Appendix

cpp symbols and *make* variables defined in the X configuration files are listed and described here. Much of this information can be found in those files and in *mit/config/README*.

There are a number of places where I confess ignorance. If you fill me in on the missing pieces, I'll revise this document.

Besides symbols used for rule macros, symbols may be classed in three categories. Some are used as booleans and are defined as YES or NO. Some are simply defined as nothing or left undefined. Boolean and defined/not-defined symbols are rarely if ever associated with variables appearing in *Makefiles*; rather, they are used for control purposes to modify the configuration and build process.

The other symbols are general-purpose, usually (but not always) having a value that is assigned to some *Makefile* variable. The value may be empty (that is, the symbol may be defined as nothing):

```
#define     symbol      /* as nothing */
```

The descriptions below give symbol names, and their defaults and symbol class where applicable. A default of ''nothing'' means the empty definition. The format for symbol descriptions is:

SymbolName *symbol-class*
default: *default-value*

Note: The symbol UseInstalled is not defined in any of the configuration files. It should be defined (by you, as nothing) in *site.def* if you want to use installed versions of *imake*, *makedepend* and the configuration files. However, you should leave it undefined on the initial build, obviously, since the configuration programs and files will not have been installed yet.

1. Imake.tmpl

1.1. Global Constant Definitions

This section defines the two symbols YES and NO:

```
#define     YES     1
#define     NO      0
```

These are heavily used to define boolean symbols.

1.2. Header Blocks

Every header block in this section of *Imake.tmpl* defines two symbols:

MacroIncludeFile general
default: <generic.cf>
MacroFile general
default: generic.cf

> Both of these name the platform-specific file to be included by the template after the header block section. The defaults are deliberately rebarbative and are selected only if the correct header block for your system was not triggered. Otherwise these symbols will have values like <*platform*.cf> and *platform*.cf, where *platform* is ''ultrix'', ''sun'', ''hp'', ''apollo'', etc.
>
> There are two symbols associated with this name because it is used two different ways. MacroIncludeFile is used by *cpp* to include the platform file. MacroFile is used to generate file names.

In addition to the above two symbols, header blocks may define one or more of the following symbols to indicate the software and/or hardware platform.

AIXArchitecture	PS2Architecture
ATTArchitecture	PegasusArchitecture
ApolloArchitecture	RtArchitecture
CrayArchitecture	SGIArchitecture
HPArchitecture	SonyArchitecture
IBMArchitecture	StellarArchitecture
M4330Architecture	SunArchitecture
MacIIArchitecture	UltrixArchitecture
MipsArchitecture	VaxArchitecture

Some of these symbols may be defined in more than one header block. The platform file may also define other *xxx*Architecture symbols For instance, *sun.cf* does so to allow SPARC- or i386-dependencies to be selected.)

2. System/Build Definitions

2.1. Preprocessor Symbols

Symbols are listed in order of their appearance in *Imake.tmpl*. The default value of some symbols is determined based on whether SystemV is YES or NO. For these symbols both defaults are listed, with the SystemV=YES case first.

SystemV boolean
default: varies

SystemV indicates whether your system is System V based (YES) or not (NO). If not, that usually means BSD based. The default value of many other symbols depends on SystemV, so although it's given a value in *Imake.tmpl* if it doesn't already have one, it's usually set explicitly in the platform-specific files included earlier, to make sure it's correct.

OSMajorVersion general
default: 0

OSMinorVersion general
default: 0

These two symbols indicate the major and minor version numbers of the operating system. They are often used as control symbols in the platform files to select symbol values that vary depending on the OS version. For this reason, they are usually set in the platform file.

Example: if some feature of your OS didn't appear until version 5.7, you could test for that with:

```
/**/# these definitions would go in the platform file
# ifndef OSMajorVersion
# define OSMajorVersion 5
# endif
# ifndef OSMinorVersion
# define OSMinorVersion 7
# endif

/**/# this test would go wherever you need to test for 5.7 and up
# if OSMajorVersion > 5 || (OSMajorVersion == 5 && OSMinorVersion >= 7)
        /**/# 5.7-and-up specific stuff here
# endif
```

If your vendor names OS releases with more than two parts, you'll have to define other symbols in the platform file and use the appropriate tests. The inventiveness needed could reach heights of sublimity, for instance if you have to distinguish between SunOS 4.1.1 and SunOS 4.1.1 Rev. B. Ugh.

UnalignedReferencesAllowed boolean
default: NO

Some processors can dereference pointers to arbitrary addresses. Others cannot—inability to dereference odd addresses is a typical problem for such processors. If your pointers must be specifically aligned (e.g., on word boundaries), define UnalignedReferencesAllowed as NO. The default reflects a conservative approach.

ExecableScripts boolean
default: NO/YES

Whether the kernel *exec*() system call correctly executes scripts beginning with ''#!''. ExecableScripts determines the way the CppScriptTarget() rule macro is expanded and thus the way shell scripts are built.

Script prototypes begin with a line that says ''#!/bin/sh''. If ExecableScripts is NO, that line is stripped out and replaced with one that says '':'' instead.[1]

BourneShell general
default: /bin/sh

Path name of Bourne shell. This is used to set the value of SHELL, and in construction of some commands that run shell scripts.

ConstructMFLAGS boolean
default: YES/NO

Whether to construct MFLAGS *make* variable from MAKEFLAGS. This is used (I think) for compatibility with versions of *make* that use MFLAGS instead of MAKEFLAGS.

HasLargeTmp boolean
default: NO

Whether the file system on which */tmp* resides has lots of free space. The meaning of ''lots'' is roughly 1MB, although that may need to be revised upward for future releases. This symbol

[1] I would say that a better way to do it might be to explicitly write a line into the script using the value of ${SHELL} if ExecableScripts is YES and '':'' otherwise. This would allow the value of SHELL to propagate into scripts were it defined as something other than */bin/sh*.

determines the default value of ArCmd.

HasSockets boolean
default: NO/YES

Whether the system has BSD socket support.

HasVFork boolean
default: NO/YES

Whether the *vfork*() system call is supported. Presumably this causes *fork()* to be used instead of *vfork()* if HaveVFork is NO.

HasPutenv boolean
default: NO

Whether the *putenv*() C library call is supported.

HasVoidSignalReturn boolean
default: YES/NO

Whether the *signal*() C library call returns *void** (YES) or *int** (NO). The default is YES for System V platforms, but if you have a BSD platform, you should not assume NO. Some BSD-based systems (e.g., Ultrix and Sun) have changed from *int** to *void** in new OS releases. Some systems (e.g., Mips RISC/os) try to support both sets of semantics, which can make things fun.

HasBsearch boolean
default: YES/NO

Whether the *bsearch*() C library call is supported.

HasSaberC boolean
default: NO

Whether the system has the Saber C development environment. The default value of this symbol will usually be overridden in the platform file. **Warning:** some of the platform files define HasSaberC as YES (an MIT-ism?), so be sure to check this and change it if necessary.

