Engineer's Handbook
Makefile Type Stuff

May 5, 1994

Contents

Engineer's Handbook
Makefile Type Stuff

May 5, 1994

Building Manual Pages and Release Notes

access via: handbook buildman

probable data location: bonnie.wpd:/proj/sherwood/isms/man/doc/buildman.ps

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.

TOOLROOT 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. *Boot* 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/apps
att/usr/src/cmd
att/usr/src/cmplrs
att/usr/src/gfx
att/usr/src/head
att/usr/src/lib
att/usr/src/man
att/usr/src/sgionly
att/usr/src/stand
att/usr/src/uts/mips
```

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. RAWIDB 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 in-

stallation, 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 *features* 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}
```

Engineer's Handbook
Makefile Type Stuff

May 5, 1994

Engineer's Guide to Packaging Software for Inst

access via: handbook sw4inst

probable data location: jake.wpd:/depot/sgi_info/old_jake_att_doc/instpkg/book.out

Silicon Graphics Inc Common Makefile include files

NOTES

Publication Date: 5/6/93

Overview

- Overview
  - Why are they useful?
  - What is common?
  - Where do I go for more information?
  - What common actions are expected of you?
- **commondefs and commonrules** (page 6)
  - Used implicitly almost everywhere
  - Used in source directories (both parent and leaf directories)
  - Most typical include files
- **ismcommondefs and ismcommonrules** (page 18)
  - Used just in parent directory of ism and its build directory
- **mandefs and manrules** (page 27)
  - Used in parent man directory and leaf man directories
- **relnotesrules** (page 33)
  - Used in release notes directories
- Future Additions (page 38)

Why are they useful?

- Files to include in your makefiles
- Simplify and shorten Makefiles
- Provide common targets for common actions
- Provide Tool abbreviations and option management
- Support cross compilation
  - Run ppropriate tools from $TOOLROOT
  - Access appropriate header files and libraries from $ROOT
- NOTE: this document applies to Sherwood (5.0), or subsequent, releases

What is common?

- Some common targets are typically provided
  - **clean** remove TARGET by-products (*.o files, etc.)
  - **clobber** remove all but source
  - **rmtargets** just remove $(TARGETS)
- Other {defs/rules} files tend to extend common{defs/rules}

What common actions are expected of you?

- A common Makefile structure, namely:
  - Include the ***defs*** file
  - Add your local definitions and overrides
  - Add your default Makefile target
  - Include the ***rules*** file
  - Add your other Makefile targets
- Some common targets
  - **default** produce the targets of this directory only and pass the target to a make in appropriate subdirectories. manrules and relnoterules are an exception, they provide this target.
  - **install** same as default plus invoke $(INSTALL) on the targets, with appropriate idb tag info manrules and relnoterules are an exception, they provide this target.

Where do I go for more information?

- Use the handbook commands to access the following chapters:
  - **makeconv** SGI Makefile Conventions
  - **ism** ISM Class notes (Independent Software Module)
  - **buildman** Building Manual Pages and Release Notes
  - **SGIintro** Pointers to Engineering Resources at SGI. It has information on some of the code names for releases and products at SGI.

commondefs and commonrules

- base set of macros and targets used in most SGI Makefiles, primarily focused on producing executables
- Macros and Flags Provided
  - Compiler and Loader option flags (page 7)
  - Flag Defaults (page 9)
  - Tool Abbreviations (via macros) (page 10)
  - Combined Compiler macro name and flags (page 11)
  - Other macros (page 12)
- Other features
- Common Targets
- Makefile implications
  - Makefile Structure (page 14)
  - Overriding the default rules (page 16)
  - Sample Makefile (page 17)

Compiler and Loader option flags

- Three sources of definitions
  - **G** global, defined in commondefs
  - **L** local, defined in Makefile
  - **V** variable, defined on Makefile command line
  - V overrides L which overrides G
- Three types of compiler flags
  - **DEFS** Mechanism to pass environment variables to source
  - **INCS** Directories to search to satisfy #includes
  - **OPTS** all other compiler options
- Four types of compilers
  - **C** C
  - **C++** C++
  - **F77** FORTRAN 77
  - **P** Pascal
- Three types of loader flags
  - **LIBS** directories to search to satisfy external references
  - **DSO** for making Dynamic Shared Objects (with lib rules)
  - **OPTS** all other loader options

Compiler and Loader option flags (continued)

- Order of components to form flag name
  - source
  - compiler letters or 'LD' for loader
  - type of flag
  - Example (of C Defs local to your Makefile)
    - LCDEFS =- -DCPUTYPE
- Flags are combined for you (based on the option flags you provide)
  - **CFLAGS**
  - **C++FLAGS**
  - **FFLAGS**
  - **PFLAGS**
  - **LDFLAGS**
- **C++FLAGS** are used with YACC and LEX compilations

Flag Defaults

- use of $(ROOT)/usr/include instead of /usr/include
- for C, C++, P, F77, YACC and LEX
  - Automatic Makefile dependencies to file Makedepend (MKDEPOPT)
- for C, C++, P, and F77
  - Optimization turned on (OPTIMIZER)
  - Correct endian (based on ***PRODUCT***defs file) (ENDIAN)
- for C
  - Some warnings turned off (WOFF)
- for C++
  - use of $(ROOT)/usr/include/cc instead of /usr/include/cc
  - C++ defines:
    _MODERN_C
    _SVR4_SOURCE
    _SGI_SOURCE
- for LD
  - use of $(ROOT)/{lib, usr/lib} instead of {/lib, /usr/lib}

Flag Overrides

- OPTIMIZER, WOFF can be overridden by unsetting the macro
- To select different dependency file, set macro MKDEPFILE

Tool Abbreviations (via macros)

- Compilers
  - CC
  - C++ (or C++C)
  - PC
  - F77 (or FC)
  - HOST_CC
  - HOST_C++
- Compiler Related
  - AS
  - LEX
  - LINT
  - MKF2C
  - YACC
  - NM
- Linking and Loading
  - AR
  - LD
  - LIBSPEC
  - LORDER
  - MKSHLIB
  - SIZE
  - STRIP
  - RANLIB
- Miscellaneous
  - INSTALL
  - SHELL
- Supported, but not recommended (i.e. reference the command directly, since its function is independant of specific release or hardware platform)
  - AWK
  - NAWK
  - ECHO
  - M4
  - MKF2C

Combined Compiler macro name and flags

- $(C++F) is equivalent to $(C++) $(C++FLAGS)
- Similarly (i.e. substitute new letter in three places for C++)
  - PC for Pascal
  - LD for loader
  - AS for assembler
  - LEX for Lex
  - YACC for YACC
- Exceptions
  - $(CCF) is equivalent to $(CC) $(CFLAGS)
  - $(F77F) is equivalent to $(F77) $(FFLAGS)

Other macros

- DIRT macros -> {V,L,G,X}DIRT
  - **{V,L,G}DIRT** define target by-products to be removed by the clean target
  - **XDIRT** for use of any enclosing set of commondefs, e.g. ismcommondefs
- subdirectory support - **SUBDIRS_MAKERULE**
  - **SUBDIRS_MAKERULE**as a command for a target, will execute the same target in each subdirectory of the macro SUBDIRS (echoing what it is going to do to stdout)
  - can override the target passed to the subdirectories by setting the envariable ROOT to be the desired target
  - can override the default action for each directory by resetting the macro SUBDIR_MAKERULE
  - set macro NOSUBMESG to yes and any non-existent directory of the SUBDIRS macro will not be reported as an error.
- macros for other common Makefile include files
  - library DSO support
  - **HEADERS_SUBDIRS_MAKERULE**identical to SUBDIRS_MAKERULE except operates on HEADERS_SUBDIRS instead of SUBDIRS
  - **EXPORTS_SUBDIRS_MAKERULE**identical to SUBDIRS_MAKERULE except operates on EXPORTS_SUBDIRS instead of SUBDIRS

Other Features

- local{defs,rules}
  - included at end of corresponding common file, if present in the current directory
  - useful for defining things part of your build process, but not part of build groups build process, e.g. setting DSOREGFILE
  - CAUTION: in general do not want any differences between your build process and that of build groups build process
  - can override default file name via macros LOCALDEFS and LOCALRULES.
  - To access such a file located outside of the current directory, then set LOCALDEFS or LOCALRULES on the command line. You can also just include the file from within a local LOCALDEFS or LOCALRULES file.

Makefile Structure

- You define your source files via the macros:
  - **CFILES** for C code
  - **C++FILES** for C++ code
  - **ASFILES** for Assembly code
  - **FFILES** for FORTRAN code
  - **RFILES** for RATFOR code (see f77(1))
  - **EFILES** for RATFOR code (see f77(1))
  - **PFILES** for Pascal code
  - **LFILES** for Lex code
    implicit rules of ***name***.l into ***name***.o rather than lex.yy.o
  - **YFILES** for YACC code
    implicit rules of ***name***.y into ***name***.o rather than y.tab.o
  - **SHFILES** for a shell scripts
  - **HFILES** for header files
- You define the list of targets to be made by the Makefile
  - **TARGETS**
- You define the default rule (typically just =>
  default: $(TARGETS))

Makefile Structure (continued)

- You are provided:
  - OBJECTS macro, the list of objects produced by the makefile, derived from the FILES macros you define.
  - default targets:
  - **clean** (Standard SGI makefile target)
  - **clobber** (Standard SGI makefile target)
  - **rmtargets** (Standard SGI makefile target)
  - **fluff** run lint on source files
  - **tags** form ctags file from sources for use with vi and other editors

Overriding the default rules

- by renaming them, as well as,
- via setting macro COMMONPREF to a unique value
- For instance,

  COMMONPREF = xyzzy

  makes commonrules define

  xyzzyclean, xyzzyclobber, etc.

  instead of

  clean, clobber, etc

Sample Makefile

```
include $(ROOT)/usr/include/make/commondefs
# <definitions of this Makefile, last macro definition wins>
CFILES = foo.c bar.c
TARGETS = foo
default: $(TARGETS)
include $(COMMONRULES)
install: default
          $(INSTALL) ...
foo: $(OBJECTS)
          $(CCF) $(OBJECTS) $(LDFLAGS) -o $@
# <rules of this Makefile>
```

ismcommondefs and ismcommonrules

- used in just two directories (main Makefile and build/Makefile) to help produce installable images of a product and otherwise support the notion of Independent Software Modules.
- run **handbook-ism** for more info
- main Makefile discussion
  - Macros Provided for the main Makefile (page 19)
  - Makefile Structure for the main Makefile (page 20)
  - Suggested Makefile for the main Makefile (page 21)
- build directory Makefile discussion
  - Flags Provided for the build/Makefile (page 23)
  - Makefile Structure for the build/Makefile (page 24)
  - Suggested Makefile for the build/Makefile (page 26)

Macros Provided for the main Makefile

- **HEADERS_SUBDIRS_MAKERULE**
  - pass target to subdirectories listed in HEADERS_SUBDIRS
  - actually provided by commondefs (See page 12)
- **EXPORTS_SUBDIRS_MAKERULE**
  - pass target to subdirectories listed in EXPORTS_SUBDIRS
  - actually provided by commondefs (See page 12)

Makefile Structure for the main Makefile

- Use the default Makefile
- You define the macro:
  - **SUBDIRS**: list of subdirectories to be made
- You define the targets (defined for you if you use the default Makefile shown on the next page):
  - **default** build the ISM (Standard SGI makefile target)
  - **headers** populate $ROOT with header files needed by other ISMs
  - **exports** populate $ROOT with libraries produced by the ISM for use by other ISMs
  - **install** install what you have built (Standard SGI makefile target)
  - **rawidb** set RAWIDB=build/ISM and run **make install**
  - **ism** run **buildism** in the build subdirectory
  - **images** run **buildimages** in the build subdirectory

Suggested Makefile for the main Makefile

```
#
# Makefile for foo ism
#
# The main targets at the top level are:
#
# default:this performs a bootstrap (headers and exports) then
#    builds all of the software in place. No rawidb is generated and
#    targets are not installed anywhere. No ism image is generated
#
# headers:install all headers. This should always be run without RAWIDB set.
#
# exports:builds and installs all libraries and data files that are required
#    to build the rest of the ism. This should always be run without
#    RAWIDB set.
#
# install:builds and installs the entire ism. 'headers' and 'exports'
#    must be run previous to 'install'. With RAWIDB set this will
#    generate the rawidb for the ism. Without RAWIDB set, all the
#    ism objects will be installed in $ROOT
#
# images:descends into the build directory and generates the files needed
#    for the build of entire system.
#    'rawidb' must be run previous to 'images'.
#
# ism:descends into the build directory and generates ism images.
#    'rawidb' must be run previous to 'ism'.
#
# clean:removes .o's
# clobber:removes all non-source. The effective of 'clobber' can be checked
#    by running p_check -w after running 'clobber'
# rmtargets:removes TARGETS only
#
#
# "$Revision: 1.1 $"

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

## customize for subdirs you have (tools etc)
## i.e. replicate/rename src directory as needed
SUBDIRS=man build src
```

```
## other subdirectory list that may apply
#HEADERS_SUBDIRS=
#EXPORTS_SUBDIRS=

COMMONPREF=foo
SRC=`pwd`
IDBFILE="`pwd`/build/IDB"
ISM_NAME=foo
## override to default ALPHA envariable setting, if desired
#ALPHA=000001

default:headers exports $(_FORCE)
     $(SUBDIRS_MAKERULE)

headers:$(_FORCE)
##   uncomment next line if there are headers to be generated
#    $(HEADERS_SUBDIRS_MAKERULE)

exports:$(_FORCE)
##   uncomment next line if there are libraries to be generated
#    $(EXPORTS_SUBDIRS_MAKERULE)

install:$(_FORCE)
     $(SUBDIRS_MAKERULE)

$(COMMONTARGS):$(_FORCE) $(COMMONPREF)$$@
     $(SUBDIRS_MAKERULE)

include $(ISMCOMMONRULES)

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

rawidb: $(_FORCE)
     @RAWIDB=$(IDBFILE); export RAWIDB ; \
     if [ -f $$RAWIDB ] ; then mv $$RAWIDB $$RAWIDB.prev ; fi ;\
     echo "RAWIDB=$$RAWIDB SRC=$(SRC) $(MAKE) install" ;\
     $(MAKE) SRC=$(SRC) install

ism: $(_FORCE)
     cd build; $(MAKE) buildism

images: $(_FORCE)
     cd build; $(MAKE) buildimages
```

Flags Provided for the build/Makefile

- **GENDISTFLAGS**
  - flags for gendist commands
  - provided from {V,L,G}GENDISTFLAGS
  - you define LGENDISTFLAGS in your Makefile
  - GGENDISTFLAGS is set to **-verbose**
- **IDBJOINFLAGS**
  - flags for idbjoin commands
  - provided from {V,L,G}IDBJOINFLAGS
  - you define LIDBJOINFLAGS in your Makefile
  - if no rawidb file (build/IDB) is generated then set LIDBJOINFLAGS to **-m**

Makefile Structure for the build/Makefile

- Use the default Makefile
- You optionally define the macros:
  - **ALLIMAGES** the list of products to make, defaults to all products listed in $SPEC
  - **ALPHA** the lower six digits of the version, defaults to YDDDHH, the time of the build, where
    - Y is last digit of the year
    - DDD is julian date
    - HH is hour in military time
- You define the macros (defined for you if you use the default Makefile shown on the next page):
  - **RAWIDB** where to put the generated idb (build/IDB is the strongly preferred place)
  - **SRCIDB** where to find the checked in idb (build/idb is the strongly preferred place)
  - **SPEC** location of the checked in spec file (build/spec is the strongly preferred place)

Makefile Structure for the build/Makefile (continued)

- You are provided the targets:
  - **buildimages** invokes checkalpha and finalidb and then create images in images directory
  - **buildism** invokes checkalpha and finalidb and then creates external spec file and idb file in idbs directory (typically only used by build group in creating a release)
  - **version** documents the characteristics of the build (in $WORKAREA/**ISM_NAME**.version)
    - the RCS versions of each file in the ism
    - date
    - versions of installed isms
  - **finalidb** merges generated idb ($RAWIDB) with checked in idb ($SRCIDB)
  - **checkalpha** makes sure macro ALPHA is a six digit number

Suggested Makefile for the build/Makefile

```
#
# Makefile for foo/build to build the foo isms
#
# IF YOU USE spec.proto as your spec file THEN:
# Three products are built:
# foo_root - header files and libraries
# foo_toolroot - tools needed for development with this ism
# foo_eoe - files to ship to customers
#
#
# otherwise your Makefile install lines imply a default spec/idb structure
# "$Revision: 1.1 $"

include $(ROOT)/usr/include/make/ismcommondefs
ISM_NAME=foo
RAWIDB=IDB
SRCIDB=idb
SPEC=spec
SRC=`pwd`/..

default install:

#
# The only real rules are defined in ismcommonrules, currently:
# buildism and buildimages
#

include $(ISMCOMMONRULES)
```

mandefs and manrules

- used just in man directories that are leaf directories to produce man pages and print and view them.
- run **handbook buildman** for more info
- parent man Makefile discussion
  - just regular parent Makefile using SUBDIRS macro (see page 13)
  - Suggested Makefile for the parent man Makefile (page 28)
- leaf man directory Makefile discussion
  - Macros Provided for a man leaf Makefile (page 29)
  - Suffix Rules Provided for a man leaf Makefile (page 30)
  - Makefile Structure for a man leaf Makefile (page 31)
  - Sample Makefile for a man leaf Makefile (page 32)

Suggested Makefile for the parent man Makefile

```
#
# Makefile for foo/man
#
# $Revision: 1.1 $

include $(ROOT)/usr/include/make/commondefs
COMMONPREF=foo
NOSUBMESG=yes

SUBDIRS=man1 man1m man2 man3 man3c man3w man3x man4 man5 man7 man7m

default install $(COMMONTARGS): $(_FORCE)
     $(SUBDIRS_MAKERULE)

include $(COMMONRULES)

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

Macros Provided for a man leaf Makefile

- **GHOSTVIEW** name of ghostview program (default is ghostview)
- **GHOSTVIEWOPTS** options passed to $(GHOSTVIEW) (default is -)
- **XPSVIEW** name of xpsview program (default is xpsview)
- **XPSVIEWOPTS** options passed to $(XPSVIEW) (default is -wp -skipc -ps 8.5 9.75 -geom 649x768 -)

Suffix Rules Provided for a man leaf Makefile

- suffix rules perform most desired man page actions
  - **<name.z** created online packed version of man page
  - **<name>.p** print man page using default printer
  - **<name>.ps** create PostScript version of man page
  - **<name>.xp** view page with $(XPSVIEW)
  - **<name>.gv** view page with $(GHOSTVIEW)

Makefile Structure for a man leaf Makefile

- commonrules are not referenced by manrules
- You optionally define the macros:
  - **IDB_TAG** the idb tag to associate with all man pages in this directory (no default)
  - **LANGOPT** use if language specific man pages
  - **OPTHEADER** optional header string, usually not set
  - **OPTFOOTER** optional footer string, usually not set
  - **ODDEVEN** controls type of headers and footers for printed man pages, defaults to same for both sides.
  - **PAGENUM** used to control page numbers for printed man pages
- You define the macro:
  - **IDB_PATH** the directory where inst should place the man pages
- You are provided the targets:
  - **defaul** which produces the .z files for all manpages in the directory
  - **clean** which removes all man page targets, leaving just the source man pages
  - **clobber** same as clean
  - **rmtargets** does nothing (just present for compatibility)
  - **install** install what you have built
  - plus the targets provided by the suffix rules

Sample Makefile for a man leaf Makefile

