BEE — Compilation and Environment

Version 0-102

HERA-B Note 02-001, Software 02-001

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

January 22, 2002

Abstract

BEE is a data analysis framework for high energy physics experiments. The current note describes the compilation and environment settings needed to generate the BEE library and BEE based analysis executables.

Contents

1 Introduction

The BEE data analysis framework has been developed under the Linux operating system and consists of a set of almost independent software projects which are compiled into a library. Analysis jobs can be build from this library by the user and run e.g. in batch mode (see section 3).

The shared BEE library can be loaded as a ROOT [1] application into the ROOT framework and lots of it's classes can be used interactively in the ROOTCINT C++ interpreter (see section 4). This gives an easy access to the data stored in BEE-files and allows a rapid code development.

BEE is distributed as source code, therefore the current note will provide the neccessary information to compile and link the library.

2 Installation and Compilation

In order to install the BEE library it is recommended to obtain a source code distribution from the BEE homepage [2]. No binary distribution is provided because system dependencies are difficult to control. You have to compile the code on your target machine (see section 2.5). Compilation is managed by a set of scripts and Makefiles and should be straightforward, as BEE is implemented in standard C++ and does not use system specific code.

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

2.1 External software

In order to compile BEE a few external software packages need to be available on the user's computer system:

1. ROOT, any version above or equal to 3.00, see `http://root.cern.ch`.
2. `libmathlib.a`, `libkernlib.a` from the CERN program library [3]. The CERNLIB is only needed for the Vt++ project [4].
3. `gcc` and `g++`, recommended at least version 2.95.2.
4. A standard STL installation [5] which is in most cases part of the compiler distribution. If not, download a recent STL from: `http://www.sgi.com/tech/stl`.
5. The `bash` shell, installed in `/bin/bash`. This shell is used by the BEE scripts. A recent version of `bash` can be obtained from the homepage: `http://www.bash.de`.
6. The GNU `make` utility. In case `make` is not available on your system, get it from `http://www.gnu.org`.

The following packages are needed only for special purposes:

1. `gdb` for debugging,
2. `gprof` for code profiling
3. In order to generate the documentation, an installation of LaTeX and DOC++ is needed. LaTeX is available for almost every system, DOC++ can be downloaded from it's homepage: `http://www.zib.de/Visual/software/doc++`.

2.2 System requirements

The compilation of the BEE library puts some demands on the the computer hardware, as the generation of the ROOT dictionaries (in the gate and clue project [8]) as well as the processing of the pattern expression templates in the smatrix project [9] is cpu-time and memory intensive.

To compile the BEE project you should use a computer with at least the following properties:

1. 128 MB RAM or more (to avoid swapping),
2. 466 MHz CPU or better,
3. one or more processors.

Compilation of BEE on a Dual 466 MHz Intel Celeron Machine with 256 MB RAM usign both CPU's takes about 15 minutes. During compilation up to 110 MB RAM is used.

2.3 Environment variables

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

ROOTSYS: variable pointing to the ROOT installation, e.g.:
`bash> export ROOTSYS=/opt/root`

CERN_LIB: variable pointing to the CERN directory where the CERN libraries are located, e.g.:
`bash> export CERN_LIB=/cern/pro/lib`

HBBINTYPE: variable defining the binary type of the machine (can be set to any system specific name), e.g.:
`bash> export HBBINTYPE=Linux_intel_glibc`

In case one of these variables is not known in the shell environment, the BEE Makefile will complain.

Note: All variables are automatically set in the proper way if you are sourcing the hbprofile.sh or hbprofile.csh script from the HERA-B GLOBAL project [6] in your shell initialization file (e.g. .bashrc or .cshrc):

```
# My local Shell initialization file (.bashrc)
source <path to GLOBAL project>/GLOBAL/pro/hbprofile.sh
```

2.4 BEE distribution file

To get a tarball of the BEE project, look at the BEE homepage [2] (see table of "Official BEE releases"). Here you will also find the appropriate cluearte code (see table "Official cluearte releases"), which you need in case you want to write BEE-files from an ARTE [7] job.

The BEE tar file is named in the following way:

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

To unpack, use the GNU tar utility and type

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

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

```
BEE-<version>
```

In the BEE directory you will find several subdirectories, for each BEE project at least one. A directory listing will look like in figure 1 (red color shows executables, blue color shows directories, black color shows normal files). From the directory listing one can see that BEE consists currently

```
BEEIni.C             Makefile.setup      grelation
BEE_command.sh       Makefile.targets    grover
BEE_tar_src.sh       PACKAGES            include
BEEinstall           README              interfaces
BEEmakelib           arte                makeanalysis.sh
CVStag.BEE           cern                makeproject.sh
ChangeLog            clue                pattern
ERRORS               doc                 smatrix
Linux_intel_glibc    gate                util.sh
Makefile             gcombiner           version.sh
Makefile.BEElib      geometry            vt++
```

Figure 1: Directory structure of a BEE installation.

of 13 projects: arte, cern, clue, doc, gate, gcombiner, geometry, grelation, grover, interfaces, pattern, smatrix and vt++.

