Version

1.0

JEFFERSON LAB

Data Acquisition Group

BMS User's Guide

JEFFERSON LAB DATA ACQUISITION GROUP

BMS User's Guide

Carl Timmer
timmer@jlab.org

David Lawrence
davidl@jlab.org

10-Sep-2008

12000 Jefferson Ave
Newport News, VA 23606
Phone 757.269.5130 • Fax 757.269.6248

Table of Contents

1. Introduction 4

2. File Naming Conventions 4

2.1. Makefiles 4

2.1.1. Table: makefiles included by users 5

2.2. C and C++ Files 5

2.3. Binary Files 5

2.3.1. Table: hidden directories containing binary files 5

3. Dependencies 5

4. BMS Defined Targets 6

4.1.1. Table: predefined BMS makefile targets 6

5. Using BMS 6

5.1. Operating System and Architecture 6

5.2. What Gets Compiled? 7

5.3. Compile and Link Flags 7

5.4. Optimization and Debugging 7

5.5. Libraries 7

5.6. Executables 8

5.7. User-Defined Targets 8

5.8. 64 Bit Compilation 8

5.9. External Software Packages 8

5.10. Example Makefile 9

5.11. Installation 10

5.11.1. Table: installation location of file types 11

6. Vxworks 11

7. Makefiles at the Package Level 11

7.1. Example 12

7.2. Makefile.local 14

8. CODA-Specific Compilation 14

8.1. Package Directory Structure 14

8.1.1. Table: software package directory structure 15

8.2. CODA makefile 16

8.3. JAVA 18

8.3.1. Table: ant targets and actions 21

1. Introduction

The Build Management System (BMS) is a set of GNU makefiles which simplify and standardized the building of source code distributed throughout a directory tree. The goal of BMS is to implement makefile behavior in a generic way so that makefiles for large numbers of projects don't have to be maintained separately. Files with .c, .cc suffixes are automatically compiled. Platform dependence is handled through automatic inclusion of platform specific makefiles. Building versions for debugging is done by setting a single variable. Cross compiling for vxworks is handled by setting 2 variables.

There are actually 2 BMS systems. The first was developed by David Lawrence for Hall-D software. This BMS system was adapted from the original by the Data Acquisition Group for CODA software.

.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 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. Files which are placed into a single library must be together in one directory. Likewise, files containing related executables (containing "main" in the case of C, C++), are placed in their own directory.

Makefiles in the library and executable directories simply include appropriate makefiles from BMS. For example, an actually makefile may be as simple as the one below.

```
include $(BMS_HOME)/Makefile.common
include $(BMS_HOME)/Makefile.libs
```

2. File Naming Conventions

2.1. Makefiles

In order to allow great simplification when creating upper level makefiles, the bottom level makefiles must be named Makefile for unix and Makefile.vxworks-<arch> for cross compiling vxworks. Currently makefiles for vxworks are called Makefile.vxworks-ppc since the PPC platform is the only one in use. Generally speaking, on the lowest level there must be a Makefile.vxworks-ppc in each directory that there is a Makefile even if the vxworks makefile is a dummy which does nothing. See section 6 for further discussion on this subject.

The BMS files themselves can be checked out of the subversion repository into a single directory. To use BMS, the BMS_HOME environmental variable must be set to this directory. See the following table for makefiles used directly by the user.

2.1.1. Table: makefiles included by users

Makefile Name	Function
Makefile.common	this should be included first and contains definitions necessary for all compilation
Makefile.libs	this should be included for making static and shared libraries and allows for user-defined targets
Makefile.bin	this should be included to make executables and allows for user-defined targets
Makefile.lib	this is a variation of Makefile.libs which only creates a static library

There are other makefiles in BMS that are OS specific like Makefile.Linux, GNU specific like Makefile.GNU, and software package specific like Makefile.xerces or Makefile.cMsg. However, the user should not have to deal with these directly.

2.2. C and C++ Files

C source files must end in .c while C++ source files must end in .cc, .cpp, or .cxx. Header files should end in .h, .hh, or .hxx.