HasFortran boolean
default: NO

Whether the system has a FORTRAN compiler. Needed for GKS support.

HasNdbm boolean
default: NO

Whether the system has the NDBM database manager (supercedes DBM).

HasDESLibrary boolean
default: NO

Whether the system has a DES (digital encryption standard) library.

NeedFunctionPrototypes boolean
default: NO
NeedWidePrototypes boolean
default: YES

These are used to support use of function prototypes. Apparently they are not well-liked judging from the deprecatory comments in the latter part of *Imake.tmpl*.

HasSunOSSharedLibraries boolean
default: NO

Whether SunOS shared libraries are supported.

SharedCodeDef general
default: varies

Flags to compile sharable library code.

SharedLibraryDef general
default: varies

Flags to turn on use of shared libraries.

HasSharedLibraries boolean
default: varies

Whether shared libraries are supported. **Note:** This symbol is defined *inside* the (convoluted) definition of SharedLibraryDef.

StripInstalledPrograms boolean
default: NO

If YES, strip symbol table from installed programs, otherwise leave intact. Setting to YES can save lots of disk space on some machines, but leaving it as NO can help during debugging. This symbol affects use of InstPgmFlags.

DestDir general
default: nothing

Prefix to prepend to installation directory pathnames. Normally this is left empty. If you want to install under an alternate root, the X folks recommend that you compile with the default value, then install with

make install DESTDIR=*/alt/root/dir*

UsrLibDir general
default: $(DESTDIR)/usr/lib

Directory under which to install system libraries (e.g., *lint* libraries; see description of LintlibDir under *Project.tmpl*). This is also used as the basis for the default value of LibDir (in *xdm*, *awm*, *twm*...)

IncRoot general
default: $(DESTDIR)/usr/include

Path to root of system header file hierarchy. X11 header files are installed under this. (A default for this symbol is also included in *Project.tmpl*; you can ignore that one because the definition in *Imake.tmpl* occurs first and will override it.)

UNCOMPRESSPATH general
default: /usr/ucb/uncompress

Command to run the *uncompress* program. I think this is used to uncompress compressed font files. If you override this symbol, the definition should probably be a full path name.

OptimizedCDebugFlags general
default: −O

C compiler flags to turn on optimization.

DebuggableCDebugFlags general
default: −g

C compiler flags to turn on debugging information.

NoOpCDebugFlags general
default: nothing

C compiler flags to not turn on optimization or debugging.

DefaultCDebugFlags general
default: OptimizedCDebugFlags

C compiler flags to turn on optimization for programs.

LibraryCDebugFlags general
default: OptimizedCDebugFlags

C compiler flags to turn on optimization for libraries.

DefaultCCOptions general
default: nothing

Special C compiler flags (other than −I's and −D's? See StandardDefines, StandardCppDefines). Useful, e.g., for turning on floating point options.

LibraryCCOptions general
default: DefaultCCOptions

Special C compiler flags for compiling libraries.

ServerCCOptions general
default: DefaultCCOptions

Special C compiler flags for compiling the server. (This is X-specific and should be in *Project.tmpl*, perhaps?)

PexCDebugFlags general
default: −g

C compiler flags to turn on debugging information for PEX. (This is X-specific and should be in *Project.tmpl*, perhaps?)

InstPgmFlags general
default: −s

Flags for installing programs. The default strips symbol tables. This symbol is used in conjunction with StripInstalledPrograms to set the *make* variable INSTPGMFLAGS. Note that INSTPGMFLAGS applies to installs of binary programs and also shell scripts; some systems give a warning when you try to strip a script because they have no symbol table (such warnings can be ignored).

InstBinFlags general
default: −m 0755

Flags for setting the mode of installed executable files.

InstUidFlags general
default: −m 4755

Flags for setting the mode of installed set-user-id files (e.g., *xterm* needs to *chown* pty's).

InstLibFlags general
default: −m 0664

Flags for setting the mode of installed libraries.

InstIncFlags general
default: −m 0444

Flags for setting the mode of installed header files.

InstManFlags general
default: −m 0444

Flags for setting the mode of installed manual pages.

InstDatFlags general
default: −m 0444

Flags for setting the mode of installed data (non-executable) files.

InstKmemFlags general
default: InstUidFlags

Flags for installing programs that need to read */dev/kmem*, e.g., *xload*. The default makes such programs setuid, which is overkill, since typically all that's needed is setgid to the group having access to */dev/kmem*. InstKmemFlags is a good symbol to define in *site.def*. Be sure to include the correct group, e.g., ''−g kmem −m 2755''. The group is not set in the default value of this symbol because it's difficult to determine a good default group.

ArCmd general
default: varies

Command to create archive files. If HasLargeTmp is YES, */tmp* is used for temp file space, otherwise the current directory is used.

BootstrapCFlags general
default: nothing

Defines needed to get *ccimake* compiled and by *imake* to get *cpp* to select the correct header block in *Imake.tmpl*.

CcCmd general
default: cc

Command to run the C compiler.

HasGcc boolean
default: NO

Whether the GNU C compiler, *gcc*, is available.

ServerCcCmd general
default: CcCmd

Command to run the C compiler to compile the server. (This is X-specific and should be in *Project.tmpl*, perhaps?)

LibraryCcCmd general
default: CcCmd

Command to run the C compiler to compile libraries.

FortranCmd general
default: f77

Command to run the FORTRAN compiler.

FortranFlags general
default: nothing

General flags for FORTRAN compiler.

FortranDebugFlags general
default: nothing

Flags for FORTRAN compiler to turn on debugging information.

AsCmd general
default: as

Command to run the assembler.

CompressCmd general
default: compress

Command to run *compress*.

CppCmd general
default: /lib/cpp

Command to run the C preprocessor.

PreProcessCmd general
default: CcCmd −E

Command to preprocess input. This is used by the script version of *makedepend*.

InstallCmd general
default: $(SCRIPTSRC)/bsdinst.sh *or* install

Command to install files. It should be compatible with the BSD version of *install*.

LdCmd general
default: ld

Command to run the loader.

LintCmd general
default: lint

Command to run *lint*.

LintLibFlag general
default: −o/−C

Flags to build *lint* libraries.

LintOpts general
default: −ax/−axz

Flags to tell *lint* how picky to be.

CpCmd general
default: cp

Command to copy files.

LnCmd general
default: ln/ln −s

Command to link files. Symbolic links are used if they're available (not System V).

MakeCmd general
default: make

Command to run *make*.

MvCmd general
default: mv

Command to rename files.

RanlibCmd general
default: /bin/true *or* ranlib

Command to run *ranlib*. On System V platforms this command is missing. */bin/true*, a nop, is used instead. (*/bin/true* always succeeds, which is important for *make*.)

RanlibInstFlags general
default: nothing

Flags to pass to *ranlib* when run on libraries that have already been installed. ''−t'' is a good option here, if you have it.

RmCmd general
default: rm −f

Command to remove files.

StandardCppDefines general
default: −DSYSV/nothing

−D's for C preprocessor.

