Silicon Graphics Makefile Conventions

Tom Murphy

A reference manual of the conventions followed in SGI Makefiles.

This manual presents the conventions for all Makefiles in the SGI source tree. It assumes the reader knows how to construct a Makefile. It also assumes that the *commondefs/rules* files (make include files developed at SGI) will be used to simplify Makefile construction.

This manual numbers the SGI Makefile conventions for easy reference and provides examples of correct use. In addition, several annotated examples of common Makefiles, which can be used as templates, are presented in an appendix at the end of this document.

Standardization in Makefile construction allows developers and the Release Group to generate correct results and aids understanding Makefiles. This document is intended to educate engineers new to SGI and is also a reference point for ongoing discussion and evolution of these conventions within the SGI software engineering community.

Conventions for Constructing Correct SGI Makefiles

All the conventions mentioned in this document are in widespread use throughout SGI. Those marked (Required) are the characteristics of Makefiles that enable them to successfully pass through the SGI build process. The unmarked conventions are considered good Makefile style, but developers do not absolutely have to use them.

*TOOLROO*T is an environment variable containing the name of a directory whose contents are like the root directory of a system. The *TOOLROOT* directory contains enough tools to build all IRIX software. *ROOT* is also an environment variable that points to the top of your workstation source tree. In it the developer can find everything other than build commands, such as header files and libraries. Developers often set these two environment variables to point to the same place. They are the same environment variables used in other SGI tools, such as ctools.

The two files *commondefs* and *commonrules* form the de facto standard for SGI Makefile construction since most current SGI Makefiles use them. The "Basic Makefile"on page B-1 illustrates how to access these files from within a Makefile. The documentation for them is within the text of the files and appears as Appendices C and D. Some basic documentation also appears in the annotations of the "Basic Makefile" on page B-2.

Rules, targets and *macros* are Makefile terms. A *rule* is made of three components: a name called the *target*, a list of other *target*s or files that this *rule* depends on, and a series of shell commands that *make* executes when it updates this *target*. A *Target* describes an action or file produced when *make* executes that target's rule. A *macro* is a Makefile abbreviation for a character string. Developers often use *macros* as Makefile constants.

A *feature* is a main directory and possibly additional subdirectories of the source tree containing software tending to be independent and indivisible. Good examples are IRIX commands found in the usr/src/cmd directory. They may have subdirectories that represent libraries used to build a command, but are self-contained aside from references to global header files and libraries. The gl library (usr/src/gfx/lib/libgl) provides another example of a *feature*. A *feature* may correspond to a product we ship, but does not have to.

This document uses the word *platform* to refer to the machines that SGI ships. The environment variable *PRODUCT* contains the name of a particular SGI hardware *platform*. *PRODUCT* is used by some makefiles to produce *platform* dependent *features*. PRODUCT usually contains the name of the type of machine the engineer uses for software development. *Make*, using *commondefs/rules*, will build for that machine by default. The developer can also set it to another machine type to compile for that machine or leave it unset to compile for each machine listed in the EVERYPRODUCT macro defined by the engineer.

A *reacharound* is a relative path from one directory to another that passes through a common ancestor. An *in-scope reacharound* is one that stays within the directory hierarchy of a *feature*.

References will appear throughout this document to the makefile examples of Appendix B. The reader is not expected to follow each of these cross references. They are solely to provide complete makefile examples for the conventions presented.

Which *make* to use

Convention 1. Use *smake* where possible.

Smake is an alternative version of *make* that sometimes executes the Makefile faster than *make* since it will try to run parallel processes. This is primarily of help to the Release Group whose builds go on for hours. Unfortunately, there are some pitfalls you need to be aware of. In general, it is possible to use *smake* if:

- The order in which targets are generated is insignificant or .ORDER is used.
- The Makefile is "low" enough in the tree so that the build does not run out of processes. Typically this means a leaf directory with more than one source to compile. The "Parent Makefile" on page B-3 can be made to run in parallel. It needs to be modified so each run of *make* in the subdirectory loop is run in its own shell in the background.
- You do not use VPATH. This rules out using *smake* with the current version of "Makefile with boot target" on page B-8.

The man page for *smake* has some additional cautions you may want to be aware of. The comment line "#!smake" must be present as the first line in the Makefile to have *make* run *smake* run instead of *make*.

What to call Makefile

Convention 2. (Required) Name Makefiles "Makefile" with a capital M.

This convention clarifies which makefile is used for system builds. The Makefile name with a capital M brings it to the top of the directory listing. Developers originally based this on the idea that they could create an alternate lower case makefile for use during software development. Having two makefiles is no longer recommended. It is important that one makefile controls both development and building.

Scope of Makefiles

Convention 3. (Required) Each Makefile builds and installs only the contents of its directory.

The Makefile in a directory makes and installs the source contained within the directory and initiates a *make* within its sub-directories. For instance, a directory should not *install* something produced in a sub-directory. The subdirectory is responsible for *install*ing it.

Convention 4. (Required) **The Makefile must produce the same binaries no matter what build *platform* is used.**

Makefiles execute from a developer's workstation, a group's server or a corporate build machine. It must be possible to produce the same binaries in every case. The key to this is a consistent reference to things outside of the current directory. Tools referenced by the Makefile will sometimes be native tools on the build machine such as *sh* and *echo* and sometimes tools from TOOLROOT. The tools in TOOLROOT are typically a different version of compilers and commands than those of the build machine. *Commondefs* provides *macro* names for several commands. A list appears in the text of Convention 14. The developer must choose paths carefully for the Makefile to use correct versions of tools, header files and libraries. The remaining conventions of this section deal with specific ways that ensure correct Makefile function relative to the build host and the target platform.

Convention 5. (Required) **The Makefile must produce versions of its results that will run on every SGI *platform*.**

A basic tenet of SGI software is that we use a single version of the source code to create executables that run on all appropriate SGI *platform*s. Typically this is not difficult since most source code has no platform dependencies. Thus, the Makefile only needs to produce one set of results. The developer can reduce *platform* dependent effects further by linking with the shared C library or shared graphics library. The developer can manage cases where *make* must generate multiple versions of a *feature*. This is done by modifying the "Makefile with boot target" on page B-8.

There is a simple build convention used when *platform* dependencies exist. If the environment variable PRODUCT is set, then just a version for that *platform* is built. If the variable is not set, then the *macro* EVERYPRODUCT define the *platforms* for which to build.

Standard things to include

Convention 6. Use the *commondefs* and the *commonrules* files.

Example

```
include $(ROOT)/usr/include/make/commondefs
# next line must appear after your first rule
include $(COMMONRULES)
```

Commondefs/rules carry out the bulk of the SGI Makefile conventions in an automatic and natural way and they appear in all the examples in this document. Use of them will significantly simplify and shorten the Makefiles you write. Usage examples of *commondefs/rules* appear in "Appendix B: Some Annotated Examples".

Convention 7. Include the RCS revision comment line.

Example

```
#!smake
#    "$Revision$"
```

SGI has standardized on using RCS as part of the checkin process for everything stored in the source tree. It is convenient to have information about the current RCS revision of the Makefile textually included in the file. This is done by including the revision comment line shown in the example above. RCS rewrites this line every time the developer checks a new version of the Makefile into the tree with c_finalize from ctools. However, do not use $Author$, $Date$, $Header$, $Locker$, Log, $Source$, or $State$, as they complicate determining the differences between source trees and are in any event usually redundant.

Convention 8. A Makefile should announce the depth of its subdirectories in the source tree as it calls each one.

Example

```
default install $(COMMONTARGS): $(COMMONPREF)$$@
     @for d in $(SUBDIRS); do \
          echo "====\tcd $$d; $(MAKE) $@"; \
          cd $$d; $(MAKE) $@; cd..; \
     done
```

Typically, Makefiles will echo several equal symbols corresponding to the depth of its subdirectories from the att directory. For instance, the Makefile in directory jake:/jake/att/usr/src/cmd/spaceball echoes "====" as it passes control to each of its subdirectories.

Make Targets

Convention 9. (Required) Each Makefile generating a software *feature* must have an *install* target.

Example

```
#The target default is the one that typically generates the software
install: default
    $(INSTALL) -idb "std.sw.foo" -F /usr/sbin $(TARGETS)
```

The *install* target uses the install(1) command defined by $(INSTALL) to install the results generated by "make install". *Install* directs where to put these results as well as other attributes of the installation process. Every Makefile should pass the *install* target to any subdirectories producing results that need installing. More details on the choices of parameters for the $(INSTALL) command appear in "How to construct a $(INSTALL) line" on page 10. Further examples of install lines appear in "Appendix B: Some Annotated Examples".