2.3. Binary Files

The term "binary files" refers to library, executable and object files. In general, binary files are kept in hidden directories which are operating system and architecture dependent. These are listed as follows:

2.3.1. Table: hidden directories containing binary files

FILE TYPE	DIRECTORY
library	.$(BMS_OSNAME)/lib
executable	.$(BMS_OSNAME)/bin
object	.$(BMS_OSNAME)/object
dependency	.$(BMS_OSNAME)/depends

3. Dependencies

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 dependencies of the source files. Specifically, the system should recompile a source file if either it or any header files on which it depends have been changed. BMS does this by making use of a feature of the compilers (GNU and Solaris) to generate dependency rules by examining the source files

themselves. The dependency rules are generated and stored in files in the dependency directory listed in the table above. The depends files are given names with a .d suffix.

4. BMS Defined Targets

There are a number of standard targets already defined in Makefile.libs, Makefile.lib, Makefile.bin, and Makefile.common. These targets do commonly desired tasks such as making the source, installing libraries, uninstalling, cleaning, and the like. Below is a listing of these targets:

4.1.1. Table: predefined BMS makefile targets

TARGET	EFFECT
all	Creates directories, makes dependencies, and makes libraries, executables, and user-defined targets
env	Prints out makefile variables of interest
mkdir	Creates directories necessary for BMS. Normally done in “all”.
install	Makes all (above) and installs into INSTALL_DIR
uninstall	Removes files that were installed
clean	Deletes all executable, library, object, and dependency files
distClean	Makes clean (above) and also removes hidden OS directory
execClean	Deletes all executable and library files
relink	Deletes library and executables, then relinks object files
here	Makes all and installs into the current directory
hereClean	Removes binaries from the current directory

5. Using BMS

The behavior of BMS can be modified through the setting of environmental variables. The various ways in which users can determine this behavior are discussed below.

5.1. Operating System and Architecture

By default, the variables BMS_OS and BMS_ARCH are defined as uname and uname –m, but these can also be set by users. The variable BMS_OSNAME is set as $(BMS_OS)-$(BMS_ARCH) unless set by users. The significance of setting these is that the makefiles Makefile.$(BMS_OS), Makefile.$(BMS_ARCH), and Makefile.$(BMS_OSNAME) are all automatically included by Makefile.common in that order. If these variables are not set by the user, Makefile.common sets all variables appropriate for the local operating system and architecture. It is advisable NOT to set them unless compiling for vxworks in which case BMS_OS should be set to *vxworks* and BMS_ARCH should be set to *ppc*.

5.2. What Gets Compiled?

If users' makefiles include Makefile.libs, all source code gets compiled and placed in the library(ies) by default. If users' makefiles include Makefile.bin, all source code with the function "main" will be compiled by default. Note that this function must be declared as "int main" on one line for BMS to automatically find the executables. Either that or a space must precede main on the same line.

Users may define a space-separated list of objects to NOT be compiled in the variable NO_OBJS. Likewise, a space-separated list of objects for which dependencies should NOT be made can be placed in the variable NO_DEP_OBJS.

5.3. Compile and Link Flags

Special compile flags may be added for C and C++ compilation by adding the lines:

```
CFLAGS += myflag
CXXFLAGS += myflag
```

respectively. Be sure to use the "+=" operator or all the other necessary flags will be overwritten.

For linking, add directories in which to find libraries by adding:

```
LD_LIBS += dir     ,
```

add link flags for creating libraries by adding:

```
LD_SO_FLAGS += flag      ,
```

and add link flags for creating executables by adding:

```
LDFLAGS += flag   .
```

5.4. Optimization and Debugging

To compile optimized code, in the users' makefile, before including Makefile.common, include the line:

```
OPTIMIZE = N
```

where N can be 1,2, or 3. This adds the optimization flag –O1, -O2, or –O3 for GNU compilers or the flag –xO1, -xO2, or –xO3 for Solaris compilers. By default there is no optimization.

For debugging, simply define the DEBUG environmental variable to anything (its value is ignored). This automatically gets rid of the optimization flag and adds the –g flag to the compilers. The debug versions of binaries are given suffixes of _d to distinguish them from their non-debug counterparts and so that they may share directories with them. For example, a library name libFOO.a will have a debug version named libFOO_d.a, and an executable named BAR will have a debug version named BAR_d.

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

5.5. Libraries

By default, libraries will be named lib$(MODULE_NAME).a / .so for static and shared libraries respectively, and lib$(MODULE_NAME)_d.a / .so for debug versions.

