Writing Imakefiles Using the
WRPRC2 Configuration Files

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Revision date: 24 August 1993

This document is a tutorial showing how to write Imakefiles using Release 2 of the WRPRC imake configuration files. It explains the more commonly used rules and parameters and provides example Imakefiles illustrating how to use them.

The tutorial doesn't cover every rule and parameter that's available. A separate document serves as a reference describing the contents of the individual configuration files.

Imakefile Writing

To use *imake*, you write an *Imakefile* containing descriptions of targets you want to build. Then you run *imake* to generate a *Makefile* from the *Imakefile*. Once you have the *Makefile*, you run *make* to build your targets. The *Imakefile* is a machine-independent description of your targets. *imake* generates the properly configured machine-dependent *Makefile* that's appropriate for your system.

To generate the initial *Makefile* from an *Imakefile*, use *imboot*:

```
% imboot -C WRPRC2 topdir
```

topdir is the location of the project root directory, either relative to your current directory, or as an absolute path. If you are in the project root, *topdir* may be omitted. *imboot* runs *imake* for you with the appropriate arguments.

If you edit an *Imakefile* thereafter, you must rebuild the *Makefile* so it reflects the changes. If you've already got a *Makefile* from before, you can use it to rebuild itself with the following command:

```
% make Makefile
```

If your changes to the *Imakefile* are erroneous and result in an unusable *Makefile*, fix the *Imakefile* and use *imboot* again to regenerate a working *Makefile*.

If your directory has subdirectories in which Makefiles should be built, use this command:

```
% make Makefiles
```

If you're writing C programs, you should regenerate header-file dependencies each time you rebuild a *Makefile*:

```
% make depend
```

Building and Installing Programs

This section describes the rules provided for building programs and libraries.

Building a Single Program

To build a single program that consists of one source file, use `BuildSimpleProgram()`. This rule takes three arguments:

```
BuildSimpleProgram(prog,linklibs,deplibs)
```

The first argument is the name of the program; `BuildSimpleProgram()` assumes there is a single source file named *prog.c* and a single object file *prog.o*. The second argument names the libraries needed to link the program, and the third names the libraries the program depends on.

If a program uses no libraries, the *Imakefile* looks like this:

```
BuildSimpleProgram(myprog,NullArg,NullArg)
```

This specifies you want to build a program *myprog*. The second and third arguments are empty, which is specified explicitly using `NullArg`. The symbol `NullArg` evaluates to nothing; it's preferable to leaving the arguments empty because that causes problems building the *Makefile* on some systems.

If a program uses libraries, then you need to specify them. The libraries the program depends on are the same as the link libraries, but the value of *deplibs* is often different than *linklibs* because *deplibs* can only consist of libraries that can be named as pathnames (*make* understands pathnames only as dependencies, not libraries specified using −*l* syntax). This means that system libraries usually don't have any dependency specifier and you leave it out of *deplibs*.

Suppose a program uses the math library, you can write the *Imakefile* like this:

```
BuildSimpleProgram(myprog,-lm,NullArg)
```

The math library is specified as −*lm* for linking, but since it's a system library, there's no dependency specifier for it and *deplibs* is empty.

Suppose a program needs a library *mylib* that's located in the *lib* directory under the project root. You can refer to it as `$(TOP)`*/lib/libmylib.a*, because `TOP` always refers to the top of the project tree. Since that reference is a pathname, you can use it not only for linking, but for dependency purposes as well:

```
BuildSimpleProgram(myprog,$(TOP)/lib/libmylib.a,$(TOP)/lib/libmylib.a)
```

`BuildSimpleProgram()` can be invoked only once per *Imakefile*. It generates target entries for the following operations:

`% make prog`	*Build program*
`% make`	*Same as* `make prog`
`% make all`	*Same as* `make`
`% make clean`	*Remove object and executable files*
`% make lint`	*Run sources through* lint
`% make install`	*Install program*
`% make install.man`	*Install program's manual page*

The `install` target installs the program in the directory named by `LOCALBINDIR`, which is by default your site's normal installation directory for user programs. The `install.man` target installs the manual page in the directory named by `MANDIR`, which is your site's normal directory for user program manual pages. Of course, the `install.man` target entry only works if you write a manual page; if you do, name it *prog.man*.