Convention 10. Each Makefile should have a *clean* target.

Commonrules provides the *clean* target for you. The *clean* target removes intermediate files, such as object files, from the directory. *Commondefs/rules* will handle this for you, provided you define the *macros TARGETS* and *CFILES*. Developers use the *macro LDIRT* to add the names of additional intermediate files to *clean* (or *clobber*). For instance, "LDIRT = 4D20.O/*.o" causes *make* to remove the object files in directory 4D20.O to be removed when the developer selected *clean* (or *clobber*).

Convention 11. (Required) Each Makefile must have a *clobber* target.

Commonrules provides the *clobber* target for you. The *clobber* target removes everything from the directory except source files and the Makefile itself. This includes object files, other intermediate files, dependency files, and results created by the Makefile. *Commondefs/rules* will handle this for you, provided you define the *macros TARGETS* and *CFILES*.

Convention 12. Each Makefile generating software *features* for TOOLROOT should do so using a *boot* target.

Example

```
# in the simplest case the boot and install targets can be the same
boot install: default
    $(INSTALL) -idb "std.sw.foo" -F /usr/sbin $(TARGETS)
```

The purpose of the *boot* target is to generate just the tools that go into *TOOLROOT.* The Release Group uses this target to do the initial or "boot" phase of a complete build and to construct public *TOOLROOT*s for use throughout the company. B*oot* is always a subset of *install*. See "Makefile with boot target" on page B-6 for a larger example.

Things to consider when constructing commands

Convention 13. Use *macros* to simplify Makefiles, especially ones defined by *commondefs/rules*.

Example

```
    $(CCF) $(OBJECTS) -o $@ $(LDFLAGS)
# $(CCF) invokes the compiler with standard and extensible flags.
# $(OBJECTS) is a macro commonrules forms from user supplied $(CFILES)
# $(LDFLAGS) are the standard flags passed to the linker.
```

Make macros provide a way to create common definitions in one spot. Using *macro* names for command and flag references serves several purposes. It allows a developer to redefine things easily from one spot. It also allows more than one command to use the flags, when the flags are appropriately grouped (as *commondefs* does). For instance, creating a LCINCS *macro* allows the include directories defined to be passed both to lint and to the C compiler. Using *macros* can serve to simplify and clarify the intent of the Makefile, which simplifies future changes as well.

Commondefs further clarifies a Makefile by defining many command *macros* and flags which the engineer does not have to define. There is an extensive set of compiler and linker flags that simplify and standardize command lines which the reader can see in the above example.

Convention 14. (Required) Use the commands, tabulated within this convention, relative to *TOOLROOT.*

Example

```
# This next line uses commondefs for the command cc and its flags
    $(CCF) -c $<
# $(CCF) is defined as $(CC) $(CFLAGS)
# $(CC) is defined as $(TOOLROOT)/usr/bin/cc
# $(CFLAGS) are the flags for the C compiler
```

The commands listed in this table are used to ensure a Makefile will produce the same results on every build *platform*. They also ensure the TOOLROOT compilers will produce correct object files for the release under development. All other Makefile commands should be executed as native commands on the build host. The only exception to this is when a program needs to be executed as well as compiled on the build host. In this case, the native compilers, rather than the TOOLROOT compilers, should be used. The need for this distinction will strongly come into play if we begin producing code for both little endian and big endian machines.

An engineer can negotiate with other engineers and the build group to add a command to TOOLROOT if it causes the Makefile to function differently on the build host or it generates code that functions differently on target *platforms*.

TOOLROOT Commands

Command	Commondefs *Macro*	Description
cc	CC	C compiler
CC	C++	C++ compiler
f77	F77 or FC	FORTRAN compiler
pc	PC	Pascal compiler
as	AS	MIPS Assembler
lex	LEX	tool to generate lexical scanners
lint	LINT	C program checker
mkf2c	MKF2C	tool to generate FORTRAN-C interface routines
nm	NM	name list dump of MIPS object files
yacc	YACC	yet another compiler-compiler
ar	AR	archive and library maintainer
ld	LD	loader
libspec	LIBSPEC	library specification translator
lorder	LORDER	find ordering relation for an object library
mkshlib	MKSHLIB	create a shared library
size	SIZE	print the section sizes of an object file
strip	STRIP	remove symbols and relocation bits
m4	M4	macro processor
awk	AWK	pattern scanning and processing language
nawk	NAWK	"
oawk	OAWK	"
cps	CPS	construct C to Postscript interface
echo	ECHO	copy arguments to stdout
install	INSTALL	install files in directories
sh	SHELL	Bourne shell

Convention 15. (Required) Libraries, header files and other out-of-scope references must be relative to *ROOT*.

Example

```
$(LDF) $(OBJECTS) -o foo
# The $(LDF) macro contains both the command to execute ld as well as a
# flag to take all library references from $(ROOT)/usr/lib
```

The intent of this convention is to build the executables of a *feature* from source contained within the *feature* or from a common well defined place, $ROOT. All non-command references should be relative to the current directory or subdirectory of the *feature* (a valid in-scope reference), or relative to *$ROOT*. This ensures correct versions of things such as header files, include files and libraries. In particular, /usr/include cannot be in the compiler search path; the compiler must search $(ROOT)/usr/include instead. Also, /lib, /usr/lib, and /usr/local/lib must not be in the linker search path; the linker must search $(ROOT)/usr/lib instead.

Commondefs satisfies much of this convention automatically for the user.

Convention 16. (Required) Do not use out-of-scope reach-arounds.

Example

```
# The following line is invalid if other_foo is in a different feature
    $(CC) $(CFLAGS) -I../../../other_foo/include -c $<
```

It certainly is acceptable to use *reacharounds* within the scope of a *feature*. In fact, a developer often uses a *reacharound* to reference an include directory used by every subdirectory of a *feature*. It is not acceptable to reach around outside of the scope of a software *feature*. The intent is to have as small a locality of reference for a *feature* as possible. The list below shows some of the places in the source tree through which reacharounds should not pass.

Key Source Tree Directories

```
att/doc                  att/usr/src/lib
att/usr/src/apps         att/usr/src/man
att/usr/src/cmd          att/usr/src/sgionly
att/usr/src/cmplrs       att/usr/src/stand
att/usr/src/gfx          att/usr/src/uts/mips
att/usr/src/head
```

Convention 17. Make sure all dependencies are up to date, especially for header files.

Example

```
# Commonrules provides this target to keep dependencies current
% make depend
```

You should recalculate your dependencies anytime the include structure changes in any of your source files. An include structure is the collection of files included by a file. These are the files you explicitly include as well as nested includes. Changes to included files from nesting are hard to notice. For instance, the mkdepend command will not recognize that a header file you include just started including a new file. You can run the command mkdepend to help with the task of generating dependencies. Better yet is to use the *commonrules* targets *depend* or *incdepend* which generate dependencies using mkdepend.

How to construct a $(INSTALL) line

The conventional action of the *install* command is to install software in a target directory, setting things such as permissions, owner id and group id. It can also create directories, links and special files. The SGI *install* does these things but it also can produce information for the SGI installation database (idb). Install will produce this information when the *RAWIDB* environment variable is set. RAW-IDB contains the name of the rawidb file that stores *the idb* entries that are produced. See the man page for *install(1)* for more information.

The *commondefs/rules* convention is to use the *macro INSTALL* to invoke the *install* command. *Commondefs* provides the *INSTALL macro*.

The syntax of a $(INSTALL) line for a file is:

$(INSTALL) **-F** dir [-src path] options file ...

The syntax of a $(INSTALL) line for a directory is:

$(INSTALL) **-F** dir **-dir** options directory ...

Developers need to define directories explicitly so they can specify appropriate mode, owner and group values and so that directories get removed when the subsystem/product is removed.

The most common options are:

-**src** path — Use path as the source file's pathname when installing a regular file. This option is useful for renaming a source file in the target directory. This option may be used only when installing regular files.

-m mode — Set the mode of created files to mode, interpreted as an octal number. The default mode for regular files and directories is 755 adjusted by *umask*.

