CLAS-NOTE 1999-016
CVS: docs/cooking_proc/
Date: October 8, 1999
Version: 1.0

e1, g1, and g6 Data Processing Procedures

J.J. Manak, E.S. Smith
Thomas Jefferson National Accelerator Facility, Newport News, VA.
S. McAleer, S. Barrow
Florida State University, Tallahassee, FL.

Abstract

The purpose of this CLAS-NOTE is to give an overview of the steps required to calibrate a run period, proceed through preliminary reconstruction passes, monitor and document the data quality, and complete full scale processing of the data for the rest of the collaboration. The outcome of this procedure is the production of "cooked" data files, which contain reconstructed quantities which are deemed by the collaboration to be suitable as input to publishable physics analysis.

Contents

List of Tables

1 Introduction

This note attempts to describe the production of the so called "cooked" data for the e1, g1 and g6 data sets. The authors offer it as a model for processing all run periods' data. The production of "cooked" data involves more than providing users with files of reconstructed data. The software and procedures must ensure the proper calibration of the reconstructed data, the quality of the reconstruction code used, and the repeatability and documentation of the entire operation. These goals are achieved through the use of the software and procedures described in this note.

The first part of this note is intended to provide motivation for the note and a description of the design philosophy used in the development of the "cooking" procedures. The next section describes the databases used to steer the cooking of the data and asses the quality of said processing. Following that, the various programs developed for the processing of the e1, g1 and g6 data sets are itemized and the cooking script used for processing is described. This note concludes with a discussion of the steps required to produce a "cooked" data set using the described procedures.

1.1 Design Philosophy

The off-line processing software is based on a UNIX design philosophy which is well summarized below.

> "...at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial tasks in isolation, but, combined with other programs, become general useful tools." *Kernighan and Pike, The UNIX Programming Environment*

In the context of the off-line analysis, programs are kept relatively simple and are specialized to perform specific tasks. The output of these programs is often used as the input to other programs and can be easily manipulated by standard utilities and scripting languages. Finally, because programs perform relatively simple tasks, they are easy to control, usually with a set of command line options in a fashion similar to the UNIX command set. An understanding of this philosophy will help you better understand the design decisions made in the off-line analysis scheme, especially the off-line processing script.

2 Off-line Databases

There are two important databases for off-line processing, the Mapmanager database and the off-line database. The Mapmanager database stores calibration constants used by the reconstruction and analysis codes while the off-line database stores monitoring information in a convenient form to assess the quality of the cooked data. We note that the off-line database is an entirely distinct and separate entity from the on-line database. The on-line database is an Ingres database used to store information about the run conditions while that run is taking place.

2.1 Mapmanager Database

The Mapmanager provides a set of subroutine calls and programs which provide information from a database file according to a well defined prescription. Specifically, a piece of information is requested for a specific run-number; if the information exists in the database for that run number, the information is provided. If the database has no entry for that particular run number, however, it looks for a database entry in preceding run numbers until it finds one, and then returns that entry.

All run dependent quantities needed to correctly interpret the data should be placed into the map database. In this sense the map database is used to control the processing of a given data set. Typically, each reconstruction package (CC, EC etc...) provides a routine to initialize itself for a given run number. These routines contain calls to the mapmanager database. The master map database files are located in the */group/clas/parms/Maps* area on the jlab Common Unix Environment(CUE) machines. A single environment variable, *CLAS_PARMS* provides the location for the parms area in all analysis packages. Knowledge of what set of map database files were used to analyze a given data file is critical to a repeatable and documented analysis of the data.

Mapmanager database files stored in the *CLAS_PARMS/Maps* area have the *.map* file extension. Because the *.map* files are binary files and are not directly human readable, a collection of command line operated UNIX utilities have been developed to allow one to interact with the map database files. These utilities allow one to determine the parameters stored in a database file, extract the contents, create new map files, create new database entries and fill database files. Two of these utilities, *scan_map* and *print_map*, are particularly useful when assessing the quality of calibrations.

The *scan_map* utility allows one to determine which database entries are stored in a map file and can print a time line of when a particular database item has been updated. The usage for *scan_map* is shown below:

```
Usage:  scan_map [-t][-h] < map >
        - lists all items and subsystems in map file
        -t          Draw a timeline for each item
```

```
    -h          Print this message
```

By default, *scan_map* lists which database entries are contained within the map file. *scan_map* also has two options, -h which prints the usage and -t. The -t option prints a timeline for each database entry contained within the map file. An example of the use of *scan_map* with the -t option is shown below:

```
jlabs2>scan_map $CLAS_PARMS/Maps/GEOMETRY.map -t
Map:  /group/clas/parms/Maps/GEOMETRY.map
       Subsystem:  beam, nitems:  1
       Item:  position,  length:  3,  type:  float, narray:17
       INF <-- 17746 <-- 17591 <-- 17395 <-- 17233 <-- 17135
<-- 17094 <-- 16941 <-- 16833 <-- 16634 <-- 16506 <-- 16288
<-- 16186 <-- 16152 <-- 11000 <-- 8400
<-- 7400 <-- 1 <-- UNDEF
etc...
```

Listed are the various database entries stored in the *GEOMETRY.map* file. Each entry has two keys, the subsystem name and the item name. In the example shown, the map contains an entry with the subsystem tag *beam* and item name *position*. This beam position variable is stored in the map as an array of three floating point numbers. The *narray* variable confirms that this map entry has been updated seventeen times. The rest of the output is a time line of the run numbers that have been updated.

The *print_map* utility allows one to track the changes in a calibration constant over time. The usage for *print_map* is shown below:

```
Usage:  print_map -m <map> -s <subsystem> -i <item>

print to stdout an array element from the map as a function of time
example:
        print_map -m test.map -s testsub -i testitem -e 0
        -m <mapfile> Map file
        -s <subsys.> Subsystem name
        -i <item>    Subsystem name
        -e <#>       element number to print starting from zero
        -n <#>       number of elements to print (def.  = 1)
        -u <#>       upper limit run number (def.  = 99999999)
        -l <#>       lower limit run number (def.  = 0)
```

One often uses this command to examine how rapidly a particular calibration constant is changing with time. If a calibration constant changes too quickly between updates, it may indicate that the constant must be updated more frequently. An example of the use of this command to examine the pedestal values for the time-of-flight system is shown below:

```
jlabs2>print_map -m $CLAS_PARMS/Maps/SC_CALIBRATIONS.map -s pedestals
       -i left

18822 366.000000
18791 365.000000
18022 366.000000
17841 367.000000
16082 365.000000
16060 365.000000
15968 360.464996
14000 362.000000
12778 365.000000
etc...
```

In the first column is printed the run number and in the second column is printed the value of the first element of the array containing the mean value of the time-of-flight counter pedestals. The output format of this program (ASCII) can easily be used as input to a plotting program or a spreadsheet.

The mapmanager source code is located in *packages/Map*. The source for command line operated UNIX utilities is located in *packages/utilities/maputil*. Additional documentation for the map and its UNIX utilities is available on the off-line web pages.

2.2 Off-line database

The off-line database is stored in a Perl DBM database, a simple database management facility that allows the user to store information to disk files in key-value pairs. A very nice discussion of this type of database and how to use it in a Perl context can be found in [Perl, pages 170–172]. This database provides a record of the "cooking" process by storing such information as the executables that were run and the location of the output files from these executables. It also records information about the quality of the "cooking" by parsing the output of the monitoring programs and storing this information in the database. All of the variables that are stored in this database, along with their definitions, can be found in Tables 1 - 13 of this paper.

Several utilities have been written to make use of this database, all of which **MUST** be run on Linux machines. [1] These scripts can be found under CVS management in the directory *scripts/cooking_scripts*. The first and simplest utility is *lookatrun* which takes as an argument the database file name. *lookatrun* then simply dumps out the contents of that file in no particular order under the format:

variable has a value of *value*

While *lookatrun* simply lists the key-value pairs in a data file, the *getdb* script can produce a table of database quantities for a given set of keys indexed by run number. The usage for *getdb* is shown below.

```
usage:  getdb -K<key1> -K<Key2> <dbfile1, dbfile2>
        -K<key>         set key
        -h              Print this Help
        -F<###>         fraction of events * number
        -S<###>         Scale by intergrated_PS_scalers * <###>
```

The output of *getdb* is a table with the run number and file extention listed in the first two columns and the values of the various input database keys listed in the remaining columns. The following is a sample execution of this script and the results which could be redirected into an ASCII file:

```
ifarml2> getdb -Krms_s1_sl1 -Krms_s1_sl2 -Krms_s1_sl3 -Krms_s1_sl4 -Krms_s1_sl5
-Krms_s1_sl6 /work/clas/production/g1/pass1/database/clas*.db
    11205 0 0.065  0.075 0.126   0.123   0.174  0.158
    11206 1 0.065  0.074 0.126   0.123   0.175  0.160
    11462 0 0.061  0.070 0.115   0.113   0.175  0.165
    11462 1 0.061  0.070 0.114   0.113   0.177  0.165
    11462 2 0.061  0.070 0.115   0.115   0.174  0.167
etc...
```

[1] Due to an endian compatibility problem between SunOS and Linux, the Perl DBM files can only be used on the system on which they were created. Since our cooking code is much faster on the Linux batch farm machines than the SunOS batch machines, all of the cooking is done on Linux. Thus, all of the database files are created on Linux machines. This means that all utilities to look at these database files must be executed on Linux machines. Efforts to overcome this endian problem have failed so far. Plans are being made to keep a copy of this database in an web accessible MSQL database.

Another script which is used to examine the contents of the off-line database is *plot_strip_chart.pl*. The usage for this script is

```
Usage:plot_strip_chart.pl [-h][-R<#>][-r<#>][-f<file>][-d<dbdir>]
(-b<beam.t>)[-t<torus_current>][-nf][-ne][-A][-S] ARGS
      -h               Print this help message.
      -R<run#>         Maximum run number to include in plot.
      -r<run#>         Minimum run number to include in plot.
      -f<file>         Base file name for output.
      -d<dbdir>        Path to database directory.
      -b<beam.t>       Plot only runs with this beam energy in MeV.
      -t<torus>        Plot only runs with this torus current.
      -nf              Normalize results to Faraday cup.
      -ne              Normalize results to # of events written.
      -A               Average over all of the file extensions.
      -S               Plot variables on same page.
      ARGS             List of variables to plot on same graph.
```

The arguments used for this script are the variable names found in Tables 1 - 13 of this paper. *plot_strip_chart* then produces three files. The first is a text file very similar to the output of *getdb*. The first column of this text file contains the run numbers found in the database and the other columns contain the values for the chosen variables. However, where *getdb* prints out the values for each file extension, *plot_strip_chart.pl* sums the information from each file extension over the entire run number. The options *-A*, *-nf*, and *-ne* allow the user to either average the value by the number of file extensions, normalize the sum to the Faraday Cup information or for photon running, normalize the sum by the total number of events written out by *a1c*. The second output file created by *plot_strip_chart.pl* has the extension *.gnu*. This *.gnu* file contains a series of commands that will be used by *gnuplot* to histogram the chosen variables versus run number. *plot_strip_chart.pl* then uses *gnuplot* to create a postscript file containing a graphical representation of the information found in the *.txt* file. Examples of output from this script can be found in Figs. 5 and 8.