Note that for all installation operations, the installation directory is created if it doesn't exist. That means it's prudent to run *make −n install* (*install.man*) before you run *make install* (*install.man*).

Building and Installing Multiple Programs

The general interface for building programs is `BuildProgram()`. You can invoke this rule any number of times in an *Imakefile*, so it's suitable for building an arbitrary number of programs in a single directory.

For example, suppose you're building three programs *proga*, *progb*, and *progc*, your *Imakefile* might look like this:

```
BuildProgram(proga,proga.c,proga.o,NullArg,NullArg)
BuildProgram(progb,bmain.c bfuncs.c,bmain.o bfuncs.o,-lm,NullArg)
BuildProgram(progc,progc.c,progc.o,NullArg,NullArg)
```

The arguments to `BuildProgram()` are the program name, the program's source and object files, and the program's link and dependency libraries.

There are certain improvements we can make to the *Imakefile*. For one thing, we can make the the rule invocations more readable by using *make* variables to hold the source and object file lists, and the link and dependency libraries. Then we refer to the variables in the invocations:

```
   ASRCS = proga.c
   AOBJS = proga.o
   ALIBS =
DEPALIBS =

   BSRCS = bmain.c bfuncs.c
   BOBJS = bmain.o bfuncs.o
   BLIBS = -lm
DEPBLIBS =

   CSRCS = progc.c
   COBJS = progc.o
   CLIBS =
DEPCLIBS =

BuildProgram(proga,$(ASRCS),$(AOBJS),$(ALIBS),$(DEPALIBS))
BuildProgram(progb,$(BSRCS),$(BOBJS),$(BLIBS),$(DEPBLIBS))
BuildProgram(progc,$(CSRCS),$(COBJS),$(CLIBS),$(DEPCLIBS))
```

These changes also make the *Imakefile* easier to edit.

`BuildProgram()` generates target entries for the following operations:

`% make` *`prog`*	*Build program*
`% make all`	*Build all programs*
`% make`	*Same as* `make all`
`% make clean`	*Remove object and executable files*
`% make lint`	*Run sources through* lint
`% make lint.`*`prog`*	*Run* `prog` *sources through* lint

The commands *make* or *make all* build all programs in the *Makefile*, whereas *make* `prog` builds only `prog`. This is useful when you don't want to build everything. Similarly, *make lint* runs the source files for all programs through *lint*, whereas *make lint.*`prog` lints only the sources for `prog`. This is useful when you make a change to the sources for only one program and don't want to *lint* everything.

`BuildProgram()` doesn't generate any `depend` target, so we should invoke `DependTarget()` to do that. `DependTarget()` implicitly assumes the *make* variable `SRCS` is set to the names of all the source files for which dependencies should be generated. It's convenient to set `SRCS` to the concatenation of the variables used to name individual program sources:

```
   ASRCS = proga.c
   AOBJS = proga.o
```

```
   ALIBS =
DEPALIBS =
   BSRCS = bmain.c bfuncs.c
   BOBJS = bmain.o bfuncs.o
   BLIBS = -lm
DEPBLIBS =
   CSRCS = progc.c
   COBJS = progc.o
   CLIBS =
DEPCLIBS =
      SRCS = $(ASRCS) $(BSRCS) $(CSRCS)
BuildProgram(proga,$(ASRCS),$(AOBJS),$(ALIBS),$(DEPALIBS))
BuildProgram(progb,$(BSRCS),$(BOBJS),$(BLIBS),$(DEPBLIBS))
BuildProgram(progc,$(CSRCS),$(COBJS),$(CLIBS),$(DEPCLIBS))
DependTarget()
```

Note that `DependTarget()` shouldn't be invoked before the program-building rules or `depend` will become the default target.