-u owner — Set the owner of created files to owner, first as a user name, then as a numeric user ID if it fails to match known user names. If the superuser invokes install, the default owner is *bin*. Starting with Cypress, the default will be *root*. Otherwise the default owner is the effective user ID of the invoker.

-g group — Set the group of created files to group, first as a group name, then as a numeric group ID if it fails to match known group names. If the superuser invokes install, the default group is *bin*. Starting with Cypress, the default will be *sys*. Otherwise the default group is the effective group ID of the invoker.

-idb attribute When RAWIDB is set, create an idb entry for the files and/or directories from the idb attributes. This particular option may occur several times among the various option arguments. Multiple attributes can also appear in a quoted string. Some of the attributes are discussed below.

-dir Create directories named by prepending *$ROOT* to the file arguments.

-F dir Install files in dir, which must be a writable directory. It will create any directories in the dir pathname which do not exist. Note that such implicitly created directories will not be automatically removed when the subsystem/product is removed.

Common settings of mode, owner and group

```
These are the most common setting of mode, owner and group. Also
listed is their frequency of occurrence in the 3.3 release and the
type of files corresponding to the settings.
```

mode	owner	group	%	file type
0755	bin	bin	42	executables (pre-Cypress default)
0444	bin	bin	20	source and header files
0000	bin	bin	19	man pages
0644	guest	guest	6	/usr/people/4Dgifts read only files
0644	bin	bin	5	/usr/diags read only files
0755	demos	demos	3	/usr/demos executables
0644	demos	demos	1	/usr/demos read only files
0644	root	sys	1	/etc read only configuration files
0755	guest	guest	1	/usr/people/4Dgifts executables
			1	other
0755	root	sys	0	executables (Cypress and after default)

The *idb tag* is one of the attributes that a developer can specify using the -idb option. The build process maps the file into the *subsystem*, the *image* and the *product* to which it belongs. An example of an *idb tag* is "-idb SoftPC.sw.SoftPC". SGI uses this information as a basic part of its software distribution mechanism. The *subsystem, image* and *product* are progressively larger collections of files that may be installed.

If you are modifying an existing Makefile, you should be able to use the *idb tag* defined there. If you are creating a new Makefile then the parent directory and its previous children will give strong clues for what values to use. Otherwise, leave the *idb tag* blank and the person responsible for structuring the product will supply a correct one.

Other idb attributes that can be set are *preop, postop*, and *exitop* which specify IRIX commands which *inst* executes before installation, immediately after installation, and at exit from *inst*. The *config* attribute is used to specify what should be done when a file to be installed already exists. The *config* attribute has no effect when there is no preexisting file. It is typically used with configuration files, such as /etc/passwd. It requires one of three possible parameters: *update*, *noupdate*, or *suggest*. *Update* is the default and it dictates that the preexisting file be renamed with a ".O" suffix and the a new configuration file be installed by inst. The *noupdate* option specifies the old file to be kept and to install nothing. The *suggest* option also keeps the old file, but installs the new file with a ".N" suffix. The syntax of this attribute is "-idb config(**option**)". The *nostrip* attribute specifies that the executables should not be stripped. Do this to keep the executable's symbol table around for debugging or linking. Libraries use *nostrip* so development tools such as pixie, prof, dbx, and gldebug work correctly.

The only other major twist on the *install* line comes when you are building *platform* dependent code. You must be able to tell the installation database the hardware platforms that a Makefile's results are for. This is done via *mach tags* which are also an idb attribute. See "Makefile with boot target"on page B-8 for an example using the *mach* tag.

More information on choosing idb tags and idb attributes can be found in the document "Engineers' Guide to Packaging Software for inst".

Example

```
# The Makefile for finger provides a standard install line
install: default
	$(INSTALL) -idb "std.sw.unix" -F /usr/bsd finger
#
# This preop example shows the destination directory for the binary
# being removed, if it exits. The other install options create the
# destination directory soft linked to /usr/sbin
install: default
	${INSTALL} \
		-idb "std.sw.NeWS" preop("rm -rf $$rbase/usr/NeWS/bin") \
		-F /usr/NeWS -lns /usr/sbin bin
#
# The mach tag example shows an IO library installed for a given
# CPU board
install: default
	$(INSTALL) -m 444 -idb std.sw.lboot \
		-idb "nostrip mach($(CPUBOARD))" \
		-F /usr/sysgen/boot io.a
#
# The config example directs that the file /etc/wtmp be installed
# only if it does not already exist
IDB_TAG =std.sw.acct
INS_NOUP=$(INSTALL) -idb "$(IDB_TAG) config(noupdate)"
install: default
	$(INS_NOUP) -F /etc -u adm -g adm -m 664 wtmp
```

Policy on Makefiles generated automatically or authored elsewhere

The focus of this manual is on Makefiles that developers can freely modify. However, there are other Makefiles that cannot or should not be significantly modified. In some cases, it may be worth the one time cost of modifying the Makefile or the Makefile generator to follow the SGI Makefile conventions. In other cases, it is most sensible to modify the makefile as little as possible since the work will have to be done over and over again each time we get a new version from the outside source.

In these cases, a Makefile, called a *wrapper*, must be created which fulfills the SGI Makefile Conventions and treats the external makefile as a black box. Only one convention needs a little adjusting: "(Required) Each Makefile builds and installs only the contents of its directory."(Convention 3.). A *wrapper* must pass control to the external makefile and provide *targets* that manipulate the *features* generated by the external makefile as if they were the *wrapper*'s own. Good examples are X, distributed by MIT, and Motif, distributed by OSF. These have a still evolving directory structure and a Makefile generator. The developer and the Release Group jointly decide when to use a wrapper.

Appendix A:
Quick Reference List of Conventions

Convention 1.	Use smake where possible.
Convention 2.	(Required) Name Makefiles “Makefile” with a capital M.
Convention 3.	(Required) Each Makefile builds and installs only the contents of its directory.
Convention 4.	(Required) The Makefile must produce the same binaries no matter what build platform is used.
Convention 5.	(Required) The Makefile must produce versions of its results that will run on every SGI platform.
Convention 6.	Use the commondefs and the commonrules files.
Convention 7.	Include the RCS revision comment line.
Convention 8.	A Makefile should announce the depth of its subdirectories in the source tree as it calls each one.
Convention 9.	(Required) Each Makefile generating a software feature must have an install target.
Convention 11.	(Required) Each Makefile must have a clobber target.
Convention 11.	(Required) Each Makefile must have a clobber target.
Convention 12.	Each Makefile generating software features for TOOLROOT should do so using a boot target.
Convention 13.	Use macros to simplify Makefiles, especially ones defined by commondefs/rules.
Convention 14.	(Required) Use the commands, tabulated within this convention, relative to TOOLROOT.
Convention 15.	(Required) Libraries, header files and other out-of-scope references must be relative to ROOT.
Convention 16.	(Required) Do not use out-of-scope reach-arounds.
Convention 17.	Make sure all dependencies are up to date, especially for header files.

Appendix B: Some Annotated Examples

These are examples of the most common classes of Makefiles in the source tree. Another source of examples are the Makefiles actually in the source tree. The examples shown here are in jake:/jake/att/usr/src/templates. The templates are for developers to edit when creating new Makefiles and thus have relatively few comments. Annotations to these Makefiles are in bold and begin with "###. The title box of each example shows the name of the file in the template directory. Please refer back to the basic Makefile as you read later ones. It contains annotations that generally apply to all of the examples.

Basic Makefile

This sample Makefile is used in leaf directories of the source tree. You can simplify the Makefile when make has only one file to compile. The annotations to the sample show what you have to do. The major changes needed for this Makefile are to adjust the CFILES and TARGETS macros and customize the install line.