StandardIncludes general
default: nothing

−I's for C compiler.

StandardDefines general
default: −DSYSV/nothing

−D's for C compiler.

NdbmDefines general
default: varies

Flags to turn on NDBM support. Affected by HasNdbm.

ExtraLibraries general
default: nothing

Extra libraries needed to get programs to compile.

ExtraLoadFlags general
default: nothing

Extra flags needed to get loader to work.

LdCombineFlags general
default: −X −r

Flags for incremental loader support.

CompatibilityFlags general
default: nothing

Flags to turn on R3 backward compatibility support.

TagsCmd general
default: ctags

Command to generate tags files.

LoaderLibPrefix general
default: nothing

Flags to specify before libraries (see *cray.cf*).

TOPDIR general
default: .
CURDIR general
default: .

Top of source tree, and current directory. The *make* variables associated with these (TOP, CURRENT_DIR) are overridden by recursive *make* rules. The defaults are correct for the top of the source tree. TOP and CURRENT_DIR are reset for each *Makefile* during ‘‘make Makefiles’’. TOP is the path to the top of the source tree, from the current directory. CURRENT_DIR is the path to the current directory from the top of the source tree.

FilesToClean general
default: *.CKP *.ln *.BAK *.bak *.o core errs ,* *~ *.a .emacs_* tags TAGS make.log MakeOut

List of files to be removed by ‘‘make clean’’. Note that ‘‘*~’’ is in this list, and see discussion of the RemoveProgramTarget() rule. If you are in the habit of using ‘‘cp x x~’’ to make a backup copy of files you modify, ‘‘make clean’’ will remove those copies, which may not be what you want.

2.2. Make Variables

Many of the *make* variables in *Imake.tmpl* are simply set to the preprocessor variable of the same name (e.g., INSTBINFLAGS = InstBinFlags).

Most *make* variables referring to commands are set to the preprocessor symbol having the same name and a suffix ‘‘Cmd’’. For instance, CC and MAKE are set to CcCmd and MakeCmd, respectively.

The rest of the variables are set to preprocessor symbols with names that differ a bit from the variable name (e.g., SHELL = BourneShell), or are defined in terms of other *make* variables (e.g., MFLAGS = −$(MAKEFLAGS)), or a combination of preprocessor symbols and *make* variables (e.g., RM_CMD = $(RM) FilesToClean).

3. platform.cf

Symbols which definitely should be set here are BuildServer (and X*xxx*Server if BuildServer is YES), OSName, OSMajorVersion, and OSMinorVersion. Some other symbols you should think about setting are CCompilerMajorVersion, CCompilerMinorVersion, BootstrapCFlags, SystemV, UnalignedReferencesAllowed, SetTtyGroup, ExecableScripts, and ConstructMFLAGS.

4. site.def

You should read through the *site.def* file and follow the comments there which tell whether you should supply override definitions for a number of symbols. This is the place to indicate whether you have a large temp area (HasLargeTmp), whether you have GNU *cc* (HasGcc), whether you want to compile in R3 backward compatibility (CompatibilityFlags), etc. It's also a good place to set symbols such as InstKmemFlags and BuildExamples.

Once you've built and installed *imake*, *makedepend* and the configuration files, you can rebuild the top-level *Makefile* with *xmkmf*, and rebuild the rest with ''make Makefiles''. After that the installed versions will be used.

5. Project.tmpl

5.1. Preprocessor Symbols

Symbols are listed in order of their appearance in *Project.tmpl*.

ProjectX general
default: 4

X11 Release number. The *README* says this is a boolean, which is incorrect.

BuildServer boolean
default: YES

Whether to build the server. This should be defined as NO in *site.def* on client-only systems. On platforms which have no server implementation, the value of BuildServer may already be set to NO in the platform file.

BuildExamples boolean
default: YES

Whether to build the example programs. It will speed up your builds to override this as NO, and less disk space will be used.

It is probably possible to build selected examples only. Define BuildExamples as YES, then build all the Makefiles from the source root. Then build the ''clean'' and ''depend'' targets. Define BuildExamples as NO and rebuild the Makefiles. The examples directories should be left alone since BuildExamples is NO, but you can then go into an example directory and build the program since its *Makefile* should be correct. (This is all conjecture.)

BuildXawBC boolean
default: NO

Whether to build backward-compatible X Athena Widgets (I think).

InstallXinitConfig boolean
default: NO

Whether to install sample *xinit* configuration files.

InstallXdmConfig boolean
default: NO

Whether to install sample *xdm* configuration files.

InstallAppDefFiles boolean
default: YES

Whether to install application defaults files.

InstallOldHeaderFiles boolean
default: NO

Whether to install R3 header files.

DebugLibX boolean
default: NO

Whether to build debugging into the X library.

DebugLibXArchive boolean
default: NO

Whether to build debugging into the archive file version of the X library.

ProfileLibX boolean
default: NO

Whether to build profiling into the X library.

ProfileLibXArchive boolean
default: NO

Whether to build profiling into the archive file version of the X library.

DebugOldLibX boolean
default: NO
ProfileOldLibX boolean
default: NO
DebugLibXt boolean
default: NO
ProfileLibXt boolean
default: NO
DebugLibXaw boolean
default: NO
ProfileLibXaw boolean
default: NO
DebugLibXext boolean
default: NO
ProfileLibXext boolean
default: NO
DebugLibXmu boolean
default: NO
ProfileLibXmu boolean
default: NO

Whether to build debugging and/or profiling into the old X library, X Toolkit library, X Athena Widget library, X Extension library or the X mu library.

NeedBerklibInXlib boolean
default: NO

Whether BSD emulation is needed in Xlib. If YES, then BandAidLibrary is set to YES, some other related symbols are defined, and the BandAidLibraryMake() rule macro is defined.

ManDirectoryRoot general
default: /usr/man

Path to the top of the manual page hierarchy relative to the default root directory. See also ManPath.

ManSuffix general
default: n

Suffix for program manual pages. See ManSourcePath, ManDir.

LibManSuffix general
default: 3

Suffix for library manual pages. See ManSourcePath, LibmanDir.

UNCOMPRESSFILT general
default: −DUNCOMPRESSFILT=\"UNCOMPRESSPATH\"

−D's to run *uncompress*.

BDFTOSNFFILT (default: general
default: −DBDFTOSNFFILT=\"$(BINDIR)/bdftosnf\"

−D's to run *bdftosnf* to convert fonts from binary distribution format to server natural format.

SHELLPATH general
default: −DSHELLPATH=\"/bin/sh\"

−D's for command to run the shell.

ZBDFTOSNFFILT general
default: −DZBDFTOSNFFILT=\"UNCOMPRESSPATH" | $(BINDIR)/bdftosnf $(FONTCFLAGS)\""

−D's to run *uncompress* and *bdftosnf* to convert compressed binary distribution format font files to server natural format.

FontFilters general
default: UNCOMPRESSFILT BDFTOSNFFILT SHELLPATH ZBDFTOSNFFILT

−D's for font conversion filters.