```
# $Revision: 1.1 $

# destination for man pages of this directory
IDB_PATH=/usr/catman/u_man/cat1

include $(ROOT)/usr/include/make/mandefs
include $(ROOT)/usr/include/make/manrules
```

relnotesrules

- used just in relnote directory to manage chapters, appendices, cover page, credits, list of figures and list of tables.
- run **handbook buildman** for more info
- relnote Makefile discussion
  - Macros Provided for a man leaf Makefile (page 29)
  - Suffix Rules Provided (page 35)
  - Makefile Structure (page 36)
  - Sample Makefile (page 37)

Macros Provided

- **GHOSTVIEW** name of ghostview program (default is ghostview)
- **GHOSTVIEWOPTS** options passed to $(GHOSTVIEW) (default is -)
- **XPSVIEW** name of xpsview program (default is xpsview)
- **XPSVIEWOPTS** options passed to $(XPSVIEW) (default is -wp -skipc -ps 8.5 9.75 -geom 649x768 -)
- **IDB_TAG** (default is $(RELPROD).man.relnotes)
- **IDB_PATH** (default is /usr/relnotes/$(RELPROD))
- **OUT** names of PostScript files of chapter and appendices (default is derived from ".cmm and *.amm files of the current directory)
- **BOOK** order of files associated with book target (default is front.ps, contents.ps and $OUT)

Suffix Rules Provided

Table 1: Suffix Rules available for each file type

	.xp	.gv	.p	.ps	.z
front.x	X	X	X	X	
contents	X	X	X	X	
ch<N>.cmm	X	X	X	X	X
ap<N>.amm	X	X	X	X	X
index	X	X	X	X	
book		X	X	X	
errata	X	X	X	X	

where

.xp invokes the $(XPSVIEW) viewer on the file
.gv invokes the $(GHOSTVIEW) viewer on the file
.p prints the file
.ps generates a PostScript form of the file
.z generates a compressed online form of the file

- The files **contents** and **index** are generated by invoking the target of the same name.
- book.ps generates the individual files that comprise the book, rather than a single file (front.ps, contents.ps, ch**<n>**.ps and ap**<n>**.ps).

Makefile Structure

- you define the files
  - **front.x** the cover sheet
  - **ch<n>.cmm** the chapters (n is numbered sequentially from 1)
  - **ap<n>.amm** the appendices (n is numbered sequentially from 1)
- you define the macros:
  - **CMM** the list of the files corresponding to each chapter. By convention, the chapters are of the form ch<n>.cmm, numbered sequentially from 1.
  - **AMM** the list of the files corresponding to each appendix. By convention, the appendices are of the form ap<n>.amm, numbered sequentially from 1.
  - **RELPROD** a short installation name for your product
- you are provided with targets
  - provided via the suffix rules
  - **default** format online version of release notes (*.cmm & *.amm & table of contents)
  - **install** install online version of release notes
  - **relnotes** display online release notes using more
  - **check** if any release notes have words in angle brackets
  - **clean** remove all generated files except *.ps and *.z
  - **clobber** remove all generated files

Sample Makefile

```
# ident $Revision: 1.16 $
#
# Makefile for release notes.
# See $(ROOT)/usr/lib/doc/doc/make_targets for a list of available make
# targets.
#

#    Edit the CMM and AMM macros to contain the list of chapters and
#    appendices in this book. Do not use abbreviations (i.e. ch[1-8].cmm).
#    The sort order of the file names must be the same as the chapter
#    order, so if there are more than nine chapters, name them ch01.cmm, etc.
CMM  = ch1.cmm ch2.cmm ch3.cmm ch4.cmm ch5.cmm ch6.cmm
AMM  =

#    Set RELPROD to the short installation name for your product --
#    your release notes subsystem will be called RELPROD.man.relnotes
RELPROD=

#    If you are using this directory in an ISM, then you may want the
#    idb tag to be a short uppercase name. e.g. RELN. If you would like to
#    do so, then just uncomment the next line (and make sure you refer to this
#    in the appropriate exp line of your spec file:
# IDB_TAG= RELN

include $(ROOT)/usr/include/make/relnotesrules
```

Future Additions

- Recommendations on how to write Makefile include files that extend these basic ones
- description of kcommondefs and kcommonrules
- description of librootdefs, librootrules, libleafdefs and libleafrules

Engineer's Handbook
Makefile Type Stuff

May 5, 1994

ISM Class notes
(Independent Software Module)

access via: handbook ism

probable data location: dist.wpd:/sgi/doc/swdev/ism.ps

Silicon Graphics Inc

Independent Software Modules (ISMs)

NOTES

Publication Date: 4/29/93

Agenda

- Introduction
  - What is an ISM? (page 2)
  - What is not an ISM? (page 3)
- ISM details (page 4)
- How do I convert to an ISM? (page 15)
- Appendices
  - Example ISM - the man ISM (page A-1)
  - files produced by ism_setup (page B-1)

What is an ISM?

- A concept - Independent Software Module

  Simplify development/build/release by introducing software source structure midway between a source tree and a file. Have this structure correspond to a self contained project.
- Standardized interchange/dependence among developers
  - Functional interface (via header files and libraries)
  - Avoid direct knowledge/manipulation of data structures
  - Standardized way to specify needed build time components for other ISMs ROOT and TOOLROOT
- Standardized handoff to build and release group

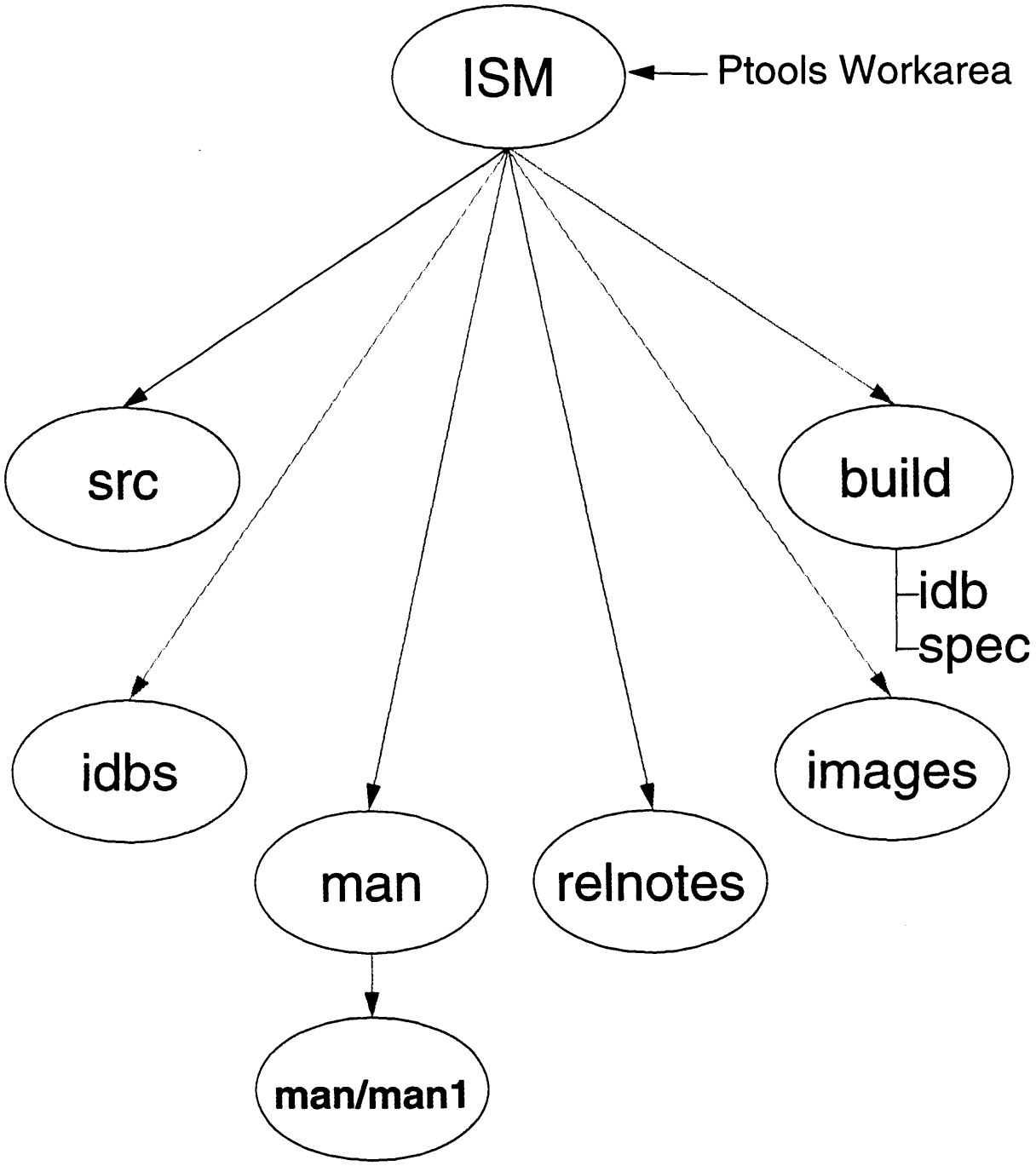
- YIELDS - movement from big bang integration to something more manageable

What is not an ISM?

- elf (replacing coff)
- Dynamic Shared Objects (DSO) (replacing shared libraries)
- MIPS ABI
- source tree reorganization
  - bonnie:/jake (instead of bonnie:/jake/att/usr/src)
  - thus shorter path names
  - sherwood will soon be cloned to jake:/jake/sherwood
  - bonnie:/depot and bonnie:/proj organizations
- Makefile changes
  - xansi C compiler (ansi + extra SGI and SVR4 system calls)
  - assembler changes (supporting kpic)
  - HOST_CC and HOST_C++
  - automatic makedepend

ISM details

What is the ISM file/directory structure?



What do I have to do for ROOTs and TOOLROOTs?

- If another internal SGI developer needs header files or libraries from you in order to build
  - put these in a <ISM_name>_root product
  - have the target ***headers***, INSTALL header files
  - have the target ***exports***, INSTALL libraries
- If another internal SGI developer needs executables from you in order to build
  - put these in a <ISM_name>_toolroot product
- Obtain your ROOT
  - from /
  - from a Public ROOT
  - construct it yourself from _root products
- Obtain your TOOLROOT
  - from /
  - from a Public TOOLROOT
  - construct it yourself from _toolroot products

How do I do the handoff to the build group?

- strongly suggested that you provide a checked in spec file
  - especially if you produce a custom product
  - provides the definition of your products structure
- strongly suggest you provide a checked in idb file
  - provides an inventory of all files in your product
  - provides easy mechanism for grouping files for product structuring with the spec file
- Provide additional Makefile targets in the main Makefile

  NOTE - use of ismcommon{defs,rules} and the template for this Makefile simplifies this for you
  - ***headers***, to add your build time header files and libraries to the build's ROOT
  - ***exports***, to add your build time tools to the build's TOOLROOT
  - ***rawidb***, to set the envariable RAWIDB and invoke the target ***install*** in all the source directories
  - ***images***, to produce inst'able images of your product
  - ***ism***, to produce external idbs and spec files used to do overall build
- Use standard build/Makefile to direct the building of your product

What normally happens

```
+cd $WORKAREA/ism
+p_tupdate
+make headers
```

- puts headers in $ROOT
- if INSTOPTS=-t,
  then it makes links to $WORKAREA instead of actual copies

```
+make exports
```

- creates libs/bins needed for build in $ROOT

```
+make rawidb (many times)
```

- creates a file $WORKAREA/ism/build/IDB

```
+cd $WORKAREA/ism/build
+make finalidb
```

- merges IDB with idb and results in any of the following files:
  - extra -> new files, change of structure
  - missing -> build broke, change of structure
  - finalidb -> everything is perfect

If there are extra files

```
+cd $WORKAREA/ism/build
+vi extra
```

- remove mach, postop, symval, exitop....tags
- add your ALLCAPS tags
- write the file

```
+p_modify idb
+vi idb
```

`<shift>g`	go to end
`:r extra`	append extra
`:.,$!sort +4 -6 -u`	from line after comment to end, sort
`:wq`	write file

```
+make finalidb
```

Tricks

- To see what your idb lines look like when you add the idb tag in Makefile

```
+cd $WORKAREA/ism/dir_you_want_to_test
+make SRC=$WORKAREA/ism RAWIDB=/tmp/foo.idb install
+vi /tmp/foo.idb
```

- How ALPHA is set when you use default setting

```
ALPHA=1007`date +%y%j%H | sed -e 's/.//'`
```

  - year, day of year, hour (00-23)
- Avoid traversals of the tree
  - critical for build group
  - irrelevant for small ISM
  - try to cd to directory and do patch work there

How do I convert to an ISM?

- Prerequisites
  - make sure you are running sherwood on your workstation
  - get the ismtools (page 16)
- Set it up
  - create an ISM template (page 17)
  - setup your source directories (page 18)
  - setup your man pages (page 19)
  - setup your relnotes (page 20)
- do your other sherwood changes and development
- Structure your product
  - setup your spec file (page 21)
  - setup your idb file (page 22)
  - create images (page 23)

get the ismtools

```
% su
# inst -a -f dist.wpd:/sgi/ismtools
# exit
% rehash
```

create an ISM template

- using ism_setup command
  - creates needed directories and basic files
- using p_setup (or ism_workarea)
  - creates .workarea file
  - set census file format to 2 (i.e. add this line to .workarea)
    ```
    census_db.output_format : 2
    ```
  - consider using newdirorg to set environment variables
    ROOT to /
    TOOLROOT to /
    WORKAREA to your ISM
  - see 'handbook ptools' for more info
- see manual pages for ismtools and ism_setup for more info

setup your source directories

- copy your source directories to your ISM workarea
- remove the template directory, src, if you are not using it
- edit the SUBDIRS macro in the main Makefile
  - change reference to src to be your source directories
- Done if your source directories are SGI Makefiles, i.e
  - respond correctly to <default>, install and clobber targets
  - see 'handbook makeconv' for more info

setup your man pages

- copy your man pages to appropriate directory
  (e.g. man/man1)
- man/Makefile and man/man<n>/Makefile use man{defs,rules}
  - means man pages become very low maintenance
  - see example on page A-9, page A-10, and page A-11
- see 'handbook buildman' for more info

create ism

- from top of ISM

```
% make ism
```

- this is what build group will invoke to create the files needed to generate their release, namely:
  - idbs/*.idb (the release idb is concatenation of these from all ISMs

Engineer's Handbook
Makefile Type Stuff

May 5, 1994

Common Makefile include files
Class notes

access via: handbook commoninclude

probable data location: dist.wpd:/sgi/doc/swdev/commonrules.ps

Engineer's Guide to Packaging Software for `inst`

Susan Ellis

Version 2.1
January 1992

Contents

List of Figures

Chapter 1

Introduction

This guide explains the task of packaging a software product for `inst`(1M), Silicon Graphics' tool for installing software. It includes step-by-step procedures and examples of a software package called *rfind* at each step, a comprehensive glossary that defines all the terms you need to know, a troubleshooting appendix that addresses common mistakes and problems that may arise during the packaging process, and an index.

1.1 Audience

If you are an engineer in Engineering, Marketing, or Education and you want to package your software for installation by `inst`, this guide is for you. The software can be Silicon Graphics system software, software option products, software being sent to customers outside the software release process, or software that is intended to be shared internally.

NOTE

The material in this document accommodates products based on 4.0 or later. If your product is based on 3.3, some information described in Section 3.5 will be slightly different: the names of subsystem flags in spec files, the ID string conventions for products, images, and subsystems, and the `oldvers` keyword is new. Pre-4.0 products used a different mechanism for generating maintenance releases (Chapter 4).

1.2 Scope

This guide covers only those topics that are specific to packaging your software product for `inst`. It includes the step-by-step procedures for starting with a source tree or a binary tree and creating, verifying, and testing software products that can be installed by `inst`.

This guide assumes that you have chosen a Marketing name for your product and have a pretty good idea what's in it.

This guide does not describe how to create Makefiles or how to install your product.

For information on how to create Makefiles, see the *Silicon Graphics Makefile Conventions* document. For installation information, see the *IRIS Software Installation Guide.*

1.3 Organization

First, to give you a feel for how a product gets packaged (this is what you need to do to put your software in a form that enables it to be installed by `inst`), an overview is presented in Chapter 2, "Product Packaging Overview." It also introduces some special terminology that you need to understand in order to follow the steps in this document.

Chapter 3, "Creating a Software Product Release," describes the actual steps you need to take to package your software in preparation for `inst`.

Chapter 4, "Creating a Maintenance Release," describes the packaging process for maintenance releases.

Appendix A, "Glossary," includes a comprehensive list of terms and definitions used in this guide.

Appendix B, "Troubleshooting," helps you solve problems you may encounter along the way. The most common mistakes and problems are addressed.

Appendix C, "Conventions for ID Strings in Spec Files," is a reference that defines the content and syntax of ID strings in spec files.

Appendix D, "The idb Language," is a reference that describes the language used to specify the files that go into a subsystem.

Appendix E, "The idb File," is a reference that defines the contents of a generated idb (installation database) file.

There is a complete index at the end of this guide.

If your software has never been packaged for `inst`, you'll need to start at Chapter 2 to learn some terminology and understand the process, then the packaging

procedure in Chapter 3.

If your software has been packaged before, but has never been packaged as a maintenance release, you should probably read Chapter 2 to become familiar with the terminology and process, then go to Chapter 4. Section 3.8 will tell you how to verify your maintenance release and then you're done.

If your task is to *repackage* a product (significantly change the content and/or names of subsystems), you should also start with Chapter 2, but then skip to Section 3.5 and go on from there since most of the work you will be doing will involve making changes to the spec file.

1.4 Required Tools and Disk Space

In order to package software for `inst`, you'll need some or all of these packaging and installation tools: `gendist`, `idbdiff`, `idbedit`, `idbscan`, `inst`, `install`, `instid`, `showprod`, and `versions`. The subsystems noship.misc.distgen and noship.misc.idbtools contain many of these tools. They require about 2 1/2 Mb of disk space. Other tools you need are included in eoe1.sw.unix and always installed on your workstation. The manual pages for these programs are in the noship subsystems with the software and in eoe1.man.unix.

In order to package software, you must have enough disk space for your (uncompiled) source tree and twice the additional disk space required to build your software. If you want to install the software distribution on your workstation, you'll need about three times as much disk space as it takes to build the software rather than twice as much: space for the binaries in the source tree, space for the binaries in the packaged software distribution, and space for the installed binaries.

1.5 Conventions

You will find the following font treatments throughout this guide.

- *italic*
  Used in two distinct ways:
  - to indicate a variable that you would substitute with an actual value.
  - to indicate a "first time usage" of a new term, which is then followed immediately by a brief definition.
- `constant width`
  Used to indicate the names of commands, command arguments, examples of commands, and actual text that displays on your computer screen.

  Constant width font is also used for file names and their contents.

Lines in some examples and commands have been wrapped because they don't fit on a single line. In these cases the continuation line is indented.

1.6 About the Examples

The examples in this document illustrate the packaging of a tool called `rfind`. This tool is used as a faster equivalent to the `find`(1) command for searching source trees. It has two parts: a server running on a source machine and a client which runs on users' workstations. The server monitors the file system at periodic intervals and keeps a database of information about files that is used to respond to queries. Clients communicate requests to the server.

Chapter 2

Product Packaging Overview

This chapter defines some terminology that is very specific to packaging and installing software at Silicon Graphics and presents an overview of the process of packaging a software release for `inst`. Additional terms are defined in the Glossary in Appendix A.

In order to make it easy for you to look up the definitions of the terms presented in this chapter, each term is defined immediately after its first introduction with the term in bold to the left of the definition.

2.1 Basic Terminology

Let's begin the definition of terms by talking about the goal of packaging software for `inst`. The goal is to start with a *source tree* for one or more *products* and generate a *software distribution* from it.

source tree
: Usually, a *source tree* contains all of the source code for your product. By convention at Silicon Graphics, when you give the command `make install` at the highest level (root) directory in the source tree, the `make install` command is propagated throughout the tree and builds all of the binaries and other files which are created in that tree. In some cases, the term source tree as used in this guide refers just to the hierarchy of created files, in these cases it doesn't matter whether or not the source files are present. (This case is known as "using `idbgen`" and is defined later.) The Makefile conventions for source trees are listed in the document *Silicon Graphics Makefile Conventions*.

product
: A product is the software that customers receive when they order a software option like Fortran from the Silicon Graphics Price Book. It is explained in more detail below and in Appendix A.

software distribution
: A software distribution is the software for one or more products after they've been packaged. It is a proprietary format and contains information about what's in the distribution and how the files in the product should be installed. `inst` and `distcp`(1M) are the only tools that understand this format. Software distributions are copied onto tapes and CD-ROM discs and shipped to customers, and can also be stored on

disk.

Assuming that you've used `inst`, we can easily relate these terms to things you know and introduce some other important terminology.

Examples of source trees in use at Silicon Graphics as of this writing are `bonnie:/jake/att/usr/src` and `bonnie:/cypress/att/usr/src`. The source tree for *rfind* is located (as of this writing) in `bonnie:/jake/att/usr/src/cmd/rfind`.

One software distribution in current use at Silicon Graphics is stored in `bacall:/4D/test/cypress`. Most of the files in that directory are *product descriptions, idb files,* or *images.*

product description
: Product description files have the name *"product"*, where *product* is a short name for your product. It is a small file that contains product and installation information in a binary format.

idb file
: idb (installation database) files have the name *"product*.idb". They contain installation information about every file in your product and are in an ascii format.

image
: Images contain the files in your product that will be installed on users' systems. Each image is usually a subset of the files in a product, although a product can have just one image. Image file are named *"product.image"*. Common values of *image* are "sw" and "man".

After a software distribution is installed on a workstation, you can use the command `versions -a` to get another view of the software distribution you just installed and possibly others as well. Figure 2-1 is an example of `versions -an` output for the *rfind* product after two of its three subsystems have been installed.

