BEE 0-95

Thorsten Glebe
Max-Planck-Institut für Kernphysik, Heidelberg
T.Glebe@mpi-hd.mpg.de

September 14, 2001

BEE is a ROOT [1] based analysis framework for data analysis in the HERA-B experiment. It is completly independent from ARTE and other HERA-B software and can thus be easily installed on any computer.

Contents

1 Introduction

2 Installation and Compilation

In the following the necessary steps to install and compile BEE are explained.

2.1 External software

In order to install BEE, you need a few external software packages installed:

1. `ROOT`, any version above or equal to 2.25.03, see [1];
2. `libmathlib.a`, `libkernlib.a` from the CERN program library, see [3], recommended is a full installation of the CERNLIBs;
3. `gcc` and `g++`, recommended at least version 2.95.3.

2.2 Environment variables

Before compilation, make sure you have set the following environment variables properly:

ROOTSYS variable pointing to the ROOT installation, e.g.:

```
bash> export ROOTSYS=/opt/root
```

CERN_LIB variable pointing to the CERN libraries, e.g.:

```
bash> export CERN_LIB=/cern/pro/lib
```

HBBINTYPE variable defining the binary type of the machine

All variables are automatically set if you are sourcing the `hbprofile.sh` or `hbprofile.csh` script from the HERA-B `GLOBAL` project [4].

2.3 BEE distribution file

To get a tarball of the BEE project, look at the BEE homepage [5]. The tar file is named like

```
BEE-<version>.src.tar.gz
```

To unpack, just type

```
bash> gtar tvzf <BEE tar file>
```

The code will be unpacked in a subdirectory of the current directory with the name

```
BEE-<version>
```

No binary distribution is provided. You have to compile the code on the target machine.

In the BEE directory you will find several subdirectories, for each BEE project at least one. An `include` directory will be created during compilation, which will point to the project include directories via symbolic links. This allows to specify a single path for all BEE include files. Example:

```
g++ -I<path-to-BEE>/BEE-<version>/include myfile.C -o myfile.o
```

In case you want to include a header file from the BEE project, you have to specify the BEE project in the `#include` statement. Example:

```
// This is myfile.C
...
// include RecoTrack.hh from BEE project CLUE:
#include "clue/RecoTrack.hh"
```

2.4 Make BEE

To compile the BEE project you should use a computer with the following properties (recommended):

1. 128 MB RAM or more,
2. 466 MHz CPU or better,
3. one or more processors.

To build BEE, just type

```
bash> make
```

this will build the BEE library.

In case you want to compile on a multiprocessor machine, you can specify the number of CPUs to use on the command line:

```
bash> # compilation on a 4 processor machine:
bash> make PROC=4
```

Per default, `PROC` is 2.

All object files and libraries will end up in a directory called `$HBBINTYPE/lib` which is located in the `BEE-<version>` directory.

Further make targets are:

slib build the shared libraries of the projects,

alib build the static libraries of the projects,

clean remove the `$HBBINTYPE` and `include` directory,

all build all static and shared libraries,

remake build all shared project libraries plus `libBEE-<version>.so`,

docu build all documentation files (needs `doc++` and LaTeX),

cleandoc remove all documentation files,

debug build all shared libraries compiled with `-g -ggdb`,

tar generate a tar file of all sources in the directory `../export`,

check print out settings of makefile variables.

soexe build an executable linked with shared BEE library.

saexe build a statically linked executable.

run run an executable.

rungdb run an executable in `gdb`.

In case you want to compile only a single project, change directory to the project directory and type `make`. Most of the above mentioned targets are there available, too.

In the BEE toplevel directory, several Makefiles are available:

Makefile main makefile, build the whole BEE project.

Makefile.setup contains all variables which are needed during compilation.

Makefile.targets contains all targets needed to compile a project.

Makefile.BEElib used to build `libBEE-<version>.so` from all projects.

