ANGLO-AUSTRALIAN OBSERVATORY AAO/DRAMA_MAKE_10
DRAMA Software Report 10
Version 1.0

Tony Farrell
14-Aug-2000

Creating Makefiles for Drama programs

Contents

Revisions:

V0.0 19-Jul-1993 Original Version.

V0.1 04-Dec-1995 General update. Added Fixed Lex/Yacc macros C++ stuff, DramaCheckTarget and local configuration files.

V1.0 14-Aug-2000 Added Java support targets.

V1.0.1 25-Aug-2000 Add `JDIRFLAG` macro and add `DramaJavaReleaseTo` macro. These combine to support Jeremy's style of java developement.

1 Introduction

DRAMA is a complex system which is available on machines with various architectures. Over the years, various techniques have arisen to help write portable code and **DRAMA** takes advantage of many of these.

Unfortunately, after writing your portable code, you often find that you have to go to some trouble to to actually build it. The preferred way is to use **make**. Normally, a makefile exists for each system being built. This makefile must be configured for the machine on which the software is being built. This works well for simple systems but causes problems for larger systems.

The most obvious large system in the style of **DRAMA** is the **X11** windowing system. **X11** is considerably larger then **DRAMA** and like **DRAMA** is arranged in a number of sub-system. Despite its size, it is remarkable for its portability. This author was surprised that after obtaining a copy of **X11**, he was able to build it by simply going to the appropriate directory and typing

```
make World
```

In addition, on a probably configured **X11** system, it is possible to grab almost any public domain **X11** program from the internet and build it with only-

```
xmkmf
make
```

Over time, I investigated the **X11** configuration system to determine if it is suitable for use in **DRAMA** and decided that the basic technique was what was required. This document first has a quick look at what is done in **X11**. It then describes the **DRAMA** approach and how to use it.

2 X11 Configuration Management

[1] describes the X11 configuration system. The basic principles it follows are-

- Use existing tools to do the build (e.g. *make*) where possible; writing complicated new tools simply adds to the amount of software that has to be bootstrapped.
- Keep it simple. Every platform has a different set of extensions and bugs. Plan for the least common denominator by only using the core features of known tools; don't rely on vendor-specific features.
- Providing sample implementations of simple tools that are not available on all platforms (e.g. a BSD-compatible *install* script for System V) is very useful.
- Machine-dependencies should be centralized to make reconfiguration easy.
- Site-wide options (e.g. default parameters such as directory names, file permissions, and enabling particular features) should be stored in only one location.

- Rebuilding within the source tree without losing any of the configuration information must be simple.
- It should be possible to configure external software without requiring access to the source tree.

The approach used by **X11** is to use existing tools (*make* and *cpp*), and a very simple program called *imake* (written by Todd Brunhoff of Tektronix). *Imake* uses the C pre-processor to combine a common template file , site and machine specific files , and source files known *Imakefiles* to produce a *Makefile*.

Each sub-system (library/package etc.) contains a file named *Imakefile*. *Imake* is run on *Imakefile* to produce a *Makefile* configured for the target machine. *Imake* is usually run using a simple script interface called *xmkmf* (For X MaKe MakeFile).

Below is the *Imakefile* used to build a manual page browser named *xman* (written by Chris Peterson of the MIT X Consortium, based on an implementation for X10 by Barry Shein):

```
DEFINES =  -DHELPFILE=\"$(LIBDIR)$(PATHSEP)xman.help\"
LOCAL_LIBRARIES =  $(XAWLIB) $(XMULIB) $(XTOOLLIB) $(XLIB)
SRCS =  ScrollByL.c handler.c man.c pages.c buttons.c help.c menu.c search.c \
        globals.c main.c misc.c tkfuncs.c
OBJS =  ScrollByL.o handler.o man.o pages.o buttons.o help.o menu.o search.o \
        globals.o main.o misc.o tkfuncs.o
INCLUDES =  -I$(TOOLKITSRC) -I$(TOP)

ComplexProgramTarget (xman)
InstallNonExec (xman.help, $(LIBDIR))
```

Lines such as `DEFINES = ...` will be copied directly to the resulting *Makefile*. The function syle lines, such as `ComplexProgramTarget` are C pre-processor macros. This particular one will be translated into the Make lines required to produce the program xman.

3 DRAMA Configuration

In addition to the problems faced by **X11**, **DRAMA** faces a couple of extra problems -

- The **X11** configuration system only works on Unix and similar systems. **DRAMA** is expected to work on **VMS**, which is sufficiently different in its command structure to cause problems
- **DRAMA** will also be using in embedded systems, such as **VxWorks**. This means we need to be able to configure and build software which will run on a different machine then the machine we are building it on.
- **DRAMA** uses a release version management system which is designed to ensure the correct versions of sub-systems are used, allowing reversion to previous versions if necessary.

My approach to solving these problems was to rewrite the **X11** configuration files (which set Project, Site and Target specific information) in a way appropriate to **DRAMA**. In addition, a new driver script - *dmkmf* (for Drama MaKe MakeFile) was written. In addition to running *imake* with a different set of configuration files, *dmkmf* also accepts options which allows *Makefiles* to be built for targets other then the machine on which it is being run. This allows for example, the generation of *Makefiles* for **VxWorks** targets on a sun host.

In addition, if you specify a **VMS** target, you can generate *descrip.mms* files suitable for use with the **VMS** *MMS* command.

4 Overview

Appendix D contains an example of a *dmakefile*. This one for a Generic Camera Package. This particular example is a bit more complex then the minimum necessary, which does not require the release targets. Netherless, it is a good example of how many *dmakefiles* for **DRAMA** programs will be written. We will work though this file and how to use it.

At this stage, I expect you to be somewhat familiar with *make* or a similar program, such as *MMS*.

First comments. You can you the standard C comment convention - `\* comment *\`. Comments made this way will not appear in the resulting *Makefile*. Alternately, comments introduced by a "#" on the first position of a line, indicating that the entire line is a comment, will be copied to the *Makefile* generated from this file.

Note that unless otherwise noted, where I refer to *make*, I also mean *MMS* under VMS. Likewise for *Makefile* and *descrip.mms*.

4.1 The configuration section

This first part of the file is the part between the special comments `#BeginConfg` and `#EndConfig` -

```
#BeginConfig
RELEASE=t0_1                    /* Release of this system       */
SYSTEM=gcam                     /* System name (for release     */
EmbeddedOnly(TWODF=/home/aaossc/tjf/scratch/2dF)
INCLUDES= DramaIncl EmbeddedOnly(-I$(TWODF)/drivers)

USERCCOPTIONS = AnsiCFull()     /* Enable Full Ansi C           */
#EndConfig
```

Everything on the left an an equals sign will become *make* macros.

The first two lines (RELEASE= and SYS=) set the release (or version) number (`t_1`) and the system name (`gcam`). Both the release number and system name must be a suitable as file names on the most restrictive system to which this package will be ported. As a result, use only lower case letters (which avoids unix/vms file name compatibility problems) and only use underscores (`_`) as word separators. These two lines are only needed if you later specify one of the *release*, *enable* or *dramadirs* target function (which we will mention later).

The rest of the configuration section will be *make* macro definitions, particularly macros used by the rules defined by the project. For example, `INCLUDES` and `USERCCOPTIONS` are used by the definition of C compilation rule. `TWODF` is used by `INCLUDES`. They are put in the configuration section since things put here are placed at the beginning of the resulting *Makefile*, ensuring that macro definitions occur before the macro is used.