```
I = Installed, R = Removed

   Name                 Version   Description

I  rfind                681523105  rfind Remote Fast Find 1.0
I  rfind.man           1006000106  rfind Man Pages
I  rfind.man.rfind     1006000106  rfind Man Pages
I  rfind.sw            1006000106  rfind Software
I  rfind.sw.client     1006000106  rfind Client Command
   rfind.sw.server     1006000106  rfind Server Support (Only)
```

Figure 2-1. `versions -an rfind` Listing

This `versions`(1M) listing shows the names of software products, images, and *subsystems* with version numbers, installation status and a description for each line.

subsystem
: The files in a product are divided into images and further divided into subsystems. The term subsystem is used to describe one of these collections of files, such as "rfind.sw.client" and is also the name of the third part of the three part name (i.e. *product.image.subsystem* where the *subsystem* in this case is "client").

Using `rfind.sw.client`, the parts of each name tell you the following:

`rfind`
: is the name of the product. All items that share this part of the name are considered to be one product for packaging purposes.

`rfind.sw`
: is the name of an *image*—in this case, software. All items with two-part names are images.

`rfind.sw.client`
: is the name of a *subsystem*—in this case, client software. Users install subsystems (groups of files) rather than individual files.

There are two types of software releases: *software product releases* (sometimes called base releases) and *maintenance releases*.

software product release
: A software product release is a complete set of files for a product. Prior to installing it, `inst` automatically removes previously installed versions, if any.

maintenance release
: A maintenance release contains only files that include bug fixes, new features, or support for new hardware. When you install a maintenance release, the files in the maintenance release overwrite existing versions of those files. No previously installed files are removed.

A maintenance release can include files from many products, but is packaged as one or sometimes two products. Product names of a maintenance releases typically are "`maint`" followed by a digit. This digit has no inherent meaning. Image names are created by taking the original product names and image names and joining them with an underscore (_) rather than a period. Subsystem names remain the same. For example, the subsystem `eoe1.sw.unix` in a maintenance release is named `maint1.eoe1_sw.unix`.

Maintenance releases are for a particular base release. They are usually cumulative which means that all fixes in the previous maintenance release for a base release are included in the current maintenance release.

There are two slightly different methods that can be followed to package software for `inst`. The first procedure that we'll present is by far the most commonly used. In this procedures you put installation specification information into the Makefiles in your source tree. This information is given as an argument to the `-idb` option of the `install`(1M) command. Thus, the first method is known as "using

`install -idb`".

The second way to package software for `inst` is used in cases where you are creating software distributions of files that you do not build, do not have Makefiles for, or for some reason you do not want to modify the Makefiles used to build the software. The Education group and Move Over Mac folks use this method to create custom software distributions for their users. It is used with third party software released by Silicon Graphics in some cases as well. This method is known as "using `idbgen`".

Section 2.2 gives an overview of the packaging procedure when using `install -idb`. Section 2.3 presents overview of the procedure for using `idbgen`. It focuses on the differences between using `install -idb` and using `idbgen` so if you are thinking of using `idbgen`, you should read the section on `install -idb` first.

2.2 Using `install -idb` for Packaging

A dataflow diagram of the packaging process when using `install -idb` is shown in Figure 2-2. The solid boxes represent steps in the process and the dashed boxes are data items: your original source tree and other trees or files that get created during the process. Each step is noted at the upper left of the box and explained in Chapter 3. In the discussion below, some of the finer detail of the process is intentionally left out in order to make it easier to grasp the big picture.

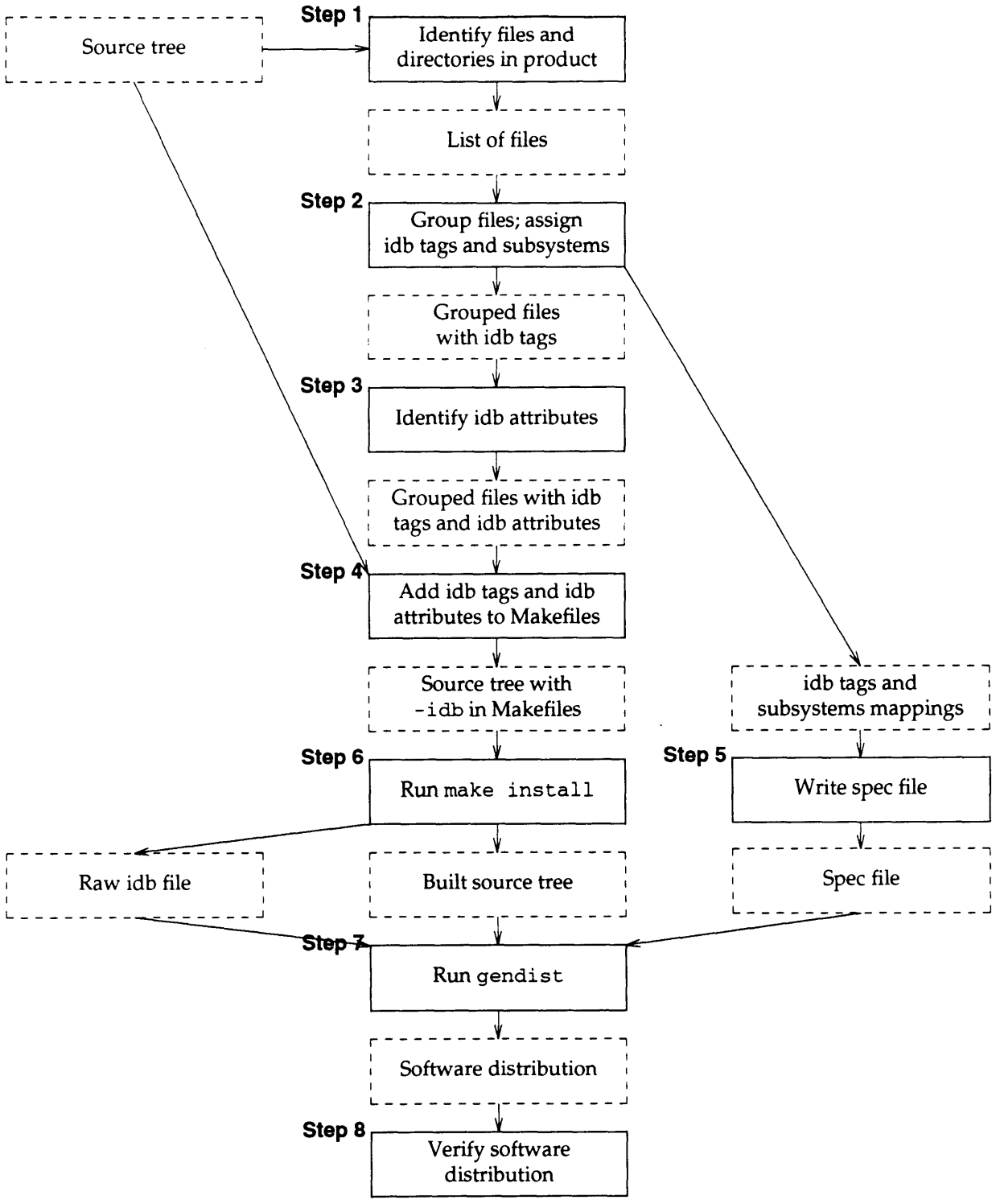


Figure 2-2. Data Flow Diagram of Packaging using `install -idb`

Step 1 of the packaging process is to identify the files that you want to include in your product (Section 3.1). Every file that should appear on a user's disk as a result of installing your product should be included in this list. Looking at Makefiles in the source tree to see what they build is one way of identifying all of the files.

Step 2 (Section 3.2) is to group the files into logical groups. These groups are given *idb tag* and product.image.subsystem names in this step, too.

idb tag An idb tag is a name given to a group of files that is a subsystem or a subset of a subsystem. It is one of the arguments to the `install -idb` option.

Next, the *idb attributes*, if any, of the files in the product are identified (Section 3.3).

idb attribute
Idb attributes describe special processing that is required during installation. They mark machine-specific versions of files, files that are not shared by diskless clients, files that are treated specially because they are likely to be modified by users (configuration files), and other special classes of files.

After the idb tags and idb attributes are determined for all of the files in the product, they are added to the source tree in the form of `install -idb` arguments in Makefiles (Section 3.4).

The list of idb tags and product.image.subsystem names created in Step 2 is used when writing a *spec file* (Step 5, Section 3.5).

spec file A spec file is a text file, created by hand, that specifies the product, image, and subsystem hierarchy for your product and gives installation information for it. Examples of the type of information in a spec file are the descriptions given in a `versions` listing, the relative order in which images should be installed, and what older subsystems should be removed when specific subsystems are installed. None of this information is for specific files, it applies to subsystems, images, and products only.

At this point in the process, all of the "manual" steps of defining a software product are done. They are typically done once in a product's lifetime, although they may require bug fixing and enhancement later. The `-idb` arguments and spec file will change only for the same kinds of reasons that other parts of the source tree change.

Steps 6 through 8 of the procedure, on the other hand, will be repeated many times, each time you want to create a new software distribution, in fact. Collectively, these steps are known as creating an *alpha release* (Sections 3.6 through 3.8).

alpha release
An alpha release is an installable snapshot of your product. It's a version of your product that might be released or just used for testing purposes while working on a new software release.

The first step in creating an alpha release is to give the command `make install` at the top of your source tree with various environment variables set properly (Section 3.6). This builds all of the files in your product and creates a file called a *raw idb file*.

raw idb file

The content of a **raw** idb file is very similar to the content of an idb file (defined above), but it is a different file. It is created by `install` and used as input to `gendist`(1M). It contains idb tags, idb attributes, mode, owner and group for every file and directory in the product.

At this point you've created all of the things required by `gendist`, the tool that creates software distributions. You can run `gendist` and create your alpha release software distribution (Step 7).

Step 8, the final step in the software distribution creation process, is to verify your distribution. This involves some checks to make sure that all of the files in your product made it into the distribution and installing the software to make sure that all of the installation details (idb attributes, replacement rules, etc.) are correct.

2.3 Using `idbgen` for Packaging

The procedure for packaging software using `idbgen` uses the same eight step model as the procedure for packaging software using `install -idb`. Under some circumstances, you'll have no choice but to use this packaging method, but it is not the preferred method because it has less flexibility than the `install -idb` method and it often requires work for each alpha release you create that would be automated if you were using `install -idb`.

A dataflow diagram of the packaging process when using `idbgen` is shown in Figure 2-3 and the remainder of this section discusses the differences when using `idbgen`.

Figure 2-3. Data Flow Diagram of Packaging using `idbgen`

In Step 1, you start with an already compiled source tree or a tree that mimics the locations of the files in your product after they are installed. Ideally, this tree has been created using `install` commands that have set the permissions, owners, and groups of the files in your product. Using `idbgen`, you create a list of files with mode, owner, group, and location information that is similar to the raw idb file (Section 3.1.2).

In Steps 2 and 3 you define subsystem names and idb attributes for your files as when using `install -idb`, then add them to the list of files created in Step 1 to form a complete raw idb file.

Step 4 is omitted when using `idbgen` since your raw idb file is already complete.

Step 6, running `make install` is recommended if you need to populate your tree with the latest binaries, but is not required.

Writing a spec file (Step 5), creating a software distribution with `gendist` (Step 7) and verifying it (Step 8) have the same tasks no matter which packaging method you choose.

Chapter 3

Creating a Software Product Release

This chapter gives complete directions for packaging your software into a software product release (base release) type of software distribution. The procedure is broken into eight major steps. These steps are shown in Figures 2-2 and 2-3. Each of these steps contains several tasks.

This guide assumes that you are following this convention in the document *Silicon Graphics Makefile Conventions*: a target called "install" in your Makefiles has a dependency on all of the binaries and other files created in that directory and has rules that invoke `$(INSTALL)` (`install`(1)) to "install" the files that are released.

3.1 Step 1. List the Files in the Product

In this step, the task is to identify all the files that you want to include in your finished product. Section 3.1.1 describes the procedure if you are using `install -idb` to generate a raw idb file and Section 3.1.2 describes the procedure if you are using `idbgen` to generate it.

3.1.1 Identifying the Files in Your Product (`install -idb`)

After your source tree has been built using `make`, all of the files that will be in your product should be in the source tree in the directories where they were built. To complete this step, you need to perform these tasks:

1. Begin to make a list of the compiled files in your product. One way to do this is to use `find` on a built source tree. This will create a list of all files in the tree.

   ```
   cd sourcedir
   find * .??* -print > filelist
   ```

   where

   sourcedir is the top of the source tree for your product.

   filelist is the name of a temporary file.

2. Prune the list by editing *filelist* and taking out directories, `.c, .h, and .o` files, `.a` files (if they aren't part of your product), Makefiles, and other files that

aren't part of the finished product.

Directories and symbolic links that are part of your product will be added later. Hard links cannot be included in your product.

The directory portion of each pathname isn't particularly important, so if you want to create the list another way and include just the last component of the pathname, that's fine. If file names do include the directory where they are created, don't edit them out because they will be useful later. Figure 3-1 below shows *filelist* after pruning for *rfind*, our example product.

```
cmd/rfind/fsdump/crontab
cmd/rfind/fsdump/fsdump
cmd/rfind/config/rfindd
cmd/rfind/init.d/rfindd
cmd/rfind/lib/librfind.a
cmd/rfind/man/fsdump.1m
cmd/rfind/man/rfind.1
cmd/rfind/man/rfindd.1m
cmd/rfind/rfind/rfind.aliases
cmd/rfind/rfind/rfind
cmd/rfind/rfindd/forward
cmd/rfind/rfindd/passwd.add
cmd/rfind/rfindd/rfindd
cmd/rfind/rfindd/README
cmd/rfind/rotatelogs/rotatelogs
```

Figure 3-1. Files in the *rfind* Product

3. Look over your list of files and make sure all of the files in your product are present. Don't forget manual pages, online help files, Release Notes, data files and anything else included in your product. Once again, it is the *file* component of the pathname that is important rather than the exact directory where it is built.

3.1.2 Identifying the Files in Your Product (`idbgen`)

If you manually create idb information, you will use `idbgen` to create the list of files in your product.

1. First, identify a directory hierarchy that contains the the files in your product. The best choice for the directory hierarchy is a destination tree for your product after a `make install` has been run. This tree can be rooted anywhere, and is easiest to work with if it contains only files and directories in your product. We'll call the root *sourcedir* below and it will contain `etc`, `usr`, `usr/lib`, and other directories.

   The permissions, owner, and group of each file should be set to what they should be after the files are installed on a user's workstation. The easiest way to do this is to have `make install` install the files with the correct permissions, owner, and group, but if this is not practical, the modes, owners, and groups can

be indicated in task 5 below.

2. Symbolic links in your product should be included in your tree. Hard links are not permitted in your product.

3. Change to the root of your directory hierarchy (*sourcedir*).

```
cd sourcedir
```

4. Create a zero-length temporary file, *idb1*, in a directory that is not in your directory hierarchy.

```
cat > idb1
^D
```

5. Create a list of files and directories in your directory hierarchy with this command:

```
find * .??* -print idb1 > idb2
```

6. Sort the list of files in your directory hierarchy, pipe the output to `idbgen`, sort it again, and process it with `nawk`. The intent of the processing is to replace every occurrence of the word "unknown" with a copy of the pathname that precedes it. The output is put in the file *idb3*.

```
sort idb2 | idbgen -i idb1 -r sourcedir | sort +4 | \
nawk 'if ($6 == "unknown") print $1,$2,$3,$4,$5,$5; \
else print $0}' > idb3
```

`idbgen` runs a stat on all the files to get the appropriate information, then creates an entry for each file. `idbgen` produces six fields, which are as follows:

type mode owner group destination source

destination and *source* will be the same. The contents of *idb3* are shown in Figure 3-2.

