Using Imake to Configure the X Window System Version 11, Release 4

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Document revision: 1.03
Revision date: 3 August 1990

ABSTRACT

The X Window System† is a large software project that, despite its size, is remarkable in its portability. Much of this is due to the method employed to configure the X distribution for building and installation: use of a small number of tools and isolation of machine dependencies into a small number of files that can be easily maintained and modified. This document discusses one of those tools, *imake*, and the design of the configuration files used in conjunction with it.

1. Introduction

This document describes how *imake* configuration files are set up in the X Window System. It is assumed that you know something about what *imake* is supposed to be used for (in general), and that you're reading this to find out something about how that is accomplished in X (in particular). The description applies to X Version 11, release 4 configuration files, which are organized quite differently than those from previous releases.

Other documentation (from the X11R4 distribution) that you may find useful includes:

- Section 2 of ''X Window System, Version 11 Release 4 Release Notes,'' by Jim Fulton.
- *mit/doc/config/usenixws/paper.ms*: ''Configuration Management in the X Window System,'' also by Jim Fulton.
- *mit/config/README*: ''X Window System Imake Configuration Guide, Release 4.''
- *mit/config/imake.man*: manual page for *imake*.
- *mit/util/makedepend/mkdepend.man*: manual page for *makedepend*.
- *contrib/doc/imake/imake.tex*: ''An *Imake* Tutorial,'' by Mark Moraes. This was written between R3 and R4, but is worthwhile reading, nevertheless.

You should also have access to the X configuration files (in *mit/config*). These are not really documentation as such, but it is expected that you have them at hand, for comparison against the descriptions below.

imake is generally conceded to have a pretty steep learning curve. The *README* referred to above notes that *imake* ''can be somewhat tricky to master,'' an observation attested to by many who use it. There

† X Window System is a trademark of the Massachusetts Institute of Technology.

seem to be three camps regarding *imake*; those who think it's wonderful, those who think it's wretched, and those who suspect it might be useful (else why would it be used to configure a major publicly-available effort like X?) but are puzzled by it. The present document was written to fill in some of the gaps in the existing documentation, in order to try to swell the ranks of the first group by depleting the membership of the third. (Those who despise *imake*—members of the second group—have good reasons for doing so and the present effort is not likely to sway them. It is worth noting that the author of *imake* is in this group.)

Where appropriate, the current X configuration files are compared to those from previous releases to show how limitations of earlier configuration architectures have been eased. There are also occasional comments to indicate how *imake* and the X configuration files might be adapted for use in contexts other than building X.

This document is independent of the efforts of the MIT X Consortium. There are no doubt errors lurking within, both of understanding and of fact; they are my own. Please send corrections, criticisms and comments to the address above.

2. The tools: imake, makedepend, xmkmf

imake is a configuration tool—the main tool used to configure X. It is not a replacement for the *make* program, but it works in an environment that assumes the availability of *make* as the tool used to direct project building and installation. *imake* eliminates the need to write *Makefiles* directly. The name means ''include-make''—*imake* uses the C preprocessor *cpp*, taking advantage of its include-file and macro-processing facilities for the purpose of generating *Makefiles* suitable for *make*.

For each directory in the X distribution, *imake* reads a bunch of configuration files that go to a lot of trouble to compensate for and adjust to the individual characteristics of your system. These are combined with an *Imakefile* from the current directory and the whole mess is sent through *cpp* to build a *Makefile* there. The same configuration files are used to build every *Makefile*; they are kept together in *mit/config*, isolating machine dependencies in one location to ease development and maintenance. In contrast, the *Imakefile* is directory-specific (it specifies the targets to be built in that directory) and is machine-independent.

This means that when the distribution is built on a different machine, only the configuration files need be changed. The *Imakefiles* do not need to be. If X configuration were specified directly using *Makefiles*, this would not be true. Because the configuration information in *Makefiles* is not portable, each one would have to be edited individually—bad enough for a single *Makefile*, but an overwhelming prospect for a project the size of X.

The value of being able to localize machine-dependent configuration information into one place should not be underestimated, particularly as X becomes more sophisticated and the number of systems on which it runs increases. The number of *make* variables used to build X has increased from approximately 60 in R1 to 100 in R3 to 150 in R4.

Another tool, *makedepend*, is used to generate header file dependencies for C source files in each directory after the *Makefile* has been built. Dependencies of targets upon object files can be statically listed in the *Imakefile*, but not those for header files. Different systems organize these differently so dependencies on them must be generated dynamically.

While X is being built, *imake* itself is located with the configuration files in *mit/config*. *makedepend* lives in *util/makedepend* if you use the compiled version (preferred), or in *util/scripts* if you use the shell script version (slower).

When X is installed, copies of *imake*, *makedepend*, and a related tool, *xmkmf*, are placed in a public directory, and the configuration files are copied to */usr/lib/X11/config*. *xmkmf* is used to bootstrap a *Makefile* from an *Imakefile* using the installed configuration files,[1] and can be used to build programs from outside of

[1] Normally a new *Makefile* is produced with ''make Makefile'', an operation that presumes you already have a properly configured *Makefile* containing the rules necessary to run *imake*. Hence the bootstrap problem. Obviously, if you know the

the X source tree, such as you might write yourself or get from comp.sources.x on Usenet.

The X configuration files make very few assumptions about the capabilities of *make* itself. Although several enhanced versions of *make* provide special features or extensions, any *Makefile* that relies on universal availability of those features will fail on systems with less-capable versions of *make*. *Makefiles* produced by *imake* in X do not use any of these constructs, so they should work with even the lowest-common-denominator *make* program.

Since *imake* passes its input through *cpp*, the configuration file writer can take advantage of several features not present in plain-vanilla *make*, such as parametrized macros (using #define), file inclusion (using #include) and conditional testing (using #if, #ifdef, #ifndef). This ameliorates many of the consequences of having to assume that such a *make* is the only one available.

Use of *cpp* can produce problems, too, of course.

(1) How to specify comments that should end up in the *Makefile*. *make* comments are introduced by a ‘‘#’’ character—unfortunately, *cpp* treats lines beginning with ‘‘#’’ as preprocessor directives. A comment of the form

```
# if your system doesn't have ranlib, use /bin/true
```

is considered a symbol test by *cpp* and is gobbled up, while a comment like

```
# The next rule is a workaround for a broken compiler
```

generates a *cpp* syntax error. To get around this, comments should be preceded by ‘‘/**/#’’ instead of just ‘‘#’’. *cpp* will strip off the ‘‘/**/’’ (the empty C comment) and won’t treat the line as a directive (though it’s still liable to symbol substitution).

