). Analoop has two parts a header file and a source file:

1. Analoop.h is the header file. This is where you would declare but not initialize histograms. IE you would say something like:

   ```
   TH1F * MyHistogram
   ```

   but **not**

   ```
   TH1F *MyHistogram = new TH1F("MyHistogram","Title",10,0,10)
   ```

2. Analoop.cc is the source file. This where you "new" the histograms and where you make cuts and fill histograms. See below for details.

The places in the code where you make changes are clearly commented. In the header file the only changes you need to make are declaring histograms. In the source file the changes you need to make are "new-ing" the histograms you've added in the header file and filling them in the main analysis function. These changes are more complicated so we will go into more detail.

5.2.3 Analoop.cc in detail

Since Analoop.cc is where the really physics analysis will occur it is important to go into some details. There are two important functions defined (they are the first two in file and are commented): SlaveBegin and Process. Slave begin is called once at the start of the program. It is where histograms are allocated. In SlaveBegin there is already some code that allocates histograms for standard Lenda quantities for all the bars (TOFs, PulseHeights...). This is done under the comment that says

Make histograms for standard quantities (TOFs and PulseHeights).

Below that is another comment specifying a good place to "new" more histograms that you have declared in the header. This is where you would say some thing like:

MyHistogram = new TH1F("MyHistogram","Title",100,-10,10)

The second important function is Process. This function is called for each entry in the ROOT tree (each event). There is already code in there to fill the "standard Lenda quantities". It has the general structure:

```
for (int i =0;i<NumBarsInEvent;i++){ //Loop over all Bars in the Lenda event
   int NumTopsInBar = lendaevent->Bars[i].NumTops;
   int NumBottomsInBar = lendaevent->Bars[i].NumBottoms;
   int BarId = lendaevent->Bars[i].BarId; //The ith Bars Unique Bar Id

   for (int t=0;t<NumTopsInBar;t++){ // Loop over all the TOPS in the Bar
     TopEnergies[BarId]->Fill(E);
     //Do stuff for Tops
   }//End for over tops

   for (int b=0;b<NumBottomsInBar;b++){//Loop over all the Bottoms in the Bar
     BottomEnergies[BarId]->Fill(E);
     //Do Stuff For bottoms
   }//End for over bottoms

   if (lendaevent->Bars[i].SimpleEventBit){//has 1 top and 1 bottom
     AvgTOFs[BarId]->Fill(lendaevent->Bars[i].GetAvgTOF());
   }
}//End Loop Over Bars
```

Here we see that for each entry in the tree the Lenda information can be accessed through the lendaevent pointer. In order to extract all the channel firings from the Lenda event, one must loop over all bars and for each bar all tops and bottoms. This is what is done above. As an example of how to add code to this structure we will fill our example histogram with the top bottom time difference for bar south Lenda 5 (SL05). Here is the same code again with the added lines:

```
for (int i =0;i<NumBarsInEvent;i++){ //Loop over all Bars in the Lenda event
   int NumTopsInBar = lendaevent->Bars[i].NumTops;
   int NumBottomsInBar = lendaevent->Bars[i].NumBottoms;
   int BarId = lendaevent->Bars[i].BarId; //The ith Bars Unique Bar Id

  for (int t=0;t<NumTopsInBar;t++){ // Loop over all the TOPS in the Bar
     TopEnergies[BarId]->Fill(E);
     //Do stuff for Tops
  }//End for over tops

  for (int b=0;b<NumBottomsInBar;b++){//Loop over all the Bottoms in the Bar
     BottomEnergies[BarId]->Fill(E);
     //Do Stuff For bottoms
  }//End for over bottoms

  if (lendaevent->Bars[i].SimpleEventBit){//has 1 top and 1 bottom
     AvgTOFs[BarId]->Fill(lendaevent->Bars[i].GetAvgTOF());
     //Can only have top bottom time difference for 1 top and 1 bottom
     if (lendaevent->Bars[i].Name=="SL05"){
        MyHistogram->Fill(lendaevent->Bars[i].GetDt());
     }
  }
}//End Loop Over Bars
```

5.3 From Evt file to Histogram file

In this example the whole process for processing data is shown step by step. For this example run 394 will be used (this was one of the runs from the test run).

1. Log on to R00TLe user

   R00TLeLogon.sh sam

   This should output:

   ```
   Setting environment Variables
   User sam found.
   Switching to working directory...
   Generating ROOT start up file...
   Generating ROOT logon file...
   Found histogrmer already. Won't over write
   ```

2. Build Raw ROOT file

```
BuildRaw.sh 394
```

This should output something like:

```
Build Raw Root File
Info in <Evt2Raw>: Input file: ./evtfiles/run-0394-00.evt
Info in <Evt2Raw>: Input file size: 124.79 MB
Info in <Evt2Raw>: Output file: ./rootfiles/run-0394-00-RAW.root
Info in <Evt2Raw>: Total of 37686 data buffers (124.79 MB read)636.91 buffers/sec...
1.16 sec to go
Info in <Evt2Raw>: Total of 34864 raw events written
Info in <Evt2Raw>: Decoded 6786.683594 buffers/sec.
Real time 0:00:05.552930, CP time 4.680
```

3. Build Calibrated ROOT file:

```
BuildData.sh 394
```

Assuming that you have provided appropriate DDAS map and corrections files and S800 calibration files this will build the calibrated trees. You should see an output like:

```
Build From Raw Root
Info in <Raw2Cal>: Input file: ./rootfiles/run-0394-00-RAW.root
Info in <Raw2Cal>: Input file size: 74.85 MB
Info in <Raw2Cal>: 34864 entries in the input file
Info in <Raw2Cal>: Output file: ./../rootfiles/run-0394-00.root
Info in <LendaPacker>: Using MapFile MapFile.txt and CorrectionsFile Corrections394.txt
Info in <Raw2Cal>: Reading a parameter file /user/lipschut/R00TLe/prm/Raw2Cal.prm
Real time 0:01:37.661684, CP time 43.450==================]
```

4. Open ROOT

```
root
```

5. run the histogram dumper on run 394

```
.x main.C(394)
```

This should output

```
Creating TChain... Done.
List of files:
Collection name='TObjArray', class='TObjArray', size=100
 OBJ: TChainElement     caltree ./rootfiles/run-0394-00.root
Info in <TUnixSystem::ACLiC>: creating shared library /user/lipschut/R00TLe/-
users/sam/src/Analoop_cc.so
Info in <Analoop::Notify>: File: ./rootfiles/run-0394-00.root ( 147.13 MB)
Info in <Analoop::Notify>: 34864 entries (100.00%) in this tree
Info in <Analoop::Notify>: 34864 entries (100.00%) up to this tree
File will be saved in /user/lipschut/R00TLe/users/bob/histograms/-
HistogramsFromRun394.root
Info in <WriteHist>: Histograms have been written to /user/lipschut/R00TLe/-
users/sam/histograms/HistogramsFromRun394.root
Info in <Analoop::Analoop>: Destructing.
```

After this there will be a HistogramsFromRun394.root file in the local histograms directory that contains the default histograms