```
d 0755 root sys etc -
d 0755 root sys etc/config -
f 0644 root sys etc/config/rfindd etc/config/rfindd
d 0755 root sys etc/init.d -
f 0755 rfindd nuucp etc/init.d/rfindd etc/init.d/rfindd
d 0755 root sys etc/rc0.d -
l 0000 root sys etc/rc0.d/K98rfindd - symval(/etc/init.d/rfindd)
d 0755 root sys etc/rc2.d -
l 0000 root sys etc/rc2.d/S98rfindd - symval(/etc/init.d/rfindd)
d 0755 root sys usr -
d 0755 root sys usr/lib -
f 0644 bin bin usr/lib/rfind.aliases usr/lib/rfind.aliases
d 0755 root sys usr/lib/rfindd -
f 0644 rfindd nuucp usr/lib/rfindd/.forward usr/lib/rfindd/.forward
f 0644 rfindd nuucp usr/lib/rfindd/README usr/lib/rfindd/README
f 4111 root sys usr/lib/rfindd/fsdump usr/lib/rfindd/fsdump
f 0755 bin bin usr/lib/rfindd/passwd.add usr/lib/rfindd/passwd.add
f 0755 rfindd nuucp usr/lib/rfindd/rfindd usr/lib/rfindd/rfindd
f 0755 rfindd nuucp usr/lib/rfindd/rotatelogs usr/lib/rfindd/rotatelogs
d 0755 root sys usr/local -
d 0755 root sys usr/local/bin -
f 0755 bin bin usr/local/bin/rfind usr/local/bin/rfind
d 0755 root sys usr/man -
d 0755 root sys usr/man/u_man -
d 0755 root sys usr/man/u_man/man1 -
f 0644 bin bin usr/man/u_man/man1/fsdump.1m usr/man/u_man/man1/fsdump.1m
f 0644 bin bin usr/man/u_man/man1/rfind.1 usr/man/u_man/man1/rfind.1
f 0644 bin bin usr/man/u_man/man1/rfindd.1m usr/man/u_man/man1/rfindd.1m
d 0755 root sys usr/spool -
d 0755 root sys usr/spool/cron -
d 0755 root sys usr/spool/cron/crontabs -
f 0444 rfindd nuucp usr/spool/cron/crontabs/rfindd usr/spool/cron/crontabs/rfindd
```

Figure 3-2. List of Files Produced by `idbgen` after Pruning

7. Edit *idb3* if you need to to modify the *destination* fields so that they have the full pathname that the file will have after installation. For instance, if all of the files are installed in `/usr/people/tutor`, but your list from the previous step doesn't include that part of the *destination*, add it now. It is not necessary to add a leading / to the pathname.

8. Edit *idb3* if necessary to change any of the other information such as the mode, owner, or group. This step is not necessary if you've used `make install` and put this information in the `install` command lines.

9. Check for missing files in your list (same as task 2 in Section 3.1.1) and add them to your list.

10. Check for already-in-use full pathnames of files (violations of the Golden Rule, defined below). No file in your product can have the same full pathname (*destination*) as any other file in any other product. It doesn't matter whether the contents are the same or different. The idb files for other products (*product*`.idb`) contain the full pathnames of the files in those products, so using `grep` to search the *product*`.idb` files of existing products for duplicates of full pathnames

in your product is the best way to check that your file names are unique at this stage of packaging. Note that the Golden Rule does not apply to directories.

3.2 Step 2. Choose Subsystem Names

In this step, you will separate the files in your list (and directories if you are using `idbgen`) into subsystems and assign them *product*, *image*, and *subsystem* names. These subsystem names will be used as *idb tag* arguments to `install -idb` as well.

Normally, there is a one-to-one mapping between idb tags and subsystem names. The discussion below and examples will cover this case. Section 3.5.2 explains how to create mappings between idb tags and subsystem names that are not one-to-one.

1. Divide the files from the list you created in Step 1 (if you used `idbgen`, use the list in *idb3*) into subsystems using this procedure:

   a. First, divide your list into three parts: Release Notes files, manual pages, and everything else (software).

   b. Take the software list and divide it into to these two parts (if it makes sense for your product): files that every user of your product will use (we'll call this the "basic" list) and files that only some users will use (the "optional" list).

   c. Look at the optional software list (or the whole software list if you didn't divide it into basic and optional), think about the functions that your product provides, and try to divide the list by function. A good example is the eoe2 product which has subsystems for accounting software, CD-ROM software, Graphics Library tools, and others. Think about how the user is going to use your product and try to group files into logical groupings. Other considerations are given in the list of guidelines below. It is OK to move some of the files back to the basic list if they fit better there.

   d. Divide the list of manual pages into groups that match the software groups if possible. For instance, if you have a group of printer software files, put the manual pages that go with that software into its own group.

   Here are some general guidelines that can help you make decisions:

   - Each list should be constructed so that if users need any of the files in the list, they are likely to need most of the files in the list.

   - Try to formulate each list so that there are few, if any, dependencies on other subsystems in your product.

   - Keep your scheme for dividing files into subsystems simple. Remember that usually the only information a user has when installing your product are the Release Notes (maybe), and what they see when using 7inst: the subsystem name, the 30 character description of the subsystem, its size, and if it is "required", "default" or optional. Information about dependencies

and incompatibilities is not immediately available.

- Avoid having a set of files take up an inordinate amount of disk space. Keep in mind that the primary purpose of dividing products into separately installable subsystems is to facilitate conserving the users' disk space. Check your sets of files to see how much disk space they will use and if it is a lot, examine the list to see if it can be broken up based on usage patterns (see next bullet).

- Look at the usefulness of the groups of files. Is each group of files useful by itself? (If not, you should probably not make it a separate group.) Are there sets of files than can never be used together? (If so, the sets should be different groups.)

- DO NOT mix manual page files with software files! This convention makes it easy for users to find manual pages and to install them or not install them as they wish independent of whether or not they install the software.

- Release Notes are always in their own subsystem.

- DO NOT put a file in more than one group! When a product is installed, there is only one copy of each file on the system. One file cannot be shared by subsystems because if you decide to remove one subsystem, the common file is removed, so the remaining subsystem(s) has a hole in it. This is the Golden Rule of packaging software for `inst`. If you violate it, you are likely to cause yourself and your users incredible amounts of grief. Note that this rule applies to files only, not directories.

Figures 3-3 and 3-4 show the list of files and directories from Step 1 after it has been divided into subsystems. Figure 3-3 is the list from using `install -idb` and Figure 3-4 is the list from `idbgen`.

server:
cmd/rfind/fsdump/crontab
cmd/rfind/fsdump/fsdump
cmd/rfind/config/rfindd
cmd/rfind/init.d/rfindd
cmd/rfind/rfindd/forward
cmd/rfind/rfindd/passwd.add
cmd/rfind/rfindd/rfindd
cmd/rfind/rfindd/README
cmd/rfind/rotatelogs/rotatelogs

noship:
cmd/rfind/lib/librfind.a

manpages:
cmd/rfind/man/fsdump.1m
cmd/rfind/man/rfind.1
cmd/rfind/man/rfindd.1m
client:
cmd/rfind/rfind/rfind.aliases
cmd/rfind/rfind/rfind

Figure 3-3. *rfind* Subsystems (`make -idb`)

```
server:
d 0755 root sys etc -
d 0755 root sys etc/config -
f 0644 root sys etc/config/rfindd etc/config/rfindd
d 0755 root sys etc/init.d -
f 0755 rfindd nuucp etc/init.d/rfindd etc/init.d/rfindd
d 0755 root sys etc/rc0.d -
l 0000 root sys etc/rc0.d/K98rfindd - symval(/etc/init.d/rfindd)
d 0755 root sys etc/rc2.d -
l 0000 root sys etc/rc2.d/S98rfindd - symval(/etc/init.d/rfindd)
d 0755 root sys usr/lib/rfindd -
f 0644 rfindd nuucp usr/lib/rfindd/.forward usr/lib/rfindd/.forward
f 0644 rfindd nuucp usr/lib/rfindd/README usr/lib/rfindd/README
f 4111 root sys usr/lib/rfindd/fsdump usr/lib/rfindd/fsdump
f 0755 bin bin usr/lib/rfindd/passwd.add usr/lib/rfindd/passwd.add
f 0755 rfindd nuucp usr/lib/rfindd/rfindd usr/lib/rfindd/rfindd
f 0755 rfindd nuucp usr/lib/rfindd/rotatelogs usr/lib/rfindd/rotatelogs
d 0755 root sys usr/spool -
d 0755 root sys usr/spool/cron -
d 0755 root sys usr/spool/cron/crontabs -
f 0444 rfindd nuucp usr/spool/cron/crontabs/rfindd usr/spool/cron/crontabs/rfindd

manpages:
d 0755 root sys usr/man -
d 0755 root sys usr/man/u_man -
d 0755 root sys usr/man/u_man/man1 -
f 0644 bin bin usr/man/u_man/man1/fsdump.1m usr/man/u_man/man1/fsdump.1m
f 0644 bin bin usr/man/u_man/man1/rfind.1 usr/man/u_man/man1/rfind.1
f 0644 bin bin usr/man/u_man/man1/rfindd.1m usr/man/u_man/man1/rfindd.1m

client:
d 0755 root sys usr -
d 0755 root sys usr/lib -
f 0644 bin bin usr/lib/rfind.aliases usr/lib/rfind.aliases
d 0755 root sys usr/local -
d 0755 root sys usr/local/bin -
f 0755 bin bin usr/local/bin/rfind usr/local/bin/rfind
```

Figure 3-4. *rfind* Subsystems (idbgen)

There are some differences between the two lists that illustrate some important points about these lists:

- The `idbgen` list contains a lot more directories than the `install -idb` list. This is what you get naturally when using `idbgen` and is fine as long as all of the directories have the right modes, owners, and groups. When using `install -idb`, you need not add directories such as `/bin`, `/usr/lib`, and `/etc` to your list, but any directory that is used only by your product or by few other products should be added in Step 4.

- The `install -idb` list in this example includes a category labeled "noship". The intent is to indicate files that built, but not distributed externally. This is implemented in two ways: it can be a subsystem in a different product (called "noship" by convention), or these files can be in subsystems that are distributed, but have an idb attribute (explained in Section 3.3) of `noship` that prevents it

from being included in the software distribution. These unshipped files do not get included when you use `idbgen`.

2. Assign a subsystem name to each group of files.

   After you create groupings, you need to give each group a three part subsystem name (idb tag). The guidelines for *product.image.subsystem* (subsystem) names are:

   - DO NOT begin the name of *subsystem* with a number.
   - Product is a short name (three to five lower case characters is recommended).
   - The image is the middle name of the three-part subsystem name; typically, it is `sw` (for software) or `man` (for manual pages). Other image names are not recommended.
   - A three to six character lower case subsystem name is recommended.
   - The image and subsystem names of Release Notes are "man.relnotes".

   The subsystem names for the files and directories in *rfind* are shown in Figure 3-5.

```
server:     rfind.sw.server
noship:     rfind.sw.internal
manpages:   rfind.man.rfind
client:     rfind.sw.client
```

Figure 3-5. Subsystem Names for *rfind*

3. If you are using `idbgen`, edit the file *idb3* and add the subsystem names you have chosen to the end of each line. Figure 3-6 shows the result for *rfind*.

```
d 0755 root sys etc - rfind.sw.server
d 0755 root sys etc/config - rfind.sw.server
f 0644 root sys etc/config/rfindd etc/config/rfindd rfind.sw.server
d 0755 root sys etc/init.d - rfind.sw.server
f 0755 rfindd nuucp etc/init.d/rfindd etc/init.d/rfindd rfind.sw.server
d 0755 root sys etc/rc0.d - rfind.sw.server
l 0000 root sys etc/rc0.d/K98rfindd - symval(/etc/init.d/rfindd) rfind.sw.server
d 0755 root sys etc/rc2.d - rfind.sw.server
l 0000 bin sys etc/rc2.d/S98rfindd - symval(/etc/init.d/rfindd) rfind.sw.server
d 0755 root sys usr/lib/rfindd - rfind.sw.server
f 0644 rfindd nuucp usr/lib/rfindd/.forward usr/lib/rfindd/.forward rfind.sw.server
f 0644 rfindd nuucp usr/lib/rfindd/README usr/lib/rfindd/README rfind.sw.server
f 4111 root sys usr/lib/rfindd/fsdump usr/lib/rfindd/fsdump rfind.sw.server
f 0755 bin bin usr/lib/rfindd/passwd.add usr/lib/rfindd/passwd.add rfind.sw.server
f 0755 rfindd nuucp usr/lib/rfindd/rfindd usr/lib/rfindd/rfindd rfind.sw.server
f 0755 rfindd nuucp usr/lib/rfindd/rotatelogs usr/lib/rfindd/rotatelogs rfind.sw.server
d 0755 root sys usr/spool - rfind.sw.server
d 0755 root sys usr/spool/cron - rfind.sw.server
d 0755 root sys usr/spool/cron/crontabs - rfind.sw.server
f 0444 rfindd nuucp usr/spool/cron/crontabs/rfindd usr/spool/cron/crontabs/rfindd
       rfind.sw.server
d 0755 root sys usr/man - rfind.man.rfind
d 0755 root sys usr/man/u_man - rfind.man.rfind
d 0755 root sys usr/man/u_man/man1 - rfind.man.rfind
f 0644 bin bin usr/man/u_man/man1/fsdump.1m usr/man/u_man/man1/fsdump.1m rfind.man.rfind
f 0644 bin bin usr/man/u_man/man1/rfind.1 usr/man/u_man/man1/rfind.1 rfind.man.rfind
f 0644 bin bin usr/man/u_man/man1/rfindd.1m usr/man/u_man/man1/rfindd.1m rfind.man.rfind
d 0755 root sys usr - rfind.sw.client
d 0755 root sys usr/lib - rfind.sw.client
f 0644 bin bin usr/lib/rfind.aliases usr/lib/rfind.aliases rfind.sw.client
d 0755 root sys usr/local - rfind.sw.client
d 0755 root sys usr/local/bin - rfind.sw.client
f 0755 bin bin usr/local/bin/rfind usr/local/bin/rfind rfind.sw.client
```

Figure 3-6. Subsystem Names for *rfind* Added to *idb3*

3.3 Step 3. Create idb Attributes for Your Files

Each file in your product has an idb tag (assigned in Step 2) and idb attributes which are assigned in this step. These idb attributes specify operations to be done at installation time, identify configuration files, specify hardware dependencies, and so on. Many files will not need idb attributes at all.

1. First, review the idb attributes below to become familiar them. The idb attributes are:

   - `config`(*argument*)
     Identifies a configuration file. A configuration file is included in the software distribution but is likely to be modified by the user of a workstation to reflect site and machine-specific information. There are three ways that configuration files can be handled, and they are identified by one of the following arguments to the `config` idb attribute: `update`, `noupdate`, and `suggest`. The `config` attribute requires an argument.

- `update`
  The new version of this file needs to be installed. The existing file is saved as *file.O* (old).

- `noupdate`
  If there is a version of this file that is installed already, it is retained. The new file is not installed.

- `suggest`
  The new file has features that a user may want, but are not required. The new file is installed as *file.N* (new), and the user's file stays.

- `mr`
  Identifies a file that should be included in the miniroot.

- `preop`(*shell command*)
  Identifies an operation that is to be performed prior to installing this file.

- `postop`(*shell command*)
  Identifies an operation that is to be performed after installing this file. A configuration file cannot have a `postop`.

- `exitop`(*shell command*)
  Identifies an operation that is to be performed just before exiting from the installation program.

NOTE

When deciding on operations to perform in a `preop`, `postop`, or `exitop`, exercise caution. For example, commands that modify files are not recommended. As another example, if you compress a file, you may run out of disk space when the file is uncompressed.

- `noshare`
  In a diskless installation, identifies files that are duplicated for each client.

- `noship`
  This idb attribute is part of a two-part mechanism that specfiles that the file should not be included in the software distribution. The other part of the mechanism is that the `exp statement in the spec file (see Section 3.5.1) must contain an expression that excludes files with this idb attribute.`

- `nostrip`
  Identifies a file that should not be stripped during software distribution creation. During software distribution creation, by default all binaries are stripped This can be overridden for all files with `gendist -nostrip` or by including the `nostrip` idb attribute for individual files.

- `symval`(*shell command*)
  Identifies a symbolic link. If you are using `idbgen`, it has created lines

with `symval` idb attributes automatically, so you do not need to add this idb attribute manually in this case.

- `mach(CPUBOARD=`*type*`)`
  `mach(GFXBOARD=`*type*`)`
  `mach(SUBGR=`*type*`)`
  `mach` idb attributes specify that the file being installed is machine-specific. All machines have a CPUBOARD type, a GFXBOARD type, and a SUBGR type, but you need not specify all of them. When files are machine-specific, typically several versions of the file are created so that versions of the file for each type of hardware are included in the software distribution.

  In the case where you restrict installation to several types of CPUBOARD, GFXBOARD, or SUBGR, you can specify them as follows:

  `mach(CPUBOARD=`*type1* `CPUBOARD=`*type2*`...)`

  OR

  `mach(GFXBOARD=`*type1* `GFXBOARD=`*type2*`...)`

  The values of CPUBOARD, GFXBOARD, and SUBGR for different groups of machines are specified in the files *usr/src/head/make/*defs* on the source tree. The *IRIS Software Installation Guide* also contains useful information about `mach` values.

  Subgroups are not required for every graphics board; however, when you do require one, you cannot have more than one subgroup per graphics board.

  The combinations of CPUBOARD, GFXBOARD, and SUBGR for all Silicon Graphics workstations as of the writing of this guide are shown in Figure 3-7.

CPUBOARD	GFXBOARD	SUBGR
IP4	CLOVER1	IP4G
IP4	CLOVER2	IP4GT
IP5	CLOVER2	IP5GT
IP5	STAPUFT	IP7GT
IP6	ECLIPSE	ECLIPSE
IP7	CLOVER2	IP7GT
IP7	STAPUFT	IP7GT
IP7	STAPUFT	SKYWR
IP9	CLOVER2	IP9GT
IP9	STAPUFT	IP7GT
IP12	ECLIPSE	ECLIPSE
IP12	LIGHT	LIGHT
R2300	CLOVER1	IP4G
R2300	CLOVER2	IP4GT

Figure 3-7. CPUBOARD, GFXBOARD, and SUBGR Values

2. If you are using `install -idb`, review the list of files in your product from Step 2 and decide file by file if any of your files need idb attributes. Add the idb attributes to the list. Figure 3-8 shows the list for *rfind*.

```
rfind.sw.server:
cmd/rfind/fsdump/crontab config(suggest)
cmd/rfind/fsdump/fsdump
cmd/rfind/config/rfindd
cmd/rfind/init.d/rfindd config(update)
cmd/rfind/rfindd/forward config(noupdate)
cmd/rfind/rfindd/passwd.add exitop("chroot $rbase /bin/sh /usr/lib/rfindd/passwd.add")
cmd/rfind/rfindd/rfindd config(noupdate)
cmd/rfind/rfindd/README config(noupdate)
cmd/rfind/rotatelogs/rotatelogs

rfind.sw.internal:
cmd/rfind/lib/librfind.a

rfind.man.rfind:
cmd/rfind/man/fsdump.1m
cmd/rfind/man/rfind.1
cmd/rfind/man/rfindd.1m

rfind.sw.client:
cmd/rfind/rfind/rfind.aliases config(suggest)
cmd/rfind/rfind/rfind
```

Figure 3-8. idb Attributes for *rfind* (`install -idb`)

3. If you are using `idbgen`, review the files (not directories or symbolic links) in *idb3* and decide file by file if any of your files need idb attributes. Add idb attributes at the end of the line for each file that needs them. Figure 3-9 shows the result for *rfind*.