Makefile.basic

```
### Commondefs provide macros for commands and flags. Commonrules
### provides standard targets and Makefile goo to support them.
### Together they are a common SGI foundation for Makefile development.
### Please see Appendices C and D for listings of these files.

### Remove or comment out.
     The standard sample Makefile
### Causes processing of the Makefile using smake instead of make.
#!smake
#
### Add comments to describe this Makefile and directory.
# Makefile for foo(1).
#
### This is so RCS will update the line with the current rev number.
#    "$Revision$"

### This brings in the SGI Common Makefile definitions.
### Include it before any of your own definitions.
include $(ROOT)/usr/include/make/commondefs.

### What follows are some sample uses of commondefs.
### To pass an environment variable to a C program:
### LCDEFS = -DFOO
### To have the compiler keep symbolic info for debuggers:
### LCOPTS = -g
### To have an alternate include directory:
### LCINCS = -I../include

### You need to tell commonrules what your source files are so it can
### generate a list of your objects files, $(OBJECTS).
### There are several lists possible: CFILES, C++FILES,
### ASFILES, YFILES, etc. A complete list is in commondefs.
CFILES=foo.c foosubs.c

### This line defines the results made by this Makefile.
TARGETS=foo

### If you have a TARGET compiled from a single source
### file, then uncomment the LMKDEPFLAGS macro. It means you
### don't have to write a target rule for that target and
### that make will not produce any intermediate .o files.
# uncomment the following line if compiling only single file programs.
# LMKDEPFLAGS= $(NULLSUFFIX_MKDEPFLAG)

### You must define at least one rule before including commonrules.
### (otherwise the first commonrules rule will be the default rule)
default: $(TARGETS)

### The macro COMMONRULES contains the pathname of the
### commonrules file.
include $(COMMONRULES)

### The install target installs the Makefile's results. For more info
### please see "How to construct a $(INSTALL) line" on page 10.
install: default
     $(INSTALL) -idb std.sw.foo -F /usr/sbin $(TARGETS)

### This is the rule to make the TARGET of this Makefile.
### It uses several macros defined by commondefs.
### Don't forget the LDFLAGS macro when you are linking.
foo: $(OBJECTS)
     $(CCF) $(OBJECTS) $(LDFLAGS) -o $@
```

Parent Makefile

There are two sample parent directory Makefiles. The first parent Makefile just passes the targets to its subdirectories. The second one passes the targets to its subdirectories and also produces some results of its own. Together they represent the most common cases of interior directories in the source tree.

The major changes needed to this Makefile are to customize the SUBDIRS macro to reflect the list of subdirectories to be made.

Makefile.parent

```
### Please see "Basic Makefile" on page B-2.
### for general information not repeated here.
     Sample Makefile for a parent directory just making its children
#!smake
#
# Makefile for foo(1).
#
#    "$Revision$"

include $(ROOT)/usr/include/make/commondefs

### These are the subdirectories in which to run make.
SUBDIRS= subdir1 subdir2 subdir3

### This ensures no attempt is made to rm these directories when
### the Makefile is interrupted.
.PRECIOUS: $(SUBDIRS)

### Commonrules allows the targets it defines to be used both
### in the current directory and the designated subdirectories.
### This is activated by defining the COMMONPREF macro.
### Targets the user created for the current directory need to have
### the COMMONPREF preceding their name. The value you choose for
### COMMONPREF should be unique. A good choice is the current
### directory name.
COMMONPREF=foo

### $(COMMONTARGS) are targets defined in commonrules.
### Namely: clobber, clean, rmtargets, depend, incdepend, fluff, tags.
### fluff runs lint on the file, tags generates ctags info.
###
### This loop will propagate any of the targets default, install and
### $(COMMONTARGS) to the subdirectories mentioned in $(SUBDIRS).
### Note the equal symbols in the comment line which are used to
### reflect the depth of the Makefile's directory in the source tree.
default install $(COMMONTARGS):
     @for d in $(SUBDIRS); do \
          echo "====\tcd $$d; $(MAKE) $@"; \
          cd $$d; $(MAKE) $@; cd..; \
     done

### This target allows subdirectories to be made individually.
$(SUBDIRS): $(_FORCE)
     cd $@; $(MAKE)

include $(COMMONRULES)
```

There are a few differences between Makefile.parent+ and the preceding one. You now need to include the CFILES and the TARGETS macros as done with the "Basic Makefile"on page B-2. The COMMONPREF macro is a device that allows each of the standard targets to generate itself in the subdirectories as well as within the directory. A good common prefix to use is the directory name. A target passed to the Makefile first generates the target within the directory because of the dependency on the COMMONPREF'd target. It then generates the target in each of the subdirectories. You will need to edit the install line to handle the *feature*s the Makefile produces.

Makefile.parent+

```
### This Makefile is basically a merger of "Basic Makefile" on page B-2
### and "Parent Makefile" on page B-3.
     Sample Makefile for a parent directory generating features
#!smake
#
# Makefile for foo(1).
#
#     "$Revision$"

include $(ROOT)/usr/include/make/commondefs

### Define sources and targets as in "Basic Makefile" on page B-2.
CFILES=foo.c foosubs.c
TARGETS=foo

### Define subdirectories as in "Parent Makefile" on page B-3.
SUBDIRS= subdir1 subdir2 subdir3

.PRECIOUS: $(SUBDIRS)

COMMONPREF=foo

### The dependency on $(COMMONPREF)$$@ causes make to execute
### the target first in this directory.
default install $(COMMONTARGS): $(COMMONPREF)$$@
     @for d in $(SUBDIRS); do \
          echo "====\tcd $$d; $(MAKE) $@"; \
          cd $$d; $(MAKE) $@; cd ..; \
     done

include $(COMMONRULES)

$(SUBDIRS): $(_FORCE)
     cd $@; $(MAKE)

### Define the target default for this directory.
$(COMMONPREF)default: $(TARGETS)

### Define the target install for this directory.
$(COMMONPREF)install: $(COMMONPREF)default
     $(INSTALL) -idb std.sw.foo -F /usr/sbin $(TARGETS)

foo: $(OBJECTS)
     $(CCF) $(OBJECTS) $(LDFLAGS) -o $@
```

Shell Script Makefile

The shell script Makefile is the simplest of all the sample Makefiles. The major changes needed are to define the source shell scripts in the SHFILES macro, the corresponding executable shell scripts in the TARGETS macro and to customize the install line.

Makefile.sh

```
### Please see "Basic Makefile" on page B-2.
### for general information.
    Sample Makefile for a shell script
#
# Makefile for foo(1).
#
#    "$Revision$"

include $(ROOT)/usr/include/make/commondefs

### This makefile uses the default make rule for shell files.
### foo.sh is the nonexecutable source form of the shell script.
### foo is the executable (not intended for editing) shell script.
SHFILES= foo.sh
TARGETS = foo

default: $(TARGETS)

include $(COMMONRULES)

### define the target install for this directory.
install: default
    $(INSTALL) -idb std.sw.foo -F /usr/sbin $(TARGETS)
```

Makefile with boot target

The SGI source tree build model has a core build environment that includes many of the most recent header files, libraries and compilers. A feature that produces results for inclusion in this core set of tools and objects may need to support a *boot* target that builds and installs its results. The Makefile should do this in addition to its other functions, whether as an interior or leaf directory in the source tree. Makefile.boot is an example of a Makefile for an interior directory that makes some files as well as its subdirectories. The BOOT_DIRS macro defines those directories that need to install targets during boot. The *boot* target simply makes each of these directories with an *install* target. A *boot* usually generates a subset of what *install* does. Sometimes it may generate the same thing that *install* does.

Makefile.boot

```
### This bears some similarity to the "Basic Makefile" on page B-2.
     Sample Makefile for a parent directory with boot target
#!smake
#
# Makefile for foo(1).
#
#    "$Revision$"

include $(ROOT)/usr/include/make/commondefs

SUBDIRS= subdir1 subdir2 subdir3
BOOT_DIRS = subdir1

CFILES=foo.c foosubs.c
TARGETS=foo

.PRECIOUS: $(SUBDIRS)

COMMONPREF=foo

default install $(COMMONTARGS): $(COMMONPREF)$$@
     @for d in $(SUBDIRS); do \
          echo "====\tcd $$d; $(MAKE) $@"; \
          cd $$d; $(MAKE) $@; cd..; \
     done

include $(COMMONRULES)

### The purpose of the boot target is to install just
### the features that go in $TOOLROOT and $ROOT.
### In this case, an install is done within this directory and boot
### is passed to the subdirectory listed in $(BOOT_DIRS).
boot: $(COMMONPREF)install
     @for d in $(BOOT_DIRS); do \
          echo "====\tcd $$d; $(MAKE) $@ (boot)"; \
          cd $$d; $(MAKE) boot; cd ..; \
     done

$(SUBDIRS): $(_FORCE)
     cd $@; $(MAKE)

$(COMMONPREF)default: $(TARGETS)

$(COMMONPREF)install: $(COMMONPREF)default
     $(INSTALL) -idb std.sw.foo -F /usr/sbin $(TARGETS)

foo: $(OBJECTS)
     $(CCF) $(OBJECTS) $(LDFLAGS) -o $@
```

Platform-Dependent Makefile

Most directories don't need to do explicit machine dependent makes. The shared libraries libc_s and libgl_s, which are the standard C and SGI graphics libraries, will typically handle any platform dependencies present. These shared forms of the libraries work on all machine types, or have a different versions installed on each machine type. However, there are cases where make must compile with explicit knowledge of hardware platforms, such as when these shared libraries are themselves compiled.