The third line specified a *make* macro definition which is only used when you build a make file directed at an embedded system. The Make macro will be `TWO=/home/aaossc/tjf/scratch/2dF`. By wrapping the definition in the function `EmbeddedOnly()`, the line is only output when you are building an embedded system. Similar macros exist for a number of other possibilities.

The macro `INCLUDES` specifies the directories in which to search for include files during C compilations. Note that the format `-Idir` will work even on VMS systems. This is due to the way the C compiler is run by the resulting *descrip.mms* file. In this case, we only use that format for the stuff in the `EmbeddedOnly()` function.

The function `DramaIncl` will insert the appropriate line to compile using the **DRAMA** include files[1].

You can use the function `IDir()` to add other directories. It considers its argument to be either a logical name (**VMS**) or a environment variable (**unix**) pointing to a directory which you wish added to the include file list.

The Make macro `USERCCOPTIONS` is used to add extra options to the *cc* command. In this case, the function `AnsiCFull()` is used, which causes the compiler to use strict Ansi-C (if possible). What this translates to depends on the system and compiler in use. You could also use the function `AnsiC()` for ansi C but not strictly, or `TraditionalC()` for traditional (K&R) C.

4.2 Rules, Objects and Sources

The line below enables the normal set of Make rules for compiling C programs. All *dmakefiles* should have one, but it is made explicit so that they can be overridden if necessary.

```
NormalRules()
```

Next we specify some *make* macros to be used later in the program -

```
OBJECTS = Obj(gcam) Obj(gcamcentroid) Obj(vfg)
EmbeddedOnly(GRAB_OBJS = Obj($(TWODF)/drivers/grabberDrv)
                                           Obj($(TWODF)/drivers/pixel_box))
SRC1=  gcam.c gcamcentroid.c vfg.c
```

[1]Under unix, this translates to `‘$GIT_DIR/git_cc‘`. Under **VMS**, it translates to `-IDRAMA_INCLUDES` .

As we will see later on, `OBJECTS` is the list of objects which make up the *gcam* system library. The name `OBJECTS` is not significant, it is just what is specified later. All object files are specified using the `Obj()` function. This produces the correct name for the corresponding object file on the target machine [2].

`GRAB_OBJS` is used later, when the target is an embedded system. Note that this need to be on the one line in the actual file, since you cannot break function arguments over more then one line.

Next we have the definition of the list of source files. Each file should define *make* macros of the form `SRCn=`, where n is 1 to 5. The `SRCn` *make* macros are used in a target named *depend.* When you do *make depend*, the source file lists are run though the *makedepend* program, which adds a list of include file dependencies to the makefile. This will ensure that *make* correctly updates targets when include files are changed. Each `SRCn` macro should be restricted to about 2 lines of file names. When `SRC1` fills about two lines, start with `SRC2` etc.

4.3 The All target

The first target in a *Makefile* is the default target, which is built when no argument is specified to the *make* command. By convention, there is often a target named *All*, which builds the system and is also the default target. The line-

```
DummyTarget(All, includes Lib(gcam))
```

defines the target *All.* The function `DummyTarget` takes two arguments (separated by commas). The first argument is the target to be built - *All.* The second argument is a space separated list of dependencies. In this case *includes* and `Lib(gcam)`. The `DummyTarget` function produces a *make* dependency with no update rule, i.e. a dummy target. *includes* is just another target. The function `Lib()` is used to generate an object library name[3].

4.4 The includes target

Many of my *dmakefile*s have a target named *includes*, specified in a `DummyTarget()` macro -

```
DummyTarget(includes, gcam_err.h gcam_err_msgt.h vfg_err.h vfg_err_msgt.h
                                          gcam.h vfg.h gcaminfocreate.h)
```

The dependencies for this target is just a list of the includes files which may have to be built or fetched from *SCCS*. It will not be needed when *make depend* has been done, but I find it convenient. Note that this definition should be one only one line in the actual *dmakefile*, since you cannot spead function arguments over more then one line.

[2]Under **unix**, `Obj(file)` is converted to “file.o”, while under **VMS**, it is converted to “file.obj”.

[3]Under **VMS**, `Lib(gcam)` translates to “gcam.olb”, while under **unix**, it translates to “libgcam.a”.

4.5 Include file generate

Under drama, some include files are generated automatically from other sources. The lines-

```
ErrorIncludeFiles(gcam_err)
ErrorIncludeFiles(vfg_err)
```

cause error message include code files to be generated. In the first case, the files "gcam_err.h" and "gcam_err_msgt.h" are generated by running the **DRAMA** *messgen* command on "gcam_-err.msg".

In a similar way, the file "gcaminfocreate.h" is generated from "gcaminfo.h" using the **DRAMA** *sdsc* command (sds compiler) with the line

```
SdsIncludeFile(gcaminfocreate.h, gcaminfo.h)
```

4.6 An Object library target

One of the major things we build are object libraries. The following line-

```
ObjectLibraryTarget(gcam, $(OBJECTS),)
```

generates the gcam object library. The actual filename for the library is based on the first argument to the function `ObjectLibraryTarget()`. It is the same as would have been generated by function `Lib()`. The second argument is a space separated list of object files to be placed in the library. In this case, we use the *make macro* "`OBJECTS`", which we defined earlier. The third argument is a list of other dependencies for the library. Normally this is empty.

This function will create the library, load the objects into it and if necessary, run *ranlib*.

4.7 A DRAMA program target

The gcam system contains a simple **DRAMA** program, named *vfg*. This is built using the gcam library, the **DRAMA** supplied *git* library and the drama libraries themselves.

The line to build this program is

```
DramaProgramTarget(vfg,Obj(vfgmain),Lib(gcam),LinkLib(gcam) $(GRAB_OBJS) $(LIB_GIT),)
```

The function `DramaProgramTarget()` will build a program against the **DRAMA** libraries. It takes five arguments, being -

1. The target program. (no file extensions). In this case - *vfg*.
2. A space separated list of object files to build the program from. In this case, only `vfgmain` is required.

3. A space separated list of libraries on which the program is dependent. In this case, the program is dependent on the gcam library (i.e., make ensures the gcam library is up to date as part of building this target).
4. A space separated list of local libraries to link against and objects which are not to be built automatically. These will normally include the libraries the target is dependent upon, but specified with the function `LinkLib()` instead of `Lib()`. This is required as the link specification is different from the dependency specification on some machines. In this case, we link the library gcam and the objects specified in the *make* macro `GARB_OBJS`. We also specify the git library using the *make* macro `LIB_GIT`, which is set up automatically.
5. The last argument is normally used to link extra system libraries, such as the maths library under **unix**. In this case, we don't need any.

It is important to understand the order in which objects and libraries are linked, as the correct order must be used. The order is

```
argument2 argument4 $(DRAMA_LIBS) $(LDLIBS) argument5
```

Where `DRAMA_LIBS` and `LDLIBS` are macro macros defined automatically[4]. As a result, anything which uses the **DRAMA** libraries, such as the git library, must be in argument 2 or argument4. You should always ensure they order of libraries is such at the library containing a routine is searched after the first routine which invokes the routine is linked in.

There also exists a function named `NormalProgramTarget()`, which only differs from `DramaProgramTarget()` in that is does not link against the **DRAMA** libraries. Use it for simple, non **DRAMA** based programs.

