Rootbeer Manual

S. Fegan[1], K. Livingston[2], and W. Booth[1]

[1]School of Physics Engineering and Technology, University of York, UK
[2]SUPA School of Physics and Astronomy, University of Glasgow, UK

March 14, 2025

Abstract

This is an updated version of the Rootbeer manual, based on Ken Livingston's original (October 2005), distributed with various versions of the Rootbeer source code and previously available online. The intention of this note is to provide a static version of this guide matching the use cases and workflows of Rootbeer to the current (2025) CLAS6 software container, and the standalone version, built against ROOT version 6.

1 Introduction

Rootbeer (suggested meaning 'ROOT Bank Event Extraction Routines') is a package for the analysis and conversion of CLAS BOS format data [1] using ROOT [2]. It aims to make it easy to write ROOT based code to analyse CLAS data in BOS format and to produce ROOT DSTs (Data Summary Tapes). Here are some advantages.

- Independent of the CLAS software - only require `clasbanks.ddl` to build rootbeer.
- Make rootDST files which hold BANKs plus any other desired ROOT objects.
- Read rootDST or .bos files with the same code.
- Simple access to BANK data.
- Write ROOT code to be interpreted or compiled.
- Write ROOT based executables.
- Modularise quickly and simply by writing ROOT libraries.

These use of rootbeer is best illustrated with some code, and some starter examples are available in the `sample_code` folder (see section 3). As illustrated in the sample code, for each bank defined in `clasbanks.ddl` there is a structure `BANK_t`, and within `libRootBeer.so`, variables like `PART_t *PART`, `int PART_NH` etc. are defined and made globally availble.

In user code there is never the need to explicitly request a BANK, as this is handled by the `rootbeer->GetEvent()` function, which does a lot of stuff with tables of `void**`'s and makes the `BANK_t` structs point to the data. These are the 2 aspects of rootbeer which it is hoped make it easy to use:

1. `rootbeer->GetEvent()` gets all the required banks for the next event.
2. All BANK variables are made globally availble by including the header `RootBeerUtil.h`. They are accessible to the root prompt (i.e. in ROOTCINT), and in compiled code linked with `libRootBeer.so`. The aim is to make it simple to develop new code, and simple to modularise analysis by writing small function libraries.

2 Installation

2.1 Prerequisites

- `clasbanks.ddl` file describing the bank structure of the data files you want to analyse (several variants exist with minor changes to the bank variables).
- An installation of ROOT, the version of which depends on whether you are using the legacy container version or the currently maintained version (see next section).

2.2 Choose your version

Two versions of Rootbeer currently exist;

- **Rootbeer2.1** - Legacy version for use in the CLAS6 software container. This builds on Centos7.9 against ROOT version 5.34. Features and development frozen, save for bugfixes affecting operation in the container.
- **Rootbeer6.0** - "Actively" maintained version, combining various efforts of Glasgow and York/Edinburgh over the years to continue to use Rootbeer from ROOT version 6 onwards. New libraries added, including g13 momentum corrections.

2.3 Installation

1. Download the latest version from the Rootbeer repository:
   `https://github.com/JeffersonLab/rootbeer2.1.git`
   for Rootbeer2.1. And
   `https://github.com/JeffersonLab/rootbeer6.0.git`
   for Rootbeer6.0
2. You may want to edit the following files although the defaults are probably OK to begin with.

   (a) `include/clasbanks.ddl`. This is from the CLAS packages, and should be consistent with the BANKs which are in the data. If you know of any new banks or want to copy over a different `clasbanks.ddl` file now is the time to do it.

(b) `dat/singles.dat` This is a list of "single sector" banks. It is assumed that banks are multi-sector (i.e. there may be several instances of the BANK in one event) unless the are listed in this file. I've put in ones I'm sure are singles, but may have missed a few. It doesn't matter really, but singles are referred to as `BANK[row].member` Whereas multi-sector banks are referred to as `BANK[sector][row].member`. If a single doesn't get flagged in singles.dat, it has to be referred to as `BANK[0][row].member`.

