BMS: Build Management System

D. Lawrence
Jefferson Lab

May 3, 2005

Abstract

The BMS Build Management System is a set of GNU Makefiles which simplify and standardize the building of source code distribruted throughout a directory tree. Files with .c, .cc, or .F suffixes are automatically compiled. Platform dependance is handled through automatic inclusion of platform specific makefiles. Building versions for debugging and profiling is done by setting a single variable. The system also includes rules to automatically generate ROOT dictionaries from any .h file which includes the *ClassDef* keyword.

Contents

1 Introduction

The BMS is a set of GNU Makefiles which can be used to compile libraries and link exectables from a collection of source files. In practice, most projects need to do the same thing 99% of the time. Namely, compile all of the source files in a directory and then either archive them into a library. or link them into an executable. The goal of BMS is to implement this default behaviour in a generic way so that makefiles for large numbers of projects don't have to be maintained separately.

In contrast to a typical makefile, the BMS makefiles contain no information about the names of the files that they need to compile. Rather, they assume that all source files (identified by a .F, .c, or .cc suffix) in a directory should be compiled. It is believed that this can help lead to better maintenance of the source tree as files which should not be compiled must not be kept among those that should.

A typcial users' makefile will look like this:

Listing 1: Example Makefile

```
PACKAGES = ROOT:DANA

include $(HALLD_HOME)/src/BMS/Makefile.bin
```

In the example in listing 1, one or more excutables will be made from the source code. The $PACKAGES = ROOT : DANA$ line tells BMS to include the *Makefile.ROOT* and *Makefile.DANA* from the BMS directory.

BMS was designed for use in the Hall-D source code environment. However, it is general enough to be used in other applications.

2 File Naming Conventions

2.1 Makefiles

The user makefiles (as shown in listing 1) can potentially have any name. The recommendation however is to name them *Makefile*. This name is one of a list that gmake looks for if one is not excplicitly passed to it on the command line.

The BMS files themselves are kept in the src/BMS directory and have names of the form *Makefile.** where the * represents one of:

- the functionality provided by the makefile (e.g. Makfile.lib)
- the "OSNAME" of the platform the makefile is being invoked on (e.g. Makefile.SunOS)
- the package name of a package specific makefile (e.g. Makefile.ROOT)

The file *Makefile.ROOT* is part of BMS and provides the needed flags and link options fo building a ROOT enabled executable.

2.2 FORTRAN Files

BMS automatically invokes the fortran compiler for all files with a ".F" suffix. Files with a ".f" suffix are ignored. This is done for two reasons. The first is that the GNU F77 compiler uses the suffix to determine whether or not to run the file through the preprocessor. The ".F" files are preprocessed while the ".f" files aren't. The preprocessor's job is to replace # directives such as #include, #define, and #if which would cause errors in the g77 compiler proper.

The second reason is that PAW's built-in FORTRAN interpretor can be used to process FORTRAN files which are never intended to be compiled into an executable. By adopting the convention that ".f" files are for macros and ".F" files are for compiling, the macros and source files can coexist in the same directory.

2.3 C/C++ Files

C source files should end in ".c" while C++ source files should end in ".cc"[1]. The prefered suffix is ".cc". Header files should end in ".h".

In order to allow ROOT C++ macros to coexist with source code, they should have the suffix ".C".

2.4 Binary Files

The term "Binary Files" here means library files (e.g. libBAR.a), executable files, and object files (e.g. foo.o). In general, the library files are kept in the directory $\$HALLD\_HOME/lib/\$OSNAME$ and the executables are kept in $\$HALLD\_HOME/bin/\$OSNAME$. The output of the executables can be controlled through the $\$HALLD\_MY$ environment vaiable as described in section 4.1. The object files for libraries are kept inside the library archives themselves. For executables, the objects are kept in the directory $obj/\$OSNAME$ relative to the source directory. See section 6 for more details and how the naming scheme for debug versions differ from the non-debug versions.