4.8 The Release Targets

DRAMA programs normally reside in a particular directory structure. This structure allows the implementation of a version control system and provides a standard organisation. It is described in [3]. Given this organisation, it proves possible to automate the release of software into the release directories. Various functions are implemented to support this. Gcam uses the following functions-

```
DramaReleaseCheck()
DramaReleaseCommon(gcam_err.h gcam_err_msgt.h gcam.h gcaminfocreate.h)
DramaReleaseCommon(vfg_err.h vfg_err_msgt.h vfg.h)
DramaReleaseTarget(vfg,Lib(gcam),,)
DramaReleaseDramaStart()
```

The function `DramaReleaseCheck()` will check to ensure you are not overwritting a previous release. If you are, it will prompt for confirmation. For this to work, it must be the first `DramaRelease` macro.

[4]`DRAMA_LIBS` links the core drama libraries. `LDLIBS` is defined in a complex manner but normally the resulting definition is empty.

The function `DramaReleaseCommon()` takes one argument which is a space separated list of files. It generates a target named *release*. This is a double colon target - which means that a *Makefile* can have multiple ways of updating the target. Hence we can have the multiple invocations of `DramaReleaseCommon()`. On a unix machine, this particular function creates, if it does not exist, the directory *Project*/`local/gcam/`*release*. Here, *Project* is the location of the drama project and *release* is the definition of the make macro `RELEASE` (see configuration section above). The files specified are then copied to the directory.

The function `DramaReleaseTarget()` works in a similar way. In this case, the directory is *Project*/`local/gcam/`*release*/*target*, where *target* is the target type. Target types are described in more detail later, but basically, this directory is the location for files specific to a particular target machine, where as the common directory is common to all targets. This function takes four arguments-

1. A space separated list of executable programs. In this case, just *vfg*.
2. A space separated list of object libraries. In this case, the gcam library.
3. A space separated list of executable scripts.
4. Any other files.

Any but not all of these may be null. The use of multiple arguments in this function allows, for example, *ranlib* to be run on libraries after they are moved and scripts to be set executable.

The last function - `DramaReleaseDramaStart()` releases the files to be used by the *dramastart* command. Under **VMS**, this is the file "`GCAM_DRAMASTART.COM`". Before being released, lines containing "`RELEASE=`" will be replaced by "`RELEASE=`*release*" where *release* is the value of the make macro `RELEASE`. Under **unix**, a file named gcam_dramastart.rel containing the line "`RELEASE=`*release*" will be **created**.

4.9 Enabling the release

The function `DramaEnable()` generates the target *enable*. It causes the file generated by the `DramaReleaseDramaStart()` to be copied to *Project*/`local/gcam`. Once here, it will be picked up by the *dramastart* command, making it available to users.

4.10 DramaDirs

The last function - `DramaDirs()` is a special target. It can be used by shell scripts which automatically build a number of sub-systems. When doing this, it is common that some sub-systems will be dependent on other sub-systems. You will want to do the equivalent of the *dramastart* command between building each sub-system. This macro will generate a target named *dramadirs*. If your **unix** build script does-

```
eval ‘make dramadirs‘
```

after making a release of gcam, then the same effect will occur as if you had executed dramastart after having enabled gcam. Why not just execute dramastart? Because it will not return to your script.

Under **VMS**, you will need a command procedure which does something like this-

```
$mms dramadirs
$@gcam_dir:gcam_dramastart
```

to get the same effect.

For example, the gcam system is dependent on the git system. If you want to build git and gcam automatically, you need to enable git before building gcam.

4.11 Using it

Now that we have a *dmakefile*, we need to generate the appropriate makefile. Currently there are three possible targets (VAX/VMS, Sun/Sparc and VxWorks 680x0) and two host machines (VAX/VMS and Sun/Sparc). We will examine how to build systems on a Sun/Sparc for either the Sun itself or for a VxWorks target. We will also look a targeting VAX/VMS.

4.11.1 Targeting a Sun/Sparc

Assuming you are in the directory containing the system you which to build (say gcam), you must first enable drama with the *dramastart* command (normally `~drama/dramastart`

You must then enable the various drama commands by sourcing a file. If you are running a *sh* compatible shell, do

```
. $DRAMA/drama.sh
```

If you are running a *csh* compatible shell, do

```
source $DRAMA/drama.csh
```

You now execute the command

```
dmkmf
```

This is a script which will run the *imake* program. You must always invoke *imake* via this script, never directly.

You now have your *Makefile.* Type *make* to build the *All* target. Type *make release* to release the system. Type *make enable* to enable the release. Type *make clean* to tidy up.

It is not normally neccessary or desirable for you to be concerned with what is actually put in the *Makefile.* They are somewhat more complex then the standard hand-generated *Makefile.*

4.11.2 Targeting a VxWorks 68020

Assuming a sun host, the procedure is the same as above, except that you must specify the argument `vw68k` to the *dramastart* command.

4.11.3 Targeting VAX/VMS

VMS is a bit more complex. Currently, the *imake* program does not exist under **VMS**. As a result, you must take your *dmakefile* to a unix machine to create a *descrip.mms* file. You can then take the *descrip.mms* file back to the **VMS** machine and run *MMS*.

The procedure to create the *descrip.mms* file is similar to that above for a sun host. The difference is the specification of the *dmkmf* command. Use-

```
dmkmf -f -t vaxvms
```

This will generate your *descrip.mms* file. The `-f` flag is optional but is recommended. It causes the file to be generated using rules which result in a much faster build.

Once you have your *descrip.mms* back on the vax, go to the directory and type *DRAMASTART* followed by *MMS* to build the target *All*. Type *MMS RELEASE* to release the system. Type *MMS ENABLE* to enable the release. Type *MMS CLEAN* to tidy up.

It is hoped that a **VMS** version of *imake* can be generated in the future. In the meantime, it is suggested that having your **VMS** discs NFS mounted on a **unix** machine reduces the work involved here.

5 How it works

Although *imake* is a fairly powerful took, it is a very simple program. All of the real work is preformed by the template and configuration files. It is here that there is a major difference between the **DRAMA** use of *imake* and the **X11** use of *imake*

The *imake* driver program - *dmkmf* first converts its own arguments into a set of C preprocessor macro definitions. It then runs the *imake* program specifying the template file *imake.tmpl* (in the **DRAMA** *config* directory) as the input file, the macro definitions it has worked out and the **DRAMA** *config* directory as the default source for C preprocessor include files. *Imake* does a bit of processing on the user's *dmakefile* and then passes *imake.tmpl* to the C preprocessor, along with the macro definitions generated by *dmkmf* and one defining the location of the processed version of the user's *dmakefile*

So in effect, *imake.tmpl* is the driver file for the C preprocessor. This template does the following to create a *Makefile*

1. Includes the file *Target.sel*. Normally, *imake.tmpl* makes use of predefined C preprocessor macros for each architecture to work out which machine it is running on. The *Target.sel* file first undefines all such macros and then uses a macro set by *dmkmf* to define macros for an architecture other then the machine it is running on - i.e. the target architecture. If no target architecture was defined, then the host machine is used.

2. Includes the file Platform.sel. This file determines the name of the configuration file for the target architecture.
3. Includes the target architecture specific configuration file. This program includes defines C preprocessor macros which are dependent on the target machine and host operating system. It is normally named *machine*.cf, where machine is something like *sun*, *vaxvms* etc.
4. Includes the file *site.def.* This file specified site specific information such as the location of the drama project. Note that some site specific information is also set in the target specific configuration file, since some such information may be machine specific.
5. Includes the file *Project.defs.* This sets C macros which defines the various functions provided by the configuration utility and also macros which define the location and names of various commands. These definitions are often overriden by defining them in the machine specific configuration file. For example, on **VMS**, many of the rules defined in *Project.defs* must be overridden.
6. Includes the file Imake.defs. This file ensures things required by *imake* are defined to default values, if not already defined in the previous files.
7. Includes the file *ImakeBasicS1.tmpl*, which defines the first part of the actual makefile, using previously defined C macros. It is this file which defines the make macros in the first part of the file.
8. Includes the modified version of the user's *dmakefile.*
9. Includes the file *ImakeBasicS2.tmpl*, which defines the bottom part of the actual makefile. Here are defined the make rules for running the targets *clean* and *depend.*

Which the exception of the user's *dmakefile*, standard C preprocessor constructs are used throughout these files.