3. Edit your `~/.cshrc` file (or equivalent) to make sure root and rootbeer are setup correctly. For example, put the following lines in `~/.cshrc`:

   ```
   setenv ROOTSYS <PATH_TO_ROOT>
   setenv LD_LIBRARY_PATH "${LD_LIBRARY_PATH}:${ROOTSYS}/lib"
   setenv PATH "${PATH}:${ROOTSYS}/bin"
   setenv ROOTBEER ${HOME}/rootbeer2.1
   ```
   (or `setenv ROOTBEER ${HOME}/rootbeer6.0`)
   ```
   source ${ROOTBEER}/scripts/rootbeer.cshrc
   ```

   Note. The final "source" command will setup several `ROOTBEER` environment variables, and modify your `PATH` and `LD_LIBRARY_PATH` appropriately. It also creates an alias to start rootbeer in the appropriate directory, with the correct setup macro:
   ```
   alias rootbeer "cd /home/user/rootbeer2.1;root -l RootBee\
   rSetup.cxx"
   ```
   (substitute `rootbeer6.0` for rootbeer6.0)

   Open a new shell to work in, since you need to pick up the new environment, etc.

4.
   ```
   >cd $ROOTBEER
   >make
   ```
   This will generate the some header files in the current directory and the following files;

   - `${ROOTBEER_SLIB}/libRootBeer.so`
     A shared library which gets loaded into ROOT to allow handling of BANKs in bos or rootDST files. Executables must be linked with this.
   - `${ROOTBEER_BIN}/bankdump`
     An executable to dump BANKs from a bos or rootDST file.

- `${ROOTBEER_BIN}/getbanks`
  Run on bos or rootDST file to produce list of banknames in the file.

If running csh or tcsh, do
```
>rehash
```
To make sure you pick up the executables in your path.

If you also want to build the root documentation for the classes in `libRootBeer.so`, do
```
>make htmldoc
```
This will create an htmldoc directory.

For other options in the `Makefile` see section 4.

2.4 Testing

To test your installation, try dumping 20 events of the test DST file rbtestDST.root:
```
>bankdump -N20 rbtestDST.root
```
Repeat this process selecting some banks, for example:
```
>bankdump -N20 -GEPIC -GTESC rbtestDST.root
```
Try dumping 20 events of a bos file:
```
>bankdump -N20 <bosfile>
```

3 Tutorial

The `sample_code` directory contains some examples to use for getting started. It is assumed that development of code will be done interactively at the root prompt using macros, but for anything useful these should be turned into shared libraries or executables as appropriate. Both are illustrated here. Examples shown can be run on either BOS or rootDST files. There's also an example of how to make rootDST files (from BOS, or existing rootDST files).

Start rootbeer

```
>root RootBeerSetup.cxx                   // Start root up in rootbeer mode
```

Or if you have set up an alias;

```
>rootbeer                                  // Start root up in rootbeer mode
```

ROOT will open and execute the script `RootBeerSetup.cxx`, the format of the terminal prompt may also be modified, depending on your shell environment. For example, you may have the prompt

`rootbeer>` or `rootbeer [0]` or similar.

Load and run a macro

```
rootbeer> .L rbtest.C                      // Load in the sample macro
rootbeer> rbtest(1000, "rbtestDST.root","test.root")
   // Run 1000 events on rbtestDST.root, output to test.root
rootbeer> b = new TBrowser()               // Start a ROOT TBrowser
```

In the browser select "ROOT Files", then double click on "test.root". Now you can look at the histograms you produced by running the macro (see Figure 1).

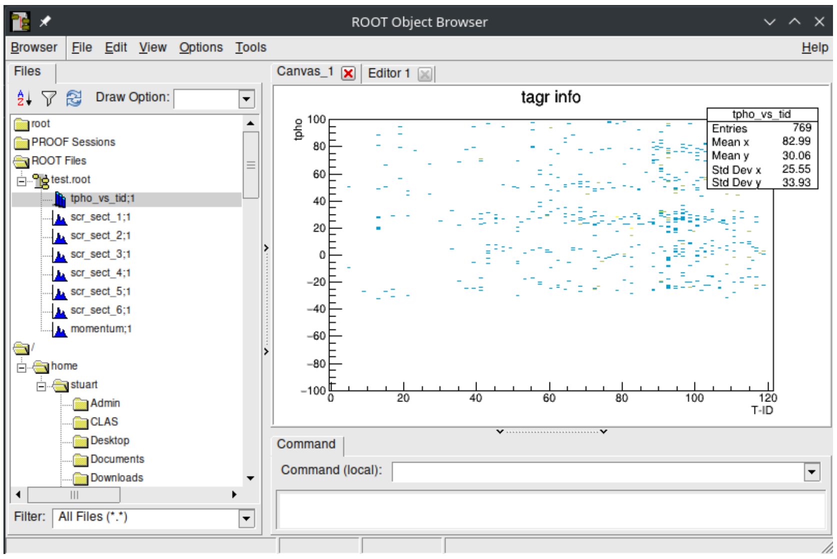

Figure 1: ROOT TBrowser showing the output root file from running the `rbtest.C` macro on the rbtestDST.root sample DST file. A 2D histogram of photon tagger information is plotted.