However, users can supersede this by setting SHLIB_NAME for a shared library and LIB_NAME for a static library. Also the .so suffix of shared libraries can be superseded by setting SHLIB_SUFFIX.

Before including Makefile.libs, defining MAKE_SHARED_LIB as any value will make a shared library. Similarly, defining MAKE_STATIC_LIB as any value will make a static library.

5.6. Executables

Before including Makefile.bin, defining the variable OTHER_DEPS as a space-separated list of files will cause the executables to be remade from the object files when the files listed in OTHER_DEPS change.

5.7. User-Defined Targets

Before including either Makefile.bin or Makefile.libs, users can define OTHER_TARGETS as a space-separated list of additional targets. ***After*** including Makefile.common, users need to define the targets themselves. If a target is defined before Makefile.common is included, it becomes the first and therefore default target. This will break the expected behavior. “Make install” will install them and “make uninstall” will uninstall them as well.

The best way to define your target is to precede it with the proper directory: $(LIB_DIR) when making an library and $(BIN_DIR) when making an executable. That way the user-defined target gets stored in the same os and architecture dependent directory as the rest of the compiled code. See the example makefile and its OTHER_TARGET in section 5.10

5.8. 64 Bit Compilation

Setting the variable CODA_USE64BITS to anything (value not used) will cause flags to be set that will compile the code to be 64 bits. Note that trying to compile for 64 bits on a 32 bit machine causes an error on Solaris while Linux just creates 32 bit code. Vxworks ignores CODA_USE64BITS.

5.9. External Software Packages

When making libraries or executables that depend on an external software package, users simply need to define the variable PACKAGES in the makefile. Define PACKAGES as a colon separated list of the needed software packages. For example, if one needs the et and cMsg libraries for an executable or the et and cMsg headers to make a library, the line:

```
PACKAGES = et:cMsg
```

must be present in the makefile before including Makefile.common. What this does is include the files Makefile.et and Makefile.cMsg in that order in Makefile.common. These makefiles must exist in the BMS directory to work and the package names are case sensitive.

Another way to access external libraries and include directories is by adding them to the variable MISC_LIBS:

```
MISC_LIBS += -LmyLibDir -lmyLib    .
```

5.10. Example Makefile

The following makefile is an example of all the issues discussed up to this point. It is taken from the et library source directory but with the irrelevant material removed:

```
1  ifndef BMS_HOME
2    $(error "Need to define BMS_HOME")
3  endif

4  ifeq ($(BMS_OS), vxworks)
5    $(error "This makefile is for unix, use Makefile.vxworks-ppc")
6  endif

7  TOPLEVEL = ../..
8  MODULE_NAME = et
9  NO_OBJS = et_remoteclient.o

10 include $(BMS_HOME)/Makefile.common


11 # objects for making a remote library
12 ROBJS = et_statconfig.o \
13        et_openconfig.o \
14        et_init.o \
15        et_sysconfig.o \
16        et_remote.o \
17        et_network.o \
18        et_remoteclient.o

19 RDOBJS = $(addsuffix $(DEBUG_SUFFIX).o,$(basename $(ROBJS)))
20 OTHER_TARGETS += $(LIB_DIR)/libet_remote$(DEBUG_SUFFIX).so
21 RLINK_OBJS = $(addprefix $(OBJ_DIR)/,$(RDOBJS))

22 MAKE_SHARED_LIB = make_me_please
23 MAKE_STATIC_LIB = make_me_please

24$(LIB_DIR)/libet_remote$(DEBUG_SUFFIX).so: $(RDOBJS)
25     $(LD) $(LD_SO_FLAGS) $(LD_DIRS) $(RLINK_OBJS) $(LD_LIBS) -o $@

26 include $(BMS_HOME)/Makefile.libs
```

Lines 1-3: The first order of business is to know where the BMS makefiles are since Makefile.common and Makefile.libs must be included. Thus the environmental variable BMS_HOME must be defined or an error is returned.

Lines 3-6: When run, GNU make first looks for the file GNUmakefile, then for makefile, and finally for Makefile. In the CODA BMS system all makefiles are called either Makefile, Makefile.local, or Makefile.vxworks-ppc. Thus, make will use Makefile by default – which is the unix makefile. These lines do not allow the unix makefile to be used with vxworks.

Line 7: Makefile.local is included if make is run in the package or source code level directories. The variable TOPLEVEL specifies where the software

package's top level directory is so the location of Makefile.local is known. See section 7.2 for more on Makefile.local.