PrimaryScreenResolution general
default: 72

Resolution of default server screen, in dots per inch (dpi). This number applies both to horizontal and to vertical resolution; X assumes square pixels.

DefaultFontPath general
default: varies

Path to font directory. If not set somewhere else, a default is guessed, based on PrimaryScreenResolution, to select between 75 dpi and 100 dpi fonts.

DefaultRGBDatabase general
default: $(LIBDIR)/rgb

Path to RGB color database.

FontCompilerFlags general
default: −t

Flags for *bdftosnf*.

ConnectionFlags general
default: −DTCPCONN −DUNIXCONN

What kinds of connections the server will accept. Common values are −DTCPCONN, −DUNIX-CONN, −DSTREAMSCONN. More than one can be specified. If the server is built, at least one must be specified.

FontDefines general
default: −DFONT_SNF −DFONT_BDF −DCOMPRESSED_FONTS

−D's for font options.

ExtensionDefines general
default: nothing

−D's for server extensions. Default is nothing, for ''no extensions''. This is typically overridden in the platform file for servers that know about particular extensions, or in *site.def*.

ServerCDebugFlags general
default: OptimizedCDebugFlags

C compiler flags to turn on optimization/debugging in server.

LibraryDefines general
default: StandardDefines

−D's for compiling libraries.

ServerDefines general
default: StandardDefines ExtensionDefines

−D's for compiling the server.

CppSourcesPresent boolean
default: NO

Whether you have BSD *cpp* sources. If you define this as YES, make sure CppSources is defined correctly.

CppSources general
default: /usr/src/lib/cpp

Path to BSD *cpp* sources. Not used if CppSourcesPresent is NO.

BinDir general
default: $(DESTDIR)/usr/bin/X11

Where to install programs.

IncRoot general
default: $(DESTDIR)/usr/include

IGNORE the definition of this symbol in *Project.tmpl*; it's redundant. See its definition in the system/build section of *Imake.tmpl*.

BuildIncRoot general
default: $(TOP)

Relative path to (parent of) top of header file tree within source tree.

BuildIncTop general
default: ..

Top of header file tree within source tree, relative to $(TOP)/X11.

LibDir general
default: $(USRLIBDIR)/X11

X library installation directory (other directories are installed under this, e.g., FontDir, XinitDir).

ConfigDir general
default: $(LIBDIR)/config

Where to install configuration files.

LintlibDir general
default: $(USRLIBDIR)/lint

Where to install lint libraries. Note that the default uses USRLIBDIR, not LIBDIR. Also note that the name of this symbol is LintlibDir, not LintLibDir.

FontDir general
default: $(LIBDIR)/fonts

Where to install fonts.

AdmDir general
default: $(DESTDIR)/usr/adm

Where system log files are found.

XinitDir general
default: $(LIBDIR)/xinit

Where to install *xinit* files.

XdmDir general
default: $(LIBDIR)/xdm

Where to install *xdm* files.

XdmConfigurationSubdirectory general
default: config/default

Default *xdm* configuration directory.

AwmDir general
default: $(LIBDIR)/awm

Where to install *awm* files.

TwmDir general
default: $(LIBDIR)/twm

Where to install *twm* files.

GwmDir general
default: $(LIBDIR)/gwm

Where to install *gwm* files.

ManPath general
default: $(DESTDIR)ManDirectoryRoot

Full path to top of manual page hierarchy.

ManSourcePath general
default: $(MANPATH)/man

Path to top of manual page hierarchy, plus the common prefix of manual page directories there.

ManDir general
default: $(MANSOURCEPATH)ManSuffix

Where to install program manual pages.

LibmanDir general
default: $(MANSOURCEPATH)LibManSuffix

Where to install library manual pages. Note that it's LibmanDir, not LibManDir, which is inconsistent with being defined in terms of LibManSuffix.

XAppLoadDir general
default: $(LIBDIR)/app-defaults

Where to install application defaults files.

OsNameDefines general
default: nothing

I don't know what this is used for. It's referenced in *lib/Xdmcp/Imakefile* and *lib/Xmu/Imakefile*.

DefaultUserPath general
default: :/bin:/usr/bin:$(BINDIR):/usr/ucb

Default *xdm* user PATH environment variable.

DefaultSystemPath general
default: /etc:/bin:/usr/bin:$(BINDIR):/usr/ucb

Default *xdm* system PATH environment variable.

DefaultSystemShell general
default: BourneShell

Default shell.

UseCCMakeDepend boolean
default: NO

Whether to use the slow shell script version of *makedepend* (uses *cc −E*). It's best to use the compiled version if it works on your system. If the compiled version does not work as supplied with the distribution, it's probably worth a little effort to see if you can get it to work (particularly if you can't get the shell script version to work, either, which sometimes happens!). Possible problems: */usr/include* is hardwired into the code; if you use *gcc*, the library path is hardwired, too.

DependDir general
default: varies

Path to *makedepend* sources. The default is dependent on UseCCMakeDepend.

ContribDir general
default: $(TOP)/../contrib

Top of user-contributed source tree.

DependCmd general
default: $(DEPENDSRC)/makedepend

Command to run *makedepend.*

RemoveTargetProgramByMoving boolean
default: NO

This symbol affects expansion of the RemoveProgramTarget() rule. ‘‘Program’’ in that rule’s name is misleading; it can be used to remove any file, not just programs.

AllocateLocalDefines general
default: nothing

−D’s to turn on *alloca*().

SharedLibX boolean
default: varies
SharedOldLibX boolean
default: varies
SharedLibXt boolean
default: varies
SharedLibXaw boolean
default: varies
SharedLibXmu boolean
default: varies
SharedLibXext boolean
default: varies

Whether to build shared versions of the X, old X, X Toolkit, X Athena Widget, X mu or X Extension libraries. The defaults are YES if HasSunOSSharedLibraries is YES, NO otherwise.

SharedXlibRev general
default: 4.2
SharedOldXRev general
default: 4.0
SharedXtRev general
default: 4.0
SharedXawRev general
default: 4.0
SharedXmuRev general
default: 4.0
SharedXextRev general
default: 4.0

Version (revision) numbers of shared libraries.

HasInputExtension boolean
default: YES

Whether there are any input extensions.

DoInstallExtensionsIntoXlib boolean
default: NO

Whether the extension library should be merged into the X library.

BandAidLibraryNeeded boolean
default: YES

BandAidLibrarySources general
default: Berklib.c

BandAidLibraryObjects general
default: Berklib.o

BandAidLibraryMake() general
default: take a look

BandAidLibraryNeeded boolean
default: NO

Whether a band-aid library is needed. This will be YES if NeedBerklibInXlib is YES.

I'm not sure what all the band-aid stuff is for. Perhaps for BSD socket emulation on System V platforms.

XawClientDepLibs general
default: $(DEPXAWLIB) $(DEPXMULIB) $(DEPXTOOLLIB) $(DEPXLIB)

Library dependencies for clients using the Athena Widgets (DEPLIBS).