If the file is kept open in an editor, it can be modified, reloaded and run again using the above commands without exiting from root.

Create an executable from the macro

```
rootbeer> mkexe("sample_code/rbtest.C","")      // Compile macro as an executable
rootbeer> .q                                    // Quit root
```

Note: This makes an executable `rbtest` in `${ROOTBEER_BIN}`. The second argument is for any extra flags to be passed to the compiler (e.g. for a library which is not already loaded into root).
If `mkexe_deb()` is called instead of `mkexe()`, the executable will be built with the debugging flag on.

To see how to write the macro for use as an executable, look in the source code at the `#ifdef ROOTEXE` statements

Now run the executable:

```
> rehash                                        // To pick it up in your path
> rbtest -N1000 rbtestDST.root test.root        // Run it
```

Open `test.root` in root to see the histograms

The easiest way to make a new executable.

Start with the one you last worked on (eg `rbtest.C`).
Copy it to a new name. (e.g. `cp sample_code/rbtest.C sample_code/myana.C`)
Open the new code and do a search and replace (eg. replace all instances of "`rbtest`" with "`myana`").
Hack away and customize the new code.
Load and test it as described above.
When it works, create an executable as described above.
(The same goes for new libraries - as described below).

Creation of executables and shared libraries.

Rootbeer defines some simple functions (like mkexe, above) to allow users to make executables and shared libraries for use with rootbeer without having to fiddle about with makefiles. It doesn't do anything about dependencies, so if you remake a library, you also need to remake any executables which depend on it. The functions are as follows:

- `mkexe(char *filename, char *flags)`:
  Makes an executable with any filename suffix stripped off. Eg. `rbtest.C` gets made into `${ROOTBEER_BIN}/rbtest`.

The `flags` will be passed to the compiler (e.g. to inlcude `${ROOTBEER_SLIB}/libeloss.so`, use `"-leloss"`).

- `mkexe_deb(char *filename, char *flags)`:
  Makes an executable as above, but uses the `-g` flag instead of `-O`, to allows debugging.

- `mklib(char *filename)`:
  Similar to above, but makes a shared library.
  e.g. `mklib("sample_code/rbtest.C")` will make a shared library called `${ROOTBEER_SLIB}/librbtest.so`.
  This can be added to the rootbeer initialisation script `RootBeerSetup.cxx` to be loaded for subsequent use.

- `mklib_deb(char *filename)`:
  Makes a sharer library as above, but uses the `-g` flag instead of `-O`, to allow debugging.

Making rootDST files

There is a template code (`sample_code/dstmaker.C`). This takes a BOS file or a previously created rootDST file as input, and writes out the required banks to a new rootDST file based on any user defined cuts. The template is set up to analyse rbtestDST.root, using the TAGR, SCR and EPIC banks. It only writes events where there both TAGR and SCR had hits, or and EPIC bank was present. It drops the SCR bank from the output. It also fills a token histogram of hit multiplicity in the TAGR bank, just as an example. The code is well commented. To test this do:

```
>rootbeer
rootbeer [0] mkexe("sample_code/dstmaker.C,"")
rootbeer [1] .q
>rehash
>dstmaker rbtestDST.root test.root
```

To make a custom dstmaker, copy `sample_code/dstmaker.C` to `sample_code/mydstmaker.C`.
Open this an editor and search/replace all instances of “`dstmaker`” with “`mydstmaker`”.
There are then 3 parts which need to be customised:

1. Edit the line which begins “`char *mybanks[]={....}`”.
   This selects the banks in the input file which may be written out, or used to make cut decisions. (The getbanks utility can make a stab at

producing this line for you - type “getbanks” for the usage).
Note: The HEAD bank is always written to the DST, event if not explicitly listed in `mybanks`. If you really really don't want it you can force it out in `dropBanks()`.

2. Edit the `dropBanks()` function. This drops banks which are selected above and may be used to make cuts, but should not be written out to the rootDST.