After running *imake*, the *dmkmf* script will move the configuration section to the beginning of the make file using a *awk* script and run *sed* to remove the sequence by "`%%\`", which is used to escape quote and double quote characters.

By isolating the machine and site specifics from the programer, this tool allows properly configured *Makefiles* to be regenerated quickly and correctly.

5.1 VMS Tricks

VMS is a bit harder to handle then **unix**. There are three major problems

1. Most of the **VMS** command we use take there arguments are a comma separated list, whereas in **unix**, the arguments are normally a space separated list. This makes it hard to translate a **unix** macro definition into something which can be run on a **VMS** machine.
2. **Unix** compilers use `-I` and `-D` to specify include directories and C Pre-Processor defines respectively. **VMS** uses `/INCLUDE=` and `/DEFINE=`.

3. **VMS** tends to be a slower at running *MMS*, then **unix** is at running *Make*. The major reason is the longer image startup time on **VMS**.

Problem 1 has be avoided by running the complers and linker via an intermediate program - `RUNEM`. This program resides `DRAMAUTIL_DIR:` and is responsible for converting **unix** style argument lists into **VMS**style lists. In addition, it also handles problem 2 by converting unix style include directory and macro definitions into the **VMS** style.

Problme 3 is optionally handled when the `-f` flag is specified to *dmkmf*. This causes the compilation of C programs to be delayed so that one run of the compler can process several source files. This is handled by a pair of command scripts in `DRAMAUTIL_DIR`.

6 Writing dmakefiles

Having read section 4, you will have a good idea of what it will look like. Appendix B details the *imake* functions and *make* macros which you are likely to need. Since most *dmakefiles* look similar to the one in appendix D, you should grab a copy of it[5] and modify it as you require.

7 Local configuration

It is often nessary to locate files local to a configuration or to alter command options for a particular machine.

Consider an example from the AAO Two degree field project. At one time the system level software was located in "/instsoft/2dF" at Epping an in "/home/aatssb/2dF" at the telescope. It would be nice to be able to move dmakefile's between Epping and the telescope without having to edit them, but this difference in location made edits necessary. The **DRAMA** configuration system can assist.

If a the file "`$DRAMA_LOCAL/drama_local.cf`" exists, then it is included when *Imake* runs *cpp*, after the target has been determined and project specific configuration read. It can be used define local macros. In the above example, it can be used to define a macro which locates the AAO 2dF system software.

In addition, you may wish to change the configuration based on the node you are running on. In the above example, we could actually use the one copy of "`$DRAMA_LOCAL/drama_local.cf`" at both sites if a macro can be defined to indicate which site we are running at. This can be handled by creating an executable program named "`$DRAMA_LOCAL/drama_local.opts`". This is normally a script (note that you must set the protection such that the user can execute it). The output from this program should be a sequence of options to be passed to *imake*. These are normally macro definitions.

The AAO copy of "`$DRAMA_LOCAL/drama_local.opts`" contains

[5] Available as part of the demos which come with drama.

```
NODE=‘uname -n‘
case "$NODE" in
aao*)
    echo -DAAO_EPPING
    ;;
aat*)
    echo -DAAO_COONA
    ;;
*)
    ;;

esac
```

This results in *imake* being passed “`-DAAO_EPPING` if we are running at AAO Epping (where all node names start with `aao`) and “`-DAAO_COONA` if we are running at the telescope (where all node names start with `aat`). This leads to the following implementation of “`$DRAMA_LOCAL/drama_local.cf`”

```
#ifdef AAO_EPPING
#define AAO2dFDir /instsoft/2dF
#elif defined(AAO_COONA)
#define AAO2dFDir /home/aatssb/2dF
#endif
```

It is also possible to modify compiler options etc. if required.

A Programs

This appendix documents programs mentioned in this document.

A.1 dmkmf — Generate a Makefile from a dmakefile.

Function: Generate a Makefile from a dmakefile.

Synopsis: dmkmf [options]

Description: Run imake on the the file dmakefile in the current directory to generate a Makefile (except for vms targets when a descrip.mms file is generated)

Options:

-t target Sets the target. It defaults to the current value of `DRAMA_TARGET`. If this is not defined it defaults to the current host. Possible values are

sun4	A sun sparcstation sun4_solaris
decstation	A decstation running ultrix
vw68k	A 680x0 running VxWorks
vwppc	A PowerPC running VxWorks
mv167	A 680x0 running VxWorks
mv2700	A PowerPC running VxWorks
vaxvms	A vax running vms
alphavms	A alpha running vms

-l Set verbose mode in Makefile. Lots of comments in the resulting Makefile.

-n Make dmkmf reverts to it old noisy way of working - verbose details spilled to stdout.

-f Only effective with `VMS` targets. Causes rules to build `VMS` systems quickly to be defined, but the order of the build will change.

-g Enable debugging flags

`-O` Enable Optimization flags

-v version Define the macro Version_version, where version is normally something like `5_0` This enables selection of different target versions. It must be supported by the target cofiguration file. If not defined, use value of `DRAMA_VERSION` if this is not defined, defaults to a site specific default. Multiple specifications allowed and can also be used to pass flags to the dmakefile.

-nodepend Don't generate the include file dependency check code.

-noautotest Don't automatically test as part of build.

Author: Tony Farrell, `AAO`

B Imake Functions

This section describes the various Imake functions which you may using in your *dmakefile* and the Make macros defined and used by the resulting makefile. There are three types of items described here

Functions translated by the *imake* into appropriated lines in the *Makefile*.

Macros to set. These are *make* macros which can be or should be set by the user to appropriate values.

Macros to invoke. These are em make macros setup automatically which can be invoked by the user.

B.1 File specification

This Functions allow you to specify various files, such as executables and objects, in such a way as to be operating independent.

B.1.1 Exe

Type: Function

Description: Specifies that its argument is an executable file.

Call:
Exe(file)

Arguments:

file	The name of the executable.

B.1.2 Obj

Type: Function

Description: Specifies that its argument is an object file.

Call:
Obj(file)

Arguments:

file	The name of the object.

B.1.3 Lib

Type: Function

Description: Specifies that its argument is an object library file.

Call:
Lib(file)

Arguments:

file	The name of the library.

B.2 System Configuration

This section specifies functions which are replaced by their arguments only under certain conditions. Otherwise, the will be replaced by a blank line.

B.2.1 VmsOnly

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is a machine running **VMS**.

Call:
VmsOnly(rule)

B.2.2 NotVms

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is **NOT** a machine running **VMS**.

Call:
NotVms(rule)

B.2.3 UnixOnly

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is a machine running **unix**.

Call:
UnixOnly(rule)

B.2.4 NotUnix

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is **NOT** a machine running **unix**.

Call:
NotUnix(rule)

B.2.5 EmbeddedOnly

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is an embedded system (such as **VxWorks**)

Call:
EmbeddedOnly(rule)

B.2.6 NoEmbedded

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is **NOT** an embedded system (such as **Vx-Works**).