XawClientLibs general
default: $(XAWLIB) $(XMULIB) $(XTOOLLIB) $(XLIB)

Library needed by clients using the Athena Widgets (LOCAL_LIBRARIES).

NeedDefaultDepLibs boolean
default: YES

Affects the definitions of DEPLIBS, DEPLIBS1, DEPLIBS1, DEPLIBS3.

5.2. Make Variables

Many of the *make* variables in *Project.tmpl*, like those in *Imake.tmpl*, bear a direct name relationship to proprocessor symbols with which they are associated.

Since much of the project source and installation directory layout is specified in this file, the number of variables defined in terms of other *make* variables (e.g., XMUSRC = $(LIBSRC)Xmu, DEMOSRC = $(TOP)/demos) is quite large.

6. Imake.rules

The X *Imake.rules* file rule definitions make heavy use of the following techniques:

(1) Use of the "::" construct is quite prevalent. In a *make* rule, if a target name is followed by ":", it's assumed to be the only rule for that target. If another rule with the same target is found, it's an error. If target names are followed by "::", there may be multiple rules for the targets. The X configuration files rely on this property of "::" in the following ways:

(a) To make sure that certain targets are always present in the *Makefile*, even if they are not otherwise generated from expansion of the macros in *Imakefile*. This way, subdirectory-traversing *make* operations can rely on those targets being present. For instance, many rules that build programs automatically generate a ''clean'' target to remove the program. But if no such rule is used in the *Imakefile*, there might be no ''clean'' target in the *Makefile*, and ''make clean'' fails. That's not a disaster, but a ''make clean'' from the root of the source tree can result in a lot of apparent errors. To avoid this, an empty ''clean'' target is written to the *Makefile*, using ''clean::''. If there is another ''clean'' target, the empty one doesn't hurt anything. If there isn't, the empty one prevents error messages.

Other default targets generated are ''tags'', ''install'', ''install.man'', ''Makefile'', ''Makefiles'' and ''includes''.

(b) To allow commands like ''make all'' to trigger builds of an arbitrary number of targets. Without the ''::'' construct, the configuration process would need to be able to determine exactly which targets ''make all'' should generate, and produce a single ''all: target1 target2 ...'' rule. This would be a difficult task. Using ''::'', each program-building rule can generate its own ''all:: target*i*'' and they are all triggered.

(2) Definitions are allowed to be overridden. All rules are defined using the following construct:

```
# ifndef rulename
# define rulename(arguments) ...
# endif
```

This should look familiar because it's the same construct used in the other configuration files to allow symbol values to be overridden. Since *Imake.rules* is the last configuration file read during a *Makefile* build, *any* of its rules may be redefined. Doing so can have far-reaching consequences, of course, so that's not something you want to do without thinking it through first. Nevertheless, it's sometimes useful, or even necessary.

The likely places for such overriding to occur are:

platform.cf—if a system does things in some strange way that's not easy to compensate for by defining the usual configuration symbols properly. For example, the file *sgi.cf* redefines a couple of rules, to account for non-standard file extensions.

site.def—if a rule just isn't right for your site, for some reason.

Imakefile—it's sometimes useful to override a rule in a particular directory, which can be done by defining the rule in that directory's *Imakefile*. The file *config/Imakefile* does this, for instance. Such redefinitions do *not* carry down through subdirectories. Note that since *Imake.rules* is seen before *Imakefile*, you need to #undef the macro before redefining it to avoid ''symbol redefined'' errors.

(3) If a directory has subdirectories, and *make* operations in that directory should recurse down through the subdirectories, there should be a line that says

```
# define IHaveSubdirs
```

at the top of the *Imakefile*. This causes recursive ''install'', ''install.man'', ''clean'', ''tags'', ''Makefile'' and ''includes'' target rules to be written into the *Makefile* (see end of *Imake.tmpl*; the *make* variable SUBDIRS should set to the names of the subdirectories involved, as well). If you want CDEBUGFLAGS passed along to subdirectories, put another line at the top of the *Imakefile* that says

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

(4) As you scan through the rules, you'll notice that *if* constructs are not written like this:

if *condition* then; *action*; fi

Instead, they're written like this:

if *condition* then; *action*; else exit 0; fi

The reason for this is to deal with differing behaviors of *if* on various systems. On some systems, if you write ''if *condition*; then *action*; fi'' and *condition* is false, the value of the statement is zero (on the grounds that the test executed to completion without error). On other systems, the value of the statement is non-zero (on the grounds that the test failed). The latter behavior is broken behavior; Ultrix in particular is guilty of it.

In the X rules, *if* tests are usually used to check whether some preliminary action needs to be performed before proceeding to the next step. Systems that return false when the *if* test fails cause *make* to exit early if the preliminary action does not need to be done, and the next step is—incorrectly—never executed.

In order to get around this, all *if* statements explicitly include an *else* clause to return zero when the condition fails, so that *make* will continue whether the condition is true or not.

6.1. Commonly Used Rules

Rules for building programs:

NormalProgramTarget()
SimpleProgramTarget()
ComplexProgramTarget()
ComplexProgramTarget_1()
ComplexProgramTarget_2()
ComplexProgramTarget_3()

Rules for building shell scripts:

CppScriptTarget()
MakeScriptFromCpp()

Rules for building libraries:

NormalLibraryObjectRule()
NormalLibraryTarget()

Rules for installing files:

InstallProgram()
InstallLibrary()
InstallManPage()
InstallNonExec()
InstallSubdirs()

Rules for cleaning up:

RemoveTargetProgram()
CleanTarget()
CleanSubdirs()

Rules for generating Makefiles and dependencies:

MakefileTarget()
MakefileSubdirs()
DependTarget()
DependSubdirs()

Many rules that compile programs also generate ''clean'' and ''all'' targets.

6.2. Rule Descriptions

Each rule-generating macro in *Imake.rules* is described below, in the order in which they appear in that file. You should still read through the text of a rule before you use it to write your own *Imakefile*.

Disclaimer: I myself do not, for the most part, use the X rules, except in modified form for my own projects. There are likely to be a number of errors of understanding here.

SaberProgramTarget(program,srclist,objlist,locallibs,syslibs)

If HasSaberC is NO, this is a nop, otherwise it expands to a rule for ''saber-*program*'' and causes Saber C to load source and object files for the program. Odd that ''#'' is in front of the rules, though.

RemoveTargetProgram(program)

Removes *program* (actually this symbol is misnamed because it can be used to remove any file, not just programs). If RemoveTargetProgramByMoving (from *Project.tmpl*) is YES, *program* is moved to *program~* instead. This allows you to keep the previous version around. Note that ''*~'' is in the list of files removed by CleanTarget(), so unless you override FilesToClean, ''make clean'' will remove these previous target versions.

BuildIncludes(srclist,dstsubdir,dstupdir)

Builds the header file directories in the source tree, and populates them.