Commonrules/defs has some support for platform-dependent makes. Include the file $(PRODUCTDEFS) in your Makefile to set macros for the type of OS, graphics, CPU board, graphics board, CPU subgroup that corresponds to the PRODUCT environment variable. The most typical ones used are CPUBOARD and GFXBOARD. You can define the values of these macros in your source code or use them in your Makefile.

PRODUCT variables

PRODUCT	SYSTEM	GRAPHICS	CPUBOARD	GFXBOARD	SUBGR
4D100B	SVR3	GL4D4	IP5	STAPUFT	IP5GT
4D100	SVR3	GL4D2	IP5	CLOVER2	IP5GT
4D200B	SVR3	GL4D4	IP5	STAPUFT	IP7GT
4D200C	SVR3	GL4D4	IP5	STAPUFT	IP7GT
4D200	SVR3	GL4D2	IP5	CLOVER2	IP7GT
4D20	SVR3	GL4D3	IP6	ECLIPSE	
4D210	SVR3	GL4D4	IP5	STAPUFT	IP9GT
4D60T	SVR3	GL4D4	IP4	CLOVER1	
4D60	SVR3	GL4D	R2300	CLOVER	
4D80T	SVR3	GL4D2	IP4	CLOVER2	IP4GT
4D80	SVR3	GL4D2	R2300	CLOVER2	IP4GT
4DSERVER	SVR3				
M4D20	SVR3	GL4D2	IP6	ECLIPSE	VGR
4D30E	SVR3	GL4D3	IP12	ECLIPSE	
4D30G	SVR3	GL4D3	IP12	ECLIPSE	ECLIPSE
4D30L	SVR3		IP12	LIGHT	
4D30	SVR3	GL4D3	IP12	ECLIPSE	ECLIPSE

Another kind of platform dependent make depends on the version of the OS such as 3.3.2 or 4.0. You define the macro RELEASE by including the file $(RELEASEDEFS) in your Makefile. It is automatically included when you include $(PRODUCTDEFS).

Makefile.proddep

```
### This only vaguely resembles the "Basic Makefile"on page B-2.
     Sample Makefile for a platform dependent directory
#
#
# Makefile for foo(1).
#
#    "$Revision$"

include $(ROOT)/usr/include/make/commondefs

### There are files of macros that define hardware characteristics
### for each value of the PRODUCT environment variable.
### These product definition files are in $(ROOT)/usr/include/make.
### The macros defined give the flavor of OS (SYSTEM),
### graphics (GRAPHICS), CPU board (CPUBOARD),
### graphics board (GFXBOARD), and CPU subgroup (SUBGR).
### This Makefile just uses the CPUBOARD macro.
include $(PRODUCTDEFS)

### These are the machines for which this Makefile creates results.
EVERYPRODUCT = 4D60 4D80T 4D20 4D30 4D100

### There will be a directory for each value of CPUBOARD containing
### the results produced by this Makefile. $(OAREA) is the macro
### referring to this directory.
OAREA=$(CPUBOARD).O

### VPATH contains names of directories to look in for any files not
### found in this directory. In this case it is the directory $(OAREA).
VPATH=$(OAREA)

### $(LCDEFS) defines the C flags local to this Makefile. In this case,
### make tells the C compiler to import the value of CPUBOARD as if the
### developer #defined it in the code.
LCDEFS=-D$(CPUBOARD)

CFILES=foo.c foosubs.c
TARGETS=foo

### The target default in this Makefile tests to see if $(PRODUCT)
### is defined. If it is, then make creates results just for that
### platform. If it is not defined, then make creates results for all
### platforms mentioned in $(EVERYPRODUCT).
default: $(_FORCE)
     @if test -n "$(PRODUCT)"; then \
          exec $(MAKE) product_default; \
     else \
          exec $(MAKE) every; \
     fi

COMMONPREF=foo

include $(COMMONRULES)

### The developer must set MKDEPRULE after make includes commonrules.
### Then all dependencies can be done correctly for each platform.
MKDEPRULE=$(EVERYPRODUCT_MKDEPRULE)

### remainder of file is on next page.
```

Makefile.proddep
(continued)

```
### LDIRT defines intermediate files that make will remove
### by using the target clean. In this case, make selects the object
### files of $(OAREA) to remove.
LDIRT= $(OAREA)/*.o

### The developer must rewrite the inference rule to generate .o
### files from .c files so that it deposits object files in $(OAREA).
.c.o:
	@rm -f $*.o
	$(CCF) $< -c -o $(OAREA)/$*.o

### This is the target used to generate results for all platforms.
every: $(_FORCE)
	@for p in $(EVERYPRODUCT); do \
		echo "Making foo for PRODUCT $$p."; \
		$(MAKE) PRODUCT=$$p; \
	done

### This is the target used to generate results for just $(PRODUCT).
product_default: $(OAREA) $(TARGETS)

### The template structures this target like the target default above.
### It will install for $(PRODUCT) if it is defined, otherwise
### it will install for all platforms.
install: $(_FORCE)
	@if test -n "$(PRODUCT)"; then \
		exec $(MAKE) product_install; \
	else \
		exec $(MAKE) every_install; \
	fi

### This is the target used to install results for just $(PRODUCT).
product_install: product_default
	cd $(OAREA); \
	$(INSTALL) -idb std.sw.foo -idb "mach(CPUBOARD=$(CPUBOARD))" \
		-F /usr/sbin $(TARGETS)

### This is the target used to install results for all platforms.
every_install: $(_FORCE)
	@for p in $(EVERYPRODUCT); do \
		echo "Making install for PRODUCT $$p."; \
		$(MAKE) PRODUCT=$$p product_install; \
	done;

### This target will create $(OAREA).
$(OAREA): $(_FORCE)
	@if test ! -d $@; then \
		echo "\trm -rf $@; mkdir $@;" \
		rm -rf $@; mkdir $@; \
	fi

foo: $(OBJECTS)
	cd $(OAREA); $(CCF) $(OBJECTS) $(LDFLAGS) -o $@
```