3. Edit the `makeCuts()` function. This is where to make the decision on whether an event is written out to the rootDST. This can be simple (as in the example given), or can call some more sophisticated function defined in another library. It should return 0 if the event passes the cuts, and -1 if it fails.

4 Code reference

The use is best illustrated by example. The files `rbtest.C` and `dstmaker.C` in the `sample_code` directory are heavily commented for this purpose.

4.1 Environment and rootbeer.cshrc

To set up the environment, the environment variable `$ROOTBEER` should be defined and `rootbeer.cshrc` should be sourced; usually from `.cshrc` (see section 2). Bash users will need to create their own equivalent. The environment is set up in a similar way to the `$CLAS` environment, depending on the OS on the machine. For example, on a RH9 machine:

```
> printenv | grep ROOTBEER
```

gives this:

```
ROOTBEER_SCRIPTS=/home/kl/rootbeer2.1/scripts
ROOTBEER_OBJ=/home/kl/rootbeer2.1/obj/LinuxRH9
ROOTBEER=/home/kl/rootbeer2.1
ROOTBEER_SLIB=/home/kl/rootbeer2.1/slib/LinuxRH9
ROOTBEER_LIB=/home/kl/rootbeer2.1/lib/LinuxRH9
ROOTBEER_BIN=/home/kl/rootbeer2.1/bin/LinuxRH9
```

4.2 libRootBeer.so

All the main classes and function are contained in this library. The root generated html documentation can be generated by doing:

```
>make htmldoc
```

This puts the documentation tree in the `htmldoc` directory. Here's a description of the main classes and functions.

Class TRootBeer

The base class for RootBeer. This cannot be instantiated by itself, but is used to create derived classes to handle different types of input. Currently there are 2 input types: BOS files and rootDST files[1]. The next step will be to add an input for events from the ET system. The public functions are:

TRootBeer::TRootBeer - Class constructor
TRootBeer::~TRootBeer - Class destructor
TRootBeer::StartServer - Start to get events from the input source
TRootBeer::GetEvent - Get the next event from the server
TRootBeer::SetBankStatus - Set up banks which are to be served
TRootBeer::GetBankStatus - Get status of BANK (ie. is it being served)
TRootBeer::ListServedBanks - List the BANKs which will be served (if they're in the data file).

Class TBos

For handling input from BOS files. Inherits from **TRootBeer**. Has customised **constructor**, **destructor**, **StartServer()** and **GetEvent()** functions.

Class TDST

For handling input from rootDST files. Inherits from **TRootBeer**. Has customised **constructor**, **destructor**, **StartServer()** and **GetEvent()** functions.

Class TDSTWriter

For writing rootDST files. The public functions are:

TDSTWriter::TDSTWriter - Class constructor
TDSTWriter::~TDSTWriter - Class destructor
TDSTWriter::Init - Run before each input file
TDSTWriter::~DropBank - Drop bank from the DST file
TDSTWriter::GetTree - Get pointer to the root tree with the bank information

[1]In principle, someone could add more, like HIPO files for CLAS12, but we already have clas12root for that

TDSTWriter::~WriteTree - Drop bank from the DST file

4.3 Utility functions

These are functions which are not class member functions, but are built into `libRootBeer.so`. These are defined in `include/RootBeerUtil.h` and `src/RootBeerUtil.cxx`

TRootBeer *createBeerObject(char *filename) - Called to create the appropriate class for the input type (ie TBOS for a BOS file, TDST for a rootDST file).

int getFileMode(char *filearg) - Parses the file argument and decides whether it's a file or a list of files, and sets the mode appropriately.

int getNextFile(char *nextfile, char * filearg) - Get next file name from command line or filelist

int resetNextFile() - Can't honestly remember what the point of that is.

int getEpics(char *name, float *value) - Get the value of a variable in the EPIC bank.

int getBit(int word, int bit) - Get the status of a bit within a word (e.g. trigger bit) (0=off, 1=on)

4.4 Executables

Executables go in `$ROOTBEER_BIN`.

bankdump - a replacement for `bosdump`, which dumps the banks from either BOS or rootDST files. Usage is as follows:

```
>bankdump [-Nevents] [-Gbank1 -Gbank2 ...]  <dstfile|bosfile>
```

getbanks - This is a script which runs bankdump and pipes the output of bankdump into an awk command to generate a list of BANKS which are in the input file. It's useful as a quick way of finding out which BANKs are in a file. Also, for BOS files, it is much quicker to handle only a selection of banks rather consider all those which might be in the system (currently about 200 different bank types). For example;

```
> getbanks -N1000 cooked_048285.A06.B00
```
[2]