```
d 0755 root sys etc -  rfind.sw.server
d 0755 root sys etc/config - rfind.sw.server
f 0644 root sys etc/config/rfindd etc/config/rfindd rfind.sw.server config(noupdate)
d 0755 root sys etc/init.d - rfind.sw.server
f 0755 rfindd nuucp etc/init.d/rfindd etc/init.d/rfindd rfind.sw.server config(update)
d 0755 root sys etc/rc0.d - rfind.sw.server
l 0000 root sys etc/rc0.d/K98rfindd - rfind.sw.server symval(/etc/init.d/rfindd)
d 0755 root sys etc/rc2.d - rfind.sw.server
l 0000 root sys etc/rc2.d/S98rfindd - rfind.sw.server symval(/etc/init.d/rfindd)
d 0755 root sys usr - rfind.sw.server
d 0755 root sys usr/lib - rfind.sw.server
f 0644 bin bin usr/lib/rfind.aliases usr/lib/rfind.aliases rfind.sw.client config(noupdate
d 0755 root sys usr/lib/rfindd - rfind.sw.server
f 0644 rfindd nuucp usr/lib/rfindd/.forward usr/lib/rfindd/.forward rfind.sw.server
       config(noupdate)
f 0644 rfindd nuucp usr/lib/rfindd/README usr/lib/rfindd/README rfind.sw.server
f 4111 root sys usr/lib/rfindd/fsdump usr/lib/rfindd/fsdump rfind.sw.server
f 0755 bin bin usr/lib/rfindd/passwd.add usr/lib/rfindd/passwd.add rfind.sw.server
       exitop("chroot $rbase /bin/sh /usr/lib/rfindd/passwd.add")
f 0755 rfindd nuucp usr/lib/rfindd/rfindd usr/lib/rfindd/rfindd rfind.sw.server
f 0755 rfindd nuucp usr/lib/rfindd/rotatelogs usr/lib/rfindd/rotatelogs rfind.sw.server
d 0755 root sys usr/local - rfind.sw.server
d 0755 root sys usr/local/bin - rfind.sw.server
f 0755 bin bin usr/local/bin/rfind usr/local/bin/rfind rfind.sw.client
d 0755 root sys usr/man - rfind.sw.server
d 0755 root sys usr/man/u_man - rfind.sw.server
d 0755 root sys usr/man/u_man/man1 - rfind.sw.server
f 0644 bin bin usr/man/u_man/man1/fsdump.1m usr/man/u_man/man1/fsdump.1m rfind.man.rfind
f 0644 bin bin usr/man/u_man/man1/rfind.1 usr/man/u_man/man1/rfind.1 rfind.man.rfind
f 0644 bin bin usr/man/u_man/man1/rfindd.1m usr/man/u_man/man1/rfindd.1m rfind.man.rfind
d 0755 root sys usr/spool - rfind.sw.server
d 0755 root sys usr/spool/cron - rfind.sw.server
d 0755 root sys usr/spool/cron/crontabs - rfind.sw.server
f 0444 rfindd nuucp usr/spool/cron/crontabs/rfindd usr/spool/cron/crontabs/rfindd
       config(suggest)
```

Figure 3-9. idb Attributes for *rfind* (idbgen)

If you are using `idbgen`, the file *idb3* is now a complete raw idb file. You can skip Step 4 and go on to Step 5.

3.4 Step 4. Add idb Tags and idb Attributes to Makefiles

In this step, you will edit Makefiles so that they include the idb tags and idb attributes you selected for your files in Steps 2 and 3. If you used `idbgen` to create your raw idb file, you should skip this step and go on to Step 5.

After deciding on your idb tags and idb attributes (Steps 2 and 3), you need to add them to your Makefiles. To do this you will add the `-idb` option to all of your `install ($(INSTALL))` commands. You also might need to switch from `-f` to `-F` or add `-F` on some `install` lines, and to add some lines to your Makefiles to explicitly create the directories that you identified in Step 1.

The tasks in this step are:

1. Edit all Makefiles that generate files in your product and modify the `$(INSTALL)` lines for files so that they have the form:

   `$(INSTALL) -F` *installdir* `[`*option ...*`]` *file ...* -idb "*idbtag idbattribute ...*"

   where

   installdir is the name of the directory that *file* will be installed in, with no leading /.

   option is any other `install` option except `-f`. (`-f` is never appropriate because we want directories to be created if they don't exist.) Discussion of these options is beyond the scope of this document so see `install`(1) or see the document *Silicon Graphics Makefile Conventions*.

   file is a file in your product

   idbtag is the idb tag you chose for *file* in Step 2

   idbattribute is an idb attribute you chose for *file* in Step 3

   You can look up the idb tags from the list you created in Step 2, and get the idb attributes from your Step 3 work. Figure 3-10 shows the before and after versions of the `$(INSTALL)` lines for files in the *rfind* product.

```
Before:
        ${INSTALL} -u bin -g bin -F /usr/lib librfind.a
        ${INSTALL} -u root -g sys -F /usr/lib/rfindd -m 4111 fsdump
        ${INSTALL} ${USR_GRP} -F /usr/spool/cron/crontabs -m 644 -src crontab rfindd
        ${INSTALL} -F /etc/config -m 644  rfindd
        ${INSTALL} ${USR_GRP} -F /etc/init.d -m 755 rfindd
        ${INSTALL} -u bin -g bin -F /usr/man/u_man/man1 -m 644 rfind.1 rfindd.1m fsdump.1m
        ${INSTALL} -u bin -g bin -F /usr/local/bin -m 755 rfind
        ${INSTALL} -u bin -g bin -F /usr/lib -m 644 rfind.aliases
        ${INSTALL} ${USR_GRP} -F /usr/lib/rfindd -m 755 rfindd
        ${INSTALL} -u bin -g bin -F /usr/lib/rfindd passwd.add
        ${INSTALL} ${USR_GRP} -F /usr/lib/rfindd -m 644 -src forward .forward
        ${INSTALL} ${USR_GRP} -F /usr/lib/rfindd -m 644  README
        ${INSTALL} ${USR_GRP} -F /usr/lib/rfindd -m 755 rotatelogs

After:
        ${INSTALL} -u bin -g bin -F /usr/lib -idb rfind.sw.internal librfind.a
        ${INSTALL} -u root -g sys -F /usr/lib/rfindd -idb "rfind.sw.server" -m 4111 fsdump
        ${INSTALL} ${USR_GRP} -F /usr/spool/cron/crontabs -idb 'rfind.sw.server
               config(suggest)' -m 644 -src crontab rfindd
        ${INSTALL} -F /etc/config -m 644 -idb "rfind.sw.server config(noupdate)" rfindd
        ${INSTALL} ${USR_GRP} -F /etc/init.d -idb "rfind.sw.server config(update)"
               -m 755 rfindd
        ${INSTALL} -u bin -g bin -F /usr/man/u_man/man1 -idb "rfind.man.rfind"
               -m 644 rfind.1 rfindd.1m fsdump.1m
        ${INSTALL} -u bin -g bin -F /usr/local/bin -idb "rfind.sw.client" -m 755 rfind
        ${INSTALL} -u bin -g bin -F /usr/lib -idb "rfind.sw.client config(suggest)"
               -m 644 rfind.aliases
        ${INSTALL} ${USR_GRP} -F /usr/lib/rfindd -idb 'rfind.sw.server' -m 755 rfindd
        ${INSTALL} -u bin -g bin -F /usr/lib/rfindd -idb 'rfind.sw.server
               exitop("chroot $$rbase /bin/sh /usr/lib/rfindd/passwd.add")' passwd.add
        ${INSTALL} ${USR_GRP} -F /usr/lib/rfindd -idb 'rfind.sw.server config(noupdate)'
               -m 644 -src forward .forward
        ${INSTALL} ${USR_GRP} -F /usr/lib/rfindd -idb 'rfind.sw.server' -m 644 README
        ${INSTALL} ${USR_GRP} -F /usr/lib/rfindd -idb 'rfind.sw.server' -m 755 rotatelogs
```

Figure 3-10. $(INSTALL) Lines for Files

2. For each directory, you will have to add a `$(INSTALL)` line of this form to one of the Makefiles that makes files for each subsystem that uses that directory.

   `$(INSTALL)` [*option ...*] `-idb` "*idbtag idbattribute ...*" `-dir` *directory ...*

   where

   directory is a directory that your product is going to create on users' systems

   Figure 3-11 shows the before and after versions of the `$(INSTALL)` lines for directories in the *rfind* product.

Before:

```
${INSTALL} -u rfindd -g nuucp -dir /usr/lib/rfindd
```

After:

```
${INSTALL} -u rfindd -g nuucp -idb "rfind.sw.server" -dir /usr/lib/rfindd
```

Figure 3-11. $(INSTALL) Lines for Directories

3. Look for $(INSTALL) lines in your Makefiles that create symbolic links. Edit the lines so they have this syntax:

   `$(INSTALL) -F` *installdir* `[`*option ...*`] -lns -idb "`*idbtag idbattribute ...*`"` *file link*

Figure 3-12 shows the before and after versions of the `$(INSTALL)` lines for symbolic links in the *rfind* product.

Before:

```
${INSTALL} -F /etc/rc2.d -lns /etc/init.d/rfindd S98rfindd
${INSTALL} -F /etc/rc0.d -lns /etc/init.d/rfindd K98rfindd
```

After:

```
${INSTALL} -F /etc/rc2.d -lns -idb "rfind.sw.server" /etc/init.d/rfindd S98rfindd
${INSTALL} -F /etc/rc0.d -lns -idb "rfind.sw.server" /etc/init.d/rfindd K98rfindd
```

Figure 3-12. $(INSTALL) Lines for Symbolic Links

3.5 Step 5. Write the Spec File

The *spec file* is where you specify the product hierarchy for one or more products and define the dependencies within this product as well as on other products. Using examples throughout, this step provides comprehensive instructions on how to build your spec file. You can follow the directions in Section 3.5.1 to create a basic spec file and return to Section 3.5.2 after you have completed Steps 6, 7, and 8 if you wish, since it will be easier to debug a basic spec file initially.

3.5.1 Writing the Basic Spec File

There is a spec file template that you can start from. It is located in the source tree in `usr/src/template/template.spec`. Copy the template into your working area. To name your spec file, use the following form:

`spec.`*name*

where *name* typically is a short, descriptive name of your product. (For example, `spec.rfind`.) When you are ready to check your spec file into the source tree, check with the Release Group to see where they want it. Possible locations are

`usr/src/cmd/idbtools/dist` and `usr/src/build/specs`.

A copy of the *rfind* spec file is at the end of this section. (Figure 3-13). The number of each task below is listed on the line it applies to in the example. If you have more than one product, pick one subsystem (*product.image.subsystem*) to start with, then work your way through the tasks.

1. Specify the start of the specification of the product.

2. Specify the product ID string. The product ID string appears in `versions`(1M) listings and in the rightmost column of `inst` "list" listings. Appendix C gives the guidelines for ID strings.

3. Specify the *image* name with the name you chose in Step 2.

4. Specify the image ID string. Use the guidelines in Appendix C.

5. Associate the version number with the image. This is a ten-character number, broken down into three parts:

   MMMMmmmaaa OR *MMMMmmm*`${ALPHA}`

   where

MMMM	is the base number, specified in integers only.
mmm	is the maintenance number, specified in integers only. (For a base release, the maintenance number is 000.)
aaa	is the alpha number, specified in integers only.
`${ALPHA}`	is substituted with the appropriate value as the spec file is read by tools that generate the distribution.

   If you use `${ALPHA}`, the alpha number can easily be incremented for every image. The alpha number should be incremented for every alpha release so that previous versions get replaced.

   Version numbers are the only indication of the relationship between two versions of a subsystem that is available to `inst`. The subsystem with the higher number is assumed to be newer. A subsystem with the same version number is assumed by `inst` to have the same content. Therefore, re-using a previous alpha number or using a version number that is lower than a previous version number is not recommended because it can cause incorrect installation of the newer subsystem.

   Here are the version numbers used in the *rfind* spec file examples (Figures 3-13 and 3-14):

   `1006000106` *(base release of rfind;* alpha 106 of release 1006)
   `1006001107` *(maintenance release; alpha 107 of maintenance 1 of release 1006)*
   `1007000023` *(later base release; alpha 23 of version 1007)*

Instead of placing version number information directly in the spec file, you can define it inside a header file, then include the header file in your spec file. The header file is searched for in the same directory as the spec file. Specify the header file as follows in the spec file.

```
include spec.name.h
```

When you have to change any version number values, you only need to do it once inside the header file.

6. Specify the start of the specification of a subsystem.

7. Specify the subsystem *flags*. Flags are characteristics of a subsystem, such as under what circumstances it can be installed and when it should be installed. If your product is based on 3.3, the flag names will be different.

   `required` means that this subsystem is critical to the operation of the system. Without it, your machine will not run. If the distribution contains `required` subsystems, `inst` will not start the installation until they get selected. `inst` will not remove `required` subsystems with the "remove" and "go" commands, the "versions remove" command must be used.

   `miniroot` means that in order to install this subsystem properly, you must be in the miniroot. If this flag isn't present, the subsystem can be installed on a running machine. Use `miniroot` when successful installation of a subsystem requires rebuilding the kernel and/or rebooting the system, when daemons are included in the product, when consistency between programs and their configuration files is an issue, and when already running programs may get activated again. There may be other situations that require miniroot installation as well.

   `default` means that this subsystem is suggested for installation by customers who want to install a subset of this product. When this subsystem is available for installation for the first time on a workstation, `inst` will automatically select it for installation. The decision as to which subsystems are `default` belongs to the owner of the product.

8. Specify the subsystem ID string (see Appendix C for the guidelines). Since the subsystem ID string plus the list of files in a subsystem is often the only information an `inst` user has to help them decide whether or not to install the subsystem, make the ID string as descriptive as possible.

9. Create expressions. Expressions are the mechanism for mapping idb tags to subsystems. Appendix D contains the definition of the language that can be used in expressions.

   There are simple expressions in the *rfind* spec file. They create a "one-to-one" mapping between the idb tags and subsystem names created in Step 2. It is considered one-to-one because every idb tag name gets mapped into one

subsystem name. Use the example spec file as a reference for making the association between the idb tag and the subsystem it gets mapped to. An example expression is:

```
exp "rfind.sw.server"
```

The files specified by this expression are the files with the idb tag `rfind.sw.server`. They get mapped to the `rfind.sw.server` subsystem. Note that the name used as the idb tag need not match the subsystem name.

Later, you will see how a "many-to-one" mapping is created in an expressions line. It is different from a one-to-one mapping because files with several different idb tag names get mapped into one subsystem.

10. Specify additional subsystem information, described in the following section. You can delay this task until you get the bugs worked out of what you have created so far if you wish.

11. End the subsystem.

12. Specify the start and end of the description of subsequent subsystems within the same image. Repeat tasks 7 through 10 to fill in the description.

13. End the image.

14. Specify the start and end of the description of subsequent images within the same product. Repeat tasks 4 through 12 to fill in the description.

15. End the product.

```
1)product rfind
2)      id "rfind Remote Fast Find 1.0"
3)      image sw
4)              id "rfind Software"
5)              version 1006000106
6,7)            subsys client default
i)                      id "rfind Client Command"
9)                      exp "rfind.sw.client"
11)             endsubsys
6)              subsys server
8)                      id "rfind Server Support (Only)"
9)                      exp "rfind.sw.server"
11)             endsubsys
13)     endimage
3)      image man
4)              id "rfind Man Pages"
5)              version 1006000106
6,7)            subsys rfind default
8)                      id "rfind Man Pages"
9)                      exp "rfind.man.rfind"
11)             endsubsys
13)     endimage
15)endproduct1
```

Figure 3-13. *rfind* Basic Spec File

3.5.2 Adding Additional Information

In addition to basic spec file information, you may need to include more complicated installation to your spec file. This information includes order numbers, prerequisites, complex expressions, replacement rules, and incompatible subsystems. Order numbers are specified at the image level; the others are specified at the subsystem level.

Since this information varies with each product, your conditions will not be the same as those provided in the examples. However, you can use the examples for syntax and formatting purposes and to facilitate your understanding of the features. Again, the number of the feature is the same as the number of the line in the spec file example (Figure 3-14) at the end of this section. The features are described below.

1. **Order numbers** (for images).

   `order` numbers determine the order in which the images are installed when you have multiple products in one distribution. The `order` statement has the form:

   ```
   order n
   ```

   where *n* is an integer from 0 to 9999.

   The lower the `order` number, the earlier that image will be installed. If you do not specify an order number, the default is 9999 (the highest order, meaning the

last to be installed). Images with equal `order` numbers are installed in alphabetical order.

In the *rfind* spec file in Figure 3-14, notice that the images have been given `order` numbers that guarantee that the software image will be installed before the manual page image.

2. **Expressions**.

   The earlier discussion about expressions illustrated a one-to-one mapping between idb tags and subsystem names. But, what if you want files with different idb tags to belong to the same subsystem? Or files with the same idb tags to belong to two different subsystems? By employing operators, you can pluck files with different names from anywhere and pull them together into one subsystem or qualify which files with a particular idb tag go into a subsystem.

   The following operators are used in expressions: `!` (not), `&&` (and), `||` (or), `=~` (file name pattern matching), `!~` (file name no match), `( )` (grouping function arguments).

   The following variables can be used in expressions: *dstpath* (relative path in destination tree, and *srcpath* (relative path in source tree).

   The *rfind* spec file in Figure 3-14 below shows how the "one-to-many" mapping works. It contains these `exp` statements:

```
exp 'rfind.man.rfind && srcpath =~ "*.1"'
exp 'rfind.man.rfind && srcpath =~ "*.1m"'
```

   The basic spec file for *rfind* didn't follow the recommendation that each software subsystem have a matching manual page system, so the spec file in Figure 3-14 does some repackaging to achieve this. One idb tag was used for all manual pages, so some operators and the builtin variable *srcpath* were used in the `exp` statement to put the .1 manual pages into one subsystem and the .1m manual pages into another.

3. **Replaces**.

   The `replaces` line is very important because it is used by `inst` to determine what will or will not be selected for automatic removal. `replaces` are powerful and flexible, so taking the time now to learn how they work will facilitate changes you may have to make to your product in the future and prevent you from making mistakes that will be difficult to correct. The `replaces` line is critical when a product is *repackaged* (i.e., files in the product get moved around; subsystems get new names, and so on) for a later release.

   A `replaces` line simply specifies that you want to replace an older subsystem with a newer subsystem. Specify the name of the subsystem and the range of version numbers you want replaced. The format of the `replaces` line is as follows:

```
replaces name lowvers highvers
```

OR

```
replaces self
```

where

name is the name of the subsystem that is going to be replaced.

lowvers is the lower boundary of the range of versions of *name* that should be replaced. It can be 0, or any version number value that you supply.

highvers is the higher boundary of the range of versions to be replaced. *highvers* can be one of the following:

- `oldvers`, defined as the current version minus 1.
- an actual version number that you supply.

`self` `inst` always assumes a subsystem replaces itself. It is the default case and you do not need to specify it explicitly.

You can specify as many `replaces` as you need.

How does `inst` use `replaces` at installation time? For each subsystem that has a `replaces` line, `inst` looks to see if the subsystem specified in the `replaces` line is installed and if its version falls in the range given in the `replaces` line. If the installed version is in the range, the new subsystem is selected for installation automatically. If the new subsystem is installed, the old version is removed automatically.

There are four typical ways to use `replaces`:

1. Specify that a subsystem replaces older versions of itself.
2. Specify that a subsystem replaces maintenance versions of itself.
3. Specify that a subsystem replaces different subsystems that are now obsolete.
4. Specify replacement directions in complex repackaging situations where one subsystem has become several or several older subsystems have been restructured into several new ones.

Examples of `replaces` lines are given below.

Example 1

When you want to replace all previous versions of a subsystem, use:

```
replaces self
```

This is the default. `inst` will always replace a subsystem with a new version of itself.

Example 2

A subsystem gets a name change. In the example in Figure 3-14, `rfind.man.rfind` subsystem becomes `rfind.man.client` and `rfind.man.server`. To see that `rfind.man.rfind` is replaced properly, you have a combination of `replaces` to choose from:

```
replaces rfind.man.rfind 0 oldvers
```

OR

```
replaces rfind.man.rfind 0 1006000106
```

Example 3

Two or more subsystems get repackaged into a single subsystem. For example, say the `rfind.sw.client` subsystem and the `rfind.sw.server` subsystem are combined and the new subsystem is called `rfind.sw.rfind`. To replace the old `client` and `server`, you have to use:

```
replaces rfind.sw.client 0 oldvers
replaces rfind.sw.server 0 oldvers
```

If either of the old subsystems `rfind.sw.client` or `rfind.sw.server` is installed, `rfind.sw.rfind` will get selected for installation.

Example 4

A maintenance release of `rfind.sw.client` is to be replaced with a new base release. You need to replace both the maintenance release and the previous base release.