The exception to this ‘‘comment-commenting’’ convention is in the *Imakefile* itself, where *imake* automatically adds ‘‘/**/’’ onto lines beginning with ‘‘#’’ that are not preprocessor directives. The reason for this is to insulate the end user of *imake* (who simply writes the *Imakefile* and not the underlying configuration files) from the need to be aware of, or abide by, the commenting convention. I would hazard a guess that many people are not aware of this exception, given the high incidence of *Imakefiles* containing ‘‘/**/#’’.

Comments that are to be passed through to the *Makefile* can be written as normal C comments (text surrounded by ‘‘/*’’ and ‘‘*/’’) and will deleted by *cpp*.

(2) The configuration files define several multiple-line macros that are intended to produce multiple lines of output. *cpp* joins multi-line macros into a single line, so some post-processing is necessary.

(3) Sometimes the values of *cpp* symbols are to be concatenated. The symbol names cannot be concatenated in the configuration files, because that results in a different symbol name, so they must be kept apart somehow. The empty C comment is useful here, too. For instance, if Prefix, LibName and Suffix are defined as *lib*, *mylib* and *.a*, respectively, to obtain the concatentation value *libmylib.a* one must write Prefix/**/LibName/**/Suffix, not PrefixLibNameSuffix.

3. Configuration file architecture

To produce a *Makefile* from an *Imakefile*, the following configuration files are used:

---

proper incantation to utter, you can issue the appropriate *imake* command manually, but *xmkmf* eliminates the need. Besides, the only way you’d know which wand to wave is by having a reasonable understanding of *imake* in the first place—in which case you wouldn’t be reading this!

Imake.tmpl	master template
platform.cf	platform-specific definitions (the filename varies)
site.def	site-specific definitions
Project.tmpl	X-specific default definitions
Imake.rules	*cpp* macros to generate *make* rules

In general, if a port exists for your system, the only file that you should need to modify to build and install X is *site.def*. In some cases it may be necessary to modify the platform file. *Imake.tmpl*, *Project.tmpl* and *Imake.rules* should be left alone.[2]

If you are porting X to a new platform, you will need to write your own platform file, and modify the top part of *Imake.tmpl* slightly. If you are doing a new port of X, *or* porting the X configuration files for use with a different project, you are well-advised to study all of the configuration files thoroughly. Ignorance is not bliss in such instances.

Imake.tmpl contains an #include line for each of the other four configuration files and for the *Imakefile* in the current directory. It also contains in-line sections for global constant definitions, header block selection, description of system characteristics, build definitions, and extra *make* rules. The template is structured as follows to make everything fit together:

Imake.tmpl:

global constants
header blocks
#include <*platform*.cf>
#include <site.def>
system description and
build definitions
#include <Project.tmpl>
#include <Imake.rules>
#include "./Imakefile"
extra *make* rules

Before these sections of the template are described, there are some general principles that should be kept in mind.

Much of the flexibility of the X configuration files is achieved through the use of the following construct, which defines *symbol* only if it has not already been defined:

#ifndef *symbol*
#define *symbol value*
#endif

This construct allows any system-, build- or project-related symbol to be given a default definition if none has been supplied earlier; coupled with support for inclusion of platform- and site-specific files prior to the section in which the default is defined, the default may be overridden by a definition occurring within those

[2] It should be emphasized that ‘‘should be left alone’’ applies *only to building X itself*. If you are adapting the configuration files for use with another project, you will probably modify all of them somewhat. You may even find yourself in the position of having modified them to such an extent that they end up hacked to bits, incapable of configuring anything, with you left possessing only the faint hope that something useful will rise, Phoenix-like, from out of the remains. Be assured—it won’t. But cheer up; start again and learn from your mistakes.

files. For example, the default for the C compiler symbol occurs in the build definitions section of *Imake.tmpl* and is defined thus:

```
#ifndef CcCmd
#define CcCmd cc
#endif
```

If the GNU C compiler is to be used instead, the default definition can be overridden simply by putting the following in *site.def*:

```
#ifndef CcCmd
#define CcCmd gcc
#endif
```

Use of the #ifndef/#endif construct is pervasive throughout the X11R4 configuration files, to a much greater extent than in the X11R3 files. This is especially true in the platform-specific files.

A fairly consistent pattern followed within the system/build section of *Imake.tmpl* and within *Project.tmpl* is that definitions for *cpp* symbols are listed first, followed by definitions for *make* variables. I presume this is done because there is greater flexibility available in defining *cpp* symbols, e.g., through #ifndef conditional testing. Since *cpp* can't tell whether a *make* variable has previously been defined, the strategy adopted is to associate a *cpp* symbol with a given *make* variable, define the symbol conditionally to allow the possibility of overriding, then equate the variable to whatever value the symbol ends up with. For instance, *Imake.tmpl* contains:

```
#ifndef CcCmd
#define CcCmd cc
#endif
. . .
CC = CcCmd
```

If no earlier definition of CcCmd overrides the default (e.g., in *site.def*), CC ends up as ‘‘cc’’. On the other hand, if the default is overridden, e.g., to use ‘‘gcc’’ instead, CC ends up as ‘‘gcc’’.

Sometimes ‘‘mixed’’ symbol definitions occur, in which *cpp* symbols are defined in terms of *make* variables. There are at least two reasons to do this.

(1) To avoid order-of-definition problems. Consider the two sets of definitions below.

```
#ifndef a                    #ifndef a
#define a b                  #define a ${B}
#endif                       #endif
A = a                        A = a
. . .                        . . .
#ifndef b                    #ifndef b
#define b z                  #define b z
#endif                       #endif
B = b                        B = b
```

What ends up in the *Makefile* is:

```
A = b                                   A = ${B}
B = z                                   B = z
```

The example on the left defines a (and hence A) in terms of a symbol that hasn't been defined yet; both get the value of a literal ‘‘b’’. The example on the right works; b is defined as ‘‘z’’, B gets the same value. A becomes ‘‘${B}’’, which is not evaluated further until *make* is run. At that time A evaluates properly to ‘‘z’’, independently of the order in which a and b are defined in the configuration file.

(2) To allow for greater flexibility at installation time. For example, UsrLibDir and USRLIBDIR are defined as:

```
#ifndef UsrLibDir
#define UsrLibDir $(DESTDIR)/usr/lib
#endif
. . .
USRLIBDIR = UsrLibDir
```

Symbols such as IncRoot, BinDir, AdmDir are defined similarly, even though DestDir (to which DESTDIR is eventually equated) is defined earlier (i.e., there is no order-of-definition problem here). If a user wants the install to take place under a different root than DestDir, the command ‘‘make DESTDIR=/alternate/root install’’ suffices, by overriding the definition of DESTDIR in the *Makefile*. If UsrLibDir, etc., were defined directly in terms of DestDir rather than DESTDIR, this would not be possible.[3]

Each of the sections of *Imake.tmpl* is described below. The descriptions only contain representative examples of the symbols used in each configuration file. For exhaustive lists, consult the appendix.