Appendix C: Commondefs Listing

```
# Copyright 1990 Silicon Graphics, Inc.  All rights reserved.
#
#ident "$Revision: 1.68 $"
#
# Common makefile definitions.

# Notes:
#   - Definitions with the same names only need to be passed on the
#     command line of recursive makes if they would be redefined by
#     the sub-makefile.  Definitions passed on the command line are
#     not reset by the environment or the definitions in the makefile.
#
COMMONRULES= $(ROOT)/usr/include/make/commonrules
COMMONTARGS= clobber clean rmtargets depend incdepend fluff tags
PRODUCTDEFS= $(ROOT)/usr/include/make/$(PRODUCT)defs
RELEASEDEFS= $(ROOT)/usr/include/make/releasedefs


#
# Make tools, i.e., programs which must exist on both native and cross
# development systems to build the software.  $(ECHO) is a make tool be-
cause
# echo usage in makefiles should be portable.
#
AR   = $(TOOLROOT)/usr/bin/ar
AS   = $(TOOLROOT)/usr/bin/as
AWK  = $(TOOLROOT)/usr/bin/awk
C++  = $(TOOLROOT)/usr/bin/CC
CC   = $(TOOLROOT)/usr/bin/cc
CPS  = $(TOOLROOT)/usr/sbin/cps
ECHO = $(TOOLROOT)/bin/echo
F77  = $(TOOLROOT)/usr/bin/f77
FC   = $(F77)# for MIPS compatibility
LD   = $(TOOLROOT)/usr/bin/ld
LEX  = $(TOOLROOT)/usr/bin/lex
LIBSPEC= $(TOOLROOT)/usr/sbin/libspec
LINT = $(TOOLROOT)/usr/bin/lint
LORDER= $(TOOLROOT)/usr/bin/lorder
M4   = $(TOOLROOT)/usr/bin/m4
MKF2C= $(TOOLROOT)/usr/bin/mkf2c
MKSHLIB= $(TOOLROOT)/usr/bin/mkshlib
NAWK = $(TOOLROOT)/usr/bin/nawk
NM   = $(TOOLROOT)/usr/bin/nm
OAWK = $(TOOLROOT)/usr/bin/oawk
PC   = $(TOOLROOT)/usr/bin/pc
RANLIB= $(AR) ts > /dev/null# for IRIS 2000/3000 compatibility
SIZE = $(TOOLROOT)/usr/bin/size
STRIP= $(TOOLROOT)/usr/bin/strip
SHELL= $(TOOLROOT)/bin/sh
YACC = $(TOOLROOT)/usr/bin/yacc


#
# Cc flags, composed of variable (set on the command line), local
# (defined in the makefile), and global (defined in this file) parts, in
# that order.  This ordering has been used so that the variable or
# locally specified include directories are searched before the globally
# specified ones.
#
CFLAGS= $(VCFLAGS) $(LCFLAGS) $(GCFLAGS)

#
```

```
# Each of these three components is divided into defines (-D's and -U's),
# includes (-I's), and other options.  By segregating the different
# classes of flag to cc, the defines (CDEFS) and includes (CINCS) can be
# easily given to other programs, e.g., lint.
#
# Notes:
#   - The local assignments should be to LCOPTS, LCDEFS, and LCINCS, not
to
#     LCFLAGS, although CFLAGS will be correctly set if this is done.
#   - If a program cannot be optimized, it should override the setting of
#     OPTIMIZER with a line such as "OPTIMIZER=" in its make file.
#   - If a program cannot be compiled with prototype checking, its make-
file
#     should reset PROTOTYPES to the empty string.
#
VCFLAGS= $(VCDEFS) $(VCINCS) $(VCOPTS)
LCFLAGS= $(LCDEFS) $(LCINCS) $(LCOPTS)
GCFLAGS= $(GCDEFS) $(GCINCS) $(GCOPTS)

COPTS= $(VCOPTS) $(LCOPTS) $(GCOPTS)
CDEFS= $(VCDEFS) $(LCDEFS) $(GCDEFS)
CINCS= $(VCINCS) $(LCINCS) $(GCINCS)

#
# The nullary -I flag is defined to defeat searches of /usr/include in
# a cross development environment.  Where it is placed on the command line
# does not matter.
#
GCOPTS= $(OPTIMIZER) $(PROTOTYPES)
GCDEFS=
GCINCS= -I -I$(INCLDIR)

#
# Default optimizer and prototype options
#
OPTIMIZER  = -O
PROTOTYPES = -prototypes


#
# C++ flags are decomposed using the same hierarchy as C flags.
#
C++FLAGS  = $(VC++FLAGS) $(LC++FLAGS) $(GC++FLAGS)

VC++FLAGS = $(VC++DEFS) $(VC++INCS) $(VC++OPTS)
LC++FLAGS = $(LC++DEFS) $(LC++INCS) $(LC++OPTS)
GC++FLAGS = $(GC++DEFS) $(GC++INCS) $(GC++OPTS)

C++OPTS   = $(VC++OPTS) $(LC++OPTS) $(GC++OPTS)
C++DEFS   = $(VC++DEFS) $(LC++DEFS) $(GC++DEFS)
C++INCS   = $(VC++INCS) $(LC++INCS) $(GC++INCS)

GC++OPTS  = $(OPTIMIZER)
GC++INCS  = -I -I$(INCLDIR)/CC -I$(INCLDIR)


#
# Loader flags, composed of library (-l's) and option parts, with
# the libraries appearing last.  Both of these are divided into variable,
# local, and global parts.  The composition of LDFLAGS is done in the
# other "direction" from CFLAGS so that all the -L's, which are part of
# LDOPTS, appear before any of the -l's, which are part of LDLIBS.
# Another benefit of segregating the libraries from the remaining of the
# loader options is that the libraries alone can easily be given to
# another program, e.g., lint.
#
# Notes:
```

```
#   - -s belongs in GCOPTS or in the IDB program that does the actual
#     installation.
#   - If a program should not be linked with the shared version of libc,
#     then its make file should override the setting of SHDLIBC with a
#     line such as "SHDLIBC=".
#
LDFLAGS= $(LDOPTS) $(LDLIBS)

LDOPTS= $(VLDOPTS) $(LLDOPTS) $(GLDOPTS)
LDLIBS= $(VLDLIBS) $(LLDLIBS) $(GLDLIBS)

GLDOPTS= -L -L$(ROOT)/usr/lib
GLDLIBS= $(SHDLIBC)

#
# In order to be able to turn off shared library usage, we set up macros
# defining shared library options here.
#
SHDLIBC=-lc_s
SHDGL=-lgl_s


#
# F77 flags are just like cc flags.
#
FFLAGS= $(VFFLAGS) $(LFFLAGS) $(GFFLAGS)

VFFLAGS= $(VF77DEFS) $(VF77INCS) $(VF77OPTS)
LFFLAGS= $(LF77DEFS) $(LF77INCS) $(LF77OPTS)
GFFLAGS= $(GF77DEFS) $(GF77INCS) $(GF77OPTS)

F77OPTS= $(VF77OPTS) $(LF77OPTS) $(GF77OPTS)
F77DEFS= $(VF77DEFS) $(LF77DEFS) $(GF77DEFS)
F77INCS= $(VF77INCS) $(LF77INCS) $(GF77INCS)

GF77OPTS= $(GCOPTS)
GF77DEFS= $(GCDEFS)
GF77INCS= $(GCINCS)


#
# Pc flags are just like cc flags.
#
PFLAGS= $(VPFLAGS) $(LPFLAGS) $(GPFLAGS)

VPFLAGS= $(VPDEFS) $(VPINCS) $(VPOPTS)
LPFLAGS= $(LPDEFS) $(LPINCS) $(LPOPTS)
GPFLAGS= $(GPDEFS) $(GPINCS) $(GPOPTS)

POPTS= $(VPOPTS) $(LPOPTS) $(GPOPTS)
PDEFS= $(VPDEFS) $(LPDEFS) $(GPDEFS)
PINCS= $(VPINCS) $(LPINCS) $(GPINCS)

GPOPTS= $(GCOPTS)
GPDEFS= $(GCDEFS)
GPINCS= $(GCINCS)


#
# The install command to use.
#
INSTALL= $(TOOLROOT)/etc/install

#
# Shell script for generating make dependencies.  MKDEPEND is a shorthand
# for the tool's absolute pathname.  MKDEPENDC adds MKDEPCFLAGS and the -c
```

```
# mkdepend option to this.  The other language's mkdepend variables try to
# include their language's name in the variable names.  Unfortunately, a
# lot of makefiles already use the nondescript LMKDEPFLAGS for C language
# mkdepend options, so we initialize LMKDEPCFLAGS with $(LMKDEPFLAGS).
#
MKDEPEND     = $(TOOLROOT)/usr/sbin/mkdepend
MKDEPENDAS   = $(MKDEPEND) $(MKDEPASFLAGS) -c "$(CC) $(ASFLAGS) -M"
MKDEPENDC++  = $(MKDEPEND) $(MKDEPC++FLAGS) -c "$(C++F) -M"
MKDEPENDC    = $(MKDEPEND) $(MKDEPCFLAGS) -c "$(CCF) -M"

MKDEPASFLAGS  = $(VMKDEPASFLAGS) $(LMKDEPASFLAGS) $(GMKDEPASFLAGS)
MKDEPC++FLAGS = $(VMKDEPC++FLAGS) $(LMKDEPC++FLAGS) $(GMKDEPC++FLAGS)
MKDEPCFLAGS   = $(VMKDEPCFLAGS) $(LMKDEPCFLAGS) $(GMKDEPCFLAGS)
LMKDEPCFLAGS  = $(LMKDEPFLAGS)

GMKDEPFLAGS    = -e 's@ $(INCLDIR)/@ $$(INCLDIR)/@' -e 's@ $(ROOT)/@
$$(ROOT)/@'
GMKDEPASFLAGS  = $(GMKDEPFLAGS) -s ASM
GMKDEPC++FLAGS = $(GMKDEPFLAGS) -s C++ -e 's@\.o++: @\.o: @'
GMKDEPCFLAGS   = $(GMKDEPFLAGS)

#
# Macro to add to LMKDEPCFLAGS or LMKDEPC++FLAGS if your makefile builds
# single-source programs using null suffix rules (e.g., .c:).  This option
# works for both C and C++ make depend.
#
NULLSUFFIX_MKDEPFLAG= -e 's@\.o+*:@:@'

#
# MKDEPFILE is the name of the dependency database, included by commonru-
les.
#
MKDEPFILE = Makedepend

#
# CDEPFILES lists all C or cc-compiled source files that depend on header
# files computable by $(MKDEPENDC).  C++DEPFILES lists all C++ files hav-
ing
# dependencies computable by $(MKDEPENDC++).  If you develop yacc/C++
source,
# reset these variables after the include of commondefs and before includ-
ing
# commonrules to move $(YFILES) from the C to the C++ list.
#
ASDEPFILES  = $(ASFILES)
C++DEPFILES = $(C++FILES)
CDEPFILES   = $(CFILES) $(LFILES) $(YFILES)
DEPFILES    = $(ASDEPFILES) $(C++DEPFILES) $(CDEPFILES)

#
# Directory shorthands, mainly for make depend (see GMKDEPFLAGS above).
#
INCLDIR= $(ROOT)/usr/include


#
# Convenient command macros that include the flags macros.
#
# You should always invoke make in makefiles via $(MAKE), as make passes
# all command-line variables through the environment to sub-makes.
#
# Never use just $(CCF), etc. in rules that link executables; LDFLAGS
# needs to be included after your objects in the command line.
#
ASF  = $(AS) $(ASFLAGS)
C++F = $(C++) $(C++FLAGS)
CCF  = $(CC) $(CFLAGS)
```

```
F77F = $(F77) $(FFLAGS)
LDF  = $(LD) $(LDFLAGS)
LEXF = $(LEX) $(LFLAGS)
PCF  = $(PC) $(PFLAGS)
YACCF= $(YACC) $(YFLAGS)


#
# Local definitions. These are used for debugging purposes. Make sure that
# the product builds properly without the local definitions, unless these
# local definitions are checked in!
#
# To access a localdefs file outside the current directory, you can
# set LOCALDEFS on the command line. Similarly for localrules. Or,
# you can have the localdefs file just sinclude the appropriate other
# include file.
#
LOCALDEFS  = ./localdefs
LOCALRULES = ./localrules

sinclude $(LOCALDEFS)
```

