APPENDIX B

Khoros Imake Symbols and Variables

Khoros Imake Document

Compiled By:

Tom Sauer and Charlie Gage

Printed: March 13, 1992

Khoros Imake Symbols and Variables

A. INTRODUCTION

The *cpp* symbols and **make** variables that define the Khoros configuration files in *KHOROS_HOME/repos/config/imake_conf* are listed and described in this appendix.

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 defined as nothing, and some are 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, a short description and their defaults. A default of ''nothing'' means the empty definition. The following symbol definitions are found in the *Khoros.tmpl* file, the *Site.def* file, the *Library.def* file and the *platform.cf* files.

A.1 SYMBOL DESCRIPTIONS

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.

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.

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 it is usually set explicitly early in the platform-specific file 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 && OSMinorVersion >= 7
      /**/# 5.7-and-up specific stuff here
# endif
```

ExecableScripts boolean
default: NO(SystemV)/YES(BSD)

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 ‘‘:’’.

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(SystemV)/NO(BSD)

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

HasLargeTmp boolean
default: YES

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 determines the default value of ArCmd.

HasSockets boolean
default: NO(SystemV)/YES(BSD)

Whether the system has BSD socket support.

HasVFork boolean
default: NO(SystemV)/YES(BSD)

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.

HasRandom boolean
default: YES

Whether the *random*() C library call can be used instead of *rand().*

HasDirent boolean
default: NO

Whether the *dirent* directory structure can be used instead of *direct.*

HasBsdTypes boolean
default: YES

Identifies how the system defines *u_char, u_short,* etc.

HasWait3 boolean
default: YES

Whether the system supports *wait3* system call.

ReDefFortran boolean
default: NO

Whether C calls fortran as *name_*

HasVoidSignalReturn boolean
default: YES(SystemV)/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 machines try to support both sets of semantics.

HasBsearch boolean
default: YES(SystemV)/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, so be sure to check this and change it if necessary.

UseTiff boolean
default: YES

Whether the viff2tiff*tiff*library, and *tiff2viff* routines should be compiled. The default value of this symbol is YES (compile all tiff dependent code). This symbol should be set to NO when compiling on the Cray.

HasFortran boolean
default: YES

Whether the system has a FORTRAN compiler. Needed for Linpack, Eispack, and FFT support.

ForSysLibs general
default: -lF77 -lI77

Specifies the Fortran libraries that your machine uses. If *HasFortran* is YES, then the default above is used, otherwise it is set to *nothing*. If *ForSysLibs* is defined in the machine.cf file, that will override this definition.

ForLocalLibs boolean
default: $(LINPACKLIB) $(EISPACKLIB)

The libraries needed for Linpack, and Eispack.

ForLocalDependLibs boolean
default: $(DEPENDLINPACKLIB) $(DEPENDEISPACKLIB)

The libraries needed for Linpack, and Eispack.

SharedCodeDef general
default: varies

Flags to compile sharable library code.

SharedLibraryDef general
default: varies

Flags to turn on use of shared libraries.

SoeLim boolean
default: NO

Flag that determines if the public domain version of *soelim* in Khoros should be installed. This is a work-a-round for a bug in the SUN OS *soelim* program. This should be set to YES for a SUN and NO for all other architectues.

HasSharedLibraries boolean
default: varies

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

The following nine symbols are used to set the data transports that Khoros will use.

UseFile boolean
default: YES

This flag specifies Khoros to use the Standard file system transport. This should always be set to YES.

UseSockets boolean
default: NO

This flag specifies Khoros to use the bsd Sockets transport. If set to YES, then the socket code will be compiled, otherwise the socket code will not be compiled.

UseSharedMem boolean
default: NO

This flag specifies Khoros to use the Shared Memory transport. If set to YES, then the Shared Memory code will be compiled, otherwise this code will not be compiled.

UseSunVirMem boolean
default: NO

This flag specifies Khoros to use the SUN Virtual Memory transport. If set to YES, then the Virtual Memory code will be compiled, otherwise this code will not be compiled. This transport mechanism is not yet implemented.

UsePipes boolean