3.1. Global constant definitions

This section of *Imake.tmpl* defines two symbols (YES as 1 and NO as 0). References to these symbols are legion throughout the rest of the configuration specification and their values should not be changed.

Note: symbols #define'd in the configuration files or the *Imakefile* have nothing to do with symbols #define'd in your programs; they are entirely independent.[4]

3.2. Header block selection

The header block section of *Imake.tmpl* determines the type of machine on which *imake* is being run. This is done by looking for a *trigger*—a *cpp* preprocessor symbol that uniquely and unambiguously indicates a given platform. For instance, ‘‘sun’’ is defined only on Sun systems, ‘‘apollo’’ only on Apollo systems, and ‘‘ultrix’’ is only on Ultrix systems. Their header blocks look like this:

---

[3] Actually, it would be possible, but cumbersome: ‘‘make install BINDIR=/alt/bin/dir ADMDIR=/alt/adm/dir LIBDIR=/alt/lib/dir ...’’

[4] I once wondered about this, which seems pretty ridiculous now.

Sun:
```
#ifdef sun
#define MacroIncludeFile <sun.cf>
#define MacroFile sun.cf
#undef sun
#define SunArchitecture
#endif /* sun */
```

Apollo:
```
#ifdef apollo
#define MacroIncludeFile <apollo.cf>
#define MacroFile apollo.cf
#undef apollo
#define ApolloArchitecture
#endif /* apollo */
```

Ultrix:
```
#ifdef ultrix
#define MacroIncludeFile <ultrix.cf>
#define MacroFile ultrix.cf
#ifdef vax
#undef vax
#define VaxArchitecture
#endif
#ifdef mips
#undef mips
#define MipsArchitecture
#endif
#undef ultrix
#define UltrixArchitecture
#endif
```

The trigger symbol might be predefined by *cpp* itself; if no such predefined symbol is available,[5] *imake* is built to explicitly pass a definition for one to *cpp* to cause the correct header block to be selected.

If no trigger is defined, a generic configuration header block is selected instead. Since that means the platform wasn't properly determined, a warning is written into the *Makefile*:

```
# WARNING:  Imake.tmpl not configured; guessing at definitions!!!
# This might mean that BOOTSTRAPCFLAGS wasn't set when building imake.
```

The resulting generically-configured *Makefile* will probably fail in one of a number of strange and wondrous ways when used to try to build anything, With any luck, the hapless user will examine the *Makefile* to figure out what went wrong, see the warning, and realize that the platform wasn't determined properly.

Failure to find a defined trigger symbol might occur because no port exists for the machine in question, and thus no header block for it exists. (Each time X is ported to a new platform, a new header block is added to this section of *Imake.tmpl*.) Failure will also occur if *imake* was not built properly (i.e., it doesn't pass the proper trigger symbol definition to *cpp*).

Assuming the proper header block is selected, several things happen inside of it. The name of the associated platform-specific *.cf* file is defined for later inclusion by the template; one or more architecture indicator symbols are defined that can be used by later configuration sections to test for particular software or hardware platforms; the trigger symbol is undefined so it won't trigger any other header block.

A header block may define a single architecture indicator symbol that refers both to the software and hardware, or separate indicators may be defined for an operating system that runs on multiple hardware types, For example, ''ultrix'' indicates an Ultrix system, but it is no longer true (as it once was) that a VAX can be assumed as the hardware platform—RISC Ultrix systems run on MIPS chips now. Thus UltrixArchitecture is defined as the software indicator symbol, and VaxArchitecture or MipsArchitecture as the hardware indicator.

Software indicator symbols should be unique. Hardware indicator symbols are not necessarily unique in themselves and may be shared across different software platforms, e.g., MipsArchitecture can be defined for Ultrix or SGI systems, VaxArchitecture for Ultrix or BSD systems.

Imakefiles should always test for indicator symbols rather than trigger symbols, since the former are more reliable (the latter are undefined by the header block, anyway).

---

[5] A goal ANSI C will help us reach?

The use of predefined or *imake*-supplied *cpp* trigger symbols is fraught with peril, and one must construct the header blocks carefully, for two reasons:

(1) Symbols may be ambiguous. For instance, ‘‘vax’’ can be defined both under Ultrix and under BSD. Thus, the Ultrix header block must come first. It tests for ‘‘ultrix’’ and if that succeeds, defines UltrixArchitecture and undefines ‘‘vax’’ so the BSD block will fail.

(2) Symbols may become ambiguous in unanticipated ways in the future. The ‘‘mips’’ symbol is an example. In R3 it was used to unambiguously detect a true MIPS machine. This can no longer be done given the presence of that symbol on other systems that also run on MIPS chips now.

When porting X to other platforms, it is sometimes difficult to know either what the trigger symbol should be, or what indicator symbol(s) to use. Consider MIPS machines running RISC/os. ‘‘mips’’ is insufficient as a trigger, because it is does not unambiguously define MIPS systems. The hardware indicator symbol is clear enough, MipsArchitecture, but the software indicator is less so. The name of the MIPS operating system suggests that RiscosArchitecture might be a good choice. Guess again. Acorn Computers Ltd. has an OS with a similar name, ‘‘RISC OS’’.[6]

3.3. platform.cf

After the header block section has determined which platform-specific file to use, that file is then included by *Imake.tmpl*:

```
/**/##########################################################################
/**/# platform-specific configuration parameters - edit MacroFile to change
#include MacroIncludeFile
```

where MacroIncludeFile has been defined properly by the header block.

The platform file contains definitions needed to make X build and install correctly on a particular system. Some common things found here are definitions for the operating system name version numbers (major and minor), C compiler version numbers, and workarounds for missing commands. Overall, these files are really quite a hodgepodge of different things, and it is difficult to describe them in any general way. You really should read through the *.cf* file for your system before you try to compile anything, to get an idea what can be done with it. (It doesn’t hurt to read *all* the *.cf* files, as a matter of fact, especially if you are doing a new port.)

It is important that version numbers in the platform file accurately reflect your system. Some symbol definitions are contingent upon the OS or C compiler version, to accommodate system changes, deficiencies, or bugs, so you want to make sure you get the right definitions. For example, *sun.cf* contains the following:

```
#if OSMajorVersion <= 3
#define HasVoidSignalReturn NO
#else
#define HasVoidSignalReturn YES
#endif
```

This reflects a change in the return type of the *signal()* system call (from int to void) on Sun systems beginning with SunOS 4.0.

---

6 My own preference is to use ‘‘mipsriscos’’ as the trigger symbol, MipsArchitecture as the hardware indicator, and MipsRiscosArchitecture (which seems suitably unique, but has the disadvantage of being quite long, and uneuphonious to boot) as the software indicator.