Line 8: Setting the module name to "et" means that the libraries made in this directory will be called libet.so and libet.a .

Line 9: NO_OBJS contains the objects NOT to be included in the libraries.

Line 10: Once NO_OBJS is defined, Makefile.common can be included. It contains all the basic definitions needed for BMS makefiles to work.

Lines 11-18: In the "et" software package, not only must a shared and a static library be made, but the third library called libet_remote.so must be made as well. The third library contains a subset of the files in the regular libraries along with the object specifically excluded from the regular libraries. These lines list all the needed objects for libet_remote.so in ROBJS.

Line 19: RDOBJS adds the debug suffix (if any) to the objects needed in libet_remote.so

Line 20: OTHER_TARGETS lists the libet_remote.so library as an extra target to be made in addition to the 2 libraries normally make when including Makefile.libs. Notice the prefix of $(LIB_DIR)/ which is the default location of libraries in BMS

Line 21: RLINK_OBJS adds a prefix of $(OBJ_DIR)/ to the objects listed in RDOBJS. This is where BMS stores object files.

Line 22: Defining MAKE_SHARED_LIB to any value tells Makefile.libs to make a shared library.

Line 23: Defining MAKE_STATIC_LIB to any value tells Makefile.libs to make a static library.

Lines 24-25: Define the targets listed in OTHER_TARGETS. The target must be defined ***after*** including Makefile.common otherwise it would become the first and default rule. It may come after Line 26 if desired.

Line 26: Now that OTHER_TARGETS, MAKE_STATIC_LIB, and MAKE_SHARED_LIB are defined, include the BMS makefile that will make libraries – Makefile.libs.

5.11. Installation

Define the variable INSTALL_DIR to the desired installation directory. If this is not defined, Makefile.common will set this to the value of the variable CODA_HOME. If CODA_HOME is not defined either, an error will be printed and the build will be stopped. A "make install" will place everything made and headers into predefined places as seen in the table following.

5.11.1. Table: installation location of file types

File Type	Installation Location
Library	$(INSTALL_DIR)/$(BMS_OSNAME)/lib
Executable	$(INSTALL_DIR)/$(BMS_OSNAME)/bin
Include	$(INSTALL_DIR)/include
Documentation	$(INSTALL_DIR)/doc/<package name>

6. Vxworks

BMS can cross compile for vxworks with a little bit of effort. Remember that by default the operating system and architecture are taken from the local machine. For vxworks, the following two environmental variables must be set by hand to:

```
BMS_OS = vxworks
BMS_ARCH = ppc
```

Very often what gets compiled for vxworks is significantly different than what gets compiled for unix. This means that in order to avoid "ifdef vxworks" type constructs in makefiles, the user must create a whole new makefile for vxworks. For consistency and simplicity, these makefiles must be named Makefile.vxworks-ppc. Other than that there are no differences with the makefiles for unix.

In some directories there are no files to be compiled for vxworks. In such cases a dummy vxworks makefile can make upper level makefiles easier to create. An example of a good dummy vxworks makefile is given below:

```
#
# This is a dummy makefile that does nothing
#

dummy:
    @echo "No vxworks stuff in directory $(shell pwd)"

%:
    @echo "No vxworks stuff in directory $(shell pwd)"
```

Its default target is "dummy" which prints out a message saying there is no vxworks stuff in the directory. It also has a pattern rule which matches all targets. Thus users can type "make <anything>" and the same message gets printed out.

7. Makefiles at the Package Level

The BMS files Makefile.common, Makefile.libs, Makefile.lib, and Makefile.bin are designed to be used at the bottom level – the directories in which the source code lives. It would be awkward to go into each source directory by hand to make the multiple parts of

one software package, so an upper level makefile must exist for each independent software package.