2.3 Combining Off-line and Mapmanager databases

Due to the nature of the "cooking" process, it is sometimes necessary for the off-line and Mapmanager databases to share information. Most of the time the Mapmanager database requires information from the off-line database. The framework for some of this exchange is written as *UpdateCalibMaps.pl* with the usage:

```
Usage:UpdateCalibMaps.pl -h-P0x# -d<dbdir> [-t#] [-f<string>]
   [-pdu<string>]

   -h              Print this help message.
   -d<dbdir>       Path to database directory.
   -f<string>      Intermediate file name needed for put_map
                   command (default = alibaba).
   -pdu<string>    PDU wire file including path to use for
                   DC_STATUS. (must be used if -P0x4 is
                   specified).
   -t#             Run number to use as time stamp for DC_STATUS.
   -P0x#           Choose maps to update.
      0x1          Update Geometry map with x,y, and z beam positions.
      0x2          Update Sync map with number of synchronization bits.
      0x4          Update DC_STATUS map from pdu wire file.
```

For the *-P0x1* and *-P0x2* options *UpdateCalibMaps.pl* retrieves values from the off-line database for each run. With the *-P0x1* option, the vertex information, obtained by running *trk_mon* is placed into the *GEOMETRY.map*. The read out controller synchronization information, obtained by running *sync*, is stored in the *SYNC.map* when the *-P0x2* option is specified. For both of these processing bits the *-d* option must also be specified to alert *plot_strip_chart.pl* to the location of the off-line database. The information for each run is then stored in the respective Map database.

The *-P0x4* option, used for updating the *DC_STATUS.map*, is performed in a different manner than the other two processing options. In order to update this map the script does not access the off-line database but rather a text file. Using the *-pdu<string>* option, the user identifies the specific pdu wire file to use. The *-t#* should also be specified allowing the user to control where the information is placed in the map. Only one entry is needed for each set of run conditions since the health of the drift chambers should not change dramatically in the middle of a run.

3 Tagging, Freezing, and Production Libraries

When analyzing data, it is important to keep a record of the analysis packages that was used to process the data. It is also important to have the ability to return to previous versions of analysis packages in order to reproduce results or to track down errors. Toward this purpose all of the CLAS analysis software is kept under the previously mentioned version management system called CVS[2]. CVS allows the user to tag a set of software for later reference by essentially keeping track of what the code looked like at different points in the past. It is then possible to retrieve exact copies of the software which were used to process the data. For example, to retrieve a copy of the code used in production version 1-6, one would type

```
cvs checkout -r prod-1-6 packages
```

The production version number in the previous example refers to a set of software libraries and executables that were used to process the data and were tagged under CVS. Generally, a new production version is not necessary for every step in the cooking process. The times when production versions are needed will be discussed in section 5. Essentially, though, a production version is needed whenever a long term copy of the "cooking" results will be kept.

It is also necessary for a record of the calibration maps and the executables used to "cook" the data to be kept as well. Unfortunately, CVS does not economically keep binary files under management. Therefore, these files must be "frozen" by the person "cooking" the files. These files are "frozen" by having the write permissions removed from the file. These files are then used to process the data and a snapshot of them is stored to the tape silo.

4 Elements of the off-line processing procedure

4.1 Executables

There are numerous programs that have been written to cook the data, calibrate the detectors, and give general monitoring overviews of the physics results from the cooked data file. It would be impractical, though, to try to incorporate all of these programs into the cooking process. A small number of executables have been chosen, though, that hopefully encompass the full spectrum of the code available. What follows is a list of all of the executables that are used in the "cooking" process along with a brief description. The process is not limited to just these programs though and others can easily be added to the "cooking".

4.1.1 Pre-Processing

In general the following programs do not require cooked data to provide useful output and can be run on the raw data file before processing.

[2]Refer to [CVS] for more information about the CVS code management system

- sync: About every thousand events or so during data taking, the readout controllers send a synchronization signal to the event builder to check whether event fragments have been lost. If one of these sync events is bad all of the data for the past thousand events could be corrupted. Therefore, it is counter productive to cook those thousand events. The sync program skims through the raw data file and writes out a text file that contains a list of event numbers between which there was a bad synchronization event. This text file can be used by a1 to skip unsynchronized events. Obviously, this program must be run before a1c is run.

- scaler_mon: *scaler_mon* integrates quantities found in the scaler banks needed for normalization. These quantities include the Faraday Cup information (for electron running) and the T counter scalers (for photon running). The text file (to stdout) produced by this program contains the integrated scalers for each data file. In addition, a Perl script called *gscalers2map* located in CLAS_SCRIPTS was developed for photon running to update the map for tagger normalization from the *scaler_mon* output text file.

4.1.2 Processing

Processing the data involves taking a file of BOS banks produced by the on-line data aquisition system and producing a file containing higher level information, such as four-vectors. This job is performed by a program called *a1c* and as such, it is the engine for the reconstruction scheme. To do this *a1c* reads an input file of BOS banks, obtains the necessary calibration constants to properly analyze these banks and produces a file of BOS banks containing various reconstruction banks. The ouput from *a1c* is then used as input for various monitoring and calibrations programs.

a1c is controlled by command line options and a single environment variable, *$CLAS_PARMS* map database files which contain the calibration constants. To obtain a list of the command line options for *a1c* type:

```
a1c -h
```

a1c contains a large number of command line options, most options in *a1c* relate to controlling which banks or events are contained in the output file, and not in the way in which those banks are created. This reflects the design philosophy for *a1c*, in that for production processing there should be only one correct way to produce any given bank and that way is the default in *a1c*. The most commonly used flag in *a1c* is the *-o<output_file>* flag. This flag specifies the name of the file of reconstruced BOS banks produced by *a1c*.

Two flags of particular importance for *a1c* are the bitwise process flags and drop flags. The bitwise process flag indicates to *a1c* which software packages will be turned on during its operation. To use the bitwise process flag, one gives the *-P* argument followed by a bit pattern. Below is the excerpt from the *a1c* command line help relating to the process flag.

```
[-P#] Bitwise Process flags, as defined below:
            0x1    Hit-based tracking (HBTR)
            0x2    Time-of-Flight (SC)
            0x4    Hit-based particle id (HBID)
            0x8    Electromagnetic Calorimeter (EC)
            0x10   Cerenkov Counter (CC)
            0x20   Time-based tracking (TBTR)
            0x40   CALL bank analysis (CL01)
            0x80   Time-based particle id (TBID, PART)
            0x100  Simple Event Builder analysis (SEB)
            0x200  LAC analysis
            0x400  Tagger analysis
            0x800  Start Counter analysis
            0x1000 Construct TRKS from HBID
            0x2000 Construct TRKS from TBID (re-does TB)
            0x4000 Construct RGLK bank(build region 1 track bank)
For example:  HBTR + HBID + TBTR: -P0x25
Default:  0xFFFF
```

To turn on a particular package one must turn on its associated bit, so for example if one wanted to process a data file turning on only hit based tracking and time-of-flight analysis one would use the bitwise process flag of *-P0x3* the 0x indicates that it is a hexidecimal number, and the three just comes from simple arithmatic ($0x1 + 0x2 = 0x3$). The bitwise drop flag removes from the output banks various banks and uses a bit pattern in a similiar fashon to the process flags. What follows is an excerpt from the command line help about the *-D* flag.

```
[-D#] Bitwise Drop flags, as defined below:
               0x1     Drop RC00->16 Banks
               0x2     Drop DC0 Bank
               0x4     Drop DC1 Bank
               0x8     Drop HBLA Bank
               0x10    Drop TBLA Bank
```

```
                0x20    Drop HBTB Bank
                0x40    Drop SC Banks
                0x80    Drop EC Banks
                0x100   Drop HBID Bank
                0x200   Drop CL01 Bank
                0x400   Drop SEB Banks
                0x800   Drop TBID,PART Bank
                0x1000  Drop HDPL Bank
                0x2000  Drop LAC Banks
                0x4000  Drop CC Banks
                0x8000  Drop ST Banks
                0x10000 Drop DHCL Banks
For example:  RC0-16 + DC0 + DC1 + HBLA: -D0xf
Default:  0x0
```

So for example if one wished to drop the HBLA and TBLA banks from the output stream on would use the flag *-D0x18* ($0x8 + 0x10 = 0x18$).

Additional flags of general interest are the *-r<bank_list>* flag which can be used to drop an arbitrary list of banks and the -s<options> flag which can be used to intruct a1c to write out only events with a specified number of hit based tracks.

An example of the operation of *a1c* using the process flags and drop flags used for general electron processing is shown below.:

```
a1c -o<output_file> -se -D0xf3cd -P0x3fb <input_file>
```

The location of the source code for a1 is in *packages/utilities/a1*. In addition a web page documenting some of the options available in *a1c* is accessable from the Jefferson Lab off-line web pages.

4.1.3 Monitoring and Calibration

Calibration programs generally read a RAW or COOKED data file of BOS banks and produce some calibration constants typically in ASCII text format. The user must then place these calibration constants into the proper map. Monitoring programs produce ASCII reports and histograms which can be archived in the off-line database. However, it must be emphasized that the distinction between monitoring and calibration programs is artificial, nothing prevents some of the information from an ASCII report produced by a monitoring program from being put into the map database. Likewise, nothing prevents a calibration program from being run on every processed data file and using the values of the constants produced as a monitor for a particular system. Since both types of programs are used for complimentary purposes, both of these types of executables are listed together in this section.

- ccmon: Produces a file of histograms and a text report that contains the T0 offsests for each Cerenkov counter detector element, which can be directly imported into the calibration map.

- elastic_mon: *elastic_mon* produces a single file of hbook histograms that can be used to obtain the elastic cross-section.

- inelastic_mon: Produces an hbook file containing plots of physics related quantities for the reactions $e'\pi^+n$, $e'\pi^0p$, $e'K^+X$, and $e'p\pi^+\pi^-$. It also produces a text report of the number of events of each topology observed, the contents of this report are placed in the off-line database and listed in Table 9.

- pdu: *pdu* finds hot and dead drift chamber wires. *pdu* produces three files the first of which is a list of all dead wires which can be used to update the DC_STATUS.map to provide a more accurate model of the acceptance. It also produces an executive summary of the number of dead and hot wires in each sector and region, as well as the change in status of the number of dead and hot wires compared to the map. The information from this data file is deposited in the off-line database and is listed in Table 8 of this note. *pdu* also produces a file containing several hbook histograms that graphically display the hot and dead wires for each run.

- photon_mon: Produces an hbook file and a text report. The hbook file contains several tagger and start counter related calibration and monitoring histograms, as well as physics related histograms for the π^+n, π^0p, K^+X and $p\pi^+\pi^-$ final states. The contents of the text report is stored in the off-line database and contains yields for various topologies and π^+n and π^0p yields by sector. *photon_mon* related database entries are listed in Table 10

- pid_mon: *pid_mon* monitors the quality of particle id and can be used to examine both pid and seb banks. *pid_mon* produces two files, a file of hbook histograms and a text file containing a table of particle yields by sector as well as fiqures of merit for the quality of particle id. The information from this text file are deposited into the off-line database during data processing. This contents of the off-line database stored for *pid_mon* are located in Tables 5 and 6.

- rfmon: *rf_mon* calculates the corrections that must be made in the rf time to determine an accurate mass and therefore an accurate particle identification.

- sc_mon: Creates a histogram file which contains scintillator occupancy plots, reconstructed occupancy plots, and performs rudimentary TOF efficiency calculations.

- trk_mon: Produces a file of hbook histograms and a report to *stdout* which can be piped into a text file. This text file contains tracking figures of merit such as the ratio of hit based tracks to time based tracks per sector, number of hits per time based track and the residuals by superlayer and sector. In addition quantites such as the beam position as determined by tracking are recorded, which can be used to update the map. The hbook file allows for a more detailed diagnosis of the tracking, in particular the time-to-distance calibration. A web page describing *trk_mon* in greater detail is accessable from the off-line web pages.

4.1.4 Filtering

The Hall B collaboration performs many experiments using CLAS at the same time. Hence, many people's analysis use only certain portions of the data. Resources would be a wasted if many people had to run through the same data file to find their hundred or so events every time they re-examined their data. It would also waste resource time if people needed to retrieve these full data files from the silo. An effort has been made, therefore, to skim through the data files to extract events that appeal to a large audience in the CLAS collaboration.

- e_filter: A large portion of the CLAS experimental program revolves around strangeness electroproduction. *e_filter* skims through the data file to keep events which pass a W cut and/or a hadron mass cut. It is currently used in the "cooking" process to keep only events which contain a kaon.

- g_filter: *g_filter* performs the same function for photon data as *e_filter* does for electroproduction.