A common missing command workaround occurs for those systems that have no BSD-compatible *install* command, such as in *cray.cf*:

```
#define InstallCmd sh $(SCRIPTSRC)/install.sh
```

It is important to set the value of the SystemV symbol to either YES or NO in the platform file because (1) the value of parameters defined in *site.def* may depend on it, and the default value is not set until the system/build definition section (i.e., after *site.def*); (2) the default values of many parameters in later parts of the template depend on the value of SystemV. For instance, there is usually no *ranlib* under System V, so the default is defined as a no-op:

```
#ifndef RanlibCmd
#if SystemV
#define RanlibCmd /bin/true
#else
#define RanlibCmd ranlib
#endif
#endif
```

It is perhaps safer to rely on predefined *cpp* symbols (should they exist) in the platform-specific files than in the header block section of *Imake.tmpl* since once you know the system type the universe of such symbols is more constrained and their meanings cannot clash with those of symbols defined on other vendors' systems.

The platform files were significantly reorganized between R3 and R4. In R3, each platform file was expected to contain a definition for each of the build parameters (C compiler, loader, link command, etc.). Whenever a change was made that affected one platform file, it usually affected them all. Since the definitions were usually broadly similar across platforms, this was more work than necessary. The approach adopted in R4 is to supply best-guess values as the default definitions for these parameters in the template, which individual platform files may override as necessary. This set of defaults defines a baseline configuration. There are two advantages to this method. The platform files need only specify where they *differ* from the baseline, by overriding the default definitions; and changes or additions to the baseline do not necessarily require all platform files to be changed.

3.4. site.def

This file contains site-specific definitions. It is used to reflect local site-wide conventions, at least in theory. Should you wish to override the defaults, this is the place to indicate installation directories, whether to build the server and example programs, any special versions of programs to use during the build, etc.

The name *site.def* seems, to me at least, something of a misnomer. For instance, one of the things that is likely to be set there is HasLargeTmp, to indicate whether you have a large temp file directory. *site.def* is the place for this definition (it depends on your particular file system layout, so it doesn't go in the platform-specific file, and it's not *cpp*-guessable, so it doesn't go in the system description section). But a ''site'' can be a location at which multiple hosts are administered, and which may be configured dissimilarly. Some may have a large temp directory, others may not. Similarly, *site.def* is the logical place to specify that you want to use the GNU C compiler, but you might not necessarily want to use it on all hosts at a site. In some ways *host.def* might be a better name for this file.

3.5. System description and build parameter definitions

The platform- and site-specific files are followed by a section containing a number of default definitions. Generally, these describe system characteristics (e.g., does it have *vfork()*? does it have sockets?) or are related to management of the build and installation[7] processes (e.g., what is the name of the C compiler?

[7] Viz., *how* things should be installed, not *where*; defaults for the latter are in *Project.tmpl*.

does the loader need any special flags?). These are not X issues and so are segregated into their own section.

This section of *Imake.tmpl* should not be modified; definitions should be overridden in platform- or site-specific files.

3.6. Project.tmpl

This file contains definitions for parameters that are specific to X. Examples: whether to build debuggable versions of libraries, the screen resolution, what sorts of connections to accept (UNIX, TCP or DECnet sockets), where programs or libraries should be installed.

A number of the *make* variables in *Project.tmpl* are directory locations. Most of them are named in one of two ways, *XXX*DIR or *XXX*SRC. Variables of the first form are defined by equating them to the values *cpp* symbols having similar names (e.g., LIBDIR=LibDir). The *cpp* symbols are defined in the usual default-provided-but-override-allowed manner. This is because these variables refer to where things are to be placed at installation time, something which the installer might wish to change.

Variables of the second form, *XXX*SRC, indicate the layout of various source directories within the X distribution. They are not supposed to be changed, so there is no provision for overriding them; they are simply defined directly (e.g., SERVERSRC=$(TOP)/server), rather than being equated to *cpp* symbols.

Project.tmpl should not be modified; definitions should be overridden in platform- or site-specific files.

3.7. Imake.rules

This file contains *cpp* macros used to generate (sometimes quite complicated) *make* rules from concise descriptions of targets and dependencies. Since all the *cpp* symbol substitution functionality is available, these macros can be very powerful. On the other hand, the syntax used to express them is somewhat strange, in order to keep *cpp* from totally destroying the usefulness of the output for *make* purposes. In fact, one of the functions of *imake* is to undo some of the damage done by *cpp*; an uneasy alliance indeed.

Rules may be defined over multiple lines by appending ‘‘\’’ to the end of all lines but the last. Since *cpp* collapses all multi-line macros into a single line, a special trick is used to allow *imake* to post-process *cpp* output to figure out where to put the newlines back so that proper *make* syntax is preserved: ‘‘@@’’ in a rule is replaced by a newline in the resulting *Makefile*.[8]

Example: A simple rule to compile a program might be (this isn’t an actual X rule):

```
# define CompileProgram(program,objects,libraries)          @@\
program: objects                                            @@\
        cc –o program objects libraries
```

If invoked in an *Imakefile* as:

```
CompileProgram (xproga, main.o, –lm)
```

this would expand in the *Makefile* to:

[8] This all seems quite natural and obvious, now that I have worked with these things for a while. I remember, though, that the first few times I looked at the X *imake* rules, especially the more complicated ones, the formatting and syntax seemed so bizarre to me that a switch flipped in my brain, which then refused to comprehend anything at all. Repeated exposure gradually inured me to the infelicities of the syntax.

```
xproga: main.o
        cc –o xproga main.o –lm
```

If invoked as:

```
CompileProgram (xprogb, main.o parse.o scan.o, –lm –ll –ly)
```

this would expand to:

```
xproga: main.o parse.o scan.o
        cc –o xproga main.o parse.o scan.o –lm –ll –ly
```

All the rules in *Imake.rules* are defined using the standard override construct:

#ifndef *rulename*
#define *rulename* ...
#endif

Thus, even rules are subject to being overridden (something that was not true in R3). If a rule needs to be overridden for a particular platform, the default definition will be placed in the platform-specific file (*sgi.cf* does this). If a rule only needs redefining in a single directory, it may be better to undefine it and redefine right it in that directory's *Imakefile* (see *mit/config/Imakefile* for an example).

Imake.rules should not be modified; definitions should be overridden in platform- or site-specific files.

3.8. ./Imakefile

The last file included by the template is the *Imakefile* from the current directory. As already mentioned, *make* comments in this file are made *cpp*-safe by preceding them with ''/**/'' if necessary. The *Imakefile* indicates targets to be built and installed, and their dependencies, in terms of the *cpp* macros defined in *Imake.rules*. There may also be targets for generating dependencies, creating installation directories, creating lint libraries or tag files, etc.