Produces the following output:

[2]This is a real data file from the g8b run period

```
file = cooked_048285.A06.B00
Sorting file - cooked_048285.A06.B00
done 1000
Sorted 1000 events from file:  cooked_048285.A06.B00
char *mybanks[]={"MVRT","TGPB","VERT","DCPB","TGTL","ST1 ",
"ECT ","HDPL","HBTR","ECPB","TRPB","HBID","TAGR","TAGE","ECPO",
"SCPB","PART","STN0","SCR ","RFT ","ECPC","STN1","EC ",
"TBER","TGBI","SCT ","SC ","MTRK","TGTR","TAGI",
"SCRC","TRL1","TDPL","TBTR","STR ","HEVT","ECHB",
"DSTC","CL01","TBID","EVNT","SC1 ","HEAD","ECPI","STPB","null"};
```

The list of banks (`char *mybanks[]`) can be pasted into a macro or executable to deal with that file.

4.5 Makefile

In the current version, a simple `>make` command will build `${ROOTBEER_SLIB}/libRootBeer.so`, `${ROOTBEER_BIN}/bankdump`, and `${ROOTBEER_BIN}/getbanks`.

Other basic makefile options are:

`>make htmldoc` - Makes the root html documentation for classes in `libRootBeer.so`
`>make clean` - Well have a guess.
`>make tar` - Makes a tar file for distributing.

For simple root based analysis code it is recommended to use the `mkexe` utility within rootbeer (see following section). For more complicated things there are examples in the Makefile:

`>make PolHandler` - An example of a class to handle data from linearly polarised photon experiments. Rather messy code.
`>make PolHandler_clean` - Cleans this.

Some CLAS packages have been checked out and put in the `extra_packages` directory. Wrappers have been constructed to allow them to be made into shared libraries. These are packages which are more or less independent of the rest of the CLAS software trees. Headers for them need to go into `${ROOTBEER}/include`.
Object files go into `$ROOTBEER/extra_packages/<package_name>`. The resulting shared libraries go in `${ROOTBEER_SLIB}`.

`>make eloss` - Makes a shared library (`${ROOTBEER_SLIB}/libeloss.so`) from Eugene Pasyuk's eloss package [3].
`>make eloss_clean` - Cleans it

`>make ExpTable` - Makes a shared library (`${ROOTBEER_SLIB}/libExpTable.so`) from the ExpTable package for reading run tables. See, e.g. the g9/FROST wiki for an example [4]. Builds automatically in Rootbeer6.0
`>make ExpTable_clean` - Cleans it

`>make g10pcor` - Makes a shared library (`${ROOTBEER_SLIB}/libeloss.so`) from the g10 momuntum correction package.
`>make g10pcor_clean` - Cleans it

`>make g13pcor` - Makes a shared library (`${ROOTBEER_SLIB}/libeloss.so`) from the g13 momuntum correction package (Rootbeer6.0 only).
`>make g13pcor_clean` - Cleans it

`>make c_bos_io` - Makes a shared library (`${ROOTBEER_SLIB}/libc_bos_io.so`) from the CLAS c_bos_io package. This makes wrappers for the c_bos_io functions and allows other CLAS packages which depend on c_bos_io to be compiled if they are linked with `libc_bos_io.so`. This gets rather far away from the rootbeer objective of being more or less independent of the CLAS software tree. Not very rigorously tested.
`>make c_bos_io_clean` - Cleans it

4.6 RootBeer utilities

The aim is to be able to write, test and debug user code as root macros and then compile to make executables or shared libraries. Within RootBeer there is a scheme to do the making of libraries and executables without having to keep adding things to the makefile. Here's how to use this:

1. Write my code as macros in the `sample_code` directory.
   (a) Begin by copying a template (or a recently created piece of code).
   (b) eg. `>cp sample_code/rbtest.C sample_code/myana.C`
   (c) Open in my favoutite editor: `>emacs sample_code/myana.C`
   (d) Search replace all instances of `rbtest` with `myana`
   (e) Edit the code to do the analysis I want

2. Start rootbeer in a separate terminal

   (a) `>rootbeer`

3. Load the macro and test the code

   (a) `rootbeer [1] .L myana.C`

   (b) call the function for a small no of events (say 5000), with dummy outfile

   (c) `rootbeer [2] myana(5000,"cooked_048285.A06.B00","")`

   (d) See if there are any errors. Check and histograms etc. (Note: If it runs but produces subtle errors then I make an executable with the debugging flag (see below), and run the code in a proper debugger - like gdb using the rather good ddd front-end).