3 BMS defined Rules and Custom Makefiles

If you wish to include BMS in situations which require more customization of the makefile, one can do so by adding a target to their makefile which uses the "all" target as a dependancy. A list of targets and rules defined by BMS is given here:

- all
- mkdirs
- clean
- %d : %.cpp
- %d : %.cxx
- %d : %.cx

[1] ".cpp" and ".cxx" are also treated as C++ source files, but their use is discouraged.

- %d : %.c
- %d : %.F
- %_Dict.cc : %.h
- env

3.1 make clean

To clean out all depends files, objects, libraries, and executables associated with a specific directory simply invoke:

```
make clean
```

This will also remove other generated files commonly left by editors, etc. including $*.bak$, $*$, *core*, *last.kumac*, *.depends* , $\#*\#$, and $obj/\$OSNAME$.

It is not a bad idea to do this every so often.

Note also that, as with other parts of BMS, doing a make clean will NOT recursively traverse directories. Only the files in the current directory are considered. Multiple directories can be "cleaned" with a single invocation of make clean by simply making a master makefile as described in section 5.

3.2 make env

The *env* rule is available for both the Makefile.lib and Makefile.bin makefiles. Invoking make with the *env* argument will print a list of variables (see section 4.1) with their values. Some are user settable and others are internal to BMS. This can be helpful when debugging the make system

3.3 Dependancy Rules

The purpose and power of a make system is to recompile only when necessary. To accomplish this, the make system must be aware of the dependancies of the source file. Specifically, the make system should recompile a source file if either it or any header files on which it depends are changed. To accomplish this, BMS makes use of a feature of the GNU compilers to generate dependancy rules by examining the source files themselves[2]. The dependancy rules are generated and stored in files in the .depends directory relative to its source. The depends files are given names with a *.d* suffix.

Building these GNU Make compatible rules is specific to the GNU compilers. As such, the BMS is hardwired to use the GNU compiler to generate them **even when using another compiler to actually compile the code**! This limits use of the BMS to systems which have the needed GNU compilers (g77, gcc, g++) installed. As these are free and available for all platforms on which GNU make can run, it should not be much of a limitation.

[2]See the $-M$ and $-MM$ options of gcc

3.4 ROOT Dictionaries

Some features of ROOT such as object I/O and the GUI classes can only be accessed if the classes are defined to ROOT through a ROOT dictionary. Dictionaries are made from the C++ header files using the *rootcint* program. To make it easier to add this functionality, BMS defines a rule to run rootcint. It only applies the rule to header files (ones ending in *.h*) that contain the string $ClassDef$. In the Hall-D source code, a Makefile is placed in the *src/libraries/include* directory specifically for this purpose. Since the default name of *libinclude.a* seems less than optimal, the makefile specifies a different name (*libHDDICT.a*) with the following Makefile:

Listing 2: Makefile from src/libraries/include

```
PACKAGES = ROOT

MODULE_NAME = HDDICT

include $(HALLD_HOME)/src/BMS/Makefile.lib
```

Here, the $MODULE\_NAME$ variable is used to specify the alternate name of the resulting library. Also the ROOT package is included via use of the PACKAGES variable.

4 Modifying the behaviour of BMS

The behaviour of BMS can be modified through the setting of environment variables, or adding specialized makefiles to the BMS directory. In general, the only files in the BMS directory to have rules are $Makefile.common$ $Makefile.lib$ and $Makefile.bin$. All others (except for very special cases) will only modify variables (see section 4.1). You may need to add a makefile when porting BMS to a new platform or you may want to compile using a different compiler. These situations are addressed in the following sections.