- physfilter: Attempts to produce a data file with an enhanced number high multiplicity events. The program can be run in several different modes, for electron processing it extracts event with at least two negative and two positive time based tracks, in photon running it selects events with at least one negative and two postitive time based tracks.

4.2 "Cooking" Scripts

Due to the amount of data collected by CLAS, initial processing of the data must be performed on a farm of batch CPU's if that processing is to be completed in any reasonable amount of time. The software package Load Share Facility(LSF), which is managed by the CEBAF Computer Center(CC), handles the prioritization and the submission of jobs on JLAB's batch farm. LSF, however, does not provide any utilities to maintain records of a large quantity of jobs submitted by a user. Toward this goal there exists a basic set of Perl scripts which control the whole "cooking" process, all of the aforementioned executables, and the user's interactions with LSF. These scripts include *ProcessRun*, *nextRun*, *makeSub*, and *RunJob* and are all found in the CLAS CVS repository in the directory *packages/scripts/cooking_scripts*.

ProcessRun starts the whole process from the CUE machines and takes as its first argument a run number. The usage for ProcessRun is as follows

```
Usage:  ProcessRun [-h] run_number +env string +exe <exec> [+o <OS>]
     [+queue <queue>][-OPTS] ARGS
        -h                   Print this message.
        run_number           Run number to be processed.
        +env <string>        File name, w/path, of environment variable
                             text file.
        +exe <exec>          Executable to be run w/full path specified.
        +o <OS>              Operating system to run job on [default
                             = Linux].
        +queue <queue>       Farm queue to run on [default = ].
        -OPTS                Options preceeded by '-' passed to a1c.
        ARGS                 Arguments are passed to executable.
```

Any arguments, options preceeded by a minus sign, or options that *ProcessRun* does not recognize are passed automatically to the executable that will be run on the batch farm.

The other way in which the "cooking" scripts share information is through the use of environment variables. Thus, *ProcessRun*'s *+env* argument is very important to the execution of all of the scripts and must be specified when running *ProcessRun*. This option allows the user to specify a text file which contains all of the environment variables used by the scripts along with the definitions of these variables. These variables are

- CLAS_DB: This variable defines the location of the Perl off-line database to be used. This directory also contains the *list*, *done*, and *skip* files which will be discussed next. Lastly, this directory is the location of the submit files which will also be discussed later.

- CLAS_PROG: This variable defines the path to the directory where the progress files for every file submitted to the farm will be kept. The *running* file is also stored in this directory.

- CLAS_PARMS: The location of the CLAS calibration directory.

- CLAS_DATA: This variable defines the initial location of the data files to be processed.

- CLAS_OUT_PATH: The location on a CUE disk to where histogram and text files should be copied. When using *RunJob*, the subdirectories under this path will be in monitor/ and the name of the executable. For example, pid_mon histograms will be stored in CLAS_OUT_PATH/monitor/pid_mon.

- CLAS_TAPE_PATH: This variable specifies the path to which the cooked files, filtered files, and ntuples will be written. Due to the size of the files they are typically written to the silo. Under this directory, using *RunJob* the files will be written to the directories CLAS_TAPE_PATH/cooked, CLAS_TAPE_PATH/Kaons, CLAS_TAPE_PATH/physfilter and CLAS_TAPE_PATH/ntuples respectively. An internal check is used by *RunJob* to determine whether a *jput* or a *mv* is needed to move the files from the farm node disk to their final location. Therefore, these files can be written to a local disk as well.

- CLAS_SCRIPTS: This variable defines the location of the suplimentary cooking scripts such as *nextRun* and *makeSub*.

- CLAS_BIN: This variable defines the location of the analysis binary executables to be run. When defining this variable it is not necessary to include the operating system name since this information will be added by *RunJob* automatically depending upon which platform is used by the farm node running the job

- FILE_FILT: When performing a preliminary pass on data it is sometimes desirable to process only a certain amount of files from each run rather than the entire run. *FILE_FILT* stores the information about which file extensions should be cooked. If, for example, only the first three files should be cooked, *FILE_FILT* should contain *A00:A01:A02*, where all the file extensions to be cooked are separated by colons. If the entire run should be cooked, *FILE_FILT* should contain a *.

The variables *CLAS_PROG*, *CLAS_DB*, *CLAS_OUT_PATH*, and *CLAS_TAPE_PATH* define the location where output from *RunJob* and the executables it runs will be placed. If these directories do not exist previous to the execution of *ProcessRun*, the processing scripts will create them as they are needed. The standard directory structure used by these scripts can be found in Fig. 1. A sample text file to use with the *+env* option can also be found under CVS with the name *scripts/cooking_scripts/ENV_SRC_FILE*.

Returning to the initial discussion, if a run number is not specified, *ProcessRun* goes to the directory specified by the *CLAS_DB* environment variable and opens the previously mentioned *list*. This ASCII file should contain a list of run numbers to be processed with one run number per line. Once *ProcessRun* has a run number it passes it to *nextRun*. *nextRun* checks that the run has not already been submitted and that it has not been chosen to be skipped. *nextRun* performs this task by searching for the run number in the files *done*, which is automatically generated by *ProcessRun* after a job is submitted to LSF, and *skip*, which can be created by the user. Both of these files have the same format as *list* and are also in the *CLAS_DB* directory. If the *skip* file does not exist, *nextRun* only looks at the *done* file. If the user supplied the run number and the run has already been processed or it has been chosen to be skipped, the user is prompted as to whether he wishes to continue submitting that particular run. If the run was picked from *list*, *nextRun* continues to grab runs from *list* until it finds a run that is deemed suitable for processing.

Once a run has been selected, *ProcessRun* calls *makeSub*. *makeSub* creates a submission file that tells the farm node how to execute the job passed to it. It takes arguments from *ProcessRun* about the operating system to run on, the queue to place the job into, and the location of the executable that will perform the job.

After this submission script is completed a file of the form

```
PROJECT: clasE1
JOBNAME: Process-16159
COMMAND: /work/clas/production/e1/pass0.x/cooking_scripts/RunJob
MAIL: claschefcebaf.gov
OS: Linux
QUEUE: redhat52
OPTIONS: 16159 +P 0x4000 +base prod17
INPUT_FILES: /mss/clas/e1b/production/pass0.x/prod17/cooked/
/run16159_prod17.A00.00 /mss/clas/e1b/production/pass0.x/prod17/
cooked/run16159_prod17.A01.00
```

exists in the *CLAS_DB* directory. The above example is for the first two files of run 16159 to be processed on a Linux machine that is part of the redhat52 queue. The options line contains arguments that were passed to *ProcessRun* which are in turn

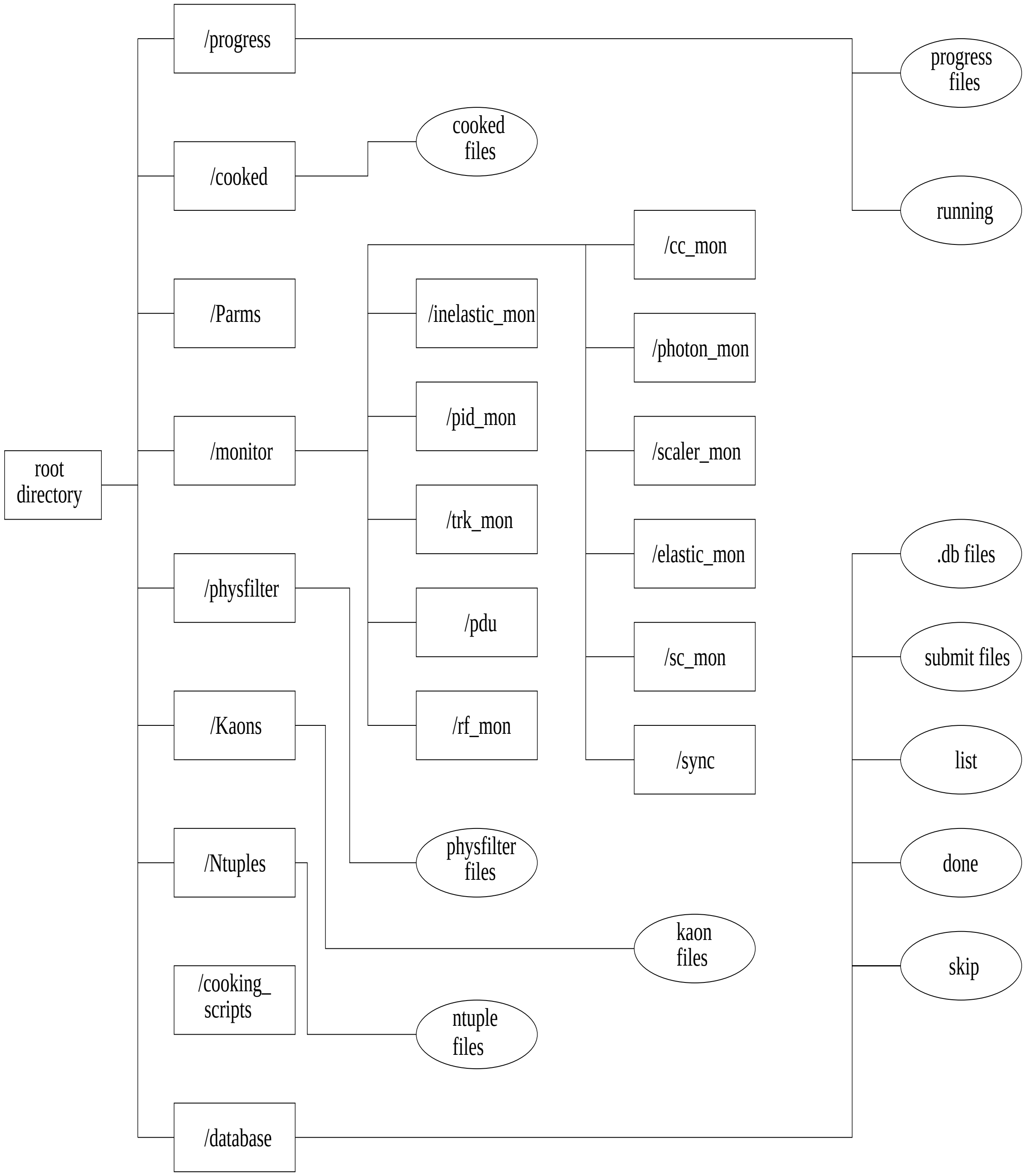


Figure 1: Standard directory structure of the off-line scripts. Rectangles represent directories while ovals represent files. If the directories have not already been created, the processing scripts will create them as they are needed in the above format. This structure is also the format that has been employed when processed run periods have been archived to the tape silo.

getting passed on to the executable running on the farm, which in this case is *RunJob*. *ProcessRun* then submits the job for the user to LSF.

In their current form, the scripts *ProcessRun*, *nextRun*, and *makeSub* can be used to process a number of runs on the batch farm using any executable. For the calibration and cooking processes, though, the program that is run on the farm nodes is the Perl script *RunJob*. *RunJob* performs the task of executing other binaries, managing their output files, and parsing their output for important information that will be stored in the off-line database. A schematic diagram of the operation of *RunJob* is shown in Fig. 2. *RunJob* has the following usage:

```
Usage:  RunJob run_num [-h] +P 0x# +env <string> [+base <string>]
        [+electron|+photon] [+S][+se | +sp]

        -h                  Print this help message.
        run_num             Run number to be processed.
        +P [0x#]            Bit wise process flag.
           0x1              Run a1c.
           0x2              Run pid_mon.
           0x4              Run pdu.
           0x8              Run scaler_mon.
           0x10             Run photon_mon.
           0x20             Run elastic_mon.
           0x40             Run inelastic_mon.
           0x80             Run cc_mon.
           0x100            Run e_filter.
           0x200            Run g_filter.
           0x400            Run physfilter.
           0x800            Run trk_mon.
           0x1000           Run rf_mon.
           0x2000           Run sc_mon.
           0x4000           Run Cole Smith's EC ntupilizer.
           0x8000           Run Sync check.
           0x10000          Run KK_filter.
           0x20000          Run Italian filter.
        +electron           Set run type as electron.
        +photon             Set run type as photon.
        +S                  Run pid_mon as seb_mon also
        +base <string>      Basename to attatch to each file
        +se                 Run standard electron processing (+electron
```

```
                    +P 0x3fdef +S -O -i -F -D0xf3dd -P0x3fb -se)
    +sp             Run standard photon processing (+photon -i
                    +P 0xb88f -T1 -st1 -O -D0x103d -cm0 -P0x1fff)
```

The executable *a1c* itself has a large number of flags which can be set. Therefore, any argument which is preceeded by a '-' that is passed to *RunJob* is automatically passed along to *a1c*. All of the other monitoring and calibration programs have their command line arguments hard coded into their specific subroutine. These subroutines are all found in *CLAS_PACK/packages/scripts/cooking_scripts*. The premise of these packages, which use the naming convention *RunNameofProgram.pl*, is that the data file that they are running on has already been "cooked" using the standard *a1c* options. Thus, they will remake the banks which are typically dropped in production "cooking".