7.1. Example

Since an upper level makefile does not actually use any of the BMS makefiles, it is technically not a part of the BMS system. However, since it must work closely with BMS, the art of constructing such a makefile is included here. Following is a makefile taken from the cMsg software package:

```
MAKEFILE = Makefile

# if using vxworks, use different set of the lowest level makefiles
ifeq ($(BMS_OS), vxworks)
  ifdef BMS_ARCH
    MAKEFILE = Makefile.$(BMS_OS)-$(BMS_ARCH)
  else
    $(error "Need to define BMS_ARCH if using BMS_OS = vxworks")
  endif
endif

# define TOPLEVEL for use in making doxygen docs
TOPLEVEL = .

# list dirs in which to run makefiles (relative to this one)
SRC_DIRS = src/regexp src/libsrc src/libsrc++ src/execsrc src/examples

# declaring a target phony skips the implicit rule search and saves time
.PHONY : first help java javaClean javaDistClean doc tar


first: all

java:
	ant;

javaClean:
	ant clean;

javaDistClean:
	ant cleanall;

doc:
	ant javadoc;
	export TOPLEVEL=$(TOPLEVEL); doxygen doc/doxygen/DoxyfileC
	export TOPLEVEL=$(TOPLEVEL); doxygen doc/doxygen/DoxyfileCC
	cd doc; $(MAKE) -f $(MAKEFILE);

tar:
	-$(RM) tar/cMsg-1.0.tar.gz;
	tar -X tar/tarexclude -C .. -c -z -f tar/cMsg-1.0.tar.gz cMsg

# Use this pattern rule for all other targets
%:
	@for i in $(SRC_DIRS); do \
	   $(MAKE) -C $$i -f $(MAKEFILE) $@; \
	done;
```

Line 1: The variable MAKEFILE is defined as Makefile which means by default, the unix makefile is run in the source directories

Lines 2-9: If BMS_OS = vxworks, then the makefile that is used in the source directories is Makefile.vxworks-ppc. This is necessary in the package level makefile for any cross compiling.

Line 11: The variable TOPLEVEL is defined as the top directory of the package. This allows the generated documentation (doxygen specifically) to place its documents in the correct location.

Line 13: The directories in which source code exists are listed in SRC_DIRS. This allows control over what gets made and what is skipped (e.g. test directories).

Line 15: Listing targets after .PHONY tells *make* that those targets don't create a file by the same name as the target. It also skips the implicit rule search. This allows targets like "clean" to be run each time and to be run much more efficiently.

Line 16: The first and default target simply executes the "all" target which does not occur in this makefile but gets run in the next level of makefiles.

Lines 17-22: These are targets that have to do with making java code. BMS is not a system for building java code. However, all of CODA uses a uniform system to make java code which will be presented later.

Lines 23-27: The "doc" target generates documentation with javadoc and doxygen.

Lines 28-30: The "tar" target creates a tar file of the package.

Lines 31-35: The last item in the makefile is the most important. It is a pattern rule which matches all targets (by using the wildcard %) except those already defined. The way it works is the following. When the user types:

```
make something ,
```

the target to be made is "something". If there is no target called "something" defined in the makefile, then it will match the given pattern rule which matches everything. What the rule actually does is loop through the list of source code directories and run make with the given target in each one. For a particular directory specified (say src/regexp), the actual command executed is:

```
make –C src/regexp -f Makefile something  ,
```

or for vxworks:

```
make –C src/regexp -f Makefile.vxworks-ppc something  .
```

The target "something" which had no match in the package level makefile is passed down to the lowest level makefiles. Of course, it is possible that the next level of makefile has no target called "something" either in which case an error will occur.

All of the targets specified in Table 4.1.1 can be used at the package level and will end up being run in the next level down.

7.2. Makefile.local

Software packages are often used independently of the full CODA distribution. In such cases, BMS must be able to build the code without reference to CODA variables. To facilitate that, in the top level of each package, a Makefile.local may be created. It is designed to overwrite CODA variables with package variables and is automatically included by Makefile.common if make is run at the package or bottom levels. Listed below is Makefile.local of cMsg which is fairly self-explanatory:

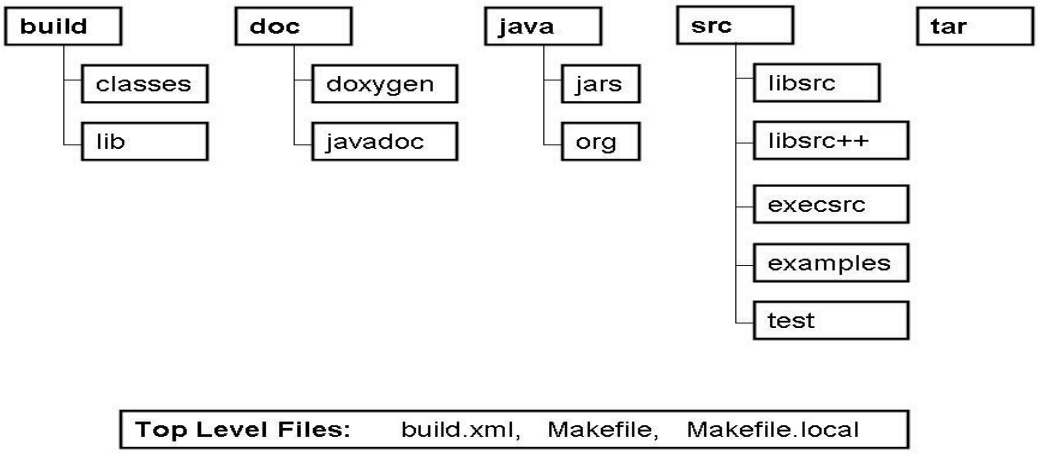
```
# If cMsg makefiles are called locally (not in a CODA build),
# override CODA variables with CMSG variables. Necessary if
# cMsg is made separately from CODA.
#
# Overwrite:
#     CODA_HOME      with CMSG_HOME
#     INSTALL_DIR    with CMSG_INSTALL
#     CODA_USE64BITS with CMSG_USE64BITS
#

ifdef CMSG_HOME
  CODA_HOME = $(CMSG_HOME)
endif

ifdef CMSG_INSTALL
  INSTALL_DIR = $(CMSG_INSTALL)
endif

ifdef CMSG_USE64BITS
  CODA_USE64BITS = $(CMSG_USE64BITS)
endif
```

8. CODA-Specific Compilation

The BMS system has nothing to say about the upper level makefiles and the directory structure of individual software packages. However, as part of the CODA software distribution, there are a number of conventions that are enforced for the sake of unity and simplicity. Also, the very top level makefile for all of CODA is contained in BMS and warrants it own section of the manual.

8.1. Package Directory Structure

Each independent software package (e.g. et, evio, cMsg) must have a directory structure that is as close as possible to the following:



8.1.1. Table: software package directory structure

Dir/File Name	Function
build	Directory in which results of a java build are stored (class files and jar file)
build.xml	Ant buildfile for building java code. Ant is used for making all java code.
doc	Directory in which to place all documentation.
doc/doxygen	Directory in which to place all doxygen files
doc/javadoc	Directory in which to place all javadoc files
java	Directory for all java files
java/jars	Directory for all jar files needed for compiling java source code
java/org/…	Directory containing all java source code (name determined by java package name)
src	Directory in which to place all sub directories which contain source code.
src/libsrc	Directory in which to place all C library code
src/libsrc++	Directory in which to place all C++ library code

src/execsrc	Directory in which to place all generally used executables' source code. Package level makefile will install these executables.
src/examples	Directory in which to place all example source code. Package level makefile will ***not*** install these executables.
src/test	Directory in which to place all test source code. Package level makefile will ***not*** install these executables.
tar	Directory in which to keep the tar file of the whole package as well as the tarexclude file which defines what stays out of the tar file.
Makefile	Top level package makefile for unix.
Makefile.local	Makefile for overriding CODA variables with package variables (optional).

The cMsg, et, evio, and codaObject software packages have all been given this same directory structure with, presumably, the rest of CODA to follow.

8.2. CODA makefile

The makefile for all of coda, though similar to the package level makefiles, is different enough to warrant it own explanation. This makefile must take 2 arguments which makes it a little more complicated. Other than as a target, the only way to pass an argument into make is to set a variable (e.g. ARG) value on the command line with the following syntax:

```
make target ARG=value
```

The CODA makefile is setup to use the software package directory name as the target and the options of Table 4.1.1 as the value of the variable ARG. For example, if the user wanted to compile the cMsg package stored in the "cMessage" directory to do a "distClean", then the command would be:

```
make cMessage ARG=distClean
```