```
replaces maint.rfind_sw.client 0 oldvers
replaces rfind.sw.client 0 oldvers
```

4. **Incompats.**

An `incompat` line specifies the version(s) of subsystem(s) that are incompatible with the subsystem you are installing. The format is the same as the `replaces` line. Specify the name of the subsystem and the range of version numbers you want to declare incompatible.

`incompat` *name lowvers highvers*

`inst` does not allow you to install subsystems that are incompatible. It checks for incompatibilities at two different times:

- When a subsystem is selected for installation, `inst` determines whether or not it is incompatible with something that has already been installed.
- When the user "quits," `inst` checks again among the subsystems it has just installed for incompatibilities.

The format of the `incompat` line is as follows:

```
incompat name lowvers highvers
```

where

name is the name of the subsystem that is incompatible.

lowvers is the lower boundary of the range of versions of *name* that is incompatible. It can be 0, or any version number value that you supply.

highvers is the higher boundary of the range of versions that is incompatible. *highvers* can be one of the following:

- `maxint`, the maximum value that a `long int` can hold.
- `oldvers`, defined as the current version minus 1.
- an actual version number that you supply.

5. **Prereqs.**

A `prereq` is a set of subsystems that needs to be present when you install the new subsystem. The format is the same as the `replaces` line, except that `prereqs` are placed inside parentheses. This is an AND `prereq` condition. It specifies that subsystems which fall into the range specified must be present in order for users to install the new subsystem.

```
prereq      (
            name  lowvers  highvers
            name  lowvers  highvers
            name  lowvers  highvers
            )
```

where

name is the name of the subsystem that is a prerequisite.

lowvers is the lower boundary of the range of versions of *name* that is a prerequisite. It can be 0, or any version number value that you supply.

highvers is the higher boundary of the range of versions that is a prerequisite. *highvers* can be one of the following:

- `maxint`, the maximum value that a `long int` can hold.
- `oldvers`, defined as the current version minus 1.
- an actual version number that you supply.

If you require an OR condition, separate the `prereqs`. This `prereq` specifies that either one subsystem or the other must be present for the new subsystem.

```
prereq      (
            name  lowvers  highvers
            )

prereq      (
            name  lowvers  highvers
            )
```

`inst` will not install a subsystem that has prerequisites unless the prerequisite subsystems are already installed or selected for installation.

In *rfind*, the `nfs.sw.nfs` subsystem is a `prereq` to `rfind.sw.server`.

```
product rfind
        id "rfind Remote Fast Find 2.0"
        image sw
                id "rfind Software"
                version 1007000023
1)              order 9980
                subsys client default
                        id "rfind Client Command"
                        exp "rfind.sw.client"
3)                      replaces maint.rfind_sw.client 0 1006999999
                endsubsys
                subsys server
                        id "rfind Server Support (Only)"
                        exp "rfind.sw.server"
3)                      replaces maint.rfind_sw.server 0 1006999999
6)                      prereq ( nfs.sw.nfs 1007000000 maxint )
                endsubsys
        endimage
        image man
                id "rfind Man Pages"
                version 1007000023
1)              order 9990
                subsys client default
                        id "rfind Client Man Pages"
2)                      exp 'rfind.man.rfind && srcpath =~ "*.1"'
3)                      replaces rfind.man.rfind 0 1006999999
3)                      replaces maint.rfind_man.rfind 0 1006999999
                endsubsys
                subsys server default
                        id "rfind Server Man Pages"
2)                      exp 'rfind.man.rfind && srcpath =~ "*.1m"'
3)                      replaces rfind.man.rfind 0 1006999999
3)                      replaces maint.rfind_man.rfind 0 1006999999
                endsubsys
        endimage
endproduct
```

Figure 3-14. *rfind* Spec File with Additional Information

3.7 Step 7. Create the Distribution

In this step, you create your software distribution. The tasks in this step are described below.

1. If you used `install -idb`, you need to sort the raw idb file. The command is:

   `sort -u +4` *rawidb* `>` *idb3*

2. Make sure you have the following as input to `gendist`(1M).

   - your spec file (*specfile*)
   - your sorted raw idb file (*idb3*)
   - your binary tree (rooted at *sourcedir*, this is a built source tree if you are using `install -idb` or a hierarchy of files in your product of you are using `idbgen`)
   - the temporary storage place for your images (*distdir*)

3. Execute `gendist` with the `-dryrun` option. `gendist` is the tool that generates your distribution. When you use the `-dryrun` option, no images will be created, but all of the error message output will be generated. The basic `gendist` command (with the `-dryrun` option on) is:

   `gendist [-v] [-dist` *distdir*`] [-sbase` *sourcedir*`] [-idb` *idb3*`] [-spec` *specfile*`] -dryrun`

   where

   `-v` specifies verbose mode.

   `-dist` specifies *distdir* as the directory in which you want the distribution created (default is `/root/usr/dist`).

   `-sbase` specifies *sourcedir* as the source tree (default is `/root/usr/src`).

   `-idb` specifies the sorted idb file *idb3* as input (default is `/root/etc/idb`).

   `-spec` specifies *specfile* as input (default is `/root/etc/spec`).

   `-dryrun` specifies that the output should be error messages only; no distribution is created.

   If `gendist` encounters problems, it will either complete and return error messages, or it will fail. Some possible sources of problems are:

   - An error in the spec file prevented `gendist` from parsing it.
   - A failure may be a sign that the idb file was not sorted. It can also mean that `gendist` cannot read the spec file properly.

- Warnings about duplicate files indicate that there are two or more files with identical pathnames in a subsystem and these files do not have `mach` idb attributes which specify that there are machine-specific versions of this file. These warnings are caused by violations of the Golden Rule. They are fixed by making the pathnames for each of the files in your subsystem unique.

- Invalid idb attributes result in messages.

- Error messages may indicate that you have empty products, images, or subsystems. Check to determine whether or not they need to exist. If not, you can just comment them out temporarily in the spec file.

The problems detected by the dryrun of `gendist` should be resolved before you continue with the next step: using `gendist` to create the distribution.

Appendix B, "Troubleshooting," contains more details on solving problems detected by `gendist`. The `gendist`(1M) manual page contains information about additional `gendist` options.

4. Execute `gendist` using the same command line as the previous step, but without the `-dryrun` option. If `gendist` creates the distribution successfully, the output is in the form of:

   - a product description file
     This is the binary form of the spec file. It is called *product*.

   - an idb file
     This file is the complete installation database. To determine if it has indeed been created, take a look inside the file. The *rfind* idb file is called `rfind.idb`.

   - image files
     This is the set of images (software and manual pages). The *rfind* set of images are `rfind.man` and `rfind.sw`.

   Here is an example of the `inst` images for *rfind*:

```
total 381
-rw-r--r--   1 root     sys          411 Oct 30 15:33 rfind
-rw-r--r--   1 root     sys         2688 Oct 30 15:33 rfind.idb
-rw-r--r--   1 root     sys        16109 Oct 30 15:33 rfind.man
-rw-r--r--   1 root     sys       174674 Oct 30 15:33 rfind.sw
```

3.8 Step 8. Verify the Distribution

Verifying your distribution is a critical step towards completing your product. You may think there is nothing wrong with your product, but you should check it out anyway. This step tests the integrity of your product and, if there are problems, sends you back to various steps in the process to enable you to make corrections and adjustments.

The goals to this step are:

- To achieve an error-free installation.
- To be able to execute and run your product without problems.

The tasks in this step are described below.

1. Execute `showprod` on the product description. `showprod` is a tool that reads and shows your product. Read through the output and verify that it is in fact a reflection of what you want.

   For more information about `showprod`, see the `showprod`(1M) manual page.

2. Verify the completeness of the idb file:

   - Compare the raw idb file and the idb file to make sure that no files were lost (possibly because they didn't build). To do this, use *awk* to extract the *destination* (fifth) field from the raw idb file and all of the idb files. Concatenate and `sort -u` the fifth fields of all of the idb files then use `diff` to compare it with the fifth fields from the raw idb file. Any *destinations* that are in the raw idb file, but not in any of the idb files indicates a problem.

   - Make sure that everything that you built ended up in the product. To locate all the executables, run `find` from the top level source directory. For every executable there, verify its presence in the idb file using `grep`.

3. Make sure that the full pathnames of the files in your product are unique across products. It is easiest to do this if you have a directory that contains the software distributions (or just idb files) for all products. (See the directions for creating distributions in the IRIS Software Installation Guide or `distcp`(1M).) Extract the destination pathnames from the idb file for your product and use them and the name of the directory that contains the software distributions as arguments to `instid`. The command to do this has the form:

   ```
   nawk '{print $5}' product.idb | xargs instid -d -f distdir
   ```

4. Install the software.

   To do your initial test installations, you can install the software from a running system (whether or not any subsystem in your product uses the `miniroot` flag) by using an `inst` command of the form:

```
inst -f distdir -r testarea
```

where

`-f` specifies *distdir* as the directory the distribution is located in (the same *distdir* directory specified when you executed `gendist` in Step 7).

`-r` specifies *testarea* as a temp directory or some other test installation place for the distribution. If you do not specify a *testarea* directory, the distribution is installed in `/`.

It is important to install the software in various situations (on systems with and without previous versions, different machine types, etc.) and to install a wide variety of other products to determine compatibility. For more information about the `inst` command, see the `inst` manual page and the IRIS Software Installation Guide.

5. After you install the product, use the `versions -a` command to list all the installed software to ensure that the old versions have been removed. If you have installed old and new versions in a test area, add the argument `-r testarea` to the `versions` command so that it will report on the test area rather than the software installed in the default locations.

6. After installing your product in the default location, start using your software and determine the following:

   - Does your product not work if certain products or subsystems are also installed?

     If so, you specified the `incompats` improperly. If you have an `incompat` and `inst` allows you to install incompatible subsystems, either you did not specify the `incompat`, or the products specified by your `incompat` have not been installed. See Step 5.

   - Do `prereqs` get reported?
     Verify that the prerequisite products are installed; otherwise, a `prereq` requirement should be reported.

   - If your product or subsystem does not work because some programs are missing, specify in a `prereq` the subsystem to which the programs belong.

Guess what? You're done!

Don't delete your built source tree at *sourcedir*. You'll need it if you ever create a maintenance release for your product.

Chapter 4

Creating a Maintenance Release

Maintenance releases are software distributions that are usually created in the following cases:

- When the quality of a base release demands it.
- When you have to change files to support new hardware platforms.

The decision to do a maintenance release typically is made by the Software QA group rather than the owner of the product.

The tasks to prepare a maintenance release are described below.

1. The first task is to prepare the spec file for the maintenance release.

   a. If you spec file doesn't already have them, add these three lines to the end of it:

      ```
      maint product
            id "ID string"
      endmaint
      ```

   b. Change the value of the maintenance number portion of the version number.

   c. Change the alpha number portion of the version number if necessary.

   d. Change ID strings if necessary for a new release number.

   e. No changes or additions of `replaces` lines are necessary since `gendist` creates the correct rules for old base and maintenance releases of the product.

   Figure 4-1 shows the *rfind* spec file after it has been modified for a maintenance release.