After a successful compilation, a library is created for each project which is listed in the `projects` variable in `Makefile.setup`. The version numbers of the projects are part of the library name. Example: for CLUE version 0-34 the library is

```
<path-to-BEE>/BEE-<version>/$HBBINTYPE/lib/libclue-0-34.so
```

In order to ease the use of the BEE libraries, a single library is build which contains all object files from the different projects. Thus you only have to link to a single library. Example: in case of BEE version 0-43, compiled on a glibc-Linux, the library is

```
<path-to-BEE>/BEE-0-43/Linux_intel_glibc/lib/libBEE-0-43.so
```

To make life even more simple, a symbolic link to the library is created:

```
libBEE.so -> libBEE-0-43.so
```

Linking your code with BEE becomes as simple like:

```
g++ -shared myfile.o -L<path-to-BEE>/.../lib -lBEE
    -o myfile
```

2.5 Create your own project

In case the user wants to create his own BEE project, choose a unique name for your project (which is not already used in BEE) and execute the `makeproject.sh` script:

```
bash> BEEROOT=$HBROOT/BEE/BEE-<version>
bash> $BEEROOT/makeproject.sh myproject
```

The `makeproject.sh` script creates a directory structure in the current directory like this:

```
myproject/Makefile
myproject/cintdict/LinkDef.hh
myproject/doc/Makefile
myproject/doc/myproject.dxx
myproject/doc/myproject.tex
myproject/include/
```

Some file templates are provided:

Makefile The project makefile. Can be used without modifications.

cintdict/LinkDef.hh ROOTCINT class declaration file, see [2].

doc/Makefile The documentation makefile. It needs a working `doc++` installation. In case the user wants to use another documentation system (e.g. `docify` or `doxygen`) the makefile must be changed accordingly. In case of `doc++`, the makefile can be used without modifications.

doc/myproject.dxx DOC++ configuration file. Here one has to insert the project header files which should be parsed by `doc++`.

doc/myproject.tex LaTeX template for a more descriptive documentation. The automatic generated documentation is in most cases more a reference manual than a full documentation.

Note: Writing software and writing documentation for software should be considered as one process, not as two independent ones. Experience shows that in many cases software documentation is considered as the second step, and sometimes the second step is never made.

Once the `makeproject.sh` script is executed, the user can create souce code in the project toplevel directory an place the header files in the `include` directory. To generate a shared library out of his code, a simple

```
bash> make
```

is sufficient. The shared library can be found in the projects `$HBBINTYPE/lib` subdirectory. When root is started, it can be loaded with the `gSystem->Load()` command:

```
gSystem->Load("libmyproject.so");
```

2.6 Create your own Analysis class

Once the user has created his own BEE project (see section 2.5), the user can start to write his own analysis code. In order to support this, the user can generate a BEE Analysis class by executing the `makeanalysis.sh` script:

```
bash> $BEEROOT/makeanalysis.sh <class name>
```

The following files will be generated in the current directory:

```
main.C
<class name>.C
include/<class name>.hh
```

The `main.C` provides a lot of functionalities and works immediatly with the generated class:

```
make soexe
<...compilation output...>
```

In the generated class you will find a lot of dummy routines where you can fill in your appropriate code:

EventSelect() : routine for event selection conditions. If routine returns `false` for a given event, the event is skipped for the analysis.

DefinePattern() : place where track and decay pattern should be defined.

DefineHistograms() : place where ROOT/PAW histograms/ntuples can be defined.

Event() : the routine where all the work is done. This routine is called once per accepted event.

EventCleanup() : called once after the Event() call. Here one can place some cleanup code.

End() : called once at the end of the job. Place to put some final calculations, or cleanup code.

For more examples on analysis classes, see the files in the `doc/examples` directory of your current BEE version.

2.7 Build a BEE executable

In order to run a batch job, two possibilities exist:

1. Use the `root -b <macro>` command to run ROOT in batch mode executing a script file (see ROOT manual).
2. Compile a root executable using a `main()` routine.

The first method is trivial. To simplify the second method (which has the advantage that no interpreter is running) BEE provides some special Makefile targets.

Note: it is recommended to use the `makeanalysis.sh` script to produce a correct `main()` routine together with an analysis class (see section 2.6).