4. Modify the code in the editor, reload the code and go through step 3 again.

   (a) Repeat this until I'm happy

5. Make an executable (or library) from the macro. This runs much faster. However, for "quickies", it might not be worth bothering with this step.

   (a) `rootbeer [15] mkexe("sample_code/myana.C","")`

   (b) examine the output for errors, perhaps a root header missing or something that works in a macro, but the compiler doesn't like.

   (c) modify the code until it compiles.

6. Test the executable version.

   (a) `>rehash`

   (b) `myana -N5000 cooked_048285.A06.B00 test.root`

The rootbeer functions for making executables and libraries are loaded at startup from `sample_code/MacroMaker.C`. These are as follows:

mkexe(char *filename, char *extraflags) - makes the executable in `${ROOTBEER_BIN}`

mkexe_deb(char *filename, char *extraflags) - make the executable in `${ROOTBEER_BIN}` with debugging flag.

mklib(char *filename, char *extraflags) - makes the shared library in `${ROOTBEER_SLIB}`

mklib(char *filename, char *extraflags) - makes the shared library in `${ROOTBEER_SLIB}` with debugging flag

The "extraflags" argument is used to pass any flags to the compiler. E.g. to make an executable which uses eloss, we need to link with eloss library like this:

```
rootbeer [15] mkexe("sample_code/myana.C","-leloss")
```

5 How it works - briefly

RootBeer was designed with two basic aims:

1. *To allow the simplest possible handling of BANKs within a root framework.*
   In particular, for users with little C++ experience, there should be no need to call C++ functions to get BANK data. Hence, all bank member data can be referred to as BANK[row].member, or BANK[sector][row].member.

2. *To be as independent as possible from the CLAS software packages.*
   To allow users to take rootDST files to their home site and perform data analysis without having to install the CLAS packages, calibration database etc.

RootBeer does not do any physics; that was not the aim. Rather, it allows easy access to BANK data, allows Bos files to be written out as rootDSTs, and code written within the rootbeer scheme can run on either bos or rootDST files. It is anticipated that rootDSTs will be mainly produced during cooking, after calibations have been finalised.

5.1 BANKs as global structures

The layout of BOS files is such that BANKs map easily to C structures. For example here is the struct for the TAGE bank:

```
typedef struct TAGE_t {
    unsigned short ID;
    unsigned short TDC;
} TAGE_t;
```

If we read data from a BOS file to memory and make a `TAGE_t` variable point to the start of the TAGE bank data like this:

```
TAGE_t *TAGE = start_of_TAGE_data;
```

We can refer to the member data for the rows in the BANK as follows

```
id = TAGE[row].ID;
```

... etc

Some other CLAS utilities do this already. Rootbeer however, does it implicitly. This means that all BANK variables are global, and with each call the to the `TRootBeer::GetEvent()` function are automatically made to point to the bank data for that event (some technical details are given below). There is minimal movement of data in memory. Large chunks of the Bos files are read in; the record, segment and data headers are scanned, and BANK structures are made to point to the correct memory addresses.

The same principle is employed for rootDST files. All banks are written into a tree as simple arrays of shorts. When rootbeer reads a DST file, it just reads these back as arrays of shorts and makes the BANK struct variables point to them as appropriate. Hence, there are no classes, streamer functions or Get() methods for BANK data. It's just always there in the global variables. Anything more would be overkill.

5.2 Some code description

Most of the source code is well commented. Anyone interested in the details should look there.

5.2.1 Code generation

Most of the code relating to banks is generated by the running the script `$ROOTBEER_SCRIPTS/bank2struct.gk` on `clasbanks.ddl`

It produces the following files:

- `bankheader.h` - Definitions of structures relating to all the banks, and a `#define BANK_ID` line to number each bank.
- `bankvars.cxx` - Defines global variables and a `BANK_Print()` function for each bank, also creates an array of `addressBanks_t` structures which contains the names and addresses of the BANK variables and

print functions. It is through the information in this array that the variables are made to point to the correct data from the input file (Bos or rootDST).