Appendix D: Commonrules Listing

```
# Copyright 1990 Silicon Graphics, Inc.  All rights reserved.
#
#ident "$Revision: 1.37 $"
#
# Common makefile rules.

# Notes:
#    - After including ${ROOT}/usr/include/make/commondefs, a makefile may
#      say ``include ${COMMONRULES}'' to get this file.
#    - It is up to the including makefile to define a default rule before
#      including ${COMMONRULES}.
#    - This file defines the following lists: SOURCES, enumerating all
#      source files; OBJECTS, the .o files derived from compilable source;
#      and DIRT, which lists intermediates and temporary files to be
#      removed by clean.
#    - The including (parent) makefile may define source file lists for
each
#      standard suffix: CFILES for .c, ASFILES for .s (named after AS-
FLAGS),
#     YFILES for .y, etc.  This file combines all such lists into SOURCES.
#    - The parent makefile must define TARGETS in order for clobber to
work.
#    - If the parent makefile must overload the common targets with spe-
cial
#      rules (e.g. to perform recursive or subdirectory makes), then set
#      COMMONPREF to some unique prefix before including ${COMMONRULES},
#      and make sure that each common target depends on its prefixed name.
#      For example, a makefile which passes common targets and install on
#      to makes in subdirectories listed in DIRS might say
#
#         COMMONPREF= xxx
#         include ${COMMONRULES}
#
#         ${COMMONTARGS} install: ${COMMONPREF}$$@
#         @for d in ${DIRS}; do \
#         ${ECHO} "\tcd $$d; ${MAKE} $@"; \
#         cd $$d; ${MAKE} $@; cd ..; \
#         done
#
#      Thus, all of the common rules plus install are passed to sub-makes
#      *and* executed in the current makefile (as xxxclean, xxxclobber,
#      xxxinstall, etc).
#
SOURCES= ${HFILES} ${ASFILES} ${C++FILES} ${CFILES} ${EFILES} ${FFILES} \
      ${LFILES} ${PFILES} ${RFILES} ${SHFILES} ${YFILES}

OBJECTS= ${ASFILES:.s=.o} ${C++FILES:.c++=.o} ${CFILES:.c=.o}
${EFILES:.e=.o} \
      ${FFILES:.f=.o} ${LFILES:.l=.o} ${PFILES:.p=.o} ${RFILES:.r=.o} \
      ${YFILES:.y=.o}


#
# C++ inference rules.  Certain of these may show up in make someday.
#
.SUFFIXES: .c++ .yuk

.c++:
     ${C++} ${C++FLAGS} $< ${LDFLAGS} -o $@
.c++.o:
     ${C++} ${C++FLAGS} -c $<
.c++.s:
     ${C++} ${C++FLAGS} -S $<
```

```
.c++.i:
	${C++} ${C++FLAGS} -E $< > $*.i
.c++.yuk:
	${C++} ${C++FLAGS} -Fc -.yuk $<

#
# Rather than removing ${OBJECTS}, clean removes ${CLEANOBJECTS} which we
# set to *.[ou] by default, to remove obsolete objects and -O3 ucode files
# after source has been reorganized.  If files ending in .[ou] should not
# be removed by clean, this definition can be overridden after the include
# of commonrules to define CLEANOBJECTS=${OBJECTS}.
#
CLEANOBJECTS= *.[ou]

#
# What gets cleaned, apart from objects.
#
DIRT= ${GDIRT} ${VDIRT} ${LDIRT}
GDIRT= a.out core lex.yy.[co] y.tab.[co] \
	${C++FILES:.c++=..c} ${C++FILES:.c++=.yuk} ${_FORCE}

#
# An always-unsatisfied target.  The name is unlikely to occur in a file
tree,
# but if _force existed in a make's current directory, this target would
be
# always-satisfied and targets that depended on it would not be made.
#
_FORCE= ${COMMONPREF}_force
${_FORCE}:

#
# File removal rules: there are three.
#	- clean removes intermediates and dirt
#	- clobber removes targets as well as intermediates and dirt
#	- rmtargets removes targets only
# One might 'make clean' in a large tree to reclaim disk space after tar-
gets
# are built, but before they are archived into distribution images on
disk.
# One might 'make rmtargets' to remove badly-linked executables, and then
# run a 'make' to re-link the good objects.
#
# If you use incdepend (see below), then 'make clobber' will remove the
# .*dependtime marker files used by incdepend to find modified ${DEP-
FILES}.
# Multi-product incremental depend uses the .*${PRODUCT}incdepend mark-
ers.
# To clobber everything but the marker files, use 'make clean rmtargets'.
#
.PRECIOUS: .sdependtime .c++dependtime .cdependtime \
	.s${PRODUCT}incdepend .c++${PRODUCT}incdepend .c${PRODUCT}incde-
pend

${COMMONPREF}clobber: ${COMMONPREF}clean ${COMMONPREF}rmtargets ${_FORCE}
	rm -rf ${MKDEPFILE} .*dependtime .*incdepend

${COMMONPREF}clean: ${_FORCE}
	rm -rf ${CLEANOBJECTS} ${DIRT}

${COMMONPREF}rmtargets: ${_FORCE}
	rm -rf ${TARGETS}

#
# Automated header dependency inference.  Most makefiles need only set the
# CFILES, ASFILES, etc. lists and include commonrules.  Those makefiles
that
```

```
# build product-dependent source (with product-dependent includes) should
set
# MKDEPRULE to ${EVERYPRODUCT_MKDEPRULE} *after* including commonrules,
and
# should set EVERYPRODUCT to the list of products that they build.
#
MKDEPRULE= NP
EVERYPRODUCT_MKDEPRULE= EP

${COMMONPREF}depend: ${_FORCE}
	@slist="${ASDEPFILES}" Clist="${C++DEPFILES}" clist="${CDEPFILES}";
\
	case ${MKDEPRULE} in \
	  NP)case "$$slist" in \
	      *.*) \
	    ${ECHO} "${MKDEPENDAS} ${MKDEPFILE} $$slist"; \
	    ${MKDEPENDAS} ${MKDEPFILE} $$slist; \
	    touch .sdependtime; \
	    esac; \
	    case "$$Clist" in \
	      *.*) \
	    ${ECHO} "${MKDEPENDC++} ${MKDEPFILE} $$Clist"; \
	    ${MKDEPENDC++} ${MKDEPFILE} $$Clist; \
	    touch .c++dependtime; \
	    esac; \
	    case "$$clist" in \