Imakefile-writing is described in more detail in a later section.

3.9. Extra make rules

The last section of *Imake.tmpl* adds some common targets to the *Makefile*, such as a ''Makefile'' target to regenerate the *Makefile* itself, and default ''tags'' and ''clean'' targets. If the directory has subdirectories, rules to recurse through them for the ''install'', ''install.man'', ''clean'', ''tags'', ''Makefiles'' and ''includes'' targets are generated. These rules are added if IHaveSubdirs is defined and SUBDIRS is equated to the list of subdirectories in the *Imakefile*.

4. Building imake

Before you can build any part of X, *imake* itself must be built. ''make World'' in the top-level X directory builds *imake* automatically (in *mit/config*). The following discussion explains what happens during that process and some of the things you may need to do in preparation for the ''World'' build.. You want to read this section if ''make World'' dies when you try it, or you want to port X to another platform, or you want to build *imake* or adapt the X configuration files for use with other projects. It is best if you are willing to look at the following files: *Makefile.ini*; *imakemdep.h*; *ccimake.c*; *imake.c*.

4.1. What you are trying to accomplish

When you build *imake* you want to (1) get it to compile successfully, so you can use it; (2) guarantee that it makes sure *cpp* knows the trigger symbol—whether *cpp* predefines it or not—so you don't end up with generically-configured *Makefiles*. Conspiring to keep you from your goal are three dilemmas.

Dilemma #1: *imake* generates *Makefiles* for use in compiling programs; *imake* is compiled using a *Makefile*.

Dilemma #2: *imake* needs to know your system's trigger symbol in order to pass it along to *cpp* for proper *Makefile* generation. But *imake* doesn't know the trigger itself if neither *cc* nor *cpp* predefine it.

Dilemma #3: Some systems require special C compiler flags to ensure proper compilation of all but the most trivial programs. One of the facilities provided by *imake*-generated *Makefiles* is that these flags can be automatically specified. Of course, that is small consolation when the program to be compiled is *imake*—which happens to be just such a non-trivial program itself.

The solution to the first dilemma is to use a handwritten minimal file *Makefile.ini* instead.[9] You might need to hack *Makefile.ini* file slightly for your system—but only slightly. If you find yourself making large changes, that is a symptom you don't understand what you're supposed to be doing.[10] Stop and think some more first.

The second and third dilemmas are solved by manually passing the trigger symbol to the *imake* compilation and using a small bootstrap program that knows what special flags are necessary to get *imake* to compile without error. The commands in *Makefile.ini* that build *imake* look something like this:[11]

```
$(CC) –o ccimake $(BOOTSTRAPCFLAGS) $(CDEBUGFLAGS) ccimake.c
$(CC) –o imake $(BOOTSTRAPCFLAGS) $(CDEBUGFLAGS) imake.c ‘./ccimake‘
```

BOOTSTRAPCFLAGS is the *make* variable used to pass the trigger value for your system to *ccimake* and *imake* so they know the platform type. It is typically ‘‘BOOTSTRAPCFLAGS=–D*trigger*’’ if your *cpp* doesn't predefine the trigger, empty otherwise.[12] *ccimake* is the bootstrap program used to figure out any extra flags needed on your platform to get *imake* to compile. It is itself—by design—so simple that it should compile without any special treatment.

BOOTSTRAPCFLAGS is supplied to the *ccimake* build so it can select the necessary extra flags by platform type. Compiled with those flags, and the trigger value in BOOTSTRAPCFLAGS, *imake* builds correctly, and learns and memorizes the trigger for itself, to pass along to *cpp*. That in turn allows *cpp* to properly determine the correct header block in *Imake.tmpl* when *Makefiles* are generated.

4.2. Lay the groundwork

First, you need to determine what the trigger symbol is for your system, as well as the value of BOOTSTRAPCFLAGS. For existing ports, you can find out the trigger symbol by examining the header block section of *Imake.tmpl*. The value of BOOTSTRAPCFLAGS can be found in the platform *.cf* file for your system. Look for a line that defines BootstrapCFlags; if *cpp* predefines the trigger, there will likely be no such line. This means BOOTSTRAPCFLAGS is empty.[13] Otherwise the value will be something like

---

[9] There might be a *Makefile* in *mit/config* already, but it might not have been generated on your system, so it's not safe to use. It has a bug in it, anyway.

[10] Or a symptom that I've not explained very clearly what is supposed to happen...of course, I prefer the first possibility!

[11] The commands actually use CFLAGS, which is defined as ‘‘CFLAGS = $(BOOTSTRAPCFLAGS) $(CDEBUGFLAGS)’’; you get the idea.

[12] The BOOTSTRAPCFLAGS mechanism was not present in some earlier versions of X11 (e.g., R1). The number of systems to which X had been ported then was smaller and apparently all of them had *cpp*'s that predefined a unique symbol.

[13] A complication likely to arise in the future with regard to predefined preprocessor symbols is that ANSI C takes a

‘‘–D*trigger*’’, e.g., ‘‘–DmacII’’, ‘‘–Datt’’, ‘‘–Daix’’.

(**Note:** *Imake.tmpl* contains some comments near its beginning pertaining to new ports. These indicate that *imake.c*, and *main.c* in the *makedepend* source, should be modified. These comments are correct for R3 but evidently were not updated for R4; in both cases the modifications should be made to *imakemdep.h*. Similarly, the *README* file refers to *ccflags.c*, the R3 equivalent of *ccimake.c*.)

For new ports, you need to invent a trigger symbol that uniquely identifies your system and define BOOTSTRAPCFLAGS accordingly. Suppose you have a Brand X system. You can use ‘‘brandx’’ as the trigger and ‘‘BOOTSTRAPCFLAGS=–Dbrandx’’. You also must modify *imakemdep.h*. This is a header file #include’d by three programs, *ccimake*, *imake* and *makedepend*, and has one section for each.

The first section of *imakemdep.h* pertains to *ccimake*; it consists of a bunch of #ifdef/#endif blocks that define *imake_ccflags* according to the trigger symbol. *imake_ccflags* is defined as the flags needed to compile *imake* on your platform. For instance, if your Brand X system is System V-based, you need to specify that:

```
#ifdef brandx
#define imake_ccflags "–DSYSV"
#endif
```

ccimake simply writes the value of *imake_ccflags* to its standard output. Common flags in this definition are –DSYSV or –DUSG to indicate System V or USG systems. Other flags might also be necessary—see the ‘‘hpux’’ and ‘‘umips’’ blocks for some particularly unpleasant examples. If no special flags are necessary to compile *imake* (e.g., under Ultrix or BSD), there does not need to be any block for your trigger symbol, and *ccimake* simply writes out a default definition of *imake_ccflags* (currently –O).