An `include` directory will be created at the top level of the BEE installation during compilation. The directory will contain symbolic links to the `include` directories of the individual projects. Thus the header files of the different projects are found in subdirectories of the top level include directory: `include/arte`, `include/cern`, `include/clue`...

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 the user wants to include a BEE header file, he has to specify the include file together with the path in the `include` directory, for example:

```
// include RecoTrack.hh from BEE project clue
#include "clue/RecoTrack.hh"
```

The advantage is that from the include statement one can directly see to which project the included header file belongs.

Note: Inclusion of BEE header files may cause problems if the order of the `#include` statements is not correct.

A library header file `BEE.hh` is provided in the top level include directory to ease the inclusion of BEE header files. In most cases it will be sufficient to just use the single statement

```
#include "BEE.hh"
```

in user code to make use of the BEE classes.

2.5 Building the BEE library

To build the BEE library, just type

```
bash> make
```

in the top level directory of the installation. This will build the shared library libBEE-<version>.so, together with the libraries of all projects listed in the projects variable in Makefile.setup: clue, gate, geometry, grover, pattern and vt++. All other projects just consist of header files and thus don't produce a library. All object files and libraries will end up in a directory called $HBBINTYPE/lib which is located in the BEE-<version> directory. The libBEE-<version>.so library is the only library which has to be used for linkage, as it contains the other libraries as well (for convenience). Thus you only have to link with a single library.

Example: In case of BEE version 0-101, compiled on a glibc-Linux, the location of the library is:

```
<path-to-BEE>/BEE-0-101/Linux_intel_glibc/lib/libBEE-0-101.so
```

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

```
libBEE.so -> libBEE-0-101.so
```

Linking your code with BEE becomes now as simple as:

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

In case you want to compile on a multiprocessor machine, you can specify the number of CPUs to use for compilation on the command line. The PROC option steers the number of processors to be used for parallel compilation. A compilation on a 4 processor machine can be issued like this:

```
bash> make PROC=4
```

Per default, PROC is set to 2.

2.6 Makefiles and Make targets

In the BEE project several makefiles exist:

Makefile Top level makefile providing necessary targets to build the current BEE installation.

Makefile.BEElib contains special make targets which allow to loop over the projects and perform certain tasks.

Makefile.setup central configuration makefile where all important variables are defined. By changing the settings and definitions in this makefile it should be possible to compile BEE under different operating systems and environments.

Makefile.targets defines all make targets which are important for the compilation of individual BEE projects or BEE based user code.

To support the makefiles, several shell scripts have been written which perform specialized tasks:

BEE_command.sh checks whether a certain command exists (similar to the `command -v` command, which does not exist on every system).

BEE_tar_src.sh creates the source code distribution tarball from the current installation.

BEEinstall script which can check out a certain project version from the development repository at the Max-Planck-Institut für Kernphysik, Heidelberg). This script relys on the presence of the `xxcvs` code organization tool [10].
Note: This script works only from whithin the MPI institute.

BEEmakelib script which checks the presence of the correct project versions and steers in case of absence the installation of the individual projects via the `BEEinstall` script.

version.sh extracts the project version from a `CVStag` file [10].

util.sh defines some useful shell functions and variables for the installation scripts.

In the above mentioned makefiles a lot of make targets have been defined. The following list explains the most important ones in alphabetically order:

alib Build the static libraries of the projects.

all Build all static and shared libraries.

check Print out settings of makefile variables.

clean Remove the `$HBBINTYPE` directory to get rid of all files created during compilation.

cleandoc Remove all documentation files which have been generated by `make docu`.

debug Build all shared libraries with the debugger flags `-g -ggdb`.

depend Create header file dependencies for makefiles. Important for a consistent recompilation of source files in case header files are changed.

docu Build all documentation files (needs `DOC++` and LaTeX).

html Generate HTML documentation of the distribution in the directory defined by the `htmldir` variable, which is defined in the `doc/Makefile` of the individual project.

remake Build all shared project libraries plus `libBEE-<version>.so`, but without running through the installation scripts again.

run Run an executable.

rungdb Run an executable in the `gdb` debugger.

saexe Build a statically linked executable (needs a user project, see section 3.1).

slib Build the shared libraries of the projects.

soexe Build an executable linked with shared BEE library (needs a user project, see section 3.1).

tar Generate a source code distribution tar file in the directory `../export`.