No installation target entries are generated by `BuildProgramTarget()`. To install a program, invoke `InstallProgram()`:

```
InstallProgram(prog,dir)
```

prog is the name of the program and *dir* is the directory in which to install it. `LOCALBINDIR` can be used to install the program in your site's "usual" user program installation directory. `InstallProgram()` generates target entries for the following operations:

```
% make install
% make i.prog
```

The first command installs all programs built by the *Makefile* and the second installs only *prog*.

If you write a manual page for *prog*, name it *prog.man* and invoke `InstallManPage()`:

```
InstallManPage(prog)
```

Note that you specify only the program name (without the *.man* suffix). `InstallManPage()` generates target entries for the following operations:

```
% make install.man
% make i.prog.man
```

The first command installs all manual pages and the second installs only the page for *prog*. The page is installed in your site's "usual" directory for user program manual pages.

After modifying the *Imakefile* to add installation rules for our programs and manual pages, it looks like this:

```
   ASRCS = proga.c
   AOBJS = proga.o
   ALIBS =
DEPALIBS =
   BSRCS = bmain.c bfuncs.c
   BOBJS = bmain.o bfuncs.o
   BLIBS = -lm
DEPBLIBS =
   CSRCS = progc.c
   COBJS = progc.o
   CLIBS =
DEPCLIBS =
      SRCS = $(ASRCS) $(BSRCS) $(CSRCS)
BuildProgram(proga,$(ASRCS),$(AOBJS),$(ALIBS),$(DEPALIBS))
```

```
InstallProgram(proga,$(LOCALBINDIR))
InstallManPage(proga)

BuildProgram(progb,$(BSRCS),$(BOBJS),$(BLIBS),$(DEPBLIBS))
InstallProgram(progb,$(LOCALBINDIR))
InstallManPage(progb)

BuildProgram(progc,$(CSRCS),$(COBJS),$(CLIBS),$(DEPCLIBS))
InstallProgram(progc,$(LOCALBINDIR))
InstallManPage(progc)

DependTarget()
```

Building and Installing a Library

To build a library, invoke `BuildNormalLibrary()`, passing it the library basename and the source file and object file lists. (If a library's name is *libxyz.a*, the basename is *xyz*). You should invoke `NormalLibraryObjectRule()` in conjunction with `BuildNormalLibrary()` so that *.o* files are built in a way suitable for libraries. You should also set the *make* variable `SRCS` and invoke `DependTarget()` so that *make depend* works. Thus, a typical library-building *Imakefile* looks like this:

```
OBJS = ...
SRCS = ...

NormalLibraryObjectRule()
BuildNormalLibrary(name,$(SRCS),$(OBJS))
DependTarget()
```

`NormalLibraryObjectRule()` may redefine the default *.c* → *.o* transformation that *make* uses to produce object files, so it's best not to build programs in a directory that's used to build a library or libraries.

`BuildNormalLibrary()` generates target entries for the following operations:

`% make lib`*`name`*`.a`	*Build library* `name`
`% make all`	*Build all libraries*
`% make`	*Same as* `make all`
`% make clean`	*Remove object and library files*
`% make lint`	*Run all sources through* lint
`% make lint.lib`*`name`*`.a`	*Run sources for library* `name` *through* lint

`BuildNormalLibrary()` does not generate any installation target entries; use `InstallLibrary()` to install a library:

```
InstallLibrary(name,dir)
```

The *`dir`* parameter is often `LOCALUSRLIBDIR`, which is by default */usr/local/lib*. `InstallLibrary()` generates entries for the following operations:

`% make install`	*Install all libraries*
`% make i.lib`*`name`*`.a`	*Install library* `name` *only*

You can invoke `BuildNormalLibrary()` any number of times to build an arbitrary number of libraries. Example:

```
OBJS1 = ...
SRCS1 = ...

OBJS2 = ...
SRCS2 = ...

 SRCS = $(SRCS1) $(SRCS2)

NormalLibraryObjectRule()

BuildNormalLibrary(lib1,$(SRCS1),$(OBJS1))
InstallLibrary(lib1,$(LOCALUSRLIBDIR))

BuildNormalLibrary(lib2,$(SRCS2),$(OBJS2))
```

```
InstallLibrary(lib2,$(LOCALUSRLIBDIR))
DependTarget()
```

Note that you invoke `NormalLibraryObjectRule()` just once.