You might have to fool around trying to compile *imake* by hand to determine the correct flags before you know how to define *imake_ccflags*.

The second section of *imakemdep.h* is for *imake*. It too has a set of #ifdef/#endif blocks, this time to select a trigger symbol definition based on the trigger symbol. That sounds circular and it is. These blocks add entries to *cpp_argv*[ ], which is an array of strings to be passed to *cpp* by *imake*. If your *cpp* predefines the correct trigger symbol automatically, you don’t need to do anything to this. Otherwise, this is where you want your trigger symbol to be listed, and there should be a block something like this:

```
#ifdef brandx
    "–Dbrandx",      /* for Brand X systems */
#endif
```

This causes *imake* to pass ‘‘–Dbrandx’’ to *cpp* to make it simulate predefinition of the trigger. When *cpp* reads the configuration files the trigger will have been defined and the correct header block in *Imake.tmpl* will be selected.

Following the sections for *ccimake* and *imake*, *imakemdep.h* contains a third section for *makedepend* (the compiled version; if you use the shell script version of *makedepend* in *mit/util/scripts*, this section of *imakemdep.h* is irrelevant). It looks for various system- and compiler-related definitions that may be predefined by *cc* and/or *cpp*. *makedepend* uses this information to be smart. Consider the following C program fragment:

---

dim view of all (or almost all) such. BOOTSTRAPCFLAGS is likely to be non-empty on more systems as implementations of ANSI C become distributed more widely.

```
#ifdef ultrix
#include "inca.h"
#else
#include "incb.h"
#endif /* ultrix */
```

If ‘‘ultrix’’ is defined, *makedepend* knows to generate a dependency for ‘‘inca.h’’ and not ‘‘incb.h’’; a dumb *makedepend* has to to assume dependencies on both.

It’s easiest simply to leave this section of *imakemdep.h* alone.

4.3. Build the program

You’re ready to begin (this should actually be simple if you’ve done the above correctly). The first thing to do is ensure the absence of any detritus that might be laying around from previous builds:

```
make –f Makefile.ini clean
```

Then build *ccimake* and *imake*. If your *cpp* predefines the trigger, you can build with:

```
make –f Makefile.ini
or
make –f Makefile.ini BOOTSTRAPCFLAGS=
```

Otherwise, specify the trigger explicitly. E.g., for Brand X, use:

```
make –f Makefile.ini BOOTSTRAPCFLAGS=–Dbrandx
```

4.4. Test your handiwork

If *imake* is supposed to pass a trigger symbol definition to *cpp*, you should test whether it actually does or not by executing the following command (**–T***/dev/null* provides an empty input template, and **–s***/dev/null* throws away the output so it doesn’t clobber the *Makefile* in your current directory):

```
imake –v –T/dev/null –s/dev/null
```

–v causes *imake* to print out the *cpp* command that it executes. If you don’t see a –D*trigger* in that command, *imake* wasn’t built properly.

4.5. Modifying other configuration files for a new platform

If you are developing a new port to another system, you have to do more than be able to compile *imake*, you also need to be able to use it in conjunction with the configuration files. First, you must modify the header block section of *Imake.tmpl* to add a block for your system. It will look something like this:

```
#ifdef brandx
#define MacroIncludeFile <brandx.cf>
#define MacroFile brandx.cf
#undef brandx
#define BrandxArchitecture
#endif /* brandx */
```

Second, you must create a platform-specific file *brandx.cf*. If your bootstrap flags are non-empty, this should contain, at minimum, ‘‘#define BootstrapCFlags –Dbrandx’’. Most certainly it will have other things in it, too. You will discover just what as you go through the process of porting the server and/or clients.

5. Building X, or, When is BOOTSTRAPCFLAGS necessary?

If you have gotten this far, you know you can get *imake* to build and can proceed to build X itself. *Before you do anything else*, copy the top-level *Makefile* somewhere safe (outside of the source tree). If something goes wrong and this file gets trashed, you want to be able to recover it.

Since this document is really about configuring X, not building it, the question addressed in this section is: how can you get all the *Makefiles* built?

The ‘‘do everything’’ operation in building X is ‘‘make World’’. It builds *imake*, generates all the *Makefiles*, removes extraneous object files, builds the header file tree, generates dependencies, and compiles everything. The only interesting part of from a configuration perspective is the first two steps.

The important questions are: (1) what is the value of BOOTSTRAPCFLAGS; and (2) when must BOOTSTRAPCFLAGS be specified? You have already answered the first question in determining how to build *imake*, so only the second is considered below.

When you build *imake* ‘‘manually’’ in *mit/config* using *Makefile.ini*, you must specify BOOTSTRAPCFLAGS explicitly if it is non-empty. What about when building from the top-level directory, e.g., ‘‘make World’’? It is a useful exercise to build *imake* by hand to test your understanding of the issues involved, but that won’t help you to do the ‘‘World’’ build—the first thing ‘‘make World’’ does, unfortunately, is throw *imake* away and rebuild it from scratch.

The X Release Notes (section 2) indicate that for ‘‘make World’’ you should specify BOOTSTRAPCFLAGS if you find a definition for BootstrapCFlags in the platform *.cf* file (which means there is no unique predefined *cpp* symbol). By implication, you do not need to specify BOOTSTRAPCFLAGS if there is no value of BootstrapCFlags in your platform *.cf* file. This means essentially that the empty value ‘‘BOOTSTRAPCFLAGS=’’ is sufficient to compile *imake* properly, which brings up a subtle point. Each *Makefile* created in the X distribution contains a line that says

> BOOTSTRAPCFLAGS=*something*

where *something* may or may not be empty. Thus, when you execute a command such as

> make World BOOTSTRAPCFLAGS=–Dbrandx

you are not just *providing* a value of BOOTSTRAPCFLAGS, you are also *overriding* the default value in the *Makefile*.

The implication that BOOTSTRAPCFLAGS need not be specified when BootstrapCFlags is not defined involves a hidden assumption, i.e., that BOOTSTRAPCFLAGS *already* has the empty value in the top-level *Makefile*. If you obtain your X distribution from a system on which X has been built and on which BOOTSTRAPCFLAGS is non-empty, that assumption is incorrect, so if you don’t explicitly provide an empty value by saying

> make World BOOTSTRAPCFLAGS=

to override the default in the *Makefile*, a value which is incorrect for your system is used instead. You can get some very strange results this way.