4.1 Variables

- OSNAME: If not set, it is set to return value of *uname* (e.g. *Linux*, *SunOS*, *Darwin*, ...)
- FC: FORTRAN compiler. Default is *g77*
- CC: C compiler. Default is *gcc*
- CXX: C++ compiler. Default is $g++$
- DFC: FORTRAN compiler used for generating dependancy rules. Default is g77 (see section 3.3)
- DCC: C compiler used for generating dependancy rules. Default is gcc (see section 3.3)
- DCXX: C++ compiler used for generating dependancy rules. Default is g++ (see section 3.3)

- HALLD_HOME: Points to the the directory which contains the *src* directory. The libraries and executables will be placed in $\$HALLD\_HOME/bin/\$OSNAME$ and $\$HALLD\_HOME/lib/\$OSNAME$
- HALLD_MY: If this is set, executables will be placed in $\$HALLD\_MY/bin/\$OSNAME$, but libraries are still linked from $\$HALLD\_HOME/lib/\$OSNAME$
- FFLAGS: Flags for FORTRAN compiler
- CFLAGS: Flags for C compiler
- CXXFLAGS: Flags for C++ compiler
- PACKAGES: Colon separated list of packages for which to include the corresponding BMS makefiles.
- LIBS: Libraries to include on the link command
- MODULE_NAME: Used as the base name for the output binary. Defaults to directory name for libraries and basename of "main" source file for executables.
- LD: Linker. Defaults to $\$CXX$
- MISC_LIBS: Extra libraries added to the end of the link list.
- DEBUG: If set, creates debug version of binaries (see section 6)
- LINK_OBJS: Extra link objects to add to the link command

4.2 Platform Dependance

Platform dependance is handled by including a platform specific makefile which is part of BMS. This done automatically with the OSNAME variable (see section 4.1). The $-include$ command is used rather than *include* (without the "-") because it will not give an error if the file does not exist. This means that new platforms for which no platform specific makefile exists at least have a chance to compile. Every effort should be made to develop the software in platform-independant ways so the content of the platform dependant Makefiles is minimized. In fact, at the time of this writing, there is no Makefile.Linux because there are no special requirements for the Linux platform. The best example of an existing BMS makefile which incorporates platform dependant settings is the Mac OS X file Makefile.Darwin (so named because uname returns "Darwin")

Listing 3: Makefile.Darwin

```
# This defines flags which can implement (or not) features
# in a way compatible with this OS

OSXFLAGS = -DBASENAME_IN_LIBGEN -DXDR_LONGLONG_MISSING -Wno-long-double

CFLAGS += $(OSXFLAGS)
CXXFLAGS += $(OSXFLAGS)
```

The important thing to note here is that the additional settings for the $CFLAGS$ and $CXXFLAGS$ variables are **appended** to the variables. This will insure settings placed there by the generic system are not overwritten.

The other point to notice in listing 3 is that "BASENAME_IN_LIBGEN" and "XDR_LONGLONG_MISSING" are features. These are used to control sections of compilation in the code by using preprocessor directives like: #ifdef XDR_LONGLONG_MISSING. You generally want to avoid using the platform itself by doing something like this: #ifdef Darwin. Compiling on features rather than platforms allows one to turn on/off sections related to single features. Using the platform as the conditional couples all conditionals for a single platform together.

4.3 Compilers

By default, the GNU compilers (g77, gcc, and g++) are used. The compilers can be changed by setting the FC, CC, and CXX variables (see section 4.1). Note that the GNU versions must still be present though since they are used to produced the dependancy files as described in section 3.3.

To override one or all of the compilers, one can set the environment variable of the appropriate name before invoking *gmake*.

It may be desirable to always use a particular compiler on a certain platform. For example, the solaris compiler on SunOS machines. In this case, the FC, CC, and CXX variables should be set in the platform specific makefile Makefile.SunOS.

5 Building across multiple directories

BMS was designed for a software heirarchy in which directories contain more or less independant packages. When executables are made, BMS assumes the libraries it needs to link against are up to date rather than checking if they need to be rebuilt. It is done this way since it can often be the case that one links against libraries stored in a group area where one lacks either the source code or the neccessary privileges to rebuild them.