```
product rfind
      id "rfind Remote Fast Find 1.0"
      image sw
            id "rfind Software"
            version 1006001107
            subsys client default
                  id "rfind Client Command"
                  exp "rfind.sw.client"
            endsubsys
            subsys server
                  id "rfind Server Support (Only)"
                  exp "rfind.sw.server"
            endsubsys
      endimage
      image man
            id "rfind Man Pages"
            version 1006001107
            subsys rfind default
                  id "rfind Man Pages"
                  exp "rfind.man.rfind"
            endsubsys
      endimage
endproduct
maint rfind
      id "rfind rfind 2.0"
endmaint
```

Figure 4-1. Spec File for *rfind* Maintenance Release

2. Build your maintenance source tree. It should be the same tree as the tree for the base release with all changes that have been made for all maintenance releases since the base release. Set the value of $RAWIDB to a file name of *rawidb2*. The name of the root directory for the built source tree is *maintdir* (this is the equivalent of *sourcedir* for the base release).

3. Get a copy of the sorted raw idb file from the base release.

4. Execute `idbdiff` on the original raw idb file and the new raw idb file from the maintenance source tree you just built to obtain the differences between the two. The file into which you redirect the output becomes the raw idb file for the maintenance release.

   `idbdiff -s` *sourcedir* `-m` *maintdir rawidb1 rawidb2* `>` *idb4*

5. Execute `gendist` to generate the maintenance images with this command:

   `gendist [-v] [-dist` *distdir*`] [-sbase` *maintdir*`] [-idb` *idb4*`]`
   `[-spec` *specfile*`] -maint`

   where

`-v`	specifies verbose mode.
`-dist`	specifies *distdir* as the directory in which you want the distribution created (default is `/root/usr/dist`).
`-sbase`	specifies *sourcedir* as the source tree (default is `/root/usr/src`).
`-idb`	specifies *idb4* as input (default is `/root/etc/idb`).
`-spec`	specifies *specfile* as input (default is `/root/etc/spec`).
`-maint`	creates a maintenance release.

The output is the actual distribution in the following files (the names of the *rfind* files are shown in parenthesis);

- a product description file (`rfind`)
  This is the binary form of the spec file.

- an idb file (`rfind.idb`)
  This is the complete installation database. To determine if it has indeed been created, take a look inside the idb file.

- image files (`maint.rfind_man`, `maint.rfind_sw`)
  This is the set of images (software and manual pages).

6. To test your maintenance release, see Step 8 in Chapter 3.

Appendix A

Glossary

alpha number

Every product in a software distribution has a three-digit alpha number. It is part of the version number that is used by users (they see it with `versions -n`) and by `inst` to identify different version numbers of software. The alpha number is set in the spec file (see Section 3.5.1) and is typically incremented each time a software distribution is created. The automatic `inst` selection algorithms do not work properly if alpha numbers decrease or if they remain the same.

attribute

See "idb attribute".

base number

Every release has a four digit base number that is the first four digits of the version number. For example, the base number for the 4D1-4.0 release is 1006. Base releases and their maintenance releases always have the same base number. Different products may or may not share the same base number. The base number does not change from alpha to alpha of the base release or for maintenance releases for the base release. It is set in the spec file (see Section 3.5.1).

base release

A base release (as opposed to a maintenance release) is a release that contains a complete set of files for a product. It obsoletes all files contained in all previous base and maintenance releases of that product unless explicitly specified otherwise in the spec file.

build tree

A build tree contains source files and the objects, binaries and other files that result from building (i.e running `make`, compiling) the tree.

config

The `config` idb attribute identifies a file as being a configuration file. It takes an argument (`update`, `noupdate` or `suggest`) which tells what type of configuration file it is. It is given an as argument to `install` commands in Makefiles or added manually to a raw idb file (see Section 3.3).

configuration file

A configuration file is a file that is part of a software product, but is likely

to require modification by the user of a workstation. These modifications are specific to that workstation or the user's site. The three types of configuration files ("update", "noupdate" and "suggest") are each handled differently during installation. See Section 3.3 for more information.

default flag

`default` is a subsystem flag is used in spec files. It tells `inst` that when this subsystem is available for the first time on a workstation, it should be selected for installation. See Section 3.5.1 and the *IRIS Software Installation Guide* for more information.

distcp

`distcp` is a tool in eoe1.sw.unix that copies software distributions or portions of them from tape, CD-ROM, or disk to tape or disk. It can also be used to compare software distributions. See `distcp`(1M) for more information.

distribution

See **software distribution**.

expression

Expressions are used in spec files to specify the files that go into a subsystem. Often this specification maps one or more idb tags to a subsystem, but there are other possibilities as well. See Sections 3.5.1 and 3.5.2, and Appendix D.

exitop

`exitop` idb attributes specify operations to be performed just prior to exiting `inst` when the file they are attached to is installed. They are given as arguments to `install` commands in Makefiles (see Section 3.4) or added manually to a raw idb file (see Section 3.3).

flag

A flag is a type of keyword in spec files. Flags describe installation characteristics of subsystems such as whether they require installation from the miniroot. The three possible flags (`miniroot`, `required`, and `default`) are described in Section 3.5.1.

gendist

`gendist` is the tool that generates software distributions. It takes binaries and other files destined for a software distribution (as specified by `install` commands in Makefiles) from a compiled build tree and creates product descriptions, idb files and images (as specified by a spec file). See `gendist`(1M) and Section 3.7 for more information.

Golden Rule

The Golden Rule of packaging software for `inst` is "Thou shalt not include two files with the same full pathname in two different subsystems." The reason for the rule is that `inst` assumes that each file is part of only one subsystem. If a file were included in two subsystems

which were installed and then one of them was removed, the common file would disappear, leaving a "hole" in the other subsystem. This rule applies only to files, not directories.

idb attribute

Idb attributes are special keywords that introduce information about files. Examples are `config` (configuration file), `mach` (machine-specific file) and `postop` (requires special processing after installation). Idb attributes are specified in arguments to `install` commands in Makefiles (see Section 3.4) or manually created raw idb files (see Section 3.3). The complete list of idb attributes is in Section 3.3.

idb file

idb files are created by `gendist` using a raw idb file as input. They are one of the components of software distributions. They are named *product*.**idb** and contain one line of information for every file, directory, link, and fifo in *product*. See Appendix E for more information. See also "raw idb file".

idb tag

Each file in a software product has a name called an idb tag associated with it. This association is done with the `install -idb` flag or by manually editing a raw idb file. idb tags can be of the form *name* or *name1.name2.name3*. All files with the same idb tag are put into a single subsystem. The mapping of idb tags to subsystems is done in `exp` statements in a spec file.

idbdiff

`idbdiff` is a tool that is used during the creation of maintenance releases to compare idb files. See Chapter 4 and `idbdiff`(1M) for more information.

idbgen

`idbgen` is a tool that is used to begin the creation of raw idb files. The output of `idbgen` needs to be edited to add idb tag and idb attribute information in order to turn it into a raw idb file. Using `idbgen` is an alternative to adding idb information to Makefiles, but is used only in unusual circumstances. Its use is explained in Section 3.1.2 and `idbgen`(1M).

image

Images are the middle level in the three-level product.image.subsystem hierarchy. They are collections of subsystems. Because of the format of software distributions, each image is a single file in a distribution. They are archives of a portion or all of the files in a software product. Images have the name *product.image*. Typical *image* names are 'sw' and 'man'.

incompat

`incompat` is a keyword that is used in spec files to declare the names and versions of subsystems that are incompatible (i.e. one subsystem can be present (installed) on a workstation at a time, not both). More

information is available in Section 3.5.2.

inst

`inst` is the program that installs software distributions and removes installed software on workstations. Its use is described in the *IRIS Software Installation Guide* and `inst`(1M). This document describes the process for preparing software distributions so they can be installed by `inst`.

instid

`instid` is a program that takes file names as input and returns all full path names that match and the names of subsystem(s) that they come from as output. More information about `instid` and its use is in `instid`(1M).

mach

The `mach` idb attribute identifies files that are machine specific. It is used as an argument to `install` commands in Makefiles or added manually to raw idb files. The `mach` idb attribute takes values called *mach tags* which are discussed in Section 3.3.

mach

Mach tags are values that are given with `mach` idb attributes. They have the form `mach(`*hardware=value*`)`. *hardware* can be "CPUBOARD", "GFXBOARD", or "SUBGR". See Section 3.3 for the values of *value*.

maintenance number

A maintenance number is a three digit number used in spec files. Each maintenance release gets one, and in combination with a base number and alpha number, it uniquely identifies a maintenance release of a product. This number does not change from alpha to alpha of the maintenance release.

maintenance release

A maintenance release is a software release designed to be used in conjunction with base releases of a collection of products. It includes bug fixes and possibly support for new hardware or software features and is a subset of the files in the products. See Section 2.1 for more of the characteristics of maintenance releases.

Makefile

Makefiles contain dependency information and commands for compiling or otherwise building source code. They also include information on what products and subsystems the results of the build should be put into and information on how the results (files) should be installed. There is one Makefile per directory in the source tree.

miniroot flag

The `miniroot` subsystem flag is used in spec files to indicate that certain subsystems can be installed only when `inst` has been invoked from the miniroot. See `inst`(1M) and the *IRIS Software Installation Guide* to find

out more about the miniroot and Section 3.5.1 to find out more about the `miniroot` flag.

mr

This idb attribute, used in `install` commands in Makefiles, identifies files that should be included in the miniroot (i.e. the Installation Tools portion of a software release which includes a kernel, `inst`, and a few other programs).

noshare

This idb attribute, used in `install` commands in Makefiles, identifies files that are duplicated for each client in a diskless installation. It is discussed in Section 3.3.

noship

The `noship` idb attribute (used in Makefiles) marks files as being for internal use only. They will not get included in any products other than the product 'noship'.

`nostrip`

The `nostrip` idb attribute (used in Makefiles) mark files During software distribution creation, by default all binaries are stripped This can be overridden for all files with `gendist -nostrip` or by including the `nostrip` idb attribute for individual files.

noupdate

`noupdate` is an argument to the `config` idb attribute in an `install` command in a Makefile. If a file is marked `config(noupdate)` and there is no version already installed, `inst` will install the current (new) version. If there is a version on the system already, it will be replaced with the new version if the user has not modified the old version. If the user modified the old version, the new version is thrown away.

operator

Operators such as !, &&, || and =~ can be used in expressions in spec files to specify the idb tags and files that are contained in (map to) a subsystem. The complete list of operators is given in Appendix D and their use is described in Section 3.5.2.

order

The `order` keyword is used in spec files. Each image can have an order number (default 9999 or "last") that specifies when the image should be installed relative to other images. Installation proceeds from lowest order number selected to highest. See Section 3.5.2 for more information.

postop

The `postop` idb attribute is an argument to the `install -idb` flag or is added manually to raw idb files. Its value is a set of shell commands to be executed immediately after the file in the `install` command is installed. See Section 3.3 for more information.

preop

This idb attribute is similar to `postop` (see "postop") except that the shell commands are executed prior to installing the file.

prereq

The `prereq` keyword and its value are used in spec files to specify subsystem dependencies. During installation, a subsystem will not be installed unless its `prereq` subsystems are already installed or are selected for installation. See Section 3.5.2 for more information.

product

A product is a collection of subsystems that have the name "first name" of their three part name (e.g. ftn is the first name of ftn.man.relnotes). Each product also has a file called a product description and a file called an installation database (idb) file which also have the same first name. There is usually a one-to-one mapping between products and software items in the Silicon Graphics Price Book. The exceptions to this are: eoe1 and eoe2 together are the software shipped with all workstations in the Price Book, a 'maint*' product contains files from a lot of other products, and sometimes several products are bundled to form one item in the Price Book.

product description

Spec files are translated into product description files as one of the functions of `gendist`. There is one product description file per product when a software distribution is on CD-ROM or disk and their names are *product*.

raw idb file

A raw idb file is a file generated by either `idbgen` or `install -idb`. It is converted into an idb file by processing it with several tools including `gendist`. See Sections 3.6 (`install -idb`) or 3.3 (`idbgen`) for more information.

rbase

rbase is a builtin variable whose value is set to the pathname of the root of the destination tree. It is used in `exp` statements in spec files.

repackage

Repackaging software is the process of taking files in released subsystems and combining their new versions in different ways to form the next release. The files might move from one product to another as well as from one subsystem within a product to another. In order for `inst` to properly update a system to a new release which contains repackaged subsystems, the spec file must contain `replaces` information which tells `inst` about the repackaging. See Section 3.5.2 for more information.

required flag

The `required` subsystem flag appears in spec files. It identifies the subsystems that are critical to the operation of the workstation. When this flag is present, `inst` users are required to have this subsystem installed

or selected when they give the "go" command.

replaces

The `replaces` keyword is used in spec files. It is used with each subsystem and specifies what subsystems to remove from a workstation when this subsystem is installed. Typically it is used to specify that all previous versions of this subsystem should be removed (`replaces self`) and to specify that it should replace versions of this subsystem that had different names in earlier releases. See Section 3.5.2 for more information.

showprod

`showprod` is a tool that displays the information in product descriptions and installation history databases. See `showprod`(1M) and Section 3.8.

source tree

A source tree contains all of the source required for a product or group of products for a particular release. It also contains previous versions of those files under RCS control.

sbase

sbase is a builtin variable whose value is set to the pathname of the root of the source tree. It is used in `exp` statements in spec files.

software

A software distribution (also known as a distribution) is one or more software products on 1/4" tape, CD-ROM or in a single directory on disk that are in the proprietary format that enables them to be installed by `inst`.

spec file

A spec file defines the products that will be turned into a software distribution by `gendist`. It gives the names of the products, their images and subsystems, information about the contents of those subsystems, and the installation procedure and environment they require. It is discussed in Section 3.5.

subsystem

A subsystem is a collection of files that belongs to an image and a product. `inst` installs subsystems selected by the user rather than individual files. Section 3.2 discusses how to choose the files that should be grouped as a subsystem.

suggest

`suggest` is one of the possible values of the `config` idb attribute. If a file is marked `config(suggest)` and there is no version already installed, `inst` will install the current (new) version. If there is a version on the system already, it will not be touched if the user has modified the old version. Instead, the new version will be installed with `.N` (newer) appended to its name. If the user did not modify the old version, the new version replaces the old version.

symval

The `symval` idb attribute is used with files that are symbolic links. It is produced by the `install -idb` command when `install` is given the `-lns` option. Alternatively, it is generated by `idbgen`. See Section 3.3 for more information.

update

`update` is one of the possible values of the `config` idb attribute. If a file is marked `config(update)` and there is no version already installed, `inst` will install the current (new) version. If there is a version on the system already, it will be thrown away if it was not modified by the user. If it was modified, it will be renamed by appending `.O` (older) to its name.

versions

`versions` is a tool (in eoe1.sw.unix) that displays information about products, images, subsystems and files. It can also be used to remove installed software. See `versions`(1M) for more information.

Appendix B

Troubleshooting

This appendix addresses some of the problems you are most likely to encounter and presents approaches to solving the problem.

Problem

Obsolete subsystems or older versions of subsystems show up in `versions -a` after installation.

Meaning

If you install all of the subsystems in a product and a `versions -an` listing shows multiple lines for products, images, and subsystems:

```
I = Installed, R = Removed

   Name                 Version   Description

I  rfind               681523105  rfind Remote Fast Find 1.0
I  rfind.man          1006000106  rfind Man Pages
I  rfind.man.rfind    1006000106  rfind Man Pages
I  rfind.sw           1006000106  rfind Software
I  rfind.sw.client    1006000106  rfind Client Command
   rfind.sw.server    1006000106  rfind Server Support (Only)
I  rfind               681523139  rfind Remote Fast Find 2.0
I  rfind.man          1007000023  rfind Man Pages
I  rfind.man.rfind    1007000023  rfind Man Pages
I  rfind.sw           1007000023  rfind Software
I  rfind.sw.client    1007000023  rfind Client Command
   rfind.sw.server    1007000023  rfind Server Support (Only)
```

then the new version of your product doesn't completely remove the old version of your product.

Corrective Action

`replaces` statements need to be added to the spec file that get rid of older versions and obsolete subsystems.

See Section 3.5 on additional information that goes into a spec file.

Problem

Files are missing from subsystems.

Meaning

Somewhere along the way, some part of the process failed.

Corrective Action

For each file that is missing, the things to check are:

1. There is an `install` line for that file in one of your Makefiles.
2. The `install` line includes the `-idb` flag and its argument includes an idb tag.
3. The idb tag is used in the spec file.
4. The `make install` command reached the directory that builds that file.
5. The files was successfully built.
6. The file is in the raw idb file.
7. The file is in the idb file.
8. The file doesn't have a `mach` tag for a machine other than the one you installed on.

Appendix C

Conventions for ID Strings in Spec Files

This appendix provides guidelines for the descriptions that appear in the right hand column of `versions` output and `inst` "list" output. The objective of the guidelines is to present descriptions that are consistent in appearance and contents across products.

There are guidelines for all ID strings and additional guidelines specific to products, images, and subsystems. They are presented in a do and don't checklist-type form with examples in each section. The left column in the Examples sections contains the product, image, and subsystem names and the right column contains ID strings that follow the guidelines.

Guidelines for all ID Strings:

Do:
- Limit string to 30 characters if possible.
- Use abbreviations if you need the space and they are standard, easily understood abbreviations.
- Use capitalization rules for titles (first letter of every word capitalized except prepositions and un-capitalized names).

Do not:
- Use punctuation.
- Include part numbers.
- Include Marketing Codes.

Examples:

eoe1	Execution Only Environment 1 4D1-4.0
eoe1.man	eoe1 Documentation
eoe1.man.relnotes	Workstation Release Notes
eoe1.man.unix	Basic UNIX Manual Pages
eoe1.sw	eoe1 Software
eoe1.sw.lib	Execution Library
eoe1.sw.quotas	BSD Disk Quotas
eoe1.sw.unix	Basic UNIX

Product ID Strings:

Do:
- Use something close to the Marketing Name of the product.
- Include the Silicon Graphics release number.

Do not:
- Use the words "option" or "version".
- Use the words "system" or "release" unless they are part of the name of the product (i.e. Maintenance Release and Network File System).

Examples:	showcase	IRIS Showcase 2.0
	nqs	Network Queueing System 2.0
	maint1	Maintenance Release 1 4D1-4.0.1

Image ID Strings:

Do: Use the product name and a description of the image (i.e. Software, Documentation, Manual Pages).

Do not: Include a version or release number.

Examples:	emacs.sw	emacs Software
	Xdev.man	Xdev Documentation

Subsystem ID Strings:

Do: Be as descriptive as you can.

Do not: Include the Marketing Name or product name.
Include a version or release number.

Examples:	eoe2.sw.demos	Graphics Demonstration Programs
	eoe2.sw.sysadm	System Administration Utilities
	nfs.man.nfs	NFS Support Manual Pages
	trans.man.relnotes	Laser Printer Release Notes

Appendix D

The idb Language

The idb language is a language that is used to create and manipulate idb files. It is used with the exp keyword in spec files, and with two tools, *idbedit*(1M) and *idbscan*(1M) which are not described in this document.

D.1 Variables and Data Types

Values are typed as integer or string, with non-zero integers and non-null strings being considered "true" in boolean tests. A reference to an idb attribute is "true" if the idb attribute is present in the database record being operated on. References to idb attribute arguments are made with a form of subscripting after the argument name, where a list of integers, or integers separated by ".." indicating a range, select specific arguments. The variable *argc* within brackets refers to the last argument. The selected arguments are concatenated with separating spaces and returned as a string value. (Note that the mechanisms of idb attribute reference just described are likely to change. They are not terribly useful as is.) Integer and string variables and constants are available, with single and double quotes being entirely equivalent around string constants.

D.2 Operators

The primary values may be combined with most of the usual operators, which behave as in C unless otherwise noted:

+ - * /	add, subtract, multiply, divide
=~ !~ // ::	pattern match, not match, substring, concatenation
& | ^ ~	bitwise and, or, exclusive or, (unary) not
&& || !	logical and, or, not
!= == <= >= < >	comparisons (on integers or strings)
? :	conditional
=	assignment
,	expression list

Parenthesis for grouping are also available.

D.3 Builtin Variables

type The file type as a one-character string; the first character of *file*, *directory*, *block device*, *character device*, (symbolic) *link*, or (named) *pipe* (i.e. fifo).

mode Permission bits. The type of this value is integer, but will be converted to a string according to context (though it will be a decimal integer, which is probably not what you want.)

owner The name of the owner. The uid is mapped through *etc/passwd*.

group The name of the group. The gid is mapped through *etc/group*.

dstpath The relative (to root) pathname of the file in the software product destination tree.

srcpath The relative pathname of the file in the source tree.

nattr The integer number of idb attributes associated with the record being operated on.

argc (Defined only within idb attribute argument list references.) The number of arguments for the current idb attribute.

sbase The pathname of the root of the source tree.

rbase The pathname of the root of the destination tree.

idb The pathname of the primary idb file.

When files are being accessed, the mapping between user and group integer ids and the *owner* and *group* string values in the idb are based on the **etc/passwd** and **etc/group** files, respectively. These are first sought under *sbase*, then under *rbase*, then under */*.

D.4 Builtin Functions

spath(s) Returns an absolute pathname for the argument; if the given value is relative, it is concatenated with the value of *sbase*. This is useful in converting a *srcpath* value into an absolute pathname.

rpath(s) Returns an absolute destination pathname, concatenated with the value of *rbase*.

putrec() Prints the current record in standard format (i.e. on one line, packed).

printf(f,a...)
Formatted print (subset of stdio printf). Recognizes field widths with leading zero pad indicator, types **%s**, **%d**, **%o**.

print(a...)
Unformatted print. Prints values as decimal integers or strings, separated by spaces, terminated with newline.

bytes(s) Returns the size, in bytes, of the given file, or -1 if not found.

blocks(s)
Returns the size, in blocks, of the given file, or -1 if not found.

access(s,m)
Returns the value of the *access*(2) system call.

D.5 Statements

The following statements are implemented as in C: *if [else], while, for, break, continue, return*, grouping with braces, and expressions.

Appendix E

The idb File

The idb file is the installation database, and it contains an entry for each file in your product that describes how the files in your distribution are to be installed. The idb file is the only place where you can find the complete distribution information. It is produced by the `gendist` command.

This appendix describes the contents of the idb file. It is nearly identical to the raw idb file, but it has additional information that is created by `gendist`.

The idb file contains one entry for each file, directory, device node, link, or fifo to be installed. Typically, each entry is one line long; however, some entries span several lines.

The idb entries are in the following form:

type mode owner group destination source subsystem idbattribute ...

Each field is separated by one space. The fields are as follows.

type The file types are: `f` (file), `d` (directory), `l` (link), and `p` (fifo).

mode This follows standard UNIX permission naming.

owner The owner of this file.

group The group owner of this file.

destination
This is the name of the file as it is to be installed on the destination system.

source This is the name of the file as it is found in the build tree

In addition to the idb attributes described in Section 3.3, the following idb attributes are added automatically when the file is run through `gendist`.

`cmpsize` The compressed size of the file.

`size` The uncompressed size of the file.

`checksum`
There are two checksum idb attributes, `sum` for a 16-bit machine, and `f` for a 32-bit machine.

If there is a complete entry for each file, directory, etc. in your distribution, the idb file is complete.

The following figure shows a few lines in the *rfind* idb file.