I submit that, in general, the deciding factor determining whether to specify BOOTSTRAPCFLAGS is not so much whether BootstrapCFlags is defined or not, but whether you've built your *Makefiles* yet. If not, you need to specify BOOTSTRAPCFLAGS, *even if it's empty*. Otherwise, you don't have to. In practice, this means that when you are building X on a new system, you should assume BOOTSTRAPCFLAGS in the *Makefile* is incorrect and explicitly override it on the command line. Normally this will be on the first ''World'' or ''mastermakefiles'' build.

The Release Notes do not cover the general case. Rather, they assume you are working from a virgin distribution (where the default BOOTSTRAPCFLAGS is in fact empty).

Once the *Makefiles* have BOOTSTRAPCFLAGS set properly, they will take care of propagating the correct value for you automatically—forever, even into another operation that rebuilds *imake* and the *Makefiles*. You can test this for yourself on a machine that has a non-empty BOOTSTRAPCFLAGS. Do ''make mastermakefiles BOOTSTRAPCFLAGS=–D*trigger*'', then ''make mastermakefiles''. You'll see that in the second case, BOOTSTRAPCFLAGS is supplied for you.

Note 1: The above discussion assumes that if BOOTSTRAPCFLAGS is not empty, BootstrapCFlags is defined correctly in the platform *.cf* file, because BOOTSTRAPCFLAGS normally gets its value from that symbol when *Makefiles* are generated.

Note 2: It is not necessary to do a ''World'' build to build *imake* and the *Makefiles*; there is a ''mastermakefiles'' target that will do so. This is a far more modest undertaking than doing the ''World'' build and you can check a few of the *Makefiles* afterward to see if the ''World'' build is likely to succeed or not: If BOOTSTRAPCFLAGS is correct and the correct header block was selected, you can then go ahead and do ''make World'' without having to specify BOOTSTRAPCFLAGS.

5.1. Alternatives to BOOTSTRAPCFLAGS

There are two alternatives to specifying BOOTSTRAPCFLAGS explicitly on the command line when you build the *Makefiles*, both sneaky and both deprecated. They are mentioned here in order to point out why you shouldn't use them.

First, you can edit *Makefile.ini* manually to set the value of BOOTSTRAPCFLAGS directly. The reason this is not a good idea is that if you give your X distribution to somebody else that doesn't have the same kind of machine, you've given away an explicitly misconfigured *Makefile.ini*. The recipient might fail to appreciate the subtle irony of this gesture.

Second, you can forget about BOOTSTRAPCFLAGS entirely. That's correct—you can build *imake* with wild abandon and total lack of regard for whether it knows about the correct trigger symbol or not. ''But on some systems, *imake* must pass the trigger definition to *cpp* explicitly,'' you say, ''and *imake* compiled in such a reckless and irresponsible manner may not do the job.'' Quite right. However... *imake* also examines your environment, and the value of IMAKEINCLUDE, if defined there, is passed to *cpp*. The value of IMAKEINCLUDE must begin with ''–I'' (or *imake* will reject it), but you can set it to ''–I. –Dbrandx'' and the trigger symbol will be passed through to *cpp* and your *Makefiles* will build happily.

This is sneaky because it uses the environment to affect the build process in a way that is not evident. Worse yet, if you actually install an *imake* built this way, it won't work for anyone else who isn't privy to the IMAKEINCLUDE convention.

(A third alternative is to edit BOOTSTRAPCFLAGS to the proper value in the top-level *Makefile* before ''make World''. This is functionally equivalent to specifying it on the command line and doesn't really gain you anything.)

(Might want to discuss here how BOOTSTRAPCFLAGS propagates during those *make* operations through use of the ImakeDependency() rule. Discuss what that rule checks.)

Discuss *xmkmf*, IMAKE_CMD and operation of UseInstalled here.

6. Imakeconoclasm—X configuration bugs

Ugh. Highly redundant.

‘‘make World’’ does a ‘‘make clean’’ after making all the *Makefiles*. This means that if you need to remake a *Makefile*, *imake* will have been removed. The rules for ‘‘make Makefile’’ include a check to see whether *imake* needs rebuilding, which is clever enough. Unfortunately, when they cause *imake* to be rebuilt they may not pass BOOTSTRAPCFLAGS. This means that every time you build a ‘‘Makefile’’ target, BOOTSTRAPCFLAGS needs to be specified if it’s not empty. **Give the fix.**

[Bug section] If what you are doing won’t build *imake*, you don’t need BOOTSTRAPCFLAGS. But it is not always apparent whether a given target will build it or not. If you don’t have UseInstalled defined, the rules for the‘‘Makefile’’ target (present in virtually every *Makefile*) contain a check to see whether *mit/config/imake* is actually built; if not, they build it. Thus even an apparently innocuous ‘‘make Makefile’’ in some obscure directory somewhere might trigger an *imake* build.

One bit of confusion that may strike you: Do ‘‘make World’’. This builds *imake*, generates all the *Makefiles*, cleans all the directories and finally builds the X stuff.

If you later rebuild the *Makefile* in any directory with ‘‘make Makefile’’, you might be surprised to find that *imake* is rebuilt first. This reason is that the rules for the ‘‘Makefile’’ target check whether *imake* exists, and if not they build it first. Since the ‘‘World’’ build does a clean after generating the *Makefiles*, *imake* was thrown away.

In fact, *imake* will be built the first time ‘‘make Makefile’’ is done even if the cleaning had not been done. The rules that check *imake*’s existence look for *Makefile* in *mit/config* and build *imake* with that if it’s there. (It will be because the ‘‘World’’ build generates it.) But the dependency for *imake* in *Makefile* is *imake.o* while the dependency in *Makefile.ini* (which *imake* is initially built with) is *imake.c*. This means that even if *imake* is present in *mit/config*, it will be built again because *imake.o* isn’t.

That isn’t a bug, but it can be confusing. What *is* a bug is that...

The IncRoot symbol is defined twice, once in *Imake.tmpl* and once in *Project.tmpl*. This is a minor nit, since neither file is supposed to be modified. However, if you use a copy of the configuration files as a base for your own projects, you probably *will* modify them; be aware that changing the definition in *Project.tmpl* has no effect if you leave both definitions in.

makedepend has some hardwired pathnames. This may bite you if you use *gcc* or compile on a MIPS system under BSD environment.

7. Courting disaster: How to do the wrong thing

imake is wonderful for portability when everything is configured properly. However, subtle syntax errors in the configuration files are often difficult to track down when you begin to make any significant changes to them. This section describes some of the misfortunes that may beset you should you make so bold as to engage in such an endeavor.

For X, the source of problems might be in any of *Imakefile*, *Imake.rules*, *Imake.tmpl*, the platform *.cf* file, or *site.def*. The most likely candidates are *site.def* and the platform file, though, since those are the only ones you’re supposed to edit. (If you’re writing an *Imakefile*, add that to the list, too.) Just remember that effects of mistakes in the early files may not be manifest until much later on. Problems may appear to originate at locations far from the actual cause.