Call:
NoEmbedded(rule)

B.2.7 StarlinkOnly

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is a machine that has the starlink software environment installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
StarlinkOnly(rule)

B.2.8 NoStarlink

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target machine does **NOT** have starlink software environment installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
NotStarlink(rule)

B.2.9 MotifOnly

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is a machine that has the Motif **X11** window environment installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
MotifOnly(rule)

B.2.10 NoMotif

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target machine does **NOT** have the Motif **X11** window environment installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
NoMotif(rule)

B.2.11 TclOnly

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is a machine that has the Tcl **X11** system installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
TclOnly(rule)

B.2.12 NoTcl

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target machine does **NOT** have the Tcl installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
NoTcl(rule)

B.2.13 TkOnly

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is a machine that has the Tk **X11** window environment installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
TkOnly(rule)

B.2.14 NoTk

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target machine does **NOT** have the Tk **X11** window environment installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
NoTk(rule)

B.2.15 CPlusPlusOnly

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is a machine that has a C++ compiler installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
CPlusPlusOnly(rule)

B.2.16 NoCPlusPlus

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target machine does **NOT** have a C++ complier installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
NoCplusPlus(rule)

B.2.17 FortranOnly

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target is a machine that has a fortran compiler installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
FortranOnly(rule)

B.2.18 NoFortran

Type: Function

Description: The argument to this function will be examined for further functions and the result copied to *Makefile* only if the target machine does **NOT** have a fortran complier installed. (assuming **DRAMA** is configured correctly - see the [4])

Call:
NoFortran(rule)

B.3 Drama Libraries Macros

B.3.1 NormalCRules

Type: Function

Description: Include the rules necessary for building objects from C language source files. This macro should be before the first target.

Obsolete. Please replace this by NormalRules().

Call:
NormalCRules()

B.3.2 NormalRules

Type: Function

Description: Include the rules necessary for building objects from C and Fortran language source files. This macro should be before the first target and replaced *NormalCRules()*

Call:
NormalRules()

B.3.3 JavaRules

Type: Function

Description: Include the rules necessary for building Java program. This macro should be before the first target.

Call:
JavaRules()

B.3.4 AnsiC

Type: Function

Description: This function expands to C compiler options which enable acceptance of Ansi-C code. This will only work if the compiler supports it.

Call:
AnsiC()

B.3.5 AnsiCFull

Type: Function

Description: This function expands to C compiler options which enable acceptance of Ansi-C code and also enables options to cause warning to be output when potential non-portable code is used. This will only work if the compiler supports it.

Call:
AnsiCFull()

B.3.6 TraditionalC

Type: Function

Description: This function expands to C compiler options which enable acceptance of Traditional-C and also enables options to cause warning to be output when potential non-portable code is used. This will only work if the compiler supports it.

Call:
TraditionalC()

B.3.7 ExtraCWarnings

Type: Function

Description: This function expands to C compiler options which enable acceptance of Traditional C code and also enables options to cause warnings to be output when potential non-portable code is used. This will only work if the compiler supports it.

Call:
ExtraCWarnings()

B.3.8 USERCCOPTIONS

Type: Make Macro to be set.

Description: This macro is used by the rule which updates an object file from a C source file. It should be used to indicate the any required options to the C compiler. It is normal set to one of `AnsiC()`, `AnsiCFull()` or `TraditionalC()`.

Call:
USERCCOPTIONS=

B.3.9 AnsiCC

Type: Function

Description: This function expands to C++ compiler options which enable acceptance of Ansi-C++ code. This will only work if the compiler supports it.

Call:
AnsiCC()

B.3.10 AnsiCCFull

Type: Function

Description: This function expands to C++ compiler options which enable acceptance of Ansi-C++ code and also enables options to cause warning to be output when potential non-portable code is used. This will only work if the compiler supports it.

Call:
AnsiCCFull()

B.3.11 USERCCCOPTIONS

Type: Make Macro to be set.

Description: This macro is used by the rule which updates an object file from a C++ source file. It should be used to indicate the any required options to the C++ compiler.

(Note the extra C over USERCCOPTIONS above).

Call:
USERCCCOPTIONS=

B.3.12 IDir

Type: Function

Description: Generate an include directory specification for a directory which is known by a logical name on **VMS** machines or an environment variable on **unix** machines.

Call:
IDir(directory)

B.3.13 LinkLibDir

Type: Function

Description: Generate an library specification for a library in a directory which is known by a logical name on **VMS** machines or an environment variable on **unix** machines.

Call:
LinkLibDir(directory,library)

Arguments:

directroy	The logical name/environement variable name.
library	The library name, as would be supplied to Lib().

B.3.14 DRAMA_INCL

Type: Make macro to be invoked.

Description: This macro is defined automatically to a C compiler option which will specifies the include directories for all the core drama libraries.

Call:
$(DRAMA_INCL)

B.3.15 TCLTK_INCL

Type: Make macro to be invoked.

Description: This macro is defined automatically to a C compiler option which will specifies the include directories for the Tcl and Tk libraries.

Call:
$(TCLTK_INCL)

B.3.16 INCLUDES

Type: Make Macro to be set.

Description: This macro is used by the rule which updates an object file from a C source file. It should be used to indicate the directories which should be searched for include files. Note the the format `-I`*directory* will work in all cases. (even on **VMS**).

Call:
INCLUDES=

B.3.17 Define

Type: Function

Description: Generate code designed for defining C preprocessor macros ensuring case is maintained. The definition is specified in the form `macro=definition` or just `macro` to define a macro to 1.

Call:
Define(macrodef)

B.3.18 StringDefine

Type: Function

Description: Generate code designed for defining C preprocessor macros to string values in the command line to the C compiler.

Call:
StringDefine(macro,string)

Arguments:

macro	The name of the macro to define. Case will be maintained.
string	The string to define the macro to.

B.3.19 DEFINES

Type: Make Macro to be set.

Description: This macro is used by the rule which updates an object file from a C source file. It should be used to indicate the C preprocess macros to be defined. Note the the format `-D`*name=definition* will work in all cases. (even on **VMS**), but case and strings can be a problem. Under VMS, the macro name will be converted to lower case, unless wrapped in one of the special functions above. Likewise, you should use the *StringDefine()* function above to define string macros.

Call:
DEFINES=

B.3.20 JUSEROPTIONS

Type: Make Macro to be set.

Description: This macro is used by the rule which updates a JAVA class file from a Java source file. It should be used to indicate any extra options to the "javac" command.

Call:
JUSEROPTIONS=

B.3.21 JCLASSPATH

Type: Make Macro to be set.

Description: This macro is used by the rule which updates a java class file from a Java source file. It should be used to override the classpath if required.

Note,you must include the `-classpath` option. E.g.

```
JCLASSPATH=-classpath dir1:dir2
```

Call:
JCLASSPATH=

B.3.22 JDIRFLAG

Type: Make Macro to be set.

Description: This macro is used by the rules which updates a java class file from a Java source file. It should be used to override the "`-d`" specification if required.

By default, the value of this is "`-d .`". This indicates that class files for any package are created in a sub-directory of the current directory, and the sources are found in the current directory. See the `javac` command documentation for other possiblities.

Set this value before you invoke `JavaRules()`.

Call:
JDIRFLAG=

B.3.23 JavaPackageClassesMacro

Type: Function

Description: Used to create a make macro the value of which a list of fully specified Java package class members, suitable for use with the `JavaPackageTarget()` macro.

Call:
JavaPackageClassesMacro(macro, package, contents)

Arguments:

macro	The name of the make Macro to be set.
package	The name of the Java package
contents	The classes which make up the Java package.