In case you want to write your own main routine, see the following code as an example:

```
// This is file main.C
#include <iostream>
#include "TROOT.h"
#include "TChain.h"

#include "BEE.hh"

int main(void) {

  // initialize ROOT, not needed in versions > 3.00
  TROOT myexe("myexe","A standalone ROOT application");

  // set up BEE event branch
  RtEvent* event = new RtEvent();
  TChain* f = new TChain("Clue");
  f->SetBranchAddress("event",&event);

  // load BEE files
  f->Add("<path to data>/<BEE file>.root");

  // Event loop
  Int_t nentries = f->GetEntries();
  for(int i = 0; i<nentries; ++i) {
    f->GetEntry(i); // load event

    // do something in the event loop

  } // for i
  return 0;
}
```

Note: It is important that the `main()` routine does not end up in a user library as this leads to problems when the library is used. Therefore the `main()` routine should be kept in a separate file. This file must be specified in the Makefile variable `mainsrc`:

```
// BEE user Makefile
```

```
...
##############################################
# .C file defining main() (does not end up
# in user library)
##############################################
mainsrc    = main.C
...
```

The name of the binary which will be produced can be specified in the `binary` variable (default is `a.out`):

```
##############################################
# name of binary
##############################################
binary     = myexe
```

The binary can be compiled in two ways:

make soexe compile the executable using the shared BEE library.

make saexe compile executable using the static BEE library.

Note: `make saexe` will not build a full static executable as the shared ROOT libraries are used for linkage.

To run the executable it is convenient to use the `make run` target which will set the `LD_LIBRARY_PATH` appropriate.

2.8 Command Line Options

Once you have compiled your analysis class which was generated by the `makeanalysis.sh` script (see section 2.6), several command line options are available for you. The command-line help can be printed by executing the binary with the option `-?`:

```
  bash> ./Linux_intel_glibc/bin/a.out -?

Usage: a.out [-n <events>] [-s <offset>] [-e event] [-o output file]
             [-u] files...
 <events>    : number of events to process (default: all)
 <offset>    : event number to start with (default: 0)
 files       : BEE files to process (wildcards are allowed)
 output file: ROOT histogram output file (default: first_analysis.root)
-u           : open histogram output file in UPDATE mode (default: recreate)
 event       : experiment event number of an event under investigation
Example:
   a.out -n 5000 /scratch/BEE/*.root -o out.root
```

2.9 Profiling BEE

To profile the code which is contained in the BEE library, compile a BEE library which can be analyzed by a profiler:

```
bash> make PROF=1
```

This will recompile the BEE sources using the `-pg` option. The output are static libraries with the suffix `_pg`, e.g. `libBEE_pg.a`.

In a second step, compile a statically linked BEE executable (see section 2.7) with the `PROF` variable defined:

```
bash> cd myproject
bash> make saexe PROF=1
```

This executable will be linked with `libBEE_pg.a`. Now run the executable:

```
bash> make run PROF=1
```

A file `gmon.out` will be produced which can be analyzed by a profiler like `gprof` (see `gprof` man pages).

Running `gprof` will give you now a detailed call graph of the executed code:

```
bash> gprof $HBBINTYPE/bin/myexe gmon.out | less
```

3 ROOT

Although BEE can run without ROOT, the main aspect of BEE is certainly the work within the ROOT framework. In the following, some aspects concerning ROOT are explained.

3.1 Setting up the ROOT environment

In order to use the BEE library in ROOT, you can add the BEE library path to the ROOT `DynamicPath` variable in the `.rootrc` file:

```
# ROOT Environment settings are handled via the class TEnv. To see
# which values are active do: gEnv->Print().

# Path used by dynamic loader to find shared libraries and macros
# Paths are different for Unix and Windows. The example shows the
# defaults for all ROOT applications for either Unix or Windows.
Unix.*.Root.DynamicPath:    .:$(HBROOT)/BEE/pro/$(HBBINTYPE)/lib\
                             :$(ROOTSYS)/lib
Unix.*.Root.MacroPath:      .:$(ROOTSYS)/macros:$(HOME)/rmacros
...
```