(1) *Makefile* trashing

A number of problems during ‘‘make Makefile’’ can result in a trashed *Makefile*. If you have *xmkmf* working, you can use that to regenerate the *Makefile* after fixing whatever the problem was. Otherwise you must recover it somehow. If you have a copy of *Makefile* stashed somewhere, you can use that. If not: the first thing that ‘‘make Makefile’’ does is to move the original *Makefile* to *Makefile.bak*; if the new *Makefile* isn’t created properly, you can usually recover it with ‘‘cp Makefile.bak Makefile’’. Then you can fix the problem and try again. If *Makefile.bak* is trashed as well, you can grab the *Makefile* from another directory at the same level and use that (this works because they both contain the same ‘‘make Makefile’’ rule). You can also use a *Makefile* from a directory at another level, but you need to edit the line that sets TOP to reflect where the top project directory is in relation to the current directory.

(2) Boolean vs. existent/nonexistent symbols

Some symbols are used in boolean fashion and are defined as YES or NO. They are tested by ‘‘#if *symbol*’’ or ‘‘#if !*symbol*’’. Others are turned on simply by being defined. They are tested by ‘‘#ifdef *symbol*’’ or ‘‘#ifndef *symbol*’’. Failure to distinquish the sense in which a symbol is used can lead to problems. It is necessary to define symbols properly *and* to test them properly.

Example: YES/NO symbols must be defined properly. SystemV is such a symbol, and the test is ‘‘#if SystemV’’. If you use ‘‘#define SystemV’’ thinking that will turn it on, you’ll have problems; ‘‘#if SystemV’’ turns into just ‘‘#if’’ and generates:

```
cpp: /usr/tmp/tmp-imake.nnnn:line mmmm: syntax error
```

Example: YES/NO symbols must be tested properly. If you test such a symbol with #ifdef, the test will always succeed, whether the value is YES or NO; the symbol is defined in *both* cases.

Example: Existent/nonexistent symbols must be defined properly. A symbol such as UseInstalled is simply defined (as nothing) or left undefined. If you say ‘‘#define UseInstalled NO’’, thinking that will turn it off, you will be surprised. (The test ‘‘#ifdef UseInstalled’’ will succeed.)

Example: Existent/nonexistent symbols must be properly. Such a symbol may be defined as nothing. If you test it, incorrectly, with ‘‘#if symbol’’, the test will fail. Use #ifdef.

(3) Rule definition problems.

These can occur after ‘‘make Makefile’’, if a rule in *Imake.rules* is #define’d with a space between the rule name and the argument list:

```
#define rule(arglist)        right
#define rule (arglist)       wrong!
```

Some (all?) *cpp*’s will define rulename as ‘‘(arglist)’’. When this happens, you’ll see

```
make: line nnn: syntax error
```

and you’ll find ‘‘ (arglist)’’ on line nnn of the *Makefile* because ‘‘rule’’ wasn’t expanded properly (or at least not the way you expect). Fix it and try again.

Other similar errors can occur if rules are defined or used with spaces anywhere between the end of the macro name to the closing parenthesis of the argument list. This is particularly true if a macro argument is used to generate a rule target name. **Give example.**

(4) Failure to supply *cpp* symbol default values

When a *make* variable is equated to a *cpp* symbol, that symbol must be defined somewhere, even if it’s just defined as nothing. Otherwise the make variable will be set to the literal *cpp* symbol name. That is, if you have ‘‘MAKEVAR=CppSymbol’’, it must be preceded somewhere by:

```
#ifndef CppSymbol
#define CppSymbol whatever
#endif
```

or MAKEVAR will end up with the value ‘‘CppSymbol’’—literally.

(5) Macros expansion failure

Suppose you include the following line in your *Imakefile* so that a special version of your library will be compiled:

```
DebuggedAndProfiledLibraryObjectrule()
```

After you regenerate *Makefile*, you try to use it and get the message:

```
Make: Must be a separator on rules line nnn.  Stop.
```

This is symptomatic of a spelling error. The rule is actually ‘‘DebuggedAndProfiledLibraryObjectRule()’’; compare closely with the above. If you spell a rule name incorrectly, it won’t be expanded. Fortunately, these errors are easy to find, because at least the message tells you where to look. Unfortunately, you can’t say ‘‘make Makefile’’ after you fix *Imakefile*, because the effect of the error is to make your *Makefile* unusable.

8. Writing Imakefiles.

This section assumes that *imake*, *makedepend* and *xmkmf* are installed in a public directory somewhere. (implies configuration files are installed where *xmkmf* can get at them.) This means, in particular, that you should be able to do ‘‘xmkmf’’ to build a *Makefile* from an *Imakefile* in the current directory. Detailed knowledge of the guts of the configuration process is *not* assumed.

Give example null *Imakefile* here.

To test the syntactic correctness of your *Imakefile*:

```
xmkmf ; make Makefile
```

xmkmf builds the *Makefile* from scratch and fails if the configuration is bad. *make* builds a new *Makefile* using the one built by *xmkmf*, and fails if that one is not legal. (say how that can happen, e.g., spelling errors cause it.)

Note that the following command lines are not always equivalent:

```
make Makefile Makefiles
make Makefile ; make Makefiles
```

You normally want to build the ‘‘Makefiles’’ target when there are subdirectories. Suppose you add a new directory. This is done by changing SUBDIRS in the current directory’s *Imakefile*. You then need to rebuild the *Makefile* so that the definition of SUBDIRS is reset, and then rebuild *Makefile* in each of those subdirectories. The first command line above builds the ‘‘Makefiles’’ target using the (old) value of SUBDIRS from the (current) *Makefile*, which is incorrect. The second command line rebuilds the *Makefile*, then runs a second *make* process to build ‘‘Makefiles’’. The second process sees the (new) value of SUBDIRS in the (new) *Makefile*, which includes the new subdirectory, and has the intended result.

This is an instance of the ‘‘doesn’t work first time, works second time phenomenon’’...

Some rules may not work as you expect after changing *IMakefile*. This means problems can occur the first time you try ‘‘make *op*’’ that mysteriously disappear the second time. **[Illustrate this.]**

Don’t forget ‘‘make depend’’ after ‘‘make Makefile’’.

9. Other topics to be dealt with

Listed below are some other topics that should eventually find their way into this document. What do *you* want to see?

Shortcomings. No provision for building non-shell scripts, especially if ExecableScripts is NO. List disadvantages of *cpp* scripts.

Some *cpp* symbols have no *make* variable equivalent are exist only to control construction of the *Makefile*.

IHaveSubdirs, IHaveSpecialMakefileTarget

PassCDebugFlags—but related to CDEBUGFLAGS

UseInstalled