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

Builds one program. All object files and libraries are explicitly listed in the macro invocation. *program* is the program to be built. *objects* names the object files needed to build it from, e.g., *junk.o*. *deplibs* names the libraries that must exist before linking is done. *locallibs* names the libraries in the source tree that should be linked in. *syslibs* names the system libraries (e.g., in */usr/lib*) that should be linked in. These are named with *−l*, e.g., *−lm*.

A ''clean'' rule is also generated to remove the program (but not the objects).

Since this rule does not assume any default sources, you can use it several times in an *Imakefile* to build multiple programs. You should also set SRCS to the names of all the programs' source files and invoke DependTarget() in the *Imakefile*. Also invoke AllTarget(), giving it the names of all programs to build, and you'll need to include a rule to generate an ''install'' target for each program and an ''install.man'' target if there is a manual page. If you're a lint kind of person, you'll also want LintTarget().

SingleProgramTarget(program,objects,locallibs,syslibs)

This rule is deprecated. Don't use it.

SimpleProgramTarget(program)

This macro is used when the *Imakefile* is supposed to build a single program, and that program consists of a single source file. The macro sets SRCS to *program.c*, OBJS to *program.o* and calls ComplexProgramTarget(). The effect is to set up everything to generate rules to build the program, install it and its man page, clean the program, generate dependencies, and lint the program. A real whiz bang. See description of ComplexProgramTarget().

ComplexProgramTarget(program)

This macro is used in an *Imakefile* that builds a single program. Given the name of the program, rules are generated to build it, as well as generate ''all'', ''install'', ''install.man'', ''depend'' and ''lint'' targets, and to remove the program. Several assumptions are made by this macro:

(1) SRCS has been set to the list of source files (so the ''depend'' target works).

(2) OBJS has been set to the list of object files (so the program build works).

(3) The *deplibs*, *locallibs* and *syslibs* arguments as used in NormalProgramTarget() are assumed to be DEPLIBS, LOCAL_LIBRARIES and LDLIBS.

ComplexProgramTarget_1(program,locallib,syslib)
ComplexProgramTarget_2(program,locallib,syslib)
ComplexProgramTarget_3(program,locallib,syslib)

These rules are use in an *Imakefile* that builds 1, 2 or 3 programs. Actually, just 2 or 3 programs. If you were building a single program, you'd use ComplexProgramTarget(). Each macro takes 3 arguments. *program* is the name of the program to build. *locallib* names libraries built in the source tree that should be linked in (e.g., XMULIB). *syslib* names system libraries to be linked in (e.g., *−lm*).

The values of certain *make* variables are assumed to be set (by you). SRCS1, SRCS2 and SRCS3 name the sources for each program. (These are used to set SRCS so the ''depend'' target works.) OBJS1, OBJS2 and OBJS3 name the objects for each program. DEPLIBS1, DEPLIBS2 and DEPLIBS3 name the library dependencies for each program. PROGRAMS names the programs to be built.

If you're only building 2 programs, then SRCS3, OBJS3, etc., do not need to be set and will be assumed to have the empty value. PROGRAMS should be the names of 2 programs, not 3.

ServerTarget(server,subdirs,objects,libs,syslibs)

Generates rules to build a server. *libs* doubles as *deplibs* and *locallibs* as those are used in NormalProgramTarget(). The directories named by *subdirs* are used as dependencies and must exist before the target can be built.

InstallLibrary(libname,dest)

Installs *liblibname.a* into *dest/liblibname.a* and runs the command named by RANLIB on it. The installation flags INSTLIBFLAGS are used.

MergeIntoInstalledLibrary(tolib,fromlib)

Merges *fromlib* into *tolib* using the *mergelib* script in the SCRIPTSRC directory.

InstallSharedLibrary(libname,rev,dest)

Installs *liblibname.so.rev* into *dest/liblibname.so.rev*, using INSTLIBFLAGS.

InstallSharedLibraryData(libname,rev,dest)

Installs *liblibname.sa.rev* into *dest/liblibname.sa.rev*, using INSTLIBFLAGS.

InstallLibraryAlias(libname,alias,dest)

In *dest*, makes *libalias.a* an alias for *liblibname.a* by generating a link.

InstallLintLibrary(libname,dest)

Installs *llib-llibname.ln* into the *dest* directory using library installation flags INSTLIBFLAGS.

InstallManPageLong(file,destdir,dest)

Generate rule to install a manual page and rename it. The source manual page is *file.man* and is installed in *destdir/dest.*ManSuffix using installation flags INSTMANFLAGS. This is used to install pages whose names are long enough to tickle brain damage on systems that don't understand long file names.

InstallManPage(file,dest)

Generate rule to install the manual page *file.man* in *dest/file.*ManSuffix using installation flags INSTMANFLAGS.

InstallNonExec(file,dest)

Generate rules to install a non-executable file using installation flags INSTDATFLAGS.

InstallProgramWithFlags(program,dest,flags)

Generate rules to install an executable program using the specified flags.

InstallProgram(program,dest)

Generate rules to install an executable program. The comments in *Imake.rules* for this macro say something about INSTALLFLAGS, but that variable isn't actually used in the expansion of this macro, so I claim the flags used are simply INSTPGMFLAGS until someone demonstrates otherwise.

InstallScript(program,dest)

Generate rules to install *program.script* into *dest/program* using installation flags INSTPGMFLAGS.

This rule is obsolete. Use CppScriptTarget() and InstallProgram().

LinkFileList(step,list,dir,sub)

Generates rules for a target *step* to make links to all files in *list* in directory *dir*. The links are created in directory *sub*. Judging from the definition of this macro, *sub* is presumably so named because the rule changes directories into *dir* first and appears to expect *sub* to be a subdirectory. (Strictly speaking, that need not be true, since (i) *sub* could legally be an absolute path; (ii) *sub* could have a value such as ''../..''. But both those possibilities probably subvert the intent of the macro.)

InstallMultipleDestFlags(step,list,dest,flags)

Generates rules for a target *step* (e.g., ''install'') to install all the files in *list* into the directory *dir* using installation flags *flags*. The files in *list* are made the rule dependencies.

InstallMultipleDest(step,list,dest)

Like InstallMultipleDestFlags(), but assuming INSTALLFLAGS as the installation flags. Note: INSTALLFLAGS is not set by the X configuration files; you must give it a value in your *Imakefile*.

InstallMultiple(list,dest)

Like InstallMultipleDest(), but assuming ''install'' as the *step* argument.

InstallMultipleFlags(list,dest,flags)

Like InstallMultipleDestFlags(), but assuming ''install'' as the *step* argument.

InstallMultipleMan(list,dest)

Like InstallMultipleDest(), but assuming ''install.man'' as the *step* argument.

InstallAppDefaults(class)

If InstallAppDefFiles (from *Project.tmpl*) is #define'd, generates an ''install'' target to install *class.ad* into *$(XAPPLOADDIR)/class* using installation flags INSTAPPFLAGS.

INSTAPPFLAGS appears to be deprecated in *Project.tmpl*.