One other *RunJob* option which deserves mention is the flag +base. All of the programs which *RunJob* executes allow the user to specify the output file name. The +base flag for *RunJob* allows the user to select a standard name that will be tacked on to these output files in order to identify the file. The run number is also an automatic addition to the names of each of the output files. When using *a1c*, if the +base option is specified for *RunJob*, it is not necessary to specify the *-N* option for *a1c*. Similarly, if *RunJob* executes both *a1c* and *sync*, it is not necessary for the user to specify the *-y* option for *a1c*. This will also be automatically taken care of for the user by *RunJob*.

The most important task that *RunJob* performs is to monitor and report back the status of the executables that it runs. The first method that *RunJob* utilizes is to constantly update a progress file specific to the data file being processed. This ASCII file is located in the *CLAS_PROG* directory and gives the chef an invaluable tool for analysis of the health of a particular job. Primarily what is stored in this file are time stamps when each executable was begun and completed. The second method that *RunJob* uses to monitor the executables it runs is to parse the output of those executables and store it in the Perl off-line database that was previously discussed in this paper.

5 Sequence of Operations

The production of a set of high quality calibrated data involves several steps. Each step (with the exception of step 0) involves processing some fraction of the data. Steps which involve writing data to the tape silo are more formal than other steps, and require that the version of the software libraries and production executables used for these steps be tagged under CVS management. A copy of the calibration map database used for these cooking passes should be frozen also. To emphasize the

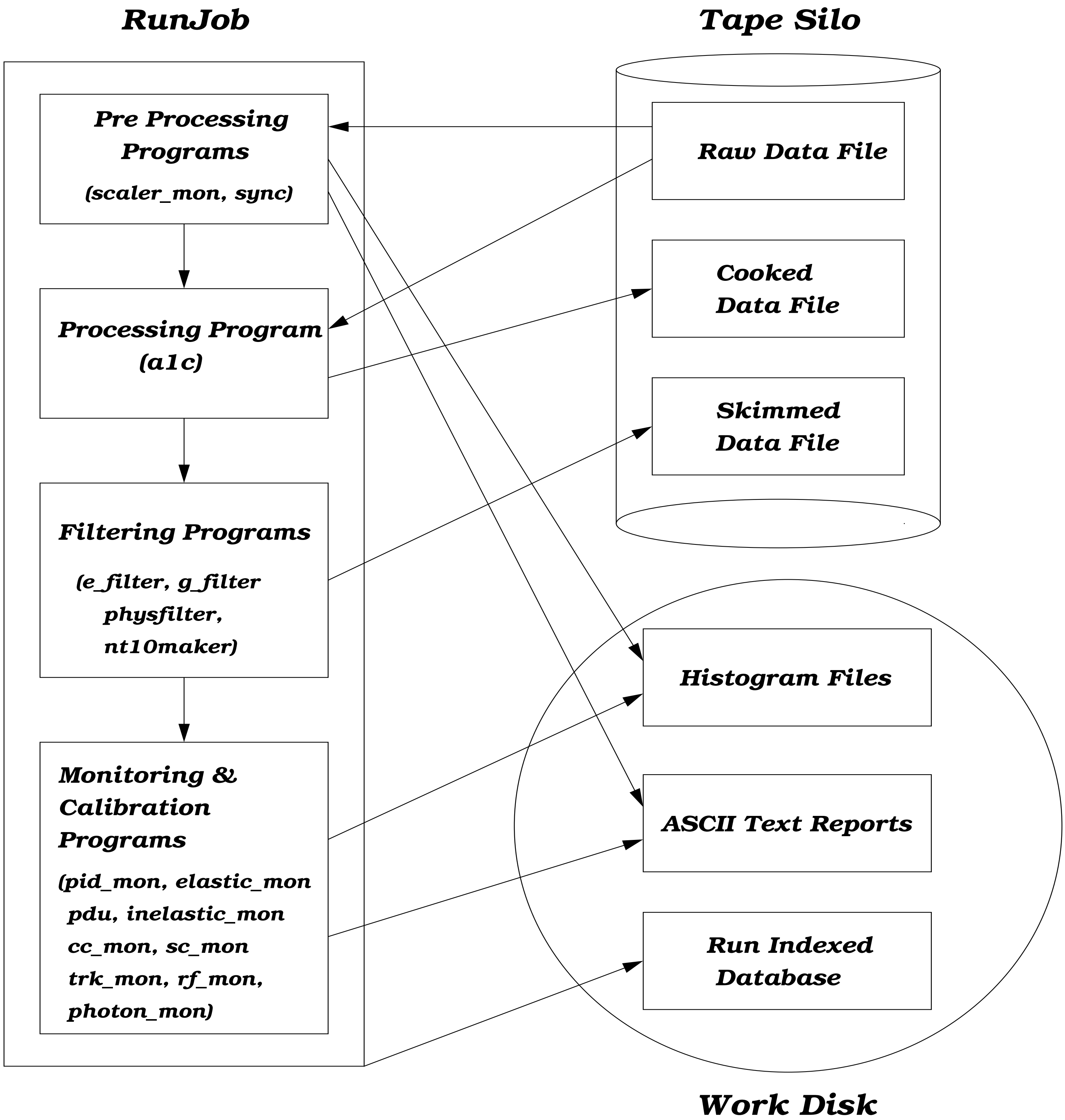


Figure 2: Schematic of the RunJob script showing the relationship of the analysis, skim and monitoring programs to the tape silo and the off-line database

importance of these steps they are called analysis passes. An outline of the processing procedure is shown in Fig. 3.

5.1 Step 0: Calibrations without tracking

The purpose of this step is to calibrate the data set to a high enough level to get an acceptable start time for time based tracking. In addition, any other calibrations that do not require the output of tracking or are derived from special calibration runs are performed. The tasks that must be performed for each system are as follows:

- All Subsystems: Copy a base set of calibrations into the map files to a run number at the beginning of your run period. The insures that your calibration constants are isolated from those of previous running periods.
- All Subsystems: Place up to date pedestals into map.
- Tracking: Choose an appropriate set of calibrations constants for time based tracking such that an inordinate number of tracks are not lost during the tracking calibration pass.
- Electromagnetic Calorimeter: Perform gain matching from special cosmic ray runs.
- Time of Flight: Perform the time walk corrections from TOF laser calibration runs.
- Cerenkov Counter: Determine the one photo-electron peak position from special pedestal runs.
- Tagger: Perform the full calibration of the tagger E and T counters from the photon normalization runs.
- Synchronize Timing (photon running): Set the relative timing between the tagger, start counter and time-of-flight.

A list of target runs suitable for production analysis should also be obtained at this step by inspection of the on-line logbook, on-line database and the progress files kept by shift takers on the on-line CLON cluster of computers.

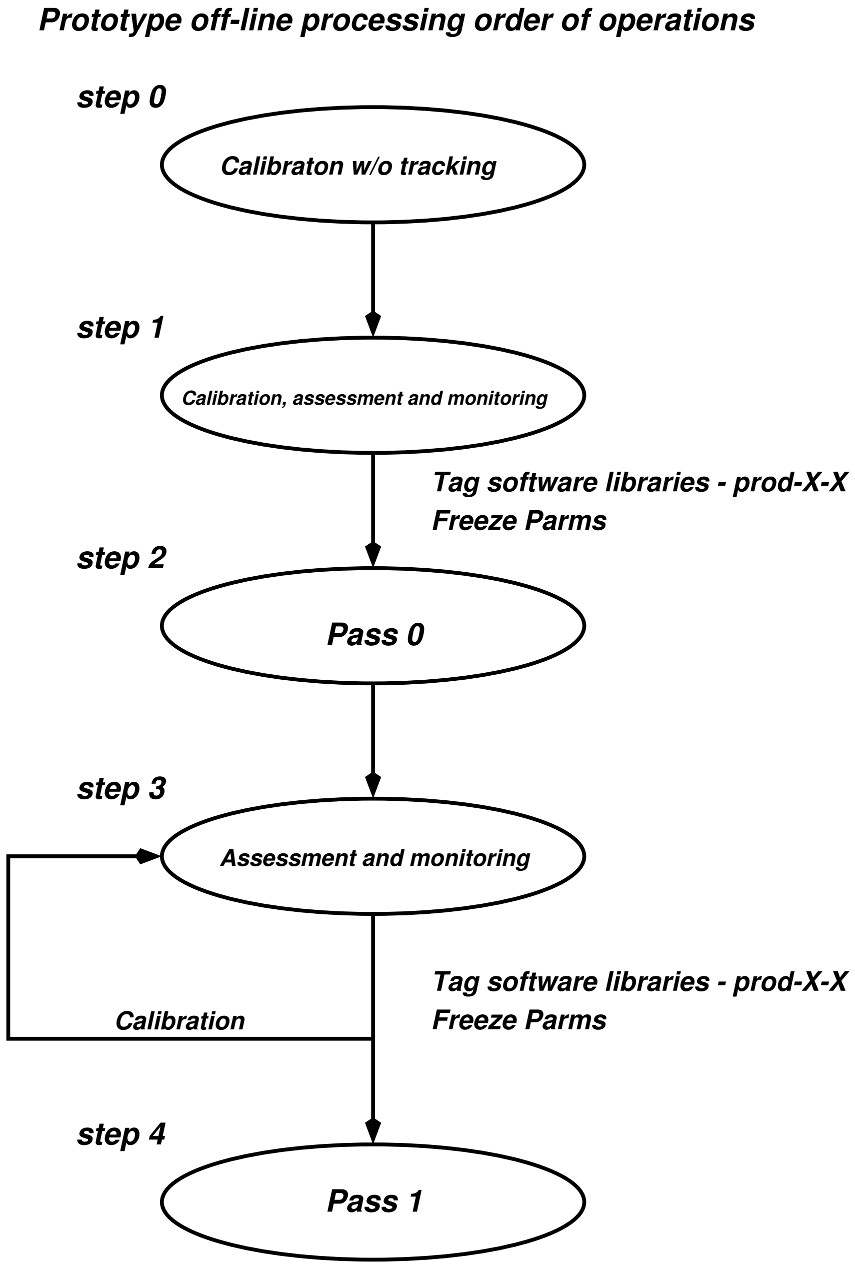


Figure 3: Flow chart for production of cooked data. A "pass" of the data implies freezing calibration constants, using tagged libraries to generate executables, and storing output files on the tape silo.

5.2 Step 1: Drift Chamber Calibration

The primary purpose of this step is to refine the time based tracking calibration. Specifically, the calibration constants for the time to distance function need to be calibrated and checked. This involves choosing calibration runs evenly spaced throughout the running period and the processing of at least 100K events from each of those runs. The exact frequency by which runs should be calibrated depends on the running period. Obviously, if the calibration constants change by a large amount from one calibration run to another the frequency of updating calibrations should be increased. A rule of thumb is the tracking calibrations should be checked at least once for every day of the running period.