Other Installation Rules

The WRPRC2 files provide a number of installation rules besides `InstallProgram()`, `InstallLibrary()`, and `InstallManPage()`. Some of the more common ones are:

`InstallScript(file,dir)`	*Install an executable script*
`InstallDataFile(file,dir)`	*Install a non-executable file*
`InstallIncludeFile(file,dir)`	*Install a header file*

You can install manual pages in directories other than the ‘‘usual’’ directory. To install a page explicitly in the directory for a given manual section, invoke `InstallMan`*S*`Page()`, where *S* is 1, 2, 3, 4, 5, 6, 7, 8, L, or N. As with `InstallManPage()`, the name of the file to be installed should end in a *.man* suffix, but the argument to the rule does not include the suffix. The installed manual page will be renamed appropriately for the directory into which it’s installed.

Naming WRPRC Libraries

At WRPRC, several locally-developed libraries are commonly used to build programs. The WRPRC2 configuration files provide link and dependency specifiers for them. The following table lists the specifiers provided by the WRPRC2 configuration files. For every library *XXX*, there is a *XXX*`LIB` variable for link purposes and a `DEP`*XXX*`LIB` variable for dependency purposes.

Library Name	**Link Specifier**	**Dependency Specifier**
refer record manipulation library	BIBSTUFFLIB	DEPBIBSTUFFLIB
Exception and termination manager library	ETMLIB	DEPETMLIB
Form processing library	FPLLIB	DEPFPLLIB
Form query library	FQLLIB	DEPFQLLIB
GECOS library	GECOSLIB	DEPGECOSLIB
Simple log manager library	LOGMGRLIB	DEPLOGMGRLIB
Simple memory manager library	MEMMGRLIB	DEPMEMMGRLIB
Network database service library	NDSLIB	DEPNDSLIB
Network I/O library	NIOLIB	DEPNIOLIB
Order processing library	ORDERLIB	DEPORDERLIB
Sequence number library	SEQNUMLIB	DEPSEQNUMLIB
Simple screen-management library	SIMSCRLIB	DEPSIMSCRLIB
Token scanning library	TSLIB	DEPTSLIB

Example: Suppose a program *myprog* is built from *prog.c* and the ETM, memory manager, and token scanning libraries. The *Imakefile* might look like this:

```
   LIBS = $(MEMMGRLIB) $(TSLIB) $(ETMLIB)
DEPLIBS = $(DEPMEMMGRLIB) $(DEPTSLIB) $(DEPETMLIB)
BuildSimpleProgram(myprog,$(LIBS),$(DEPLIBS))
```

Multiple-Directory Support

If a directory has subdirectories, normally you want *make* to traverse those subdirectories when you run it. Write the *Imakefile* like this:

```
#define IHaveSubdirs
#define PassCDebugFlags
SUBDIRS = list
```

where `list` is a space-separated list of the subdirectories that should be processed by *make*, in the order you want them processed. For example:

```
#define IHaveSubdirs
#define PassCDebugFlags
SUBDIRS = include lib apps doc man
```

When you build the *Makefile*, it will contain recursive entries for `all`, `depend`, `install`, `install.man`, `clean`, `tags`, and `Makefiles` targets.