To use BMS across multiple directories, a "master" makefile should be created which invokes make in the appropriate subdirectories. For example, the makefile that is in the src/ libraries directory looks like this:

Listing 4: Makefile from src/libraries

```
.PHONY: all


all:
        make -C include
        make -C BCAL
        make -C CDC
        make -C CHERENKOV
        make -C DANA
        make -C FCAL
        make -C FDC
        make -C HDDM
        make -C TAGGER
        make -C TOF
```

```
        make -C TRIGGER
        make -C UPV
        make -C TRACKING

clean:
        make -C include clean
        make -C BCAL clean
        make -C CDC clean
        make -C CHERENKOV clean
        make -C DANA clean
        make -C FCAL clean
        make -C FDC clean
        make -C HDDM clean
        make -C TAGGER clean
        make -C TOF clean
        make -C TRIGGER clean
        make -C UPV clean
        make -C TRACKING clean
```

This just runs make (or make clean) in each of the specified directories. BMS does NOT recursively traverse directory structures. Having BMS recursively search subdirectories would complicate the system for little gain. Writing makefiles such as the one in listing 4 are trivial enough and leave a level of control with the user.

This same mechanism can be employed for directory trees in which more than one level contains source code. For example: One has source code in a directory called "foo", but there is also a subdirectory of foo named "foo/bar" which contains source code. One could make a master makefile called Makefile and a BMS specific one called something like Makefile.bms. The two would look like this:

Listing 5: Makefile

```
all:
        make -C bar
        make -f Makefile.bms

clean:
        make -C bar clean
        make -f Makefile.bms clean
```

Listing 6: Makefile.bms

```
PACKAGES = ROOT:DANA

include $(HALLD_HOME)/src/BMS/Makefile.bin
```

Listing 5 shows the "master" makefile whose job is simply to invoke make in the current directory and in the subdirectory named *bar*. Listing 6 shows the makefile which implements BMS to compile the source in the current directory.

6 Building Debugging Versions

By default, files are compiled without debugging symbols and with level 2 optimization [3] ($-O2$ flag). However, if the variable DEBUG is set, then instead of the default optimization, the $-g$ and $-pg$ flags are set. The $-g$ flag adds debugging symbols while the $-pg$ flag turns on profiling[4] and includes the proper compiler libraries [5].

The DEBUG variable can be set either through the environment, or by passing it explicitly on the command line:

```
make DEBUG=yes
```

The value of DEBUG is ignored. It is only checked if it is set. Ergo, compiling with DEBUG=no would produce the same result as the above example.

The debugging versions of binaries are given suffixes of _d to distinguish them from their non-debug counter parts and so that they may share directories with them. For example, a library named *libFOO.a* will have a debug version named *libFOO_d.a*. An executable named *BAR* will have a debug version named *BAR_d*.

When making a debugging version of an executable, it is assumed that the needed debugging version of the libraries have been made. This is the same behaviour as for non-debug versions. The point being that one cannot link debug-objects with non-debug libraries.

It may be worth noting that the objects created from the source files follow a slightly different naming scheme. Objects created for libraries are stored in the library archive itself. These objects are given the same names as those in the non-debug version. For example, a file named *recon.cc* in the directory FOO will be stored as an object named *recon.o* in both *libFOO.a* and *libFOO_d.a*. By contrast, objects in directories which contain source which is to be linked into executables are stored as separate files in the *obj* directory. For these, the object files each carry the *_d* suffix. For example, a file named *myprog.cc* would be stored as *obj/Linux/myprog_d.o* (on a Linux system). This is a detail that one does not generally need to know since this is all taken car of by BMS.

7 Questions

Please refer any questions to David Lawrence at *davidl@jlab.org*.

[3]Optimization is not used for FORTRAN files since the large hddsGeant3.F file in the Hall-D source code fails to compile with optimization turned on.

[4]see *gprof*

[5]The $-g$ flag seems widely used while the $-pg$ flag may be GNU specific.