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

3 Building Executables

The user can create and run analysis jobs based on the `BEE` library which can then run as batch jobs and analyze large amounts of data. `BEE` supports the user in building such executables by providing scripts which generate code skeletons for analysis routines and a ready-to-use makefile for compilation. This allows the user to safe a lot of time and use a standardized code development environment.

3.1 Create your own Analysis Project

In case the user wants to create his own `BEE` project, e.g. in order to implement a `BEE` based analysis job, he has to choose a unique name for his project (which is not already used in `BEE`) and execute the `makeproject.sh` script which can be found in the top level directory of the `BEE` installation. In the following example the project name is chosen to be `myproject`:

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

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

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

As shown in the above example, some files are created within the directory tree. These file templates can be further edited by the user:

Makefile The project makefile. Here a user defined name for the executable can be set or user definitions can be added.

cintdict/LinkDef.hh ROOTCINT class declaration file, see [11].

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 to be provided by the author of the project. This is the place for information which is not covered by the automatic generated documentation.

Once the `makeproject.sh` script is executed, the user can create source code in the top level directory of the project and place the header files in the `include` directory. To generate a shared library from the user code a simple

```
bash> make
```

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

```
root [0] gSystem->Load("libmyproject.so");
```

Now the classes defined in the library are available within ROOT. In order to integrate the classes into ROOT I/O and RTTI, check out the instructions on the ROOT homepage or `http://root.cern.ch/root/Using.html`.

3.2 Create your own Analysis Class

Once the user has created his own `BEE` project (see section 3.1), he can start to write his own analysis code. In order to arrive at more standardized code, the `makeanalysis.sh` script is provided. The script will generate source code which is based on `BEE`'s `AnalysisIf` interface [12] and serves as a template for a user defined analysis:

```
bash> BEEROOT=$HBROOT/BEE/BEE-<version>
bash> $BEEROOT/makeanalysis.sh <class name>
```

The files generated by the script are:

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

The file `main.C` provides a lot of command line options and functionalities and works immediatly without further modifications (see section 3.4). The other two files provide the definition of an analysis class which can be run by the `main()` routine defined in `main.C`. The user can edit the class and fill the empty function definitions with his analysis code. The `AnalysisIf` interface forsees the following user editable functions:

EventSelect() Routine for event selection conditions. In case the routine returns `false`, the event is skipped for the analysis, otherwise it is accepted.

DefinePattern() In case of a pattern based analysis this is the place where track and decay pattern should be defined.

DefineHistograms() Place where ROOT/PAW histograms or NTUPLES can be defined.

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

EventCleanup() A routine which is called once after the `Event()` call. Here one can place cleanup code (for example memory deallocation).

End() This routine is called once at the end of the job (after all events have been processed). This is the place for some final calculations or cleanup code.

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

Note: It is recommended to use the makeanalysis.sh script to produce a correct main() routine together with an analysis class.

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 before compiling the user library:

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

The mainsrc variable definition can be omitted in case the user will only build executables.

3.3 Build a BEE Executable

Two possibilities exist to run a BEE based analysis job:

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

The first method is rather trivial. For more information see the examples on the ROOT homepage [1]. To simplify the second method (which has the advantage that no interpreter is running), BEE provides the possibility to automatically generate the needed code for building an executable (see section 3.2).

In order to compile the executable, the user should create the file depencencies first:

```
bash> make depend
```

Note: The dependencies should be recreated via make depend as soon as new header files are added by the user to assure a consistent recompilation of the project sources. Otherwise run-time errors might occur which are hard to debug.

To compile the executable,

```
bash> make soexe
```

must be executed in the project directory. This will generate an executable which is linked with the shared BEE library. In case a statically linked executable is needed, one can use the

```
bash> make saexe
```

command. This assumes that a statically linked BEE library exists (see the alib make target, section 2.6).

Note: Even with the make saexe command the executable will depend on shared libraries, e.g. the ROOT shared libraries.

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

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

After successful compilation, the binary can be found in the $HBBINTYPE/bin project subdirectory.

Before running the executable, make sure that the shared libraries (e.g. libBEE.so) can be found by the dynamic loader, e.g. by extending the LD_LIBRARY_PATH environment variable:

```
bash> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:
      <BEE path>/BEE-<version>/$HBBINTYPE/lib
```

To run the executable it is convenient to use the make run target which will set the LD_LIBRARY_PATH appropriate. However, with make run it is not possible to use command line options (see section 3.4).