In order to load the BEE library automatically when you start ROOT, specify a `rootlogon.cxx` script file in `.rootrc`:

```
# Rint (interactive ROOT executable) specific alias, logon and
# logoff macros
Rint.Load:                 rootalias.cxx
Rint.Logon:                $(HOME)/rmacros/rootlogon.cxx
Rint.Logoff:               rootlogoff.cxx
```

In the `rootlogon.cxx` file, you can now load the BEE library:

```
{
  // this is rootlogon.cxx
  gSystem->Load("libBEE.so");
}
```

To set up the BEE event structure, the some more code has to be added to `rootlogon.cxx` after the BEE library is loaded. The complete `rootlogon.cxx` will look like this:

```
{
  // this is rootlogon.cxx
  gSystem->Load("libBEE.so");

  // set up BEE event structure
  RtEvent* event = new RtEvent();

  TChain* f = new TChain("Clue");
  cout << "Set up event branch" << endl;
  f->SetBranchAddress("event",&event);
}
```

Thats it. Now you can start ROOT together with the BEE data analysis framework.

3.2 Writing ROOT files from ARTE

In order to convert HERA-B dst or mini files to ROOT files which can be used by the BEE framework, the `cluearte` project had been created in ARTE. Within this document we will call the ROOT files produced by `cluearte` "BEE files".

In order to write a BEE file, the following steps have to be done:

1. In the routine `usevnt_()` (located in file `usevnt.C`), uncomment the call to `clueWriteEvent()`.
2. Compile your ARTE executable.

3. In your kumac, set the filename of the BEE file via the `CLUE/SETFILE` steering card. The default name is `filename.root`.
4. Now run your ARTE job.

An ARTE kumac file which reads a mini file and produces a BEE file as output is shown in the following:

```
* ------------------------------------------------------------
*  CLUE output
* ------------------------------------------------------------
CLUE/SETFILE        rp17127_001.root

* ------------------------------------------------------------
*  mini-input-file
* ------------------------------------------------------------

IOFILE EV_IN /hb/.../run17127/rp0001_run17127_file001.mini -gi

ENDKUMAC
```

A full list of kumac steering cards for `cluearte` are given in the following:

SetFileName Value: [name]. Define filename for BEE file. Example:

```
CLUE/SETFILENAME rp17127.root
```

StoreRSEGflag Value: [1|0]. Write RSEG contents into ROOT file if set to 1. Example:

```
CLUE/STORERSEGFLAG  1
```

SetBranchStatus Value: [`bname`]. `bname` is the name of a branch. If `bname`=$*$, apply to all branches. If set to 1, branch will be processed, if set to 0 branch will not be processed. Example:

```
CLUE/SETBRANCHSTATUS Clue 1
```

This card can be ignored by the user.

SetCompressionLevel Values: [1...9]. Sets the gzip compression level for the BEE file. Default is 2. Example:

```
CLUE/SETCOMPRESSIONLEVEL 3
```

3.3 Reading BEE files in ROOT

In case the ROOT environment is set up as described in section 3.1, the user can simply add one or more BEE files to the `Clue` chain:

```
root[0] f->Add("rp17127_001.root");
root[1] f->Add("rp17127_002.root");
root[2] f->Add("rp17127_003.root");
...
```

Via the chain pointer `f`, all files are treated like a single, big file. In order to get the first event, type

```
root[3] f->GetEntry(0);
```

3.4 How to fill a BEE event

In order to write a BEE file from an ARTE MINI/DST file, one has to uncomment the call to `clueWriteEvent()` in the `usevnt.C` file. The `clueWriteEvent()` routine fills a BEE event which is then stored to a ROOT file. A BEE event contains only a part of the information stored in an ARTE event. At the moment, the following ARTE tables are stored in a BEE event:

EVHD Event header

GTAR Target wires

MTRA Monte Carlo tracks

MVER Monte Carlo vertices

RTRA Reconstructed tracks

RVER Reconstructed vertices

RSEG Reconstructed track segments (optional)

TOPP Particle data

The way how ARTE tables are stored to a BEE file is defined via the `TableFillBase` interface. The `clueWriteEvent()` routine needs a class inherited from the abstract `TableFillBase` class which defines how to store ARTE tables in BEE files. The user can define his own filling routines and is thus able to introduce track selection, clone removal or any other kind of track selection.