While the primary purpose of this step is to obtain a first set of time based tracking calibrations, this step also provides a good opportunity to assess the overall quality of all calibrations. An early assessment of the quality of all calibration allows one to direct resources in an efficient manner by identifying potential problems early on in the cooking process. During this step all of the monitoring and batch submission machinery developed for production processing of the data is used to process 100K event in each tracking calibration run. Single particle yields, cross sections, tracking figures of merit, and calibration constants can all be examined as a function of run number. Problematic running periods can be identified, and runs can be selected for the next stage of analysis. In addition, this step is early enough in the cooking process to allow for modification of the software to account for novel running conditions.

5.3 Step 2: Pass 0 (calibration pass)

The purpose of the first formal pass through the data is twofold: first to analyze enough events to fully calibrate the time of flight system with pions, and second, to provide a data set with which people can refine analysis codes and the calibrations. Because this is a formal pass through the data which will serve as a reference for later analysis the following procedure should be followed before analyzing this pass:

- Construct a tagged software library: It is important to know exactly what version of the software was used to process the data, when problems are found in the cooked data files we must be able to refer back to the software used to create the files. These libraries should be identical the standard libraries. Other users should also be able to link their software to these libraries.

- Build a set of analysis, calibration, monitoring and utility executables from the tagged software library: Obviously, all the executables used during the cooking process must use the tagged software libraries.

- Freeze a version of the CLAS_PARMS area: Again, one must have a record of all of the ingredients used to construct the processed data file, the calibrations are a major source of input into the processing code.
- Tag the version of the processing scripts: Each of the elements of the processing scripts contain hard-coded executable commands with certain flags set. In order to keep an accurate record of how the data was processed a version of these scripts must be saved also.
- Processes selected runs.
- Copy all files, libraries and executables to tape silo: The final step in the process is to document everything you have done by archiving all the files needed for and produced by the cooking procedure to the tape silo.

5.4 Step 3: Assesment and Analysis

At this stage a detailed analysis is performed on the result of pass 0. The purpose of this step is to refine the calibration constants. Periodically it will be necessary to re-analyze a subset of the events with new calibration constants to assess their impact on data quality. This reassesment would be identical to step 1, in which several hundred thousand events are analyzed from a variety of data files from throughout the running period. The off-line database and monitoring histograms can then be inspected to gauge the impact of the new, refined calibration constants. Another activity that takes place during this process is the development of a new production library to be used for final data analysis. Quite often, code refinements need to be made or new calibration procedures are added and these additions need to be incorporated into the software build. When the running group is satisfied with the quality of the calibrations, presumably after several iterations, one can proceed to the final step.

5.4.1 Database monitoring

The quickest way to asses the quality of calibrations is to examine quanitities sensitive to calibration parameters as a function of run number. Quantities found useful for assessing the quality of calibration for the e1 and g1 running periods are enumerated below.

single particle yields: Examining the normalized yeild of various pariticle types as a function of run number can quickly identify problems with the timing of the apparatus. Shown in Fig. 4 is the single particle yield for g1a pass 0 and pass 1. Several features changed between pass 0 and pass 1, for example, π^+ are more common in pass 1 compared to pass 0 due to improvments in the time-to-distance function calibration. In addition, particle yields are more uniform over the running period. Most notably, the precipitous drop in proton yield seen from run 11800 to 11950 in pass0 is gone in pass 1, and the spike in the yields for kaons and unknowns near run 11750 seen in pass 0 is gone in pass 1. These changes reflect improvements in the time-of-flight and tagger calibration. The remaining variations in particle yield seen for the pass 1 data are due primarily to changes in running condition (30% field runs near 11750, and no-field runs near 11825) and normalization. The single particle database entries are derived from the *pid_mon* program, a list of quantities is available in Table 5.

In addition to the standard particle yields, *pid_mon* also calculates a number corresponding to the average quality of the tracks for each particle type, this number is proportional to the Chi-squared per degree of freedom determined from time based tracking. When looked at versus run number, a change in the quality factor can indicate a problem in the tracking calibrations. For example, Fig. 5 shows that the quality of the electron and proton tracks for runs 16152 and 16154 are much higher than the rest of the run period. Since this effect appears for both the electron and proton particle types, the logical source of the error points to the tracking calibrations. In this case, *scan_map* revealed that the first calibration in the *DC_DOCA.map* for this run period had a time stamp associated with run 16155. Thus, runs 16152 and 16154 were using tracking calibrations based on a previous run period. This problem was easily fixed by moving the calibration from run 16155 to run 16152.

ratio of time-based to hit-based tracks, hits per time based track: Sometimes tracking calibrations can be so poor that tracks are simply not reconstructed. When this occurs one will typically notice that the number of hits per time based track will be low in some sectors and the yields for number of time based tracks per sector will vary from sector to sector because of differing levels of calibration in various regions of the detector. Improvements in the overall level of tracking calibration can be seen in Figure 4 in the increase in yield for π^+ events from pass 0 (approximately 225K events per million triggers) to pass 1 (325K events per million triggers).

residuals: For a a more detailed understanding of the resolution provided by the tracking it is neccessary to observe the residuals as a function of run number. Shown in Figure 6 is the RMS of the residuals for various superlayers in sector 1, for an early iteration of the calibration and the final production pass. Note that for pass 0 the residuals for superlayer 1 are the largest (solid circles) this was noticed to be unusual because it was expected that superlayer 1 would

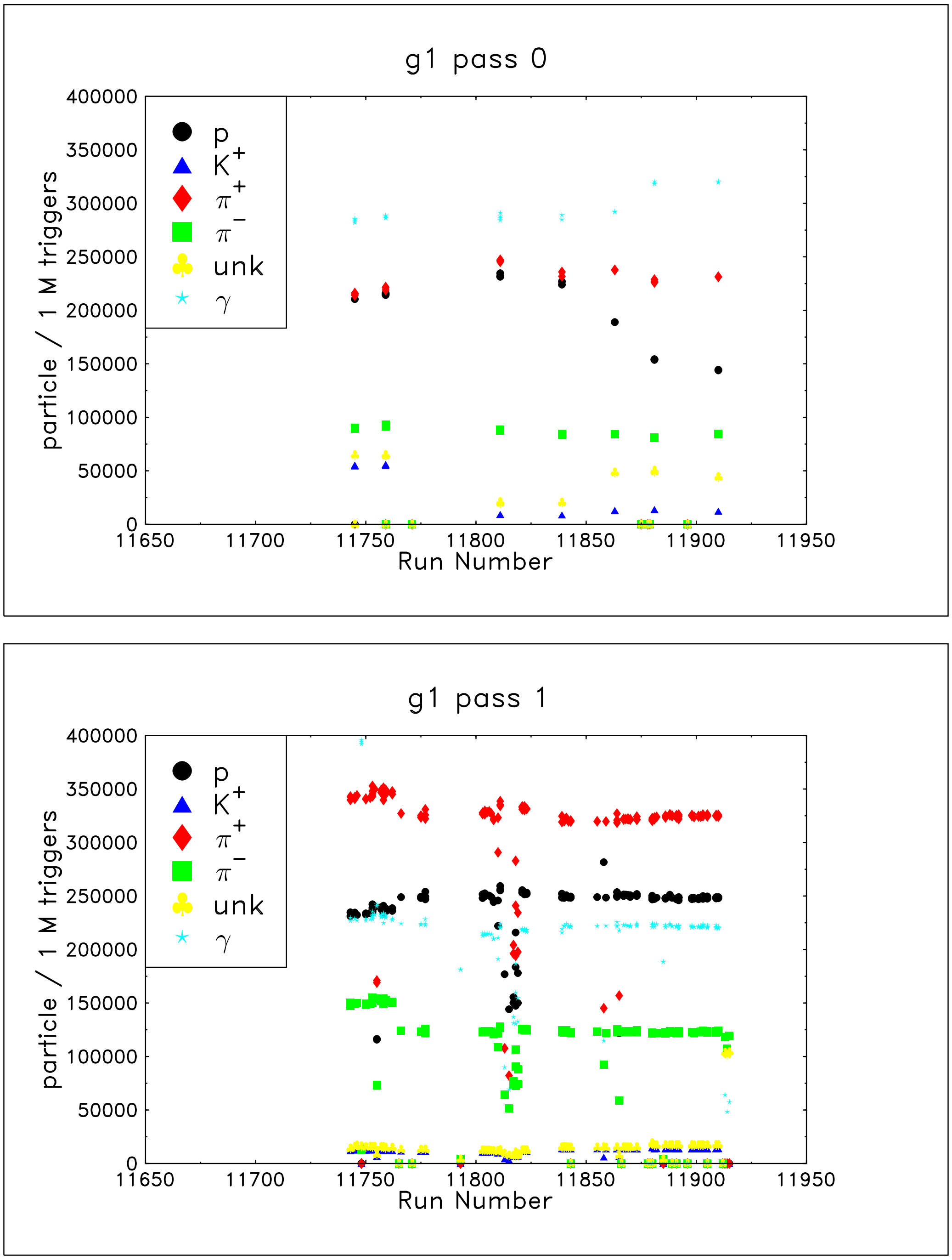


Figure 4: Particles identified by type per million triggers for the g1 period, "pass 0" results compared to "pass 1."

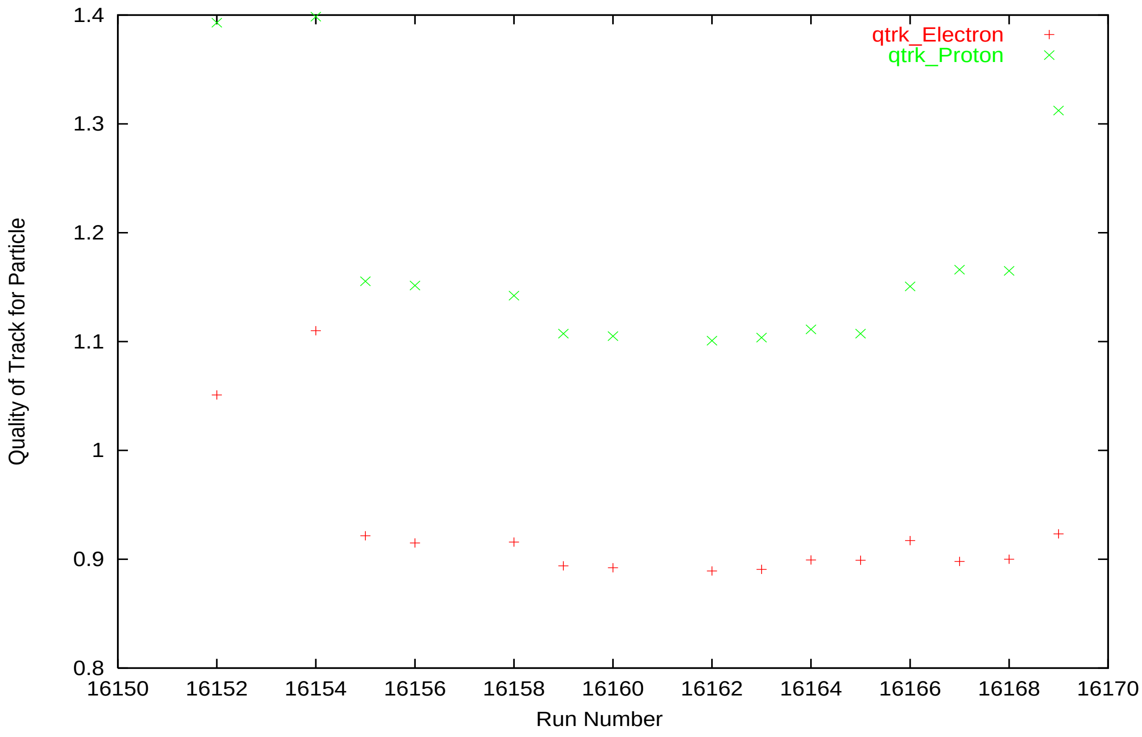


Figure 5: Quality of particle tracks from *pid_mon* for step 3 of the E1B running period using prod-1-9.

be the smallest due to its cell size. Upon closer inspection an anomaly was found in the trk_mon histograms leading to a discovery of a bug in the time based tracking software. You will note that for pass 1 of the data, superlayer 1 has the smallest residuals as expected, in addition there is now seen a clear gradation in resolution as a function of superlayer (superlayer 1 is smaller than superlayer 2 etc…).

dead wires: Observing the number of dead wires as a function of run-number is most useful for final run selection for data analysis.

yields for exclusive processes: Examining these quantitites provides similiar information to a single particle plot but can additionally provide useful information regarding the quality of the normalization of various runs.