3.4 Command Line Options

Once you have compiled your analysis class which was generated by the `makeanalysis.sh` script (see section 3.2), 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
```

The command line options allows to conveniently specify data files and parameters without having configuration files:

```
a.out -n 5000 /scratch/BEE/*.root -o out.root
```

3.5 Debugging Executables

In case an error (e.g. a segmentation fault) occurs during run time in the exectuable, the user code can be recompiled with debugger support:

```
bash> make clean; make depend; make soexe DEBUGFLAG=1
```

To run the executable in the `gdb` debugger, type

```
bash> make rungdb
```

On the `gdb` command line, start the executable with the `run` command plus the command line options:

```
(gdb) run -n 5000 /scratch/BEE/*.root -o out.root
```

In case you want to debug the `BEE` library, recompile the `BEE` library with debugger support by issuing the command

```
bash> make remake DEBUG=1
```

at the top level directory of the `BEE` distribution. This will generate a `BEE` library `libBEE_g.so` which has `gdb` support. To compile an executable with debugger support and using the `BEE` library with debugger support, use the

```
bash> make soexe DEBUG=1
```

command in the user's project directory. Now the debugger will be able to trace problems down into the `BEE` library:

```
bash> make rungdb DEBUG=1
```

3.6 Speed optimizations

In order to investigate the CPU consumption of an analysis job a code profiler can be used. To do so, the code must be recompiled with profiler support. When the executable is running, it will generate a special output file readable by the profiler. The profiler will determine afterwards the statistics of the function call tree and thus allow the user to analyze where the most time was spent in the code. Usually it makes no sense to optimize code before one has analyzed the executable with a profiler because in many cases it turns out that the most CPU intensive code is not the expected one.

To recompile BEE with profiler support, issue the following command in the top level directoy of the BEE distribution:

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

This will recompile the BEE sources using the -pg compile flag. This compiler flag supports the famous and widely available profiler gprof. The object files will be kept in separate subdirectories of $HBBINTYPE/lib (can be distinguished by the _pg suffix). The resulting BEE library will be a static library with the name libBEE_pg.a.

In a second step, a statically linked BEE executable must be compiled (see section 3.3) with the PROF variable defined:

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

The user code will be linked with libBEE_pg.a. The resulting executable with profiler support can be recognized by the extension _pg. The default name and location is: ./$HBBINTYPE/bin/a.out_pg.

When the binary is executed, it will produce a file gmon.out which contains the execution profile. Running gprof on both the executable and the exectuion profile will generate a detailed output of the call graph statistics which will allow to analyze the CPU consumption of the binary:

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

4 Setting up the ROOT environment

Although BEE can run without ROOT, the main advantage of BEE is certainly that it can be used as a ROOT application. In the following, some aspects concerning the work with BEE together with ROOT are explained.

In order to use the BEE library in ROOT, you can add the BEE library path to the ROOT DynamicPath variable in the .rootrc configuration file which should be present in the user's home directory:

```
# 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 script file you can now place the command which loads the BEE library at ROOT startup:

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

To set up the BEE event structure, 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. After startup, the TChain object f is available inside ROOT which allows to add and process BEE-files (see the section about data access in the clue manual [8]).

Note: In case you want to generate BEE-files from ARTE DST files, read the cluearte manual [13].

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

[3] http://wwwinfo.cern.ch/asd/cernlib

[4] T. Glebe, Vt++ Version 1.0, HERA-B Note 00-175, Software 00-013, 2000.

[5] Stepanov, A. and Meng Lee, The Standard Template Library, ANSI X3J16-94-0095 / ISO WG21-NO482

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

[7] http://www-hera-b.desy.de/subgroup/software

[8] T. Glebe, Clue - The BEE event model library, HERA-B Note 01-138, Software 01-019, 2001.

[9] T. Glebe, SMatrix - A high performance library for Vector/Matrix computation and Vertexing, HERA-B Note 01-134, Software 01-017, 2001.

[10] T. Glebe & M. Schmelling, Code Management with xxcvs, 1998. http://www.mpi-hd.mpg.de/herab/xxcvs.ps.gz

[11] Rene Brun, Fons Rademakers, Using ROOTCINT, http://root.cernch/root/RootCintMan.html.

[12] T. Glebe, BEE - Interfaces and Design Issues, HERA-B Note 01-136, Software 01-018, 2001.

[13] T. Glebe, Cluearte 0-42 – The ARTE interface to BEE, HERA-B Note 01-110, Software 01-004, 2001.