B.4 Objects, Libraries and Executables

B.4.1 CObject

Type: Function

Description: Expands to a *make* rule which build an object from a C language source file. This function is normally not necessary as internal *make* rules will do this automatically. This function should be used when the Object is dependent on other files as well as the C language source.

Call:
CObject(object,depends)

Arguments:

object	The name of the object file to create. *object*.c is used as the C language source file.
depends	Additional dependencies (such as include files).

B.4.2 FortranObject

Type: Function

Description: Expands to a *make* rule which build an object from a Fortran language source file.

Call:
FortranObject(object,depends)

Arguments:

object	The name of the object file to create. *object*.f is used as the Fortran language source file on **unix** machines and *object*.FOR on **VMS** machines.
depends	Additional dependencies (such as include files).

B.4.3 ObjectLibrary

Type: Function

Description: Generates an object library from a list of objects. Note that there are multiple versions of this call, which only differ in that in all but the basic call, there are multiple lists of objects on which the library is dependent. This is required as a single object list should not exceed about 200 characters.

Call:

ObjectLibrary(libname, objlist, depend)
ObjectLibrary2(libname, objlist, objlist, depend)
ObjectLibrary3(libname, objlist, objlist, objlist, depend)
ObjectLibrary4(libname, objlist, objlist, objlist, objlist depend)

Arguments:

libname	The name of the library to create. It is automatically wrapped in the `Lib()` function, so you should not do this yourself.
objlist	A space separated list of object files on which the library is dependent. Each object should be wrapped in the Obj() function. An object list should not exceed about 200 characters.

B.4.4 NormalProgramTarget

Type: Function

Description: This function will build the specified program.

Call:

NormalProgramTarget(program,objects,deplibs, localibs, syslibs)

Arguments:

program	The program to build
objects	A list of objects program is dependent on. These objects will be linked into the program.
deplibs	A list of libraries program is dependent on. These libraries will *NOT* be link to the program, you must use *locallibs* to specify which libraries are to be linked.
locallibs	A list of libraries and objects to link against.
syslibs	A List of system libraries to link against.

B.4.5 DramaProgramTarget

Type: Function

Description: This function will build the specified program and links it against the **DRAMA** libraries.

Call:

DramaProgramTarget(program,objects,deplibs, localibs, syslibs)

Arguments:

program	The program to build
objects	A list of objects program is dependent on. These objects will be linked into the program.
deplibs	A list of libraries program is dependent on. These libraries will *NOT* be link to the program, you must use *locallibs* to specify which libraries are to be linked.
locallibs	A list of libraries and objects to link against.
syslibs	A List of system libraries to link against.

B.4.6 CPPProgramTarget

Type: Function

Description: This function will build the specified program, assuming a C++ main routine.

Call:
CPPProgramTarget(program,objects,deplibs, localibs, syslibs)

Arguments:

program	The program to build
objects	A list of objects program is dependent on. These objects will be linked into the program.
deplibs	A list of libraries program is dependent on. These libraries will *NOT* be link to the program, you must use *locallibs* to specify which libraries are to be linked.
locallibs	A list of libraries and objects to link against.
syslibs	A List of system libraries to link against.

B.4.7 DramaCPPProgramTarget

Type: Function

Description: This function will build the specified program and links it against the **DRAMA** libraries, assuming a C++ main routine.

Call:
DramaCPPProgramTarget(program,objects,deplibs, localibs, syslibs)

Arguments:

program	The program to build
objects	A list of objects program is dependent on. These objects will be linked into the program.
deplibs	A list of libraries program is dependent on. These libraries will *NOT* be link to the program, you must use *locallibs* to specify which libraries are to be linked.
locallibs	A list of libraries and objects to link against.
syslibs	A List of system libraries to link against.

B.4.8 JavaProgramTarget

Type: Function

Description: This function will build the specified Java program.

Call:
JavaProgramTarget(program,depends)

Arguments:

program	The program to build
depends	A list of targets the program is dependent on.

B.4.9 JavaPackageTarget

Type: Function

Description: This function will build the specified Java package

Call:
JavaPackageTarget(package,depends)

Arguments:

program	The program to build
depends	A list of targets the program is dependent on. This should be a space separated list of the class files the package is dependent on. It should have the following format `package/classname1.class  package/classname2.class` You can create a make Macro with this format using the JavaPackageClassesMacro().

B.5 Lex and Yacc

B.5.1 YaccObject

Type: Function

Description: Creates a series of rules where an object file is dependent on a C source file and include file which were generated by the yacc utility. In a **unix** development environment, rules to generate the C code from the yacc source are output, as well as rules to generate the the object from the C code. On a **VMS** host, it is assumed the C code already exists (since **VMS** does not have yacc).

Call:

YaccObject(parser,depends)

Arguments:

parser	The name of the parser file from which the C source and object are created. The actual file must have a file type of `.y`, which should not be included in this call.
depends	Additional files the lexer object is dependend on, such as include files.

B.5.2 LexObject

Type: Function

Description: Creates a series of rules where an object file is dependent on a C source file which was generated by the lex utility. In a **unix** development environment, rules to generate the C code from the lexer source are output, as well as rules to generate the the object from the C code. On a **VMS** host, it is assumed the C code already exists (since **VMS** does not have lex).

Call:

LexObject(lexer,depends)

Arguments:

lexer	The name of the lexer file from which the C source and object are created. The actual file must have a file type of `.l`, which should not be included in this call.
depends	Additional files the lexer object is dependend on, such as include files.

B.5.3 YaccObjectFixed

Type: Function

Description: Creates a series of rules where an object file is dependent on a C source file and include file which were generated by the yacc utility. In a **unix** development environment, rules to generate the C code from the yacc source are output, as well as rules to generate the the object from the C code. On a **VMS** host, it is assumed the C code already exists (since **VMS** does not have yacc).

Unlike the plain `YaccObject` macro, This macro allows you to specify a prefix to the Yacc and Lex global function and variable names. This prefixed is applied using a sed script to edit C code which results from running Yacc. This technique allows you to link several Yacc/Lex based modules into one program. Yacc and Lex identifiers are normally named begining with `yy`, e.g. `yyparse()`. The fixed name will be `prefix-Yyparse()`. You should choose your prefix such that this is a valid C identifier.

Call:

YaccObjectFixed(parser,depends,prefix)

Arguments:

parser	The name of the parser file from which the C source and object are created. The actual file must have a file type of `.y`, which should not be included in this call.
depends	Additional files the lexer object is dependend on, such as include files.
prefix	The user selected prefixed to be applied to global Yacc and Lex identifiers.

B.5.4 LexObjectFixed

Type: Function

Description: Creates a series of rules where an object file is dependent on a C source file which was generated by the lex utility. In a **unix** development environment, rules to generate the C code from the lexer source are output, as well as rules to generate the the object from the C code. On a **VMS** host, it is assumed the C code already exists (since **VMS** does not have lex).

Unlike the plain `Lexbject` macro, This macro allows you to specify a prefix to the Yacc and Lex global function and variable names. This prefixed is applied using a sed script to edit C code which results from running Lex. This technique allows you to link several Yacc/Lex based modules into one program. Yacc and Lex identifiers are normally named begining with `yy`, e.g. `yyparse()`. The fixed name will be `prefix-Yyparse()`. You should choose your prefix such that this is a valid C identifier.

Call:
LexObjectFixed(lexer,depends,prefix)

Arguments:

lexer	The name of the lexer file from which the C source and object are created. The actual file must have a file type of `.l`, which should not be included in this call.
depends	Additional files the lexer object is dependend on, such as include files.
prefix	The user selected prefixed to be applied to global Yacc and Lex identifiers.