Most of these quantities will vary with running period and are sensitive to changes in trigger and magnetic field, however, within a running period, these quanitites should be uniform.

5.4.2 Detailed diagnosis

While the time-line based monitoring plots described in the previous section are useful for identifying problem running periods, they have only a limited capability to determine the specific cause of the problems observed.

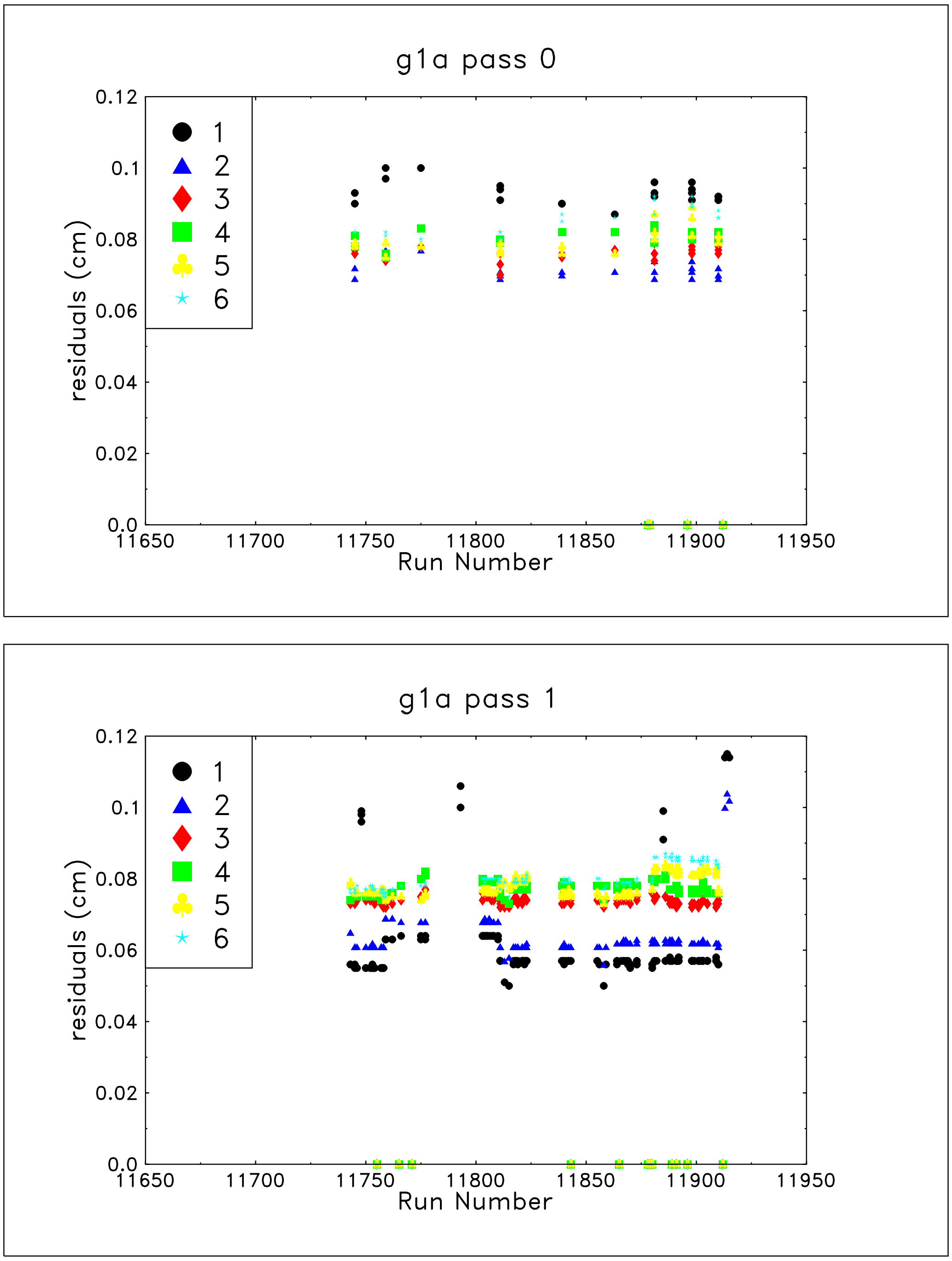


Figure 6: Top Figure: residuals in cm by superlayer for g1a pass 0, early tracking calibration before bug fix, Bottom Figure: residuals in cm by superlayer g1a pass 1, after bug fix and refined calibration procedure

Timing: The major source of problems seen in particle id, is in the relative timing of the various detector elements. Specifically, for electron beam data the primary souce of particle id problems is with miscalibrated time-of-flight paddles. In this case one will often observe an increase in the number of Kaons or unknown particles in a particular sector. For photon-beam data the situation is more complicated since the start time of the event is derived from information from the tagger, start counter and time-of-flight. A common problem seen in photon data is a misalignment in the timing between the tagger and time-of-flight. The alignment of the tagger and the time-of-flight is controlled by a single parameter contained in the map file TAG_CALIB.map in the tag2tof subsystem. A misalignment of this number is the cause of the drop in proton yield near run 11910 seen in Fig. 4. To check that the tagger and time-of-flight are aligned it is useful to look at the photon_mon histograms. Specifically, to check the tag2tof number we looked at photon_mon histogram number 109 which plots the difference between the start time determined by the tagger and the start time determined by the time-of-flight for pions. As can be seen in the top histogram in Fig. 7 the tag2tof number was misaligned by several nanoseconds for pass 0 processing of run 11910, before processing pass 1 this number was corrected.

Hardware Failure: Many times when a problem appears in the time-line based monitoring plots the possible sources of the problem can be narrowed down by looking at the *pid_mon* histogram associated with that run. For example, Fig. 8 shows the normalized number of gamma particles in each sector for the 4.46 GeV period of the E1B running period. The sudden jump in the total number of gammas at run 17746 is caused by the change in the main torus current from 3375 A to 2250 A. However, the increase in the number of gammas in sector 5 for runs 17591 and 17599 can not be written off so easily. The *pid_mon* histogram file for run 17599 contains a plot of the angular distribution of the gamma particles which can be seen in the top plot of Fig 9. This plot shows a strange pattern in sector five. This same pattern can not be seen in the electron distribution in the bottom plot of Fig. 9. Since electron reconstruction uses the full CLAS detector and gamma reconstruction uses only the calorimeter reconstruction this seems to indicate the sector 5 calorimeter as the source of the problem. At this point the EC expert was contacted and the source of the problem was identified as a set of noisy ADC boards in the calorimter.

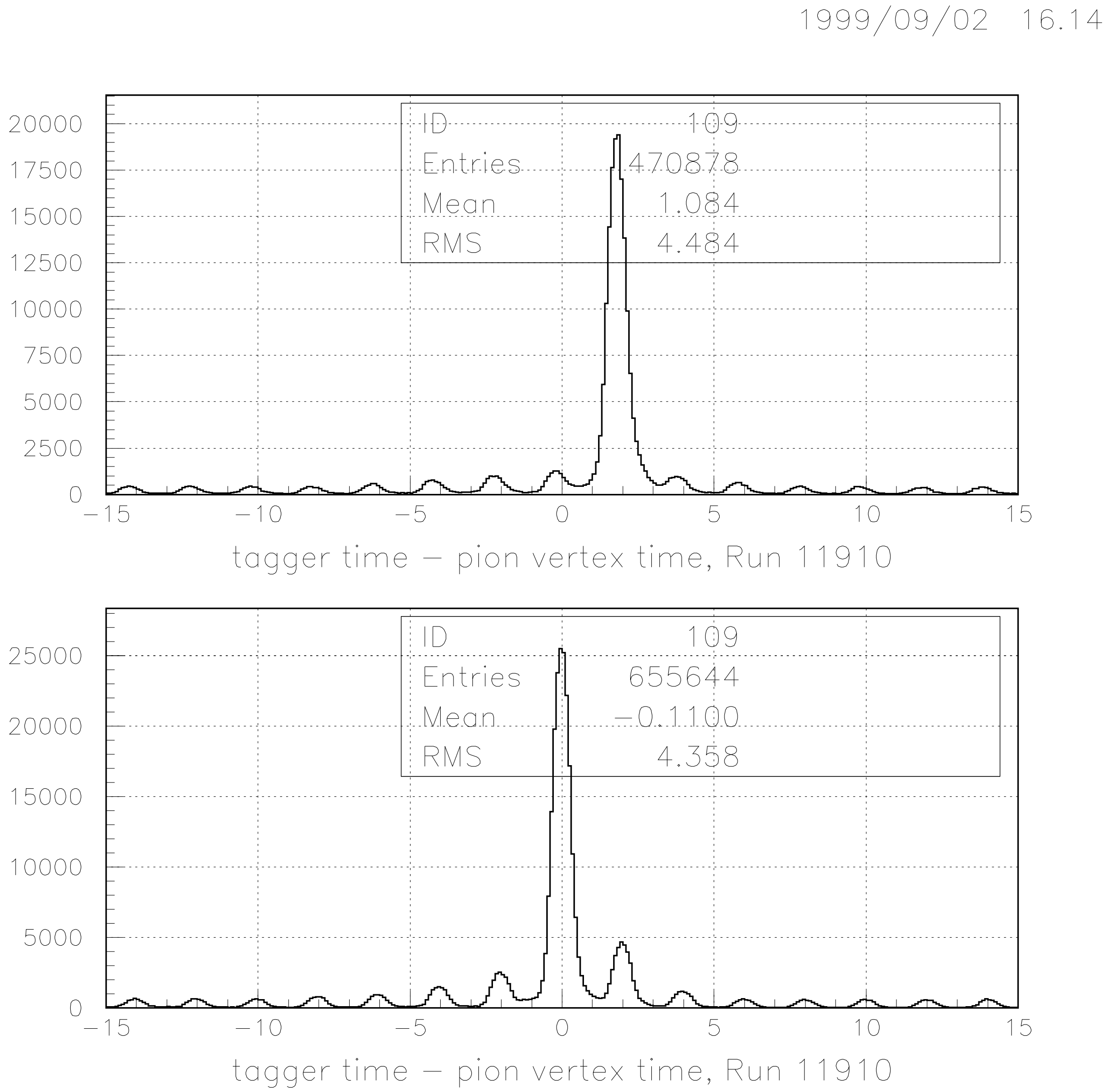


Figure 7: Top Figure: Tagger vertex time - pion vertex time, g1a pass 0, before tag2tof alignment, Bottom Figure: Tagger vertex time - pion vertex time, g1a pass 1, after tag2tof alignment