If the target is not specified, the makefile will make all the directories listed internally as comprising CODA. The makefile looks for these target directories in CODA_HOME. Take a look at the actual file (minus the irrelevant parts):

```
1  MAKEFILE = Makefile

2  # if using vxworks, use different set of the lowest level makefiles
3  ifeq ($(BMS_OS), vxworks)
4    ifndef BMS_ARCH
5      $(error "Need to define BMS_ARCH if using BMS_OS = vxworks")
6    endif
7  endif
```

```
8   ifndef CODA_HOME
9     $(warning "Should define CODA_HOME")
10    # assume we're in BMS dir & also at the top level of CODA distribution
11    CODA_HOME = ..
12  endif

13  # list of possible targets
14  TARGETS = cMsg evio et

15  .PHONY : first help tar

16  first: $(TARGETS)

17  tar:
18          -$(RM) coda-3.0.tar.gz;
19          tar -X tarexclude -C .. -c -z -f coda-3.0.tar.gz coda

20  # Use this pattern rule for all other targets
21  %:
22          cd $(CODA_HOME)/$@; $(MAKE) -f $(MAKEFILE) $(ARG);
```

Line 1: The variable MAKEFILE is defined as Makefile which means that Makefile in the top level of each software package gets run.

Lines 2-7: If BMS_OS = vxworks, then make sure that BMS_ARCH is defined so that the bottom level BMS makefiles work properly for any cross compiling.

Lines 8-12: The variable CODA_HOME should be defined as CODA's top directory. This allows the targets (names of package directories) to be found since the target names are assumed not to be absolute pathnames.

Line 14: A listing of all package directories is contained in TARGETS. This allows control over what gets made and what is skipped by default.

Line 15: Listing targets after .PHONY tells *make* that those targets don't create a file by the same name as the target. It also skips the implicit rule search. This allows *make* to be run much more efficiently.

Line 16: The first and default target is TARGET - the list of all CODA directories. Thus all of CODA will be made by default.

Lines 17-19: The "tar" target creates a tar file for all of CODA.

Lines 20-22: The last item in the makefile is the most important. It is a pattern rule which matches all targets (by using the wildcard %) except those already defined. For example, when the user types:

```
make packageDir ARG=install ,
```

the target to be made is "packageDir". If there is no target called "packageDir" defined in the makefile, then it will match the given pattern rule which matches everything. What the rule actually does is change directories to CODA_HOME/packageDir and run make with the target "install". So in this case the actual commands executed are:

```
cd $(CODA_HOME)/packageDir
make -f Makefile install  ,
```

By default then, the makefile goes to each directory and runs make on Makefile.