The `TableFillBase` class declares the following filling routine which must be defined in any derived user class:

EventSelect() Routine which returns `true` in case the event should be accepted.

Fill() Routine which is called in `clueWriteEvent()`. This routine does the filling or calls other filling routines.

FillRSEG() Routine which fills the `VDS_Track`, `OTR_SpacePoint`, `RICH_Track` and `MuonTrack` tables from the ARTE RSEG table in case the `STORERSEGFLAG` data card was specified (see section 3.2).

Note: The user cannot define the way how information is stored from EVHD and TOPP.

A default filling method is supplied by the `TableFill` class. `TableFill` has the following additional member functions which are called from the mandatory `Fill()` method:

FillWire() Routine to fill the `Wire` table from GTAR.

FillRecoTrack() Routine to fill the `RecoTrack` table from RTRA.

FillRecoVertex() Routine to fill the `RecoVertex` table from RVER.

FillMCtrack() Routine to fill the `MC_Track` table from MTRA.

FillMCvertex() Routine to fill the `MC_Vertex` table from MVER.

FillMisc() Routine to fill other tables, e.g. `EcalHit` from RCCL.

To use this filling method, create a static `TableFill` instance an specify it as an argument to `clueWriteEvent()`:

```
// this is usevnt.C
...
#include "cluearte/cluearte.hh"
#include "cluearte/TableFill.hh"
...
extern "C" void usevnt_()
{
...
  // *********************************************************
  // Step 5: Write CLUE output file
  static TableFill fillit;
  clueWriteEvent(fillit);
  // *********************************************************

}
```

To define your own filling method, inherit a class from TableFillBase[1] and overload the `Fill()`, `EventSelect()` and `FillRSEG()` method[2]:

```
// this is MyFill.hh
#include "cluearte/TableFillBase.hh"
```

[1] See the implementation of the `TableFill` class in the ARTE `cluearte` project for a complete example.

[2] `clueWriteEvent()` makes no further assuptions on the derived classes, therefore it is unlikely that the user violates the Liskov substitution principle (see [7] and [8]).

```
class MyFill : public TableFillBase {
  bool EventSelect() const {
    return true; // accept all events
  }
  bool Fill() const {
    // My way to fill BEE tables
    ...
  }
  bool FillRSEG() const {
    // My way to fill from RSEG
    ...
  }
}
```

In the `MyFill` class you can specify your own filling algorithms. To make use of your own filling method, replace `TableFill` by `MyFill` in `usevnt.C`:

```
// this is usevnt.C
...
#include "cluearte/cluearte.hh"
#include "MyFill.hh"
...
extern "C" void usevnt_()
{
...
  // **********************************************************
  // Step 5: Write CLUE output file
  static MyFill fill_obj;
  clueWriteEvent(fill_obj);
  // **********************************************************
}
```

Note: The `TableFill` class can also be used as a base class for a user defined class. This is useful in case you want to replace only a a part of the `TableFill` methods.

3.5 ROOT patches

Sometimes ROOT a version needs to be patched in order to run properly with BEE. The file `patches.txt` which is part of the BEE `doc` project describes the known patches and the ROOT versions they apply to.

References

[1] Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne,

Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch/.

[2] Rene Brun, Fons Rademakers, Using ROOTCINT, http://root.cern.ch/root/RootCintMan.html

[3] See http://...

[4] http://www.mpi-hd.mpg.de/herab/GLOBAL

[5] http://www.mpi-hd.mpg.de/herab/clue

[6] T. Glebe, GRelation1.12, HERA-B Note 00-188, December 2000

[7] R. C. Martin, The Liskov Substitution Principle, C++ Report, March 1996

[8] B. Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices, 23,5, May 1988