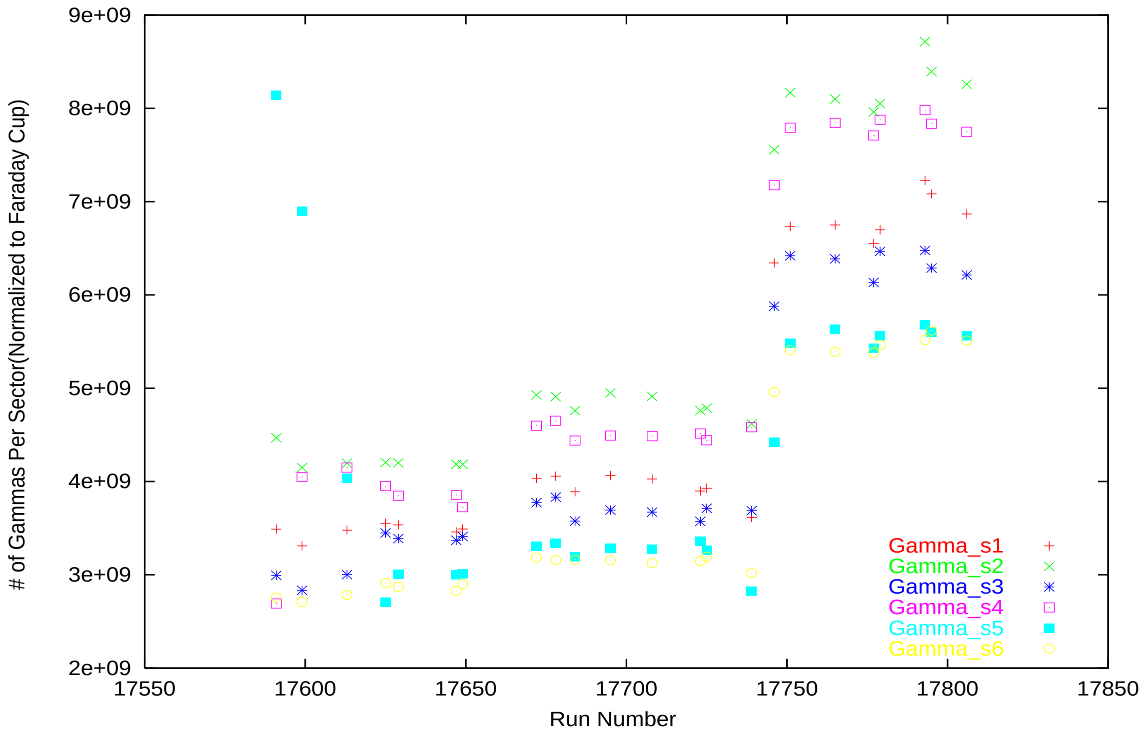


Figure 8: Number of gammas per sector normalized to the Faraday Cup for "pass 0" of the E1B running period using prod-1-7.

5.4.3 Tests Before Production

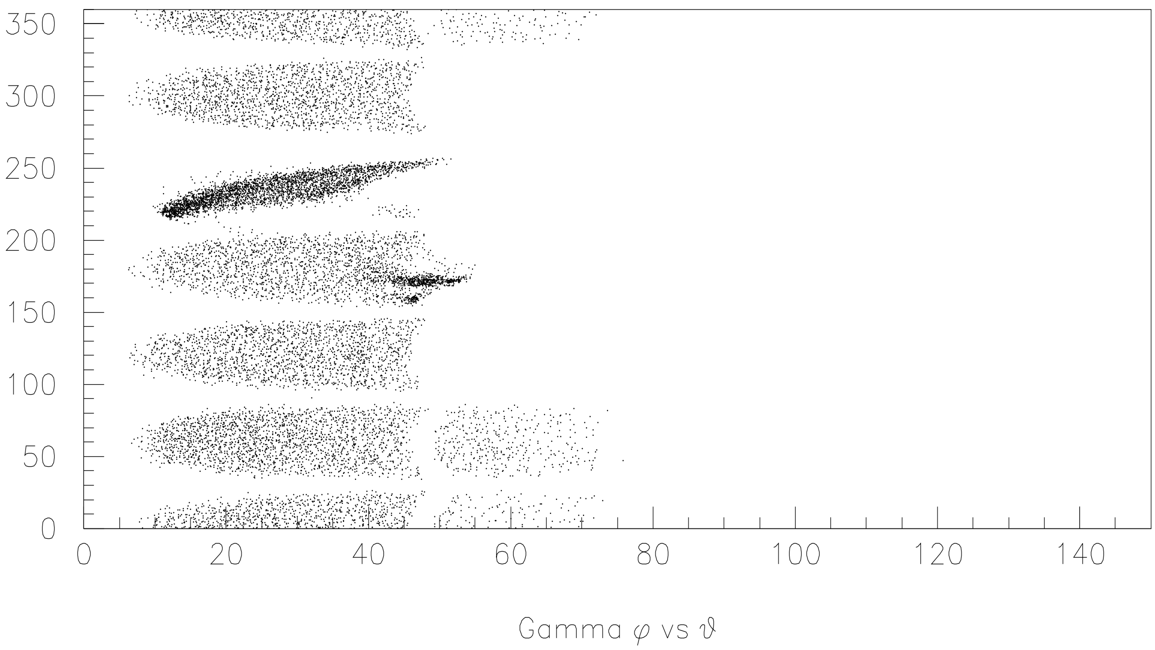
It is very important to make checks of the overall normalization of the experiment before proceeding with the production pass. Extracting absolute cross sections is a very good global check that the data make sense. Which reactions are chosen depend on the experiment. For electron scattering on hydrogen, we use the reactions ep elastic, ep $\rightarrow$ enπ^+ and ep $\rightarrow$ epπ°. For photon scattering on hydrogen γ p$\rightarrow$ nπ^+ and γ p$\rightarrow$pπ° can be used. For polarized target running, the asymmetry on the proton is a useful check.

5.5 Step 4: Pass 1 (production pass)

The final step in the analysis procedure is to perform a production analysis pass of the data with final calibrations. All runs which might have good data are processed during this step with no selection criteria. The details of this step are identical step 2 (Pass 0). The selection of runs suitable for physics analysis is the final part of this step and will be explained in some detail below.

5.5.1 Final run selection

The strategy we have found most useful for final run selection is to process ALL the data from a run period. After processing one can then use the database and tools



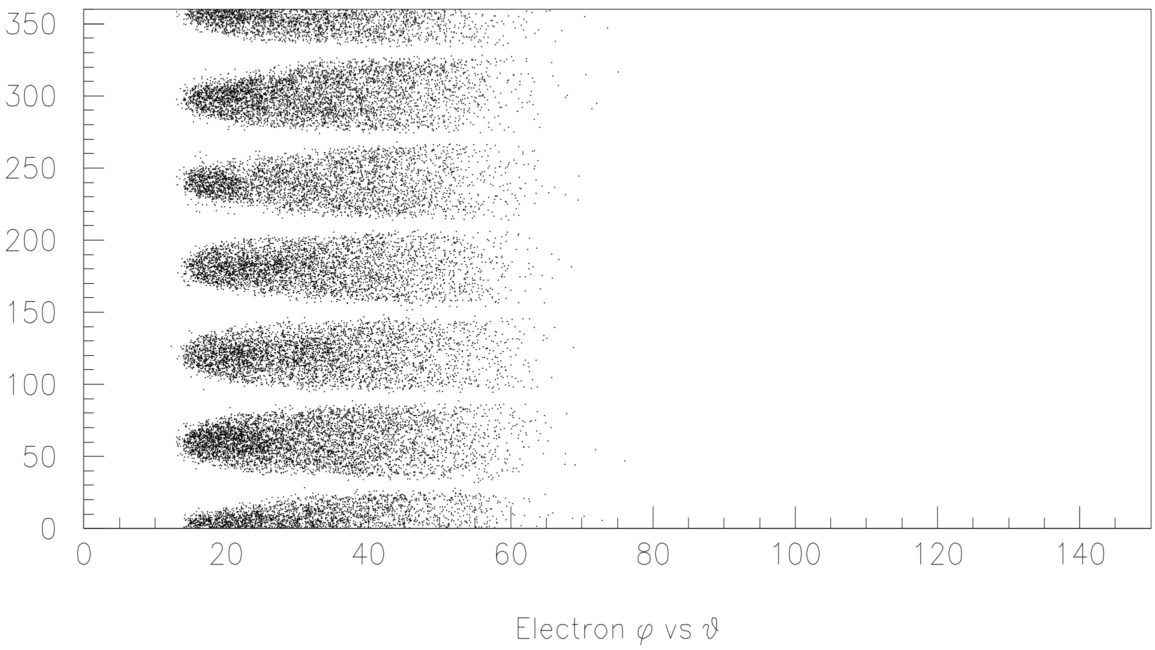


Figure 9: The top plot is the angular distribution of reconstructed gamma particles in CLAS. The bottom plot shows the angular distribution of reconstructed electrons in CLAS. Both of these plots were taken from the *pid_mon* histogram generated for run 17599 during step 3 of E1B "cooking".

developed for its inspection to generate a list of runs suitable for physics analysis. The exact values of run selection cuts vary from run period to run period below is a list of suggested quantities to cut on to generate a final run list.

- Normalized yields for single particles and exclusive final states
- ratio of time based to hit based tracks
- dead wires
- number of hits per Time based Track

6 Conclusions

The off-line software consists of a collection of programs and scripts which work together to produce the processed data. In the current scheme the main steering for the reconstruction is provided by the map database while monitoring information is provided by the off-line database. Tools described in this note have been provided to access each of these databases. A master Perl script is used to perform the production processing and construct the off-line database, this script coordinates the activity of the programs decribed in this note. An attempt was made to document only the programs and related scripts used by the master script, a large number of additional programs and scripts exist which are not described in this note. These programs are typically calibration programs which have an interactive character and hence can not be run as frequently as the codes described in this note.

In addition to describing the programs and scripts used in the off-line processing, an attempt was made to describe the use of the master script during the different stages of the cooking process. Particular emphasis is given to iterative nature of the process, because the master script can be run frequently, it can be run many times on a representative subset of the data. One can then relate problems observed in the off-line monitoring database to problems with the calibration constants stored in the map database to obtain a refined set of calibrations.

It is hoped that the processing and monitoring programs described in this note will continue to be used and refined. This project is the result of the collaboration of a large number of people in the CLAS collaboration and has proven itself to be a successful and flexible system capable of handling a large number of data sets and has proven to be amenable to expansion and modification. In particular, the authors would like to thank the following people for their contributions to the off-line montioring codes, scripts and procedures: Burin Asavapibhop (scaler_mon, tagger normalization), Rob Feuerbach(XBLA bank constructor - engine for trk_mon), Mike Klusman (photon_mon, scaler_mon), David Lawrence (run selection), Konstantin Lukashin (rf_mon, rf software), John McNabb(trk_mon), Simon Taylor (photon_mon, scaler_mon, sc_mon, sc software, tagger normalization), Richard Thompson (pdu, pid_mon and pid sofware, run selection), Dennis Weygand(prototype processing scripts).

References

[Perl] Randal L. Schwartz and Tom Christiansen. *Learning Perl* O'Reilly and Associates, Inc. 1997.

[CVS] Per Cederqvist *et al Version Management with CVS* Signum Support 1993.

A Selecting which runs to skim and calibrate

Before step 2, described in section 5, it is necessary to make a selection of runs which will be used for this calibration pass. The E1B run period will be presented as an example. The E1B run period occupied about 5 weeks of actual beam time, during which roughly 500 production runs were taken. It is neither practical nor necessary to calibrate each run individually. Instead, a subset of the production runs is chosen for calibration.

The process of choosing which runs to calibrate began with a review of the progress files for the E1B run period. All Hall B shift workers use a progress file to record the runs taken during their shifts, and those files are stored in the clasrun directory on the clon cluster. Most run periods have a specified number of triggers which should be accumulated for each run. The E1B run period used a target of ten million triggers per run, which normally corresponded to about 2 hours of data taking. However, some of the production runs taken during E1B contain much less than ten million triggers. There are many reasons runs were terminated at less than ten million triggers, and those reasons will not be discussed here. If a run does contain at least ten million triggers, it suggests the run conditions were smooth and uneventful, which are ideal conditions for deriving useful calibration constants. In addition, the runs that contain the most triggers are the runs that will dominate the scientific papers published by Hall B, so it is important for these runs to be well calibrated. The progress file were therefore perused to determine which runs possess at least ten million triggers.

The next criterium used to select which runs to calibrate is the elapsed time between the calibrations. The first time the E1B runs were calibrated, the elapsed time between the runs chosen for calibration was typically between eight and twelve hours, so either two or three runs per day of data taking were selected. The results of those calibrations determine if it is necessary to recalibrate in smaller time intervals. The Hall B online web page at http://db6.jlab.org/clas/html/clas-online/runinfo.html is consulted to determine the elasped times between the runs.