DependDependency()

If UseInstalled is not #define'd, this macro generates rules to check for the existence of *makedepend* in the source tree and build it if it's missing. This differs from the R3 rules, which failed if *makedepend* was missing.

DependTarget()

Generate rule to determine dependencies for source files named by SRCS. You have to define the value of SRCS yourself, unless you are using rules such as SimpleProgramTarget() or ComplexProgramTarget_{1,2,3}(), which do it for you. In the latter case, you need to define SRCS1, SRCS2 and SRCS3.

This macro uses DependDependency() to check for the existence of *makedepend* and build it if necessary.

CleanTarget()

Removes all the files named by FilesToClean and files whose names begin with ''#''.

To understand how this rule works, see how RM_CMD is defined in *Imake.tmpl*.

TagsTarget()

Generate rule to build a *TAGS* file from all **.c* and **.h* files.

ImakeDependency(target)

If UseInstalled is not #define'd, this macro generates rules to check for the existence of *imake* in the source tree and build it if it's missing. This differs from the R3 rules, which failed if *imake* was missing.

BuildMakefileTarget(imakefile,imakeflags)

Generate rules to build *Makefile*. The *imakefile* argument is superfluous (a bug?). Since it's ignored, *Imakefile* is assumed as the source from which to build *Makefile*. *imakeflags* is any special flags you want passed to the *imake* command.

This macro uses ImakeDependency() to check for the existence of *imake* and build it if necessary.

You don't use this rule explicitly unless you've #define'd IHaveSpecialMakefileTarget in your *Imakefile*.

MakefileTarget()

Generate rules to build a *Makefile* target from *Imakefile*, so you can say ''make Makefile''. This target is automatically included in your *Makefile* when it is rebuilt by *imake*, unless ''#define IHaveSpecialMakefileTarget'' is present in your *Imakefile*, in which case you should include an invocation of BuildMakefileTarget() to build *Makefile*.

NormalLibraryObjectRule()

Generate rule to turn a *.c* file into a *.o* file. Include this macro (once) in your *Imakefile* if you're building a library from C source.

NormalFortranObjectRule()

Generate rule to turn a *.f* file into a *.o* file. Include this macro (once) in your *Imakefile* if you're building a library from FORTRAN source. (FORTRAN? Isn't that something used to build COBOL compilers?)

ProfiledLibraryObjectRule()

Like NormalLibraryObjectRule(), but in addition to building the normal target, a profiled target is built in the *profiled* subdirectory of the current directory. *profiled* is created if necessary. A ''clean'' target to remove profiled objects is also generated.

DebuggedLibraryObjectRule()

Like ProfiledLibraryObjectRule(), but instead of profiled targets in *profiled*, debuggable targets are built in the *debugger* subdirectory.

DebuggedAndProfiledLibraryObjectRule()

A combination of ProfiledLibraryObjectRule() and DebuggedLibraryObjectRule().

SharedLibraryObjectRule()

Like ProfiledLibraryObjectRule(), but instead of profiled targets in *profiled*, sharable targets are built in the *shared* subdirectory.

SharedAndDebuggedLibraryObjectRule()

A combination of SharedLibraryObjectRule() and DebuggedLibraryObjectRule().

SpecialSharedAndDebuggedObjectRule(objs,depends,options)

Like SharedAndDebuggedLibraryObjectRule(), but allows targets (*objs*), dependencies and special compiler flags to be specified.

SpecialSharedObjectRule(objs,depends,options)

Like SharedLibraryObjectRule(), but allows targets (*objs*), dependencies and special compiler flags to be specified.

SpecialObjectRule(objs,depends,options)

Like NormalLibraryObjectRule(), but allows targets (*objs*), dependencies and special compiler flags to be specified.

SpecialProfiledObjectRule(objs,depends,options)

Like ProfiledLibraryObjectRule(), but allows targets (*objs*), dependencies and special compiler flags to be specified.

SpecialDebuggedObjectRule(objs,depends,options)

Like DebuggedLibraryObjectRule(), but allows targets (*objs*), dependencies and special compiler flags to be specified.

SpecialDebuggedAndProfiledObjectRule(objs,depends,options)

Like DebuggedAndProfiledLibraryObjectRule(), but allows targets (*objs*), dependencies and special compiler flags to be specified.

NormalLibraryTarget(libname,objlist)

Generate rules to create the library *liblibname.a* from the object files named by *objlist*. The library is RANLIB'd after being created.

NormalSharedLibraryTarget(libname,rev,solist)

Generate rules to create a shared library *liblibname.so.rev*. The library is created under a different name and then moved onto the target name to maximize availability of any existing version.

NormalSharedLibraryDataTarget(libname,rev,salist)

Like NormalSharedLibraryTarget(), but for shared data *liblibname.sa.rev*. The comments claim the library is created under a different name and then moved onto the target name as for NormalSharedLibraryTarget() but that doesn't appear to be true from the text of the macro.

NormalLibraryTarget2(libname,objlist1,objlist2)

Like NormalLibraryTarget(), but adds two lists of files to the library, in two steps. Used to create libraries with large numbers of files.

ProfiledLibraryTarget(libname,objlist)

Generate rules to create the library *liblibname_p.a* from the profiled objects in the *profiled* subdirectory.

DebuggedLibraryTarget(libname,objlist)

Generate rules to create the library *liblibname_d.a* from the debuggable objects in the *debugger* subdirectory.

AliasedLibraryTarget(libname,alias)

Generate rules to make *libalias.a* a link to *liblibname.a*.

NormalRelocatableTarget(objname,objlist)

Generate rules to produce a relocatable object file *objname.o* from the object files named in *objlist*. This is built using the loader LD, called with flags LDCOMBINEFLAGS.

ProfiledRelocatableTarget(objname,objlist)

Like NormalRelocatableTarget(), but builds *objname_p.o* and uses loader flag −X instead of LDCOMBINEFLAGS, to produce a profiled relocatable object file.

DebuggedRelocatableTarget(objname,objlist)

Like NormalRelocatableTarget(), but builds *objname_d.o* and uses loader flag −X instead of LDCOMBINEFLAGS, to produce a debuggable relocatable object file.

LintLibraryTarget(libname,srclist)

Generate rules for a ‘‘lintlib’’ target that builds *llib-llibname.ln* from the source files named by *srclist*.

NormalLintTarget(srclist)

Generate rules for a ‘‘lint’’ target that lints the source files named by *srclist*, and for a ‘‘lint1’’ target that lints the file named by the *make* variable FILE (which you must set in *Imakefile*).

LintTarget()

Generate rules to lint files named by SRCS. SRCS might be set by you in *Imakefile*, or, if you invoke macros such as SimpleProgramTarget() or ComplexProgramTarget_{1,2,3}(), it will be done for you automatically.

LinkSourceFile(src,dir)

Generate rule to make a link to a file in another directory. The macro can be used to create links to non-source files, too, of course.

MakeSubincludesForBuild(step,dir,srclist)

Generate rules for target *step* to ... to ... uh, I don’t know!