- `bankvars.h` - The file to be included by any code which needs access to the global variables defined in `bankvars.cxx`
- `banvarsLinkDef.h` - The link def file which allows the variables defined in `bankvars.cxx` to be accessed through the ROOT CINT interpreter, and therefore enables rootbeer code to run as macros.

5.2.2 TRootBeer

A base class. Not to be instantiated on it's own. Class constructor and destructors and member functions `StartServer()`, `GetEvent()` need to be customised. Functions in the base class deal with setting up the BANKs which are to be accessed, by saving pointers to the appropriate structure with names and addresses (`addressBanks_t` structs)

5.2.3 TBOS

The class which handles data from Bos files. Two threaded processes deal with the Bos input file:

- `DataServer()` - This reads large chunks (currently 2.5MB) in binary format from the Bos file into a pair of buffers. When a buffer is freed (ie the all events in it have been analysed by the user), it refilled from the Bos file stream.
- `BankScanner()` - This looks at the data buffers filled by the `DataServer()`. When a buffer is flagged as filled it scans through the record, record-segment and data-segment headers to find the pointers to the data from the BANKs which the user requires. It fills a ring buffer (of `eventInf_t` structs) with the relevant numbers and addresses.

The `GetEvent()` function takes the next event from the ring buffer (filled by `BankScanner()`) and makes the global BANK variables point to the correct addresses for the event.

5.2.4 TDST

This is the class which handles data from rootDST files. The data is saved in a root Tree (bankTree). All the member data from the banks is saved as arrays of shorts, whose length depends on the size of the BANK and the number of rows in that BANK for the event (see DSTWriter).

The `GetEvent()` function uses the `SetBranchAddress()` function to read all the BANK information contiguously into a large memory buffer and makes the global BANK variables point to the correct addresses in that buffer.

5.2.5 TDSTWriter.

This is the class which makes the rootDST. It creates TTree object, and for each BANK creates several branches; some for number of rows, sector etc, and a main branch for the actual data within the BANK. For each BANK, all the rows are made contiguous in memory (this means moving some data in memory if dealing with multi-sector banks in a BOS file). The bank data is written in a branch as an array of shorts whose length depends on the size of the BANK and the number of rows in that BANK for the event.

6 Why doesn't it work?

In general, in order of increasing likelihood:

1. The computer is screwed. Very unlikely, but something users like to say rather than blaming themselves.

2. The package you're using (RootBeer in this case) has a bug. Possibly. Please report if you think so, but please consider option 3 carefully first.

3. The user code is screwed. "...it worked yesterday and I haven't changed anything...". The most common whinge of users[3]. But, it is seldom true! Usually you have changed something! There are many reasons why user code doesn't work - and usually the user needs to figure it out.

4. You haven't "switched on" the required banks. This is the most common problem I have found with my own code. The "bank server" part of the code only takes banks from the input file if their status is set to ON. This gives a big increase in efficiency when handling data from either BOS or rootDST files. If you forget to switch a bank on, it will always be seen as having 0 hits. Doing it any other way makes the user code much more cumbersome.

[3]See also "we've tried nothing and we're all out of ideas"

7 Acknowledgements

The authors acknowledge the contributions to the Rootbeer project over the years of various CLAS collaborators and generations of PhD students. Particular thanks to Nick Zachariou at the University of York for providing the archived version of rootbeer2.1 for use in the CLAS6 container, Derek Glazier at the University of Glasgow for his contributions to porting rootbeer to its current root v6 compatible iteration, and Franz Klein at the University of Maryland for ongoing support of rootbeer during the CLAS6 era.

References

[1] D. Cords, L. Dennis, D. Heddle, and B. Niczyporuk. CLAS-note 94-012: CLAS Event Format with BOS. Technical report, 1994. URL: `http://www.jlab.org/Hall-B/notes/clas_notes94/note94-012.ps.gz`.

[2] R. Brun and F. Rademakers. ROOT - An Object-Oriented Data Analysis Framework. URL: `https://root.cern`.

[3] E. Pasyuk. CLAS-note 2007-016: Energy loss corrections for charged particles in CLAS. Technical report, Arizona State University, 2007. URL: `https://misportal.jlab.org/ul/Physics/Hall-B/clas/viewFile.cfm/2007-016.pdf?documentId=423`.

[4] CLAS Collaboration. *g9/FROST Wiki*. URL: `https://clasweb.jlab.org/rungroups/g9/wiki/index.php/Rootbeer,_including_eloss_and_ExpTable`.