Finally, the operator logbook is reviewed to see if there were any anomalous conditions during any of the runs. There were hardly any problems with the detector hardware during the E1B run period. Unstable beam conditions were probably the most common potential problem for some of the runs. As much as possible, only runs which had stable beam conditions were selected for calibration.

It may sound as if a very small fraction of the runs are being calibrated, but actually the runs selected for calibration contain about 25 percent of all the data accumulated during the E1B run period, and the vast majority of the remaining 75 percent was taken within eight hours of one of the calibrated runs. Keep in mind that ten million triggers corresponds to roughly two hours of data taking, and three runs each containing at least ten million triggers corresponds to at least six hours of data taking. During the E1B run period, the average amount of data taken per day was probably about 14 hours.

B Database Variables and Their Definitions

Table 1: General Database Entries

Host Node	Node where cooking was performed
Processing Start Date	Date when job was started
Processing Start Time	Time when job was started
Processing End Date	Date when job was completed
Processing End Time	Time when job was completed
itorus	Current in main torus in Amps
iminitorus	Current in minitorus in Amps
itagger	Current in tagger in Amps
beam energy	Energy of beam in MeV

Table 2: Database Entries for scaler_mon

scaler_mon cmd	Command used to run scaler_mon
scaler_mon hbook	Location of scaler_mon output histogram file
scaler_mon text	Location of scaler_mon output text file
Trigger Rate	Rate at which data was written to disk (Hz units in value)
Clock Live time %	Livetime based on the clock
Ungated FC counts	Ungated counts from the Faraday cup
Ungated FC Coulombs	Ungated Faraday cup value in Coulombs
FC Live time %	Livetime based on the Faraday cup

Table 3: Database Entries for a1c and sync

Cooking Start Date	Date when cooking began
Cooking Start Time	Time when cooking began
Cooking End Date	Date when cooking ended
Cooking End Time	Time when cooking ended
sync command	Command used to run sync
sync file	Location of sync output text file
Total Sync Events	Total number of synchronozitation events in raw data file
# of Bad Sync Events	Total number of bad synchronozitation events in raw data file
Fraction Bad Sync Events	Fraction of total synchronozitation events which are bad
a1c Command Line	Command used to run a1c
Output files	Location of cooked data files
# of events	Total number of events in raw data file
# of events written	Total number of events kept after cooking process

Table 4: Database Entries for rf_mon

rf_mon command	Command used to run rf_mon
rf_mon histo file	Location of rf_mon output histogram file
rf peak height	Height of rf peak
rf TOF resolution	Resolution of rf time of flight value
rf offset	Value of RF offset

Table 5: Database Entries for pid_mon PART Bank

pid PART command	Command used to run pid_mon for PART banks
pid_mon text file	Output text file from pid_mon for PART banks
pid_mon hbook file	Ouput histogram file from pid_mon for PART banks
n_Electron	Total number of electrons from PART bank in cooked file
n_PiPlus	Total number of positive pions from PART bank in cooked file
n_PiMinus	Total number of negative pions from PART bank in cooked file
n_Deuteron	Total number of deuterons from PART bank in cooked file
n_KPlus	Total number of positive kaons from PART bank in cooked file
n_KMinus	Total number of negative kaons from PART bank in cooked file
n_Unknown	Total number of unknowns from PART bank in cooked file
n_Proton	Total number of protons from PART bank in cooked file
n_Gamma	Total number of neutrals from PART bank in cooked file
Electron_sI	Number of PART electrons in sector I = 1 ... 6
Proton_sI	Number of PART protons in sector I = 1 ... 6
PiPlus_sI	Number of PART positive pions in sector I = 1 ... 6
PiMinus_sI	Number of PART negative pions in sector I = 1 ... 6
KPlus_sI	Number of PART positive kaons in sector I = 1 ... 6
KMinus_sI	Number of PART negative kaons in sector I = 1 ... 6
Gamma_sI	Number of PART gammas in sector I = 1 ... 6
Deuteron_sI	Number of PART deuterons in sector I = 1 ... 6
Unknown_sI	Number of PART unknowns in sector I = 1 ... 6
HB pos_sI	Number of positive hit based tracks in sector I = 1 ... 6
HB neg_sI	Number of negative hit based tracks in sector I = 1 ... 6
TB pos_sI	Number of positive time based tracks in sector I = 1 ... 6
TB neg_sI	Number of negative time based tracks in sector I = 1 ... 6

Table 6: Database Entries for pid_mon SEB Bank

pid SEB command	Command used to run pid_mon for SEB banks
seb_mon text file	Output text file from pid_mon for SEB banks
seb_mon hbook file	Ouput histogram file from pid_mon for SEB banks
seb_n_Electron	Total number of electrons from SEB bank in cooked file
seb_n_PiPlus	Total number of positive pions from SEB bank in cooked file
seb_n_PiMinus	Total number of negative pions from SEB bank in cooked file
seb_n_Deuteron	Total number of deuterons from SEB bank in cooked file
seb_n_KPlus	Total number of positive kaons from SEB bank in cooked file
seb_n_KMinus	Total number of negative kaons from SEB bank in cooked file
seb_n_Unknown	Total number of unknowns from SEB bank in cooked file
seb_n_Proton	Total number of protons from SEB bank in cooked file
seb_n_Gamma	Total number of neutrals from SEB bank in cooked file
seb_Electron_sI	Number of SEB electrons in sector I = 1 ... 6
seb_Proton_sI	Number of SEB protons in sector I = 1 ... 6
seb_PiPlus_sI	Number of SEB positive pions in sector I = 1 ... 6
seb_PiMinus_sI	Number of SEB negative pions in sector I = 1 ... 6
seb_KPlus_sI	Number of SEB positive kaons in sector I = 1 ... 6
seb_KMinus_sI	Number of SEB negative kaons in sector I = 1 ... 6
seb_Gamma_sI	Number of SEB gammas in sector I = 1 ... 6
seb_Deuteron_sI	Number of SEB deuterons in sector I = 1 ... 6
seb_Unknown_sI	Number of SEB unknowns in sector I = 1 ... 6
seb_HB pos_sI	Number of positive hit based tracks in sector I = 1 ... 6
seb_HB neg_sI	Number of negative hit based tracks in sector I = 1 ... 6
seb_TB pos_sI	Number of positive time based tracks in sector I = 1 ... 6
seb_TB neg_sI	Number of negative time based tracks in sector I = 1 ... 6

Table 7: Database Entries for pid_mon Quality

qtrk_Electron	Quality of electron track from PART bank
qtrk_Proton	Quality of proton track from PART bank
qtrk_PiPlus	Quality of positive pion track from PART bank
qtrk_PiMinus	Quality of negative pion track from PART bank
qtrk_KPlus	Quality of positive kaon track from PART bank
qtrk_KMinus	Quality of negative kaon track from PART bank
qtrk_Gamma	Quality of gamma track from PART bank
qtrk_Deuteron	Quality of deuteron track from PART bank
qtrk_Unknown	Quality of unknown track from PART bank
seb_qtrk_Electron	Quality of electron track from SEB bank
seb_qtrk_Proton	Quality of proton track from SEB bank
seb_qtrk_PiPlus	Quality of positive pion track from SEB bank
seb_qtrk_PiMinus	Quality of negative pion track from SEB bank
seb_qtrk_KPlus	Quality of positive kaon track from SEB bank
seb_qtrk_KMinus	Quality of negative kaon track from SEB bank
seb_qtrk_Gamma	Quality of gamma track from SEB bank
seb_qtrk_Deuteron	Quality of deuteron track from SEB bank
seb_qtrk_Unknown	Quality of unknown track from SEB bank

Table 8: Database Entries for pdu

pdu command	Command used to run pdu
pdu out file	Location of pdu output text file
pdu wire file	Location of pdu output wire map file
pdu histo file	Location of pdu output histogram file
pdu_sIrJ_dead	Number of dead drift chamber wires sector I = 1 ... 6, region J = 1 ... 3
pdu_sIrJ_hot	Number of hot drift chamber wires sector I = 1 ... 6, region J = 1 ... 3

Table 9: Database Entries for inelastic_mon

inelastic_mon command	Command used to run inelastic_mon
inelastic_mon file	Location of inelastic_mon output histogram file
ine ep	Number of inelastic e p events from PART bank
ine epi+pi-p	Number of inelastic $e'\pi^+\pi^-p$ events from PART bank
ine epi+pi-	Number of inelastic $e'\pi^+\pi^-$ events from PART bank
ine epi+	Number of inelastic $e'\pi^+$ events from PART bank
ine epi+p	Number of inelastic $e'\pi^+p$ events from PART bank
ine epi-p	Number of inelastic $e'\pi^-p$ events from PART bank
ine eK+	Number of inelastic $e'K^+$ events from PART bank
ine eK+p	Number of inelastic $e'K^+p$ events from PART bank

Table 10: Database Entries for photon_mon (PART bank only)

photon_mon command	Command used to run photon_mon
photon_mon file	Location of photon_mon output histogram file
p	Number of events with a single proton
pi+	Number of events with a single π^+
pi-	Number of events with a single π^+
K+	Number of events with a single K^+
K+p	Number of K^+p events
pi-p	Number of π^-p events
pi+p	Number of π^+p events
pi+pi-	Number of $\pi^+\pi^-$ events
pi+pi-p	Number of $\pi^+\pi^-p$ events
pi+n(f)_sI	Number of π^+n events in observed sector=I within fiducial cut
pi+n_sI	Number of π^+n events in observed in sector=I
pi0p(f)_sI	Number of π^0p events in observed in sector=I within fiducial cut
pi0p_sI	Number of π^0p events in observed in each sector

Table 11: Database Entries for trk_mon

trk_mon command	Command used to run trk_mon
trk_mon txt file	Location of trk_mon output text file
trk_mon histo file	Location of trk_mon histogram output file
nhits/trk_I	Number of hits per track for sector I = 1 ... 6
hbt/evt_I	Number of hit based tracks per event for sector I = 1 ... 6
tbt/evt_I	Number of time based tracks per event for sector I = 1 ... 6
tbt/hbt_I	Number of time based tracks per hit based track for sector I = 1 ... 6
flat_sI_slJ	Arbitrary flatness fit for DOCA sector I=1 ... 6, superlayer J=1 ... 6
rms_sI_slJ	RMS of hit and reconstruction sector I=1 ... 6, superlayer J=1... 6
3 track events	Number of three track events used for vertex reconstruction
Beam x position	Position along x axis of vertex
Beam y position	Position along y axis of vertex
Beam z position	Position along z axis of vertex
Beam fit sigma x	Sigma of fit of x position of vertex
Beam fit sigma y	Sigma of fit of y position of vertex
Beam fit sigma z	Sigma of fit of z position of vertex
Beam fit chi2 x	Chi squared of fit of vertex along x axis
Beam fit chi2 y	Chi squared of fit of vertex along y axis
Beam fit chi2 z	Chi squared of fit of vertex along z axis

Table 12: Database Entries for Other Monitoring Programs

elastic_mon command	Command used to run elastic_mon
elastic_mon histo file	Location of elastic_mon output histogram file
cc_mon command	Command used to run cc_mon
cc_mon histo file	Location of cc_mon output histogram file
cc_mon text file	Location of cc_mon output text file
sc_mon command	Command used to run sc_mon
sc_mon histo file	Location of cc_mon output histogram file
EC ntupilizer command	Command used to run EC calibration ntupilizer
EC ntuple file	Location of output EC calibration ntuple

Table 13: Database Entries for Filtering Programs

physfilter command	Command used to run physfilter
physfilter file	Location of physfilter output event file
physfilter filtered events	Total number of events that fit physfilter filtering conditions
efilter command	Command used to run e_filter
efiltered file	Location of e_filter output event file
efilter W min	Minimum W value for events kept by e_filter
efilter W max cut	Maximum W value for events kept by e_filter
efilter hadron MM$\hat{2}$ max cut	Maximum missing mass squared kept by e_filter
efilter hadron MM$\hat{2}$ min cut	Minimum missing mass squared kept by e_filter
kaon filtered events	Total number of events kept by e_filter