NamedTargetSubdirs(name,dirs,verb,flags,subname)

Generate rules for a target *name* (e.g., ‘‘depend’’, ‘‘install’’) that causes the target *subname* to be built in the subdirectories named by *dirs*. This is done by changing into each subdirectory in turn and invoking *make* for the *subname* target. *flags* is any extra flags that should be passed to those *make* processes. *verb* is the -ing form of *name*, e.g., if is ‘‘make’’, then *verb* is ‘‘making’’. This is used to echo an indication of what *make* is doing in each directory as it proceeds.

This macro is used by virtually all other macros that have names like XXXSubdirs().

NamedMakeSubdirs(name,dirs)

Invokes NamedTargetSubdirs() to generate rules for a target *name* that causes the ‘‘all’’ target to be built in the subdirectories named by *dirs*.

If you want CDEBUGFLAGS passed to the *make* processes forked in each subdirectory, the *Imakefile* should have a line at the top that says:

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

MakeSubdirs(dirs)

Equivalent to NamedMakeSubdirs(all,dirs).

DependSubdirs(dirs)

Invokes NamedTargetSubdirs() to generate rules for a ‘‘depend’’ target that causes the ‘‘depend’’ target to be built in the subdirectories named by *dirs*.

InstallSubdirs(dirs)

Invokes NamedTargetSubdirs() to generate rules for a ‘‘install’’ target that causes the ‘‘install’’ target to be built in the subdirectories named by *dirs*.

This macro is automatically invoked if IHaveSubdirs is #define’d in *Imakefile*, in which case SUBDIRS should be set to the names of the subdirectories involved.

InstallManSubdirs(dirs)

Invokes NamedTargetSubdirs() to generate rules for a ‘‘install.man’’ target that causes the ‘‘install.man’’ target to be built in the subdirectories named by *dirs*.

This macro is automatically invoked if IHaveSubdirs is #define’d in *Imakefile*, in which case SUBDIRS should be set to the names of the subdirectories involved.

IncludesSubdirs(dirs)

Invokes NamedTargetSubdirs() to generate rules for a ‘‘includes’’ target that causes the ‘‘includes’’ target to be built in the subdirectories named by *dirs*.

This macro is automatically invoked if IHaveSubdirs is #define’d in *Imakefile*, in which case SUBDIRS should be set to the names of the subdirectories involved.

NamedCleanSubdirs(name,dirs)

Invokes NamedTargetSubdirs() to generate rules for a target *name* that causes the ‘‘clean’’ target to be built in the subdirectories named by *dirs*.

CleanSubdirs(dirs)

Equivalent to NamedCleanSubdirs(clean,dirs).

This macro is automatically invoked if IHaveSubdirs is #define’d in *Imakefile*, in which case SUBDIRS should be set to the names of the subdirectories involved.

NamedTagSubdirs(name,dirs)

Invokes NamedTargetSubdirs() to generate rules for a target *name* that causes the ‘‘tags’’ target to be built in the subdirectories named by *dirs*.

TagSubdirs(dirs)

Equivalent to NamedTagSubdirs(tags,dirs).

This macro is automatically invoked if IHaveSubdirs is #define’d in *Imakefile*, in which case SUBDIRS should be set to the names of the subdirectories involved.

MakeLintSubdirs(dirs,target,subtarget)

Invokes NamedTargetSubdirs() to generate rules for a target *target* that causes the target *subtarget* to be built in the subdirectories named by *dirs*.

LintSubdirs(dirs)

Equivalent to MakeLintSubdirs(dirs,lint,lint).

This macro is *not* automatically invoked if IHaveSubdirs is #define’d in *Imakefile*; you need to do so yourself.

MakeLintLibSubdirs(dirs)

Equivalent to MakeLintSubdirs(dirs,lintlib,lintlib).

MakeMakeSubdirs(dirs,target)

Generate rules for a target *target* to generate a *Makefile* in the subdirectories named by *dirs*.

In R4, ''Makefiles'' rules handle values of ${TOP} that are either relative or absolute. The R3 rule only handled relative values.

MakeNsubdirMakefiles()

Generate a set of very very very ugly but quite wonderful rules for use by MakeMakeSubdirs(). Automatically invoked by MakefileSubdirs(); you shouldn't need to do so yourself.

MakefileSubdirs(dirs)

Generate rules to build ''Makefiles'' targets in the given subdirectories.

This macro is automatically invoked if IHaveSubdirs is #define'd in *Imakefile*, in which case SUBDIRS should be set to the names of the subdirectories involved.

CppScriptTarget(dst,src,defs,deplist)

Generate rules to build a shell script *dst* from *src*. It figures out how to do this correctly for the current platform depending on the value of ExecableScripts. *defs* is any extra flags, such as −D's, to pass through to the preprocessor. *deplist* names any targets that must exist before *dst* can be built.

MakeScriptFromCpp(name,defs)

Generate rule to build a script *name* from *name.cpp*. *defs* is as for CppScriptTarget().

MakeDirectories(step,dirs)

Generate rules for target *step* to create the directories named by *dirs*. This is often used to make sure that installation directories exist before trying to install files in them. (Not all verions of *install* create missing directories.)

MakeFonts()

Generate rules for ''fonts.dir'' target to build font database *fonts.dir*. The *make* variable OBJS must be set to the name of the files that should exist before the database is built.

InstallFonts(dest)

Generate rules to install the directory *fonts.dir* into *dest* using installation flags INSTDATFLAGS.

InstallFontAliases(dest)

Generate rules to install font alias database *fonts.alias* into *dest* using installation flags INSTDATFLAGS.

FontTarget(basename)

Generate rules to create a font file by converting *basename.bdf* to *basename.snf*.

CompressedFontTarget(basename)

Generate rules to create a compressed font file by converting *basename.bdf* (to *basename.snf.Z*, I suppose).

AllTarget(depends)

Generate rule for an ‘‘all’’ target to build targets named by *depends*.

7. Imakefile

If HasSaberC is #define’d, some special Saber-C rules are generated.

Some symbols that you may want to #define (as nothing) at the top of *Imakefile* are:

IHaveSpecialMakefileTarget

If not #define’d (the normal case), MakefileTarget() is automatically invoked. If #define’d, it’s assumed that you want to explicitly invoke a *Makefile*-building macro yourself.

MakefileAdditions

If #define’d, MakefileAdditions() is automatically invoked. However, MakefileAdditions() appears not to be defined anywhere, and I have no idea what it is supposed to do.

IHaveSubdirs

If #define’d, you should set SUBDIRS to the names of the current directory’s subdirectories, and subdirectory-traversing rules for generating ‘‘Makefile’’, ‘‘install’’, ‘‘install.man’’, ‘‘clean’’, ‘‘tags’’ and ‘‘includes’’ targets will be automatically invoked.

If you set SUBDIRS to the empty value (‘‘SUBDIRS=’’) and rebuild the *Makefile*, you will either be in for a long wait whenever you make any of the recursive targets above or you’ll get a syntax error.