B.5.5 YACC_LIB

Type: Make Macro to be invoked.

Description: This macro is defined automatically to point to the libraries which must be included in a program which uses objects generated from *yacc* sources. Normally, this macro as a system library when building such programs.

Call:
$(YACC_LIB)

B.5.6 LEX_LIB

Type: Make Macro to be invoked.

Description: This macro is defined automatically to point to the libraries which must be included in a program which uses objects generated from *lex* sources. Normally, this macro as a system library when building such programs.

Call:
$(LEX_LIB)

B.6 Release and enabling targets

In all the following targets, files are release to the directory structure in either-

- *DramaProject*/local/*subsys*/*release_num*

- *DramaProject*/release/*subsys*/*release_num*

depending on the use of the `-s` flag to *dmkmf*. *subsys* can be set with the *make* macro `SYSTEM`, while *release_num* can be set with the *make* macro `RELEASE`. Note that in both cases, these must be in lower case and should be valid file names on all architectures on which the program may be run. In the reset of the this section, this directory is referred to as *ReleaseDirectory*.

B.6.1 DramaReleaseCheck

Type: Function

Description: Check for overwriting of an existing release. This function will check if the target directory (architecture specific) exists and will prompt the user for confirmation to continue if it does.

This should be the first of the release targets in a file, excpet for ones with no libraries or exectuables to release.

Call:
DramaReleaseCheck()

B.6.2 DramaReleaseJavaCheck

Type: Function

Description: Check for overwriting of an existing release (Java files case). This function will check if the Java target directory exists and will prompt the user for confirmation to continue if it does.

This function replaces for supplments `DramaReleaseCheck()` for systems which release Java files.

Call:
DramaReleaseJavaCheck()

B.6.3 DramaReleaseCommon

Type: Function

Description: Copy a list of files to *ReleaseDirectory*

Call:
DramaReleaseCommon(stuff)

B.6.4 DramaReleaseScriptTo

Type: Function

Description: Copy a command script file to *ReleaseDirectory/target*, where *target* is the target name. The output file may be a different name to the input file. The output file is set executable.

Call:

DramaReleaseScriptTo(source,target)

Arguments:

source	Source file.
target	Name it is to be copied to.

B.6.5 DramaReleaseTargetTo

Type: Function

Description: Copy a file to *ReleaseDirectory/target*, where *target* is the target name. The output file may be a different name to the input file.

Call:

DramaReleaseTargetTo(source,target)

Arguments:

source	Source file.
target	Name it is to be copied to.

B.6.6 DramaReleaseLibTo

Type: Function

Description: Copy an object library file to *ReleaseDirectory/target*, where *target* is the target name. The output file may be a different name to the input file. *Ranlib* will be run on the output file if necessary.

Call:

DramaReleaseLibTo(source,target)

Arguments:

source	Source file.
target	Name it is to be copied to.

B.6.7 DramaReleaseCommonTo

Type: Function

Description: Copy a file to *ReleaseDirectory.* The output file may be a different name to the input file.

Call:
DramaReleaseCommonTo(source,target)

Arguments:

source	Source file.
target	Name it is to be copied to.

B.6.8 DramaReleaseTarget

Type: Function

Description: Copy particular file types to *ReleaseDirectory/target* where *target* is the target architecture.

Call:
DramaReleaseTarget(Execs, Libs, ExeScripts , Other)

Arguments:

Execs	A list of executable images.
Libs	A list of object libraries. If necessary, *ranlib* will be run after they are copied.
ExeScripts	A list of executable scripts to be copied. If necessary, they will be set executable after being copied.
Other	Any other files which may need to be copied.

B.6.9 DramaReleaseJava

Type: Function

Description: Copy all Java class files from the current directory to the Java release directory.

Call:
DramaReleaseJava(depends)

Arguments:

depends	Make file dependencies for this target. Can be empty.

B.6.10 DramaReleaseJavaTo

Type: Function

Description: Copy all Java class files from the current directory to the specified sub-directory of the Java release directory.

Call:
DramaReleaseJavaTo(depends,directory)

Arguments:

depends	Make file dependencies for this target. Can be empty.
directory	The sub-directory of the Java release directory to copy the files to.

B.6.11 DramaReleaseJavaPackage

Type: Function

Description: Copy the Java package from the current directory to the Java release directory.

Call:
DramaReleaseJava(package)

Arguments:

package	The Java package to release.

B.6.12 DramaReleaseDramaStart

Type: Function

Description: Create/Modify a dramastart file. In the case of **unix** target, a file named *subys*`_dramastart.rel` is created in the release directory. It will contain the line `RELEASE=`*release_num*. In the case of a **VMS** target, a file named *subsys*`_dramasstart.com` is copied to the release directory and any line starting with `$RELEASE=` is replace by `$RELEASE=`"*release_num*". The file is created in *ReleaseDirectory*.

Call:
DramaReleaseDramaStart(stuff)

B.6.13 DramaEnable

Type: Function

Description: Enable the current release of the subsystem being built. The file *subsys*`_dramastart.rel` (**unix**) or *subsys*`_dramastart.com` (**VMS**) is copied to *DramaProject*/local/*subsys* or *DramaProject*/release/*subsys* depending on the use of the `-s` flag to *dmkmf*. This has the effect of enabling the release when the *dramastart* command is next executed.

Call:
DramaEnable()

B.7 Include file generation

Functions are provided to allow include files to be generated using various utilities.

B.7.1 ErrorIncludeFiles

Type: Function

Description: Generate error message code include files to be generated using the *messgen* utility. The files *file*`.h` and *file*`_msg.h` are generated from *file*`.msg`

Call:
ErrorIncludeFiles(file)

B.7.2 FortranErrorIncludeFile

Type: Function

Description: Generate fortran error message code include files to be generated using the *messgen* utility. Under **unix**, the file *file* is generated from *file*.msg, while under **VMS**, the file *file*.for is generated.

Call:
FortranErrorIncludeFile(file)

B.7.3 SdsIncludeFile

Type: Function

Description: Run the *sdsc* utility on the specified input file to generate the specified output file.

Call:

SdsIncludeFile(target,source)

Arguments:

target	The name input which the result is put
source	The source file.

B.8 General Functions and Macros

B.8.1 DramaCheckTarget

Type: Function

Description: Generates a simple dependency named `checktarget` which checks the current system DRAMA Target against the target for which the Makefile was built. You can then execute the command `make checktarget` to check the environment and makefile agree.

It is generally worthwhile to include the target `checktarget` as a dependencies of the `All` target.

Call:

DramaCheckTarget()

B.8.2 SimpleDependency

Type: Function

Description: Generates a simple dependency with a rule to update the target. This function is normally used to wrap a non-standard dependency in something like `VmsOnly()`, `StarlinkOnly()` etc..

Call:

SimpleDependency(target,depends,command)

Arguments:

target	The target file
depends	A list of dependency files
command	The command to update the target from the list of dependencies.

B.8.3 DummyTarget

Type: Function

Description: Produces a rule with no update command. This is necessary because not all *make* utilities accept and on some machines it must be fudged.

Call:
DummyTarget(target,depends)

Arguments:

target	The target.
depends	A list of things the target depends on.

B.8.4 RunCurrentDir

Type: Function

Description: Generates code which will run a program known to be located in the current directory. This is normally used to run programs which have been generated as part of the build.

Call:
RunCurrentDir(program)

B.8.5 STARLINK_DIR

Type: Make macro to be invoked.