8.3. JAVA

Ant is the way to go for building java code. It compiles java code an order of magnitude faster than make and is much more powerful as well. Ant requires a build file in XML format which plays much the same role that a makefile does for make. By default ant looks for a file called build.xml. All CODA packages that contain java code use ant and each contain a very similar build file called build.xml in its top level directory. The user can simply copy an existing build.xml file from et or cMsg and make a few small corrections in order to create one for a new software package. Following is build.xml from cMsg with some commentary:

```
1   <?xml version="1.0" ?>

2   <project name="cMsg"  default="compile" basedir=".">

3       <!-- Project-wide settings -->

4       <!-- Directories -->
5       <property name="src.dir"       value="java" />
6       <property name="build.dir"     value="build" />
7       <property name="build.classes" value="${build.dir}/classes" />
8       <property name="build.lib"     value="${build.dir}/lib" />
9       <property name="doc.dir"       value="doc/javadoc" />

10      <!-- Classpath -->
11      <path id="classpath">
12          <fileset dir="java/jars">
13              <include name="**/*.jar" />
14          </fileset>
15      </path>

16      <!-- Version -->
17      <property name="cMsg.version" value="1.0" />

18      <!-- Debug -->
19      <property name="javac.debug" value="off" />

20      <!-- Targets -->

21      <!-- Help -->
22      <target name="help">
23          <echo message="Usage: ant [ant opts] &lt;target1&gt; [targ2 | targ3 | ...]" />
24          <echo message="" />
25          <echo message="       targets:" />
26          <echo message="       compile  - compile java files" />
27          <echo message="       clean    - remove class files" />
28          <echo message="       cleanall - remove all generated files" />
29          <echo message="       jar      - compile and create jar file" />
30          <echo message="       all      - clean, compile and create jar file" />
```

```
31          <echo message="       javadoc  - create javadoc documentation" />
32          <echo message="       prepare  - create necessary directories" />
33      </target>

34      <!-- Environment -->
35      <target name="env">
36          <echo message="Ant environment:" />
37          <echo message="" />
38          <echo message="    src.dir       = ${src.dir}" />
39          <echo message="    build.dir     = ${build.dir}" />
40          <echo message="    build.lib     = ${build.lib}" />
41          <echo message="    build.classes = ${build.classes}" />
42          <echo message="    doc.dir       = ${doc.dir}" />
43          <echo message="    javac.debug   = ${javac.debug}" />
44      </target>

45      <!-- Prepare things by creating directories -->
46      <target name="prepare">
47          <mkdir dir="${build.dir}" />
48          <mkdir dir="${build.classes}" />
49          <mkdir dir="${build.lib}" />
50      </target>

51      <!-- Clean by removing class files -->
52      <target name="clean" description="Remove all class files">
53          <delete failonerror="no" >
54              <fileset dir="${build.classes}" includes="**/*.class" />
55          </delete>
56      </target>

57      <!-- Clean by removing build & doc directories -->
58      <target name="cleanall" depends="clean" description="Remove all generated files.">
59          <delete dir="${build.dir}" failonerror="no" />
60          <delete dir="${doc.dir}"   failonerror="no" />
61      </target>

62      <!-- Compile all source -->
63      <target name="compile" depends="prepare" description="Compiles all source code.">
64          <javac destdir="${build.classes}" debug="${javac.debug}" >
65              <src path="${src.dir}" />
66              <classpath refid="classpath" />
67          </javac>
68      </target>

69      <!-- Create Jar file -->
70      <target name="jar" depends="compile" description="Generates jar file.">
71          <jar jarfile="${build.lib}/cMsg-${cMsg.version}.jar">
72            <!-- Dont include Consumer, Producer, and StartET.class files -->
73            <fileset dir="${build.classes}" excludes="*.class" />
74          </jar>
75      </target>

76      <!-- Clean, compile, and create jar -->
77      <target name="all" depends="clean,jar" description="Cleans, compile, & builds jar
file." />

78      <!-- Javadoc -->
79      <target name="javadoc" description="Create javadoc.">
```

```
80          <mkdir dir="${doc.dir}" />
81          <javadoc packagenames="org.jlab.coda.*"
82                   sourcepath="java"
83                   destdir="${doc.dir}"
84                   use="true" >
85              <classpath refid="classpath" />
86          </javadoc>
87      </target>

88  </project>
```

Line 2: Define the project name, the base directory that all other defined directories are relative to, and the default target.

Lines 4-9: Define some useful directories. src.dir is the directory containing the source, build.dir is the directory containing all the files generated from java like class & jar files, build.classes is the directory containing all the class files, build.lib is the directory containing the generated jar file, and doc.dir is the directory containing the generated javadoc.

Lines 11-15: A path is defined which is later used as a classpath for compiling. It tells javac to look in java/jars for all needed jar files.

Lines 17, 19: Useful variables are defined.

Lines 22-33: The ***help*** target is defined. Typing *ant help* will print out the usage information for this build file.

Lines 35-44: The ***env*** target is defined. Typing *ant env* will print out the variables used in this build file.

Lines 46-50: The ***prepare*** target is defined. This creates all the necessary directories for the build.

Lines 52-56: The ***clean*** target is defined. Typing *ant clean* deletes all class files in and under the build.classes directory.

Lines 58-61: The ***cleanall*** target is defined. Typing *ant cleanall* will delete the whole build directory and the javadoc directory – basically all generated files.

Lines 63-68: The ***compile*** target is defined which is also the default target. Typing *ant conpile* or just *ant* will compile all the java source code into class files and store it in build.classes.

Lines 70-75: The ***jar*** target is defined. Typing *ant jar* compiles and creates a jar file of all the class files created..

Line 77: The ***all*** target is defined. Typing *ant all* will clean, compile, then make the jar file.

Lines 79-87: The ***javadoc*** target is defined. Typing *ant javadoc* will generate all the javadoc documentation.

All of the targets have an attribute called depends which defines which targets are prerequisite. A full description of the powers of ant is best left to a book on the subject. The following table lists the targets and what they do.

8.3.1. Table: ant targets and actions

Ant Command	Action
ant, ant compile	Compile all java source code
ant help	Print out usage
ant env	Print out value of build file variables
ant clean	Delete all class files
ant cleanall	Remove build and javadoc directories
ant jar	Compile and create jar file
ant all	Do clean, compile, then create jar file
ant javadoc	Create javadoc documentation
ant prepare	Create necessary directories