default: NO

This flag specifies Khoros to use the Pipes as the data transport mechanism. If set to YES, then the Pipes code will be compiled, otherwise this code will not be compiled.

UseStream boolean
default: NO

This flag specifies Khoros to use Named Pipes as the data transport mechanism. If set to YES, then the stream code will be compiled, otherwise this code will not be compiled.

UseSunRPC boolean
default: NO

This flag specifies Khoros to use Sun's remote procedure call as the data transport mechanism. If set to YES, then the RPC code will be compiled, otherwise this code will not be compiled. This transport mechanism is not yet implemented.

UseKTLI boolean
default: NO

This flag specifies Khoros to use System V transport layer interface as the data transport mechanism. If set to YES, then the KTLI code will be compiled, otherwise this code will not be compiled. This transport mechanism is not yet implemented.

UseApolloRPC boolean
default: NO

This flag specifies Khoros to use Apollo's e transport layedure call as the data transport mechanism. If set to YES, then the ApolloRPC code will be compiled, otherwise this code will not be compiled. This transport mechanism is not yet implemented.

StripInstalledPrograms boolean
default: YES

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. Unused symbol in Khoros at this time.

IncRoot general
default: KHOROS_HOME/include

Path to root of the Khoros system header file hierarchy. Khoros header files are installed under this.

UNCOMPRESSPATH general
default: /usr/ucb/uncompress

Command to run the *uncompress* program. This is used to uncompress compressed postscript files for the manual. 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.

DefaultYFlags general
default: -d

Default flags for yacc. Specify the default YFLAGS to re-generate (update) the y.tab.h everytime the {file}.y is yacc'ed.

DefaultLFlags general
default: -n

Default flags for lex.

LibraryCCOptions general
default: DefaultCCOptions

Special C compiler flags for compiling libraries.

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.

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.

InstScriptFlags general
default: -m 0755

Flags for setting the mode of installed shell scripts.

ArCmd general
default: ar cq (HasLargeTmp)/ar clq (use local dir)

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 GNU CC, *gcc*, is available.

LibraryCcCmd general
default: CcCmd

Command to run the C compiler to compile libraries. If *HasGcc* is defined, then *CcCmd* is defined as, gcc -bsd -DNOSTDHDRS -fstrength-reduce -fpcc-struct-return -fwritable-strings

FortranCmd general
default: f77 -O

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.

FortranScript general
default: $(SCRIPTSRC)/fortran.sh (!HasFortran)/nothing

Flags for FORTRAN script.

CompressCmd general
default: compress

Command to run *compress*.

CppCmd general
default: /lib/cpp

Command to run 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 / install

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

LdCmd general
default: $(CC)

Command to run the loader. Default is to use CC as defined to be cc or gcc.

LibraryLdCmd general
default: LdCmd

Command to run the loader for libraries.

CpCmd general
default: cp

Command to copy files.

LnCmd general
default: ln (SystemV) / ln -s

Command to link files. Symbolic links are used if 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 (SystemV) / 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 (SystemV) / nothing

-D's for C preprocessor.

LocalDefines general
default: nothing

Local defines.

StandardIncludes general
default: nothing

-I's for C compiler.

XIncludes general
default: nothing

Includes for X11.

ExtraIncludes general
default: nothing

For other includes.

StandardDefines general
default: -DSYSV (SystemV) / nothing

-D's for C compiler.

ExtraLibs general
default: nothing

Extra libraries needed to get programs to compile.

ExtraLoadFlags general
default: nothing

Extra flags needed to get loader to work.

TagsCmd general
default: ctags

Command to generate tags files.

LocalLibs general
default: see Library.def file

Specifies the standard Khoros libraries.

LocalDependLibs general
default: see Library.def file

Specifies the Khoros depend libraries.

XLocalLibs general
default: see Library.def file

Specifies the X libraries that Khoros depends on.

XLocalDependLibs general
default: see Library.def file

Specifies the X depend libraries that Khoros uses.

LoaderLibPrefix general
default: -L

Flags to specify before libraries (see *cray.cf*). Cray uses -l

BaseName general
default: /usr/bin/basename

Specifies where basename lives on your machine.

SrcTop
default: KHOROS_HOME/src