	      *.*) \
	    ${ECHO} "\t${MKDEPENDC} ${MKDEPFILE} $$clist"; \
	    ${MKDEPENDC} ${MKDEPFILE} $$clist; \
	    touch .cdependtime; \
	    esac;; \
	  EP)nprod=`echo ${EVERYPRODUCT} | wc -w`; \
	    case "$$slist" in \
	      *.*) \
	        for p in ${EVERYPRODUCT}""; do \
	    ${ECHO} 1>&2 "Making .s depend for PRODUCT $$p."; \
	    ${MAKE} -sf ${MAKEFILE} PRODUCT=$$p _s$${p}depend; \
	        done | \
	        ${MKDEPEND} ${MKDEPASFLAGS} -p $$nprod ${MKDEPFILE}; \
	    esac; \
	    case "$$Clist" in \
	      *.*) \
	        for p in ${EVERYPRODUCT}""; do \
	    ${ECHO} 1>&2 "Making .c++ depend for PRODUCT $$p."; \
	    ${MAKE} -sf ${MAKEFILE} PRODUCT=$$p _c++$${p}depend; \
	        done | \
	        ${MKDEPEND} ${MKDEPC++FLAGS} -p $$nprod ${MKDEPFILE}; \
	    esac; \
	    case "$$clist" in \
	      *.*) \
	        for p in ${EVERYPRODUCT}""; do \
	    ${ECHO} 1>&2 "Making .c depend for PRODUCT $$p."; \
	    ${MAKE} -sf ${MAKEFILE} PRODUCT=$$p _c$${p}depend; \
	        done | \
	        ${MKDEPEND} ${MKDEPCFLAGS} -p $$nprod ${MKDEPFILE}; \
	    esac; \
	esac

_s${PRODUCT}depend: ${ASDEPFILES} ${_FORCE}
	${ASF} -M ${ASDEPFILES} | ${PRODUCT_RAWDEPFILTER}; \
	touch .s${PRODUCT}incdepend

_c++${PRODUCT}depend: ${C++DEPFILES} ${_FORCE}
	${C++F} -M ${C++DEPFILES} | ${PRODUCT_RAWDEPFILTER}; \
	touch .c++${PRODUCT}incdepend

_c${PRODUCT}depend: ${CDEPFILES} ${_FORCE}
```

```
	${CCF} -M ${CDEPFILES} | ${PRODUCT_RAWDEPFILTER}; \
	touch .c${PRODUCT}incdepend

#
# Incremental depend uses marker files to find ${DEPFILES} that are newer
# than their dependencies.  Note that the non-incremental rules, above,
also
# touch the marker files.  Care is taken not to write a product-indepen-
dent
# dependency on ${DEPFILES}, so that the list of dependent source can vary
# with each product.
#
# XXX can't run only one sub-make that depends on all .*dependtime, be-
cause
# XXX smake will parallelize and mkdepend doesn't interlock itself
#
${COMMONPREF}incdepend: ${_FORCE}
	@slist="${ASDEPFILES}" Clist="${C++DEPFILES}" clist="${CDEPFILES}";
\
	case ${MKDEPRULE} in \
	  NP)case "$$slist" in \
	       *.*) \
	     ${MAKE} -f ${MAKEFILE} _quiet.sdependtime; \
	     esac; \
	     case "$$Clist" in \
	       *.*) \
	     ${MAKE} -f ${MAKEFILE} _quiet.c++dependtime; \
	     esac; \
	     case "$$clist" in \
	       *.*) \
	     ${MAKE} -f ${MAKEFILE} _quiet.cdependtime; \
	     esac;; \
	  EP)nprod=`echo ${EVERYPRODUCT} | wc -w`; \
	     case "$$slist" in \
	       *.*) \
	         for p in ${EVERYPRODUCT}""; do \
	     ${ECHO} 1>&2 "Making .s incdepend for PRODUCT $$p."; \
	     ${MAKE} -sf ${MAKEFILE} PRODUCT=$$p _s$${p}incdepend; \
	         done | \
	         ${MKDEPEND} ${MKDEPASFLAGS} -ip $$nprod ${MKDEPFILE}; \
	     esac; \
	     case "$$Clist" in \
	       *.*) \
	         for p in ${EVERYPRODUCT}""; do \
	     ${ECHO} 1>&2 "Making .c++ incdepend for PRODUCT $$p.";\
	     ${MAKE} -sf ${MAKEFILE} PRODUCT=$$p _c++$${p}incdepend;\
	         done | \
	         ${MKDEPEND} ${MKDEPC++FLAGS} -ip $$nprod ${MKDEPFILE}; \
	     esac; \
	     case "$$clist" in \
	       *.*) \
	         for p in ${EVERYPRODUCT}""; do \
	     ${ECHO} 1>&2 "Making .c incdepend for PRODUCT $$p."; \
	     ${MAKE} -sf ${MAKEFILE} PRODUCT=$$p _c$${p}incdepend; \
	         done | \
	         ${MKDEPEND} ${MKDEPCFLAGS} -ip $$nprod ${MKDEPFILE}; \
	     esac; \
	esac

# so that make doesn't announce "`.sdependtime' is up to date."
_quiet.sdependtime: .sdependtime
_quiet.c++dependtime: .c++dependtime
_quiet.cdependtime: .cdependtime

.sdependtime: ${ASDEPFILES}
	@if test -n "$?"; then \
	     ${ECHO} "\t${MKDEPENDAS} -i ${MKDEPFILE} $?"; \
```

```
		${MKDEPENDAS} -i ${MKDEPFILE} $?; \
		touch $@; \
	fi

.c++dependtime: ${C++DEPFILES}
	@if test -n "$?"; then \
		${ECHO} "\t${MKDEPENDC++} -i ${MKDEPFILE} $?"; \
		${MKDEPENDC++} -i ${MKDEPFILE} $?; \
		touch $@; \
	fi

.cdependtime: ${CDEPFILES}
	@if test -n "$?"; then \
		${ECHO} "\t${MKDEPENDC} -i ${MKDEPFILE} $?"; \
		${MKDEPENDC} -i ${MKDEPFILE} $?; \
		touch $@; \
	fi

# you can't add dependencies to a target that begins with '.'
_s${PRODUCT}incdepend: .s${PRODUCT}incdepend
_c++${PRODUCT}incdepend: .c++${PRODUCT}incdepend
_c${PRODUCT}incdepend: .c${PRODUCT}incdepend

.s${PRODUCT}incdepend: ${ASDEPFILES}
	@if test -n "$?"; then \
		${ASF} -M $? | ${PRODUCT_RAWDEPFILTER}; \
		touch $@; \
	fi

.c++${PRODUCT}incdepend: ${C++DEPFILES}
	@if test -n "$?"; then \
		${C++F} -M $? | ${PRODUCT_RAWDEPFILTER}; \
		touch $@; \
	fi

.c${PRODUCT}incdepend: ${CDEPFILES}
	@if test -n "$?"; then \
		${CCF} -M $? | ${PRODUCT_RAWDEPFILTER}; \
		touch $@; \
	fi

#
# A sed filter that prepends ${VPATH} to object targets emitted by cc -M.
# ${VPATH} should name a directory that holds product-dependent objects.
#
PRODUCT_RAWDEPFILTER= sed -e 's:^:${VPATH}/:'

#
# Lint and C tags support.
#
${COMMONPREF}fluff: ${_FORCE}
	${LINT} ${LINTFLAGS} ${CDEFS} ${CINCS} ${CFILES} ${LDLIBS}

CTAGS= ctags

${COMMONPREF}tags: ${_FORCE}
	rm -f tags
	find . -name '*.[cfhlpy]' ! -name '.*' ! -name 'llib-*' -print | \
	    sed 's:^\./::' | \
	    xargs ${CTAGS} -a
	if test -f tags; then \
		sort -u +0 -1 -o tags tags; \
	fi

#
# Include the make dependency file if it exists.
```

```
#
sinclude ${MKDEPFILE}

#
# Local make rules
#
sinclude ${LOCALRULES}
```