If you anticipate that you'll want to build debuggable targets, define `PassCDebugFlags` like this instead:

```
#define PassCDebugFlags 'CDEBUGFLAGS=$(CDEBUGFLAGS)'
```

Then you can specify debugging flags from the *make* command line, e.g.:

```
% make CDEBUGFLAGS=-g
```

Generating a `World` Target

To generate a `World` target entry in a project's root (top-level) directory you can invoke `SimpleWorld-Target()`. Then you can say *make World* to build the entire project from scratch (including configuring its Makefiles). Note that if your project is at all complex, it may require a configure/build sequence that varies from the simple entry provided by `SimpleWorldTarget()`. In this case you'll need to write your own entry.

`SimpleWorldTarget()` takes the project name and release level as arguments:

```
SimpleWorldTarget(project,release)
```

If your project has a *config* directory for private configuration information, it's useful to use *Imake.p-params* there to override the macros that define the project name and release numbers. For example:

```
#ifndef ProjectName
#define ProjectName My Project Name
#endif
#ifndef ProjectMajorRelease
#define ProjectMajorRelease 1
#endif
#ifndef ProjectMinorRelease
#define ProjectMinorRelease 03
#endif
```

Then you can invoke `SimpleWorldTarget()` using the associated *make* variables:

```
SimpleWorldTarget($(PROJECTNAME),$(PROJECTRELEASE))
```

In this case, whenever you change the release numbers in *Imake.p-params*, you don't have the change the project root *Imakefile*. However, you do need to rebuild the *Makefile* so it gets the updated release numbers.

If your project builds targets in multiple directories, the root *Imakefile* might look like this:

```
#define IHaveSubdirs
#define PassCDebugFlags
SUBDIRS = list
SimpleWorldTarget($(PROJECTNAME),$(PROJECTRELEASE))
```

Formatting Documents

The WRPRC2 files contain rules for producing PostScript files from *troff* documents. In most cases, a *troff* document is formatted using the −*man*, −*me*, −*mm*, or −*ms* macros, possibly after having been run through one or more preprocessors such as *tbl* or *pic*.

For instance, to produce a PostScript file *doc.ps* from a document *doc.ms* that requires the −*ms* macros but no preprocessors, write this:

```
TroffMsToPostScript(doc.ps,doc.ms,NullArg,NullArg)
```

The arguments to `TroffMsToPostScript()` are the destination output file, the input source file (or files), the preprocessor list, and any dependencies that must exist before the source can be processed. The preprocessor list must be given in the form of a piece of a command pipeline, beginning with “|”. For example, to run the input through *pic*, use this:

```
TroffMsToPostScript(doc.ps,doc.ms,|$(PIC),NullArg)
```

To run the input through *tbl* and *eqn*, use this:

```
TroffMsToPostScript(doc.ps,doc.ms,|$(TBL)|$(EQN),NullArg)
```

To format a document with one of the other macro packages, change the “Ms” in the rule name to “Man”, “Me”, or “Mm”.

Default Target Entries

The WRPRC2 configuration files provide several target entries in the *Makefile* for you. Some of them actually do something, others are simply placeholders to prevent errors when certain *make* operations are performed.

- A `help` entry is provided allowing *make help*, which indicates some typical *make* commands and what they do.
- A `Makefile` entry is produced so you can run *make Makefile* to rebuild the *Makefile* after you make changes to the *Imakefile*.
- A default `clean` entry is produced that removes typical garbage files such as **.o*, **.a*, and **.bak* files, and *core*. **Note:** Unlike the default `clean` entry produced by the X11 rules, the WRPRC2 default `clean` entry does not remove *~ files, since I often save a backup copy of *file* as *file~*.
- A `tags` entry is provided so you can say *make tags*.
- If the *Imakefile* is for a directory with subdirectories, recursive `all`, `depend`, `install`, `install.man`, `clean`, `tags`, and `Makefiles` entries are produced. If the directory doesn't have subdirectories, dummy targets for these operations are produced as necessary so that *make* doesn't generate errors (this is necessary or recursive *make* operations would fail as soon as they reached a directory with no subdirectories).