Specifies the top of the Khoros source tree.

FilesToClean general
default: *.o core *.a tags TAGS

List of files to be removed by ‘‘make clean’’.

HasTempnam general
default: YES

Specifies whether to use temporary file names.

GhostWriter general
default: $(BINDIR)/ghostwriter

Specifies the path to the Khoros *ghostwriter* program.

GhostReader general
default: $(BINDIR)/ghostreader

Specifies the path to the Khoros *ghostreader* program.

GhostAns general
default: ./ghost.ans

Sepeifies the filename of the *ghost*routine answer file.

ManSuffix general
default: 1

Suffix for the Khoros program manual pages.

LibManSuffix general
default: 3

Suffix for the Khoros library manual pages.

LibraryDefines general
default: StandardDefines

-D’s for compiling libraries.

BinDir general
default: KHOROS_HOME/bin

Where to install Khoros programs.

IncRoot general
default: KHOROS_HOME/include

Where to find Khoros include files.

LibDir general
default: KHOROS_HOME/lib

Khoros library installation directory.

SysLibs -ll
default: -lm

Specifies the system libraries that your machine uses. The definition of *SysLibs* in the machine.cf file will override this definition.

XLibs -lXmu
default: -lXaw

Specifies the X libraries that your machine uses. If *XLibs* is defined in the machine.cf file, that will override this definition.

ConfigDir general
default: KHOROS_HOME/repos/config/imake_conf

Where to install Khoros configuration files.

ManDir general
default: KHOROS_HOME/man/man1

Where to install Khoros manual page files.

LibManDir general
default: KHOROS_HOME/man/man3

Where to install Khoros library manual page files.

UseCCMakeDepend boolean
default: NO

Whether to use the slow shell cc -E 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. If UseCCMakeDepend is not set, then *DependDir* is $(SCRIPTSRC). Otherwise *DependDir* is $(TOP)/bootstrap/makedepend.

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*().

A.2 MAKE VARIABLES

Many of the **make** variables in *Khoros.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 more 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).

A.3 IMAKE RULES

The *Khoros.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 Khoros 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’’ 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 *Khoros.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*.

(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 *Khoros.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).

In the Khoros 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.

A.4 COMMONLY USED RULES

Rules for building programs

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

Rules for installing shell scripts

InstallCshScript()
InstallShScript()
InstallMultiCshScripts
InstallMultiShScripts

Rules for building libraries

NormalLibraryObjectRule()
NormalLibraryTarget()
NormalLibraryLexYacRule()
NormalSharedLibraryTarget()
NormalSharedLibraryDataTarget()

Rules for installing files

InstallProgram()
InstallLibrary()
InstallManPage()
InstallNonExec()
InstallSubdirs()
InstallSharedLibrary()
InstallFortranScript()
InstallCshScript()
InstallShScript()
InstallMultiCshScripts()
InstallMultiShScripts()

Rules for cleaning up

RemoveTargetProgram()
CleanTarget()
CleanSubdirs()

Rules for generating Imakefile, Makefiles and dependencies

GenImakefileDir()
GenImakefileDirLocal()
GenImakefileProg()
MakefileTarget()
MakefileSubdirs()
DependTarget()
DependSubdirs()

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

A.5 RULE DESCRIPTIONS

Most of the common rule-generating macros in *Khoros.rules* are described below.

SaberProgramTarget(program,srclist,objlist)

Generates rules to make Saber-C read in sources and objects. 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.

NormalProgramTarget(program,objects,deplibs)

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.

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) You have set SRCS to the list of source files (so the ‘‘depend’’ target works).

(2) You have set OBJS 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.

FortranScriptTarget(program)

Generates rules for placing a "sh" script in place of a binary when Fortran is not available on the machine.

ProgramLexYacRule()

Generates rules for lex and yacc. This includes rules for the

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.

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.

InstallManPage(file,dest)

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

InstallLibManPage(file,dest)

Generate rule to install the library manual page *lfile.man* in *dest/lfile.*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.

InstallFortranScript(program,forscript,dest)