Description: This macro is setup automatically when running under **unix** to point to the Starlink directory structure (normally, `/star`).

Call:
$(STARLINK_DIR)

B.8.6 TARGET

Type: Make macro to be invoked.

Description: This macro is setup automatically to the target machine type.

Call:
$(TARGET)

B.8.7 HOST

Type: Make macro to be invoked.

Description: This macro is setup automatically to the host machine type.

Call:
$(HOST)

B.8.8 DramaDirs

Type: Function

Description: Generate a target which when invoked will output commands to set environment variables to locate the release directories.

Under **unix**, the environment variables *subsys*`_DIR` *subsys*`_LIB` and *subsys*`_DEV` are set to appropriate values. The user should do something like `eval 'make dramadirs'` to get the required effect.

Under **VMS**, the logical name *subsys*`_DIR` is set to the appropriate value. The user should do something like `mms dramadirs` to get the required effect.

After making the target resulting from this target, it should be possible to use link against objects in *subsys*. As a result, this target is used by scripts which automatically build a number of sub-systems.

Call:
DramaDirs()

B.9 Library Macros

B.9.1 LIB_MOTIF

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the Motif X-Windows Libraries.

Call:
$(LIB_MOTIF)

B.9.2 LIB_NET

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the networking libraries.

Call:
$(LIB_NET)

B.9.3 FCLIBS

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the libraries required when linking objects built from fortran source.

Call:
$(FCLIBS)

B.9.4 LIB_TCL

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to Tcl Libraries.

Call:
$(LIB_TCL)

B.9.5 LIB_TK

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the Tk X-11 Libraries. Note, Tcl is included with Tk.

Call:
$(LIB_TK)

B.10 Drama Library Macros

B.10.1 DRAMA_LIBS

Type: Make macro to be invoked.

Description: This macro is defined automatically to a C compiler option which will link all the core drama libraries. Is is not required for targets built using *DramaProgramTarget*.

Call:
$(DRAMA_LIBS)

B.10.2 LIB_DITS

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the **DRAMA** *Dits* library. Is is not required for targets built using *DramaProgramTarget*.

Call:
$(LIB_DITS)

B.10.3 LIB_DRAMAUTIL

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the **DRAMA**utilities library. Is is not required for targets built using *DramaProgramTarget*.

Call:
$(LIB_DRAMAUTIL)

B.10.4 LIB_GIT

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the **DRAMA** *Git* library.

Call:
$(LIB_GIT)

B.10.5 LIB_DTCL

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the **DRAMA** *Dtcl* library.

Call:
$(LIB_DTCL)

B.10.6 LIB_ERS

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the **DRAMA** *Ers* library. Is is not required for targets built using *DramaProgramTarget*.

Call:
$(LIB_ERS)

B.10.7 LIB_IMP

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the **DRAMA** *Imp* library. Is is not required for targets built using *DramaProgramTarget*.

Call:
$(LIB_IMP)

B.10.8 LIB_MESSGEN

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the **DRAMA** *Messgen* library. Is is not required for targets built using *DramaProgramTarget*.

Call:
$(LIB_MESSGEN)

B.10.9 LIB_SDS

Type: Make macro to be invoked.

Description: This macro is defined automatically to point to the **DRAMA** *Sds* library. Is is not required for targets built using *DramaProgramTarget*.

Call:
$(LIB_SDS)

C C compiler pre-processor Macros

When a makefile generated by *dmkmf* is executed and compiles a C source into an object, certain preprocessor macros will be defined to assist in writing portable code.

C.1 Sun target

When the target is a sun, the macros `sun` and `unix` are defined to 1. When the target is a sun running solaris 2.0 or greater, the macro `solaris_2` is defined to 1. When the target is a sun sparcstation, the macros `sun4` and `sparc` are defined to 1.

C.2 VxWorks target.

When the target is a **VxWorks** machine, the macro **VxWorks** is defined.

The macro `Version` is set to the VxWorks version number (either `5_0` or version 5.0 or `5_1` for version 5.1). In addition, the macro `VxWorks_`*version* will be defined to 1. Here, *version* is the value of the `Version` macro.

For a **Sun** host machine, the macro `HOST_SUN` will be defined to 1. The macro `CPU` will be defined to the cpu type (`MC68020`, `MC68030` etc.)

C.3 Vms targets

Under **VMS**, the macro `vms` will be defined to 1. On a vax, the macro `vax` will also be defined to 1.

D The dmakefile for Gcam

```
#BeginConfig
RELEASE=t0_1                    /* Release of this system       */
SYSTEM=gcam                     /* System name (for release     */
EmbeddedOnly(TWODF=/home/aaossc/tjf/scratch/2dF)
INCLUDES=DramaIncl EmbeddedOnly(-I$(TWODF)/drivers)

USERCCOPTIONS = AnsiCFull()     /* Enable Full Ansi C           */
#EndConfig

NormalRules()                  /* Include normal c rules       */
/*
 *      Objects to be put in the dits library.
 */
OBJECTS = Obj(gcam) Obj(gcamcentroid) Obj(vfg)
EmbeddedOnly(GRAB_OBJS = Obj($(TWODF)/drivers/grabberDrv) \
                         Obj($(TWODF)/drivers/pixel_box))

/*
 *      Sources for makedepend.
 */
SRC1=  gcam.c gcamcentroid.c vfg.c

/*
 * The target All will build the dits library, ticker and tocker and ditscmd
 *      On embedded systems, we also to libdits.o
 */
DummyTarget(All, includes Lib(gcam))
/*
 * We use an includes target for include files which need to be built.  This
 * is only really necessary if makedepnds has not been run.
 */
DummyTarget(includes, gcam_err.h gcam_err_msgt.h vfg_err.h vfg_err_msgt.h gcam.h vfg.h gcami
/*
 * Now the normal programs
 */
ErrorIncludeFiles(gcam_err)
ErrorIncludeFiles(vfg_err)

SdsIncludeFile(gcaminfocreate.h, gcaminfo.h)
/*
 * The gcam object library
 */
ObjectLibraryTarget(gcam, $(OBJECTS),)
/*
```

```
 * The vfg program
 */
DramaProgramTarget(vfg, Obj(vfgmain), Lib(gcam), LinkLib(gcam) \
                                        $(GRAB_OBJS) $(LIB_GIT),)

/*
 * Release targets
 */

DramaReleaseCommon(gcam_err.h gcam_err_msgt.h gcam.h gcaminfocreate.h)
DramaReleaseCommon(vfg_err.h vfg_err_msgt.h .h)

DramaReleaseTarget(vfg,Lib(gcam),,)
DramaReleaseDramaStart()

/*
 * Target to enable gcam.
 */
DramaEnable()
DramaDirs()
```

References

[1] Jim Fulton, X Consortium. *X11r5, Configuration Management in the X Window System.* MIT Laboratory for Computer Science. (available from X11r5/mit/hardcopy/config on the X11r5 release tape).

[2] Tony Farrell, AAO. *21-Dec-1992, Guide to Writing Drama Tasks.* Anglo-Australian Observatory **DRAMA** Software Document.

[3] Tony Farrell, AAO. *(TO BE UPDATED),* **DRAMA** *Software Organisation.* Anglo-Australian Observatory **DRAMA** Software Document.

[4] Tony Farrell, AAO. *(TO BE WRITTEN),* **DRAMA** *Software Installation.* Anglo-Australian Observatory **DRAMA** Software Document.

[5] Tony Farrell, AAO. *01-Jun-1993, Distributed Instrumentation Task System.* Anglo-Australian Observatory **DRAMA** Software Document.