Generate rules to install a *shell script* using special install flags set in *$(INSTALLFLAGS)*. This is called when Fortran is not available on the machine.

InstallCshScript(program,dest)

Generate rules to install a *CSH* shell script into *dest/program* using installation flags INSTPGMFLAGS. The program name must be in the form *program.csh*

InstallShScript(program,dest)

Generate rules to install a *SH* shell script into *dest/program* using installation flags INSTPGMFLAGS. The program name must be in the form *program.sh*

InstallMultiCshScript(list,dest)

Generate rules to install all/many *CSH* shell scripts in a single directory into their corresponding *dest/program* using installation flags INSTPGMFLAGS. The names in the list must be in the form *program.csh*

InstallMultiShScript(list,dest)

Generate rules to install all/many *SH* shell scripts in a single directory into their corresponding *dest/program* using installation flags INSTPGMFLAGS. The names in the list must be in the form *program.sh*

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. However, that need not be true, strictly speaking, 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. As noted above for InstallProgram(), INSTALLFLAGS appears to be set nowhere in the X configuration files. Perhaps it is something you are expected to give a value to 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.

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.

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*.

BuildImakefileDirLocal()

Generate a rule to update Imakefiles located in sub-directories using the imkmf utility. This is generally done automatically by the template or by any special Imakefiles.

BuildImakefileDir(list)

Generate rules to update an Imakefile for a directory using the imkmf utility. This is generally done automatically by the template or by any special Imakefiles.

BuildImakefileProg()

Generate rules to update an Imakefile for programs, X programs, libraries and scripts using the imkmf utility. This is generally done automatically by the template or by any special Imakefiles.

GenImakefileDirLocal()

Generate a rule to build Imakefiles located in sub-directories using the imkmf utility. This is generally done automatically by the template or by any special Imakefiles.

GenImakefileDir(list)

Generate rules to build an Imakefile for a directory using the imkmf utility. This is generally done automatically by the template or by any special Imakefiles.

GenImakefileProg()

Generate rules to build an Imakefile for programs, X programs, libraries and scripts using the imkmf utility. This is generally done automatically by the template or by any special Imakefiles.

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.

NormalLibraryLexYacRule()

Generates rules for lex and yacc compile rules for a library. This includes rules for the .y.o and .l.o files. This rule will substitute "library_name_" for the "yy" part of the routine name. For example, "yyparse" will renamed to "library_name_parse".

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.

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.

SharedFortranObjectRule()

Generates make rules to build shared and normal object files.

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.

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.

MakeMakeSubdirs(dirs,target)

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

MakeNsubdirMakefiles()

Generate a set of very very very ugly but quite wonderful rules for use by MakeMakeSubdirs().

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.

AllTarget(depends)

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

Makeproto(flags,list)

Generate rules for using the protoize program to generate the ANSI C prototypes automatically.

MakeProtoDir(list)

Generate rules for using Makeproto() at the directory level.

MakeGhostWriter(program,

Generate rules for using ghostwriter in the makefile.

MakeGhostWriterDir(list)

Generate rules for using ghostwriter at the directory level.

MakeGhostReader(program,

Generate rules for using ghostreader in the makefile.

MakeGhostReaderDir(list)

Generate rules for using ghostreader at the directory level.

MakeHelppage(program)

Generate rules for using helppage in the makefile.

MakeHelppageDir(list)

Generate rules for using helppage at the dir level.

MakeRCSDir(list)

Generate rules for using RCS at the directory level.

RCSRule(list)

Generate rules for using RCS co and ci.

B. REFERENCES

This document was derived from the Appendix written by:

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Revision date: 24 January 1991

CONTENTS

Chapter APPENDIX B - IMAKE -