```
l 0000 bin bin etc/rc0.d/K98rfindd - rfind.sw.server symval(/etc/init.d/rfindd)
d 0755 root sys etc/rc2.d - rfind.sw.server
f 0644 rfindd nuucp usr/lib/rfindd/.forward usr/lib/rfindd/.forward rfind.sw.server sum(23288) size(11) f(478236391) co
f 0755 bin bin usr/lib/rfindd/passwd.add usr/lib/rfindd/passwd.add rfind.sw.server sum(36831) size(2911) f(959927454)
f 0755 bin bin usr/local/bin/rfind usr/local/bin/rfind rfind.sw.client sum(24964) size(94256) f(786271820) cmpsize(53763)
```

Figure E-1. Some Lines from the *rfind* idb File

Index

H

I

M

N

O

P

R

S

T

U

V

Engineer's Handbook
Makefile Type Stuff

May 5, 1994

SGI Makefile Conventions

access via: handbook makeconv
probable data location: dist.wpd:/sgi/doc/swdev/makeconv.ps

Building Manual Pages and Release Notes

Version 3.0, March 25, 1994

CONTRIBUTORS

Written by Susan Ellis
Edited by Christina Cary

Building Manual Pages and Release Notes
Version 3.0, March 25, 1994

Contents

Figures

Examples

Tables

Chapter 1

Introduction

"Where do I start?" and "Why doesn't it work?" are frequent questions when engineers, technical writers, and production editors at Silicon Graphics are asked to prepare manual pages and release notes. With this guide and the newest versions of the tools (now conveniently packaged so that they can be installed by *inst*(1M)) the hardest part of preparing manual pages and release notes will no longer be using the tools, it will be writing the words.

This guide covers the mechanics of manual pages and release notes: how to set up a *workarea*, where to put new manual pages in a *source tree*, how to format and print manual pages, how to format and print release notes, and how to specify installation information.

The audience for this guide is engineers, technical writers, production editors, and Marketing technical support people—anyone who has to work on manual pages or release notes. Because some members of this audience have never used *ptools* (SGI's revision control tools that are based on RCS) or use it very infrequently, basic ptools information is provided in an appendix.

This guide assumes that you, build meisters, and anyone else who builds your manual pages and release notes are using the *inst*-able manual page and release notes tools and macros (described in the section "Installing Required Subsystems" in Chapter 2). This guide also assumes that you are a C shell user. If you use a different shell, you'll need to substitute equivalent commands for your shell when necessary.

This guide is not a style guide for manual pages or release notes or a description of their contents, and it doesn't teach you how to use *troff*(1) commands or macros. Appendix C, "For More Information," lists sources for this information.

Although the official term is "reference page," this guide uses the more common term "manual page."

Most readers won't need this whole guide. Depending upon the amount of experience you've had and what you need to do, you can start at the beginning or use just the sections you need. The overall organization of this guide is:

- Chapter 1, "Introduction"
- Chapter 2, "Getting Ready"
- Chapter 3, "Building Manual Pages"
- Chapter 4, "Building Release Notes"
- Chapter 5, "Specifying IDB Information for Installation"
- Appendix A, "Troubleshooting"
- Appendix B, "Using Ptools"
- Appendix C, "For More Information"
- "Glossary"

Figure 1-1 gives an overview of the major steps of working on manual pages and release notes and gives the chapter or appendix where each major step is covered.

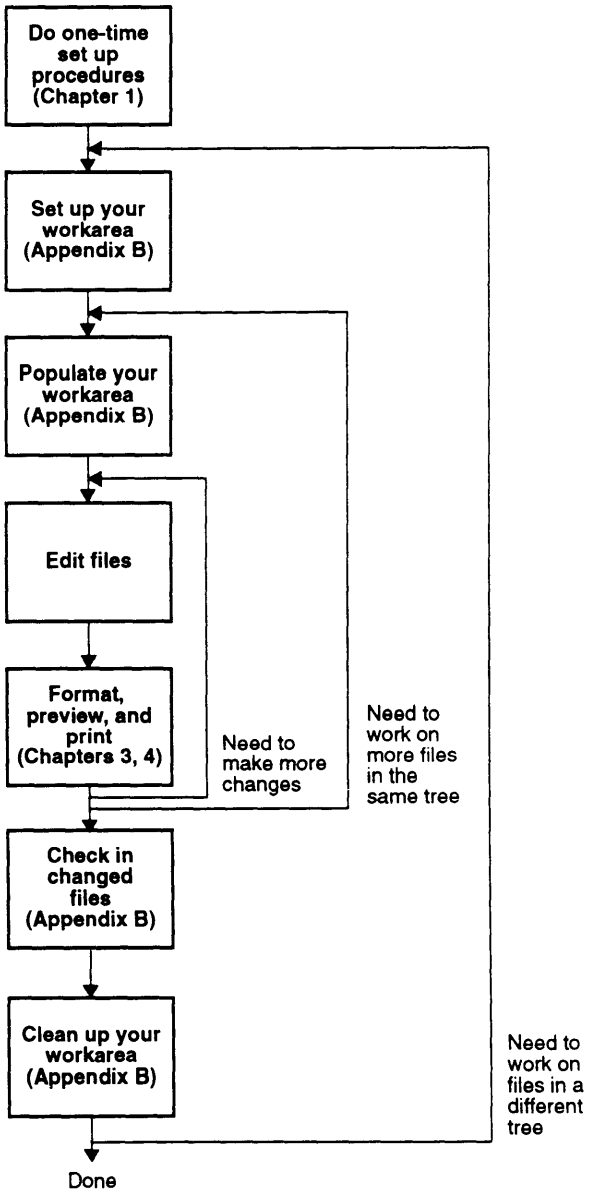


Figure 1-1 Manual Page and Release Notes Building Process

Chapter 2

Getting Ready

Note: Experienced ptools users can probably review Table 2-1 to make sure that you have all of the required subsystems installed and skip the remainder of this chapter.

This chapter explains how to get ready to work on manual pages or release notes. It assumes very little about how your system is set up. The sections in this chapter are:

- "Installing Required Subsystems"
- "Enabling Automount"
- "Setting Up Automatic Environment Variable Setting"
- "Creating and Populating a Workarea"

Installing Required Subsystems

This section lists subsystems that should be installed on your workstation if you plan to build manual pages and release notes, describes how to find out if these subsystems are already installed, outlines the basic installation sequence, and explains the steps required to complete installation.

Required Software

Building manual pages or release notes requires some software that you probably already have installed or mounted, for example NFS and ptools. Table 2-1 lists other required subsystems that are more specific to manual pages and release notes. It also lists where you install these subsystems from as of March 1994. This list applies to systems running IRIX 5.2 and later only.

Some subsystems must be installed relative to *$ROOT* or *$TOOLROOT* (by using the *inst* **-r** option). These are noted in Table 2-1 as well.

Table 2-1 Required Subsystems

Subsystem	Location	Install Relative To	Contents
build_root.sw.dev	*babylon.wpd:/usr/dist/ sherwood/ LATEST/root/build_root*	$ROOT	*/usr/include/make/commondefs*
dev.sw.make	*dist.wpd:/released/5.2/all*	/ or $TOOLROOT	*/usr/bin/smake*
dps_eoe.sw.dps	*dist.wpd:/released/5.2/all*	/ or $TOOLROOT	*xpsview*(1) (not needed if *ghostview*(1) is used instead)
dps_eoe.sw.dpsfonts	*dist.wpd:/released/5.2/all*	/	Fonts required for *xpsview* and *ghostview*
dwb.sw.dwb	*dist.wpd:/released/5.2/all*	/ or $TOOLROOT	*nroff*(1), *troff*, *eqn*(1), *neqn*(1), *tbl*(1), *pic*(1)
*ghost.sw.**	*fangio.asd:/dist/ghost*	/	*ghostview* (not needed if *xpsview* is used instead)
impr_base.sw.impr	*dist.wpd:/test/impressario/ALL*	/ or $TOOLROOT	*psdit*(1), *psroff*(1)
impr_fonts.sw.adobe22	*dist.wpd:/test/impressario/ALL*	/	Palatino fonts for previewing and printing
man_eoe.man.manpages	*babylon.wpd:/usr/dist/sherwood/ LATEST/eoe/man_eoe*	/	Manual pages for manual page and release notes tools
man_eoe.sw.doc	*babylon.wpd:/usr/dist/sherwood/ LATEST/eoe/man_eoe*	/	Manual page and release notes templates, reference list of *make*(1) targets, and this guide in PostScript
man_eoe.books.BuildMan	*babylon.wpd:/usr/dist/sherwood/ LATEST/eoe/man_eoe*	/	This guide in InSight format
man_root.sw.dev	*babylon.wpd:/usr/dist/sherwood/ LATEST/root/man_root*	$ROOT	Manual page and release notes macros and include files
man_toolroot.sw.tools	*babylon.wpd:/usr/dist/sherwood/ LATEST/toolroot/man_toolroot*	$TOOLROOT	Manual page and release notes tools
x_eoe.sw.xdps	*dist.wpd:/released/5.2/all*	/	Required for *xpsview* and *ghostview*

Checking to See If a Subsystem Is Already Installed

To check to see if a particular subsystem is already installed on your workstation, give this command:

versions -n *subsystem*

If the subsystem is installed, its name is included in the output. For example:

```
% versions -n man_root.sw.dev
...
I  man_root.sw.dev    1010854620  Man Page and Rel Notes ROOT
```

If you have installed any subsystems relative to $ROOT or $TOOLROOT with `inst -r $ROOT` or `inst -r $TOOLROOT`, you must give the **-r** option to the *versions*(1M) command, for example:

```
% versions -n -r $ROOT man_root.sw.dev
```

Installing Subsystems on IRIX 5.2 or Later

The basic installation sequence on a system running 5.2 or later is shown below (assuming the installation directories listed in Table 2-1). They assume that you set both $ROOT and $TOOLROOT and install relative to the first directory listed in Table 2-1.

```
su
inst -f dist.wpd:/released/5.2/all
Inst> keep *
Inst> install dev.sw.make dps_eoe.sw.dps dps_eoe.sw.dpsfonts
Inst> install dwb.sw.dwb x_eoe.sw.xdps
Inst> go
Inst> from dist:/test/impressario/ALL
Inst> keep *
Inst> install impr_base.sw.impr impr_fonts.sw.adobe22
Inst> go
Inst> from fangio.asd:/dist/ghost
Inst> install *
Inst> go
Inst> from babylon.wpd:/usr/dist/sherwood/LATEST/eoe/man_eoe
Inst> install *
Inst> go
Inst> quit
inst -r $ROOT -f \
```

```
babylon.wpd:/usr/dist/sherwood/LATEST/root/man_root
Inst> install *
Inst> go
Inst> quit
inst -r $TOOLROOT -f \
babylon.wpd:/usr/dist/sherwood/LATEST/tools/man_toolroot
Inst> install *
Inst> go
Inst> quit
exit
```

Give the *rehash csh(1)* command in each shell window you have open so that the new commands you just installed are recognized by the shell:

```
rehash
```

Enabling Automount

This section describes how to enable automount. Automount provides automatic NFS-mounting of directories from other workstations and is an alternative to specifying the directories to be mounted in */etc/fstab*. The instructions in the rest of this guide assume that you are using automount to mount source trees from other workstations. Using automount is not required; you can NFS-mount source trees in your */etc/fstab* file instead if you prefer.

First, check to see if automount is already enabled by giving this command:

```
chkconfig
```

If automount is enabled, the output includes this line:

```
automouunt              on
```

If automount is enabled, you can skip the rest of this section. If automount is not enabled, the line includes `off` instead of `on`. If you don't see a line that includes `automount` at all, then NFS is not properly installed or configured.

The commands for enabling automount are given below. They requires that you reboot your workstation, so you should prepare to reboot before you give these commands.

```
su
chkconfig automount on
mkdir /hosts
reboot
```

Automount needs to be enabled just once; if you install a new version of NFS, automount remains enabled.

Setting Up Automatic Environment Variable Setting

To build manual pages and release notes, the environment variables *WORKAREA, ROOT,* and *TOOLROOT* must be properly set. The values of the environment variables may be different for each manual page tree and set of release notes you work on. Setting them manually each time you work on a different tree is error-prone, so this section describes a procedure for setting up automatic setting of these environment variables.

1. Using your favorite editor, add these three lines to your *.cshrc* file (*~/.cshrc*):

   ```
   alias pushd 'pushd \!* && source ~/.set_workarea'
   alias popd  'popd  \!* && source ~/.set_workarea'
   alias cd    'cd    \!* && source ~/.set_workarea'
   ```

2. Create a new file in your home directory called *.set_workarea* and put these lines into it:

   ```
   set curdir='pwd'
   unsetenv WORKAREA
   setenv ROOT     /
   setenv TOOLROOT /
   ```

3. For each workarea, add these five lines to the end of *~/.set_workarea*:

   ```
   if ("$curdir" =~ workarea*) then
       setenv WORKAREA workarea
       setenv ROOT     root
       setenv TOOLROOT toolroot
   endif
   ```

   where *root* and *toolroot* are the values (pathnames) that you want ROOT and TOOLROOT to have, respectively. *workarea* must be a full pathname and cannot use ~ syntax. Be sure to use spaces exactly as shown in the `if` line.

4. Give these commands in each of your shell windows:

```
source ~/.cshrc
cd .
```

Each time you change directories to any directory that you use for manual page or release notes work, your new aliases for *cd*, *pushd*, and *popd* take care of setting environment variables for your new current directory. Each time you create a new workarea, edit *~/.set_workarea* as described in step 3.

Other techniques can be used instead of the one described above; however, the rest of this guide assumes you are using this technique.

Creating and Populating a Workarea

Before beginning to to work on manual pages or release notes, you need to create a workarea (if you don't already have a suitable one) and populate it (if you are editting files that already exist). If you need instructions, see Appendix B, "Using Ptools."

Chapter 3

Building Manual Pages

This chapter first describes the standard manual page building tasks in the next two sections:

- "Formatting and Previewing an Online Manual Page"
- "Formatting, Previewing, and Printing a Printed Manual Page"

Other tasks that you may need to do in special situations are described in the remaining sections:

- "Specifying Symbolic Links"
- "Updating an Old Makefile"
- "Converting Fonts in an Old Manual Page"
- "Creating a New Manual Page"
- "Deciding Where to Put a New Manual Page in the Source Tree"
- "Creating a New Makefile"
- "Creating Books of Manual Pages"

Formatting and Previewing an Online Manual Page

When you build the online version of a manual page, the manual page source gets formatted by *nroff*, compressed by *pack*(1), and stored in a file whose suffix is .z. Two make targets enable you to build an online manual page and to preview it:

make *name*.**z** Build a .z file for the manual page *name.section* in your current directory. You can preview the .z file with this command:

man -d *name*.**z**

make *name*.**m** Format the manual page *name.section* for online and display it with *man*(1). No .z file is created.

You'll get a line of output from *make* that looks like this:

`//usr/lib/doc/tools/mmdoc Op=null` *name.section* | *output_cmd*

mmdoc is a tool from *man_toolroot.sw.tools*. It creates an *nroff* command line and executes it. *output_cmd* is either a *pack* command or an approximation of the *man* command that is used to display the manual page.

Formatting, Previewing, and Printing a Printed Manual Page

When you build the printed version of a manual page, the manual page source gets formatted by *troff*. You can print the manual page, create a PostScript version, and you can preview the printed version online with your favorite PostScript previewer. For a manual page whose source file is *name.section*, the commands are:

make *name*.**p** Format and print *name.section* (no file is created; the last modification date of the source file appears at the bottom of the page).

make *name*.**xp** Format *name.section* and display it with *xpsview* (no PostScript file is created).

make *name*.**gv** Format *name.section* and display it with *ghostview* (no PostScript file is created).

make *name*.**ps** Format *name.section* and write the (PostScript) result to *name.ps*.

lp *name*.**ps** Print the PostScript file.

To print all of the manual pages in a directory, you can use the *csh* built-in command *foreach*:

```
foreach manpage (*.section)
? make $manpage:r.p
? end
```

To create PostScript files for each manual page, use `.ps` instead of `.p` in the *foreach* command above.

For the commands that use *make*, you'll get a line of output from *make* that looks like this:

`//usr/lib/doc/tools/mmdoc` *options* `Op=null` *name.section output_cmd*

mmdoc creates a *troff* command line and executes it. *options* are arguments that *mmdoc* uses to construct the command line and *output_cmd* specifies what to do with the output of the *troff* command.

Specifying Symbolic Links

When a manual page documents more than one item (i.e. there are two or more comma-separated names before the hyphen in the NAME section), you can give any of these names to the *man* command. This works because the command **make install** in a manual page source directory automatically creates symbolic links for the second and following names on a manual page to the formatted manual pages. You don't need to anything to specify that the symbolic links should be created.

By default, the symbolic link names are all lowercase. You can specify that the names have the same capitalization as the names in the NAME section by using the .upperok pseudo-macro. For example:

```
.SH NAME
.upperok
Name \- description
```

You can prevent the symbolic links from being created by using the .nolinks pseudo-macro. For example:

```
.SH NAME
.nolinks
name1, name2 \- description
```

Both pseudo-macros are described in detail in *sgiman*(5).

Updating an Old Makefile

Old-style manual page Makefiles have a line similar to this:

```
DEPTH = ../..
```

They assume that tools and macros are in your workarea, which is not the case when using *man_root* and *man_toolroot*. Example 3-1 shows an example of a basic old-style Makefile:

Example 3-1 Old, Unconverted Manual Page Makefile

```
#ident $Revision: 1.10 $
#
#
IDB_TAG=std.man.unix
IDB_PATH=/usr/share/catman/u_man/cat1

DEPTH=../..
include $(DEPTH)/mandefs
include $(DEPTH)/manrules

VERSION=
RELEASE=4.0.1
DATE=November 1991
MMFLAGS=-rs2
```

The steps to convert a Makefile to use the *man_** tools and macros are listed below. The section "Creating a New Makefile" later in this chapter gives more information about the Makefile variables mentioned in these steps.

1. Review the comments at the beginning of the Makefile. Add comments or fix up existing comments as necessary.
2. Remove the line that sets IDB_TAG the variable. (Chapter 5, "Specifying IDB Information for Installation," contains information on how to specify the subsystem that should include these manual pages.)
3. Remove the setting of DEPTH, NDEPTH, or other similarly-named variable.
4. Change the `include` lines so that they look like this:

   ```
   include $(ROOT)/usr/include/make/mandefs
   include $(ROOT)/usr/include/make/manrules
   ```

5. Remove any lines that set the variables VERSION, RELEASE, DATE, and MMFLAGS.
6. If the old Makefile sets the variable LANGOPT to a language, don't change the line. If it sets LANGOPT to nothing (`LANGOPT =`), change the line to:

   ```
   LANGOPT = null
   ```

   When LANGOPT is set, it must appear after the `include` lines. If LANGOPT is not included in an old Makefile, don't add it. For an alternative method of setting LANGOPT and dealing with multiple language manual pages, see the *gfx/man* tree.
7. If you want to use new options for headers and footers (an additional field in the header, an additional field in the footer, and/or odd/even headers and footers), add OPTHEADER, OPTFOOTER, and ODDEVEN to the Makefile as described in the section "Creating a New Makefile" later in this chapter. Most Makefiles should not set these variables.

Example 3-2 shows the Makefile in Example 3-1 converted to use the *man_** tools and macros.

Example 3-2 Old Manual Page Makefile after Conversion

```
#ident $Revision: 1.10 $
#
# Makefile for section 1 manual pages in eoe1 and eoe2.
#

IDB_PATH=/usr/share/catman/u_man/cat1

include $(ROOT)/usr/include/make/mandefs
include $(ROOT)/usr/include/make/manrules
```

Converting Fonts in an Old Manual Page

The default font for manual pages used to be Times, but now it's Palatino. Some manual pages explicitly request Times fonts and need to be changed. The Times fonts appear "squished" when compared to Palatino on a printed manual page. Table 3-1 shows the explicit Times requests and the Palatino requests that they should be converted to.

Table 3-1 Incorrect and Correct Font Change Requests

Times Requests (Incorrect)	Palatino Requests (Correct)	Font
`\fR`	`\f1`	Roman
`\fI`	`\f2`	Italic
`\fB`	`\f3`	Bold
`.ft R`	`.ft 1`	Roman
`.ft I`	`.ft 2`	Italic
`.ft B`	`.ft 3`	Bold

An easy way to convert the explicit requests for Times fonts to requests for font positions 1, 2, and 3 is to use the *cvtfonts*(1) program included in *man_toolroot.sw.tools*.

Using *cvtfonts* to convert fonts to Palatino is easy—you don't need to figure out which files need converting and the tool can run *p_modify*(1) automatically. In a directory that contains manual page source files, give the command:

```
$TOOLROOT/usr/local/bin/cvtfonts -m *.section
```

where *section* is the source files' section number suffix. The **-m** option specifies that *p_modify* should be run on un-writable files that need modification. By default, the file is modified. If the **-o** option is given, the modified version is written to standard output instead. No changes are made to files that do not contain requests for Times fonts.

The *cvtfonts*(1) manual page included in *man_eoe.man.manpages* gives more details about *cvtfonts*.

Creating a New Manual Page

If you are creating a manual page from scratch, start by copying a similar manual page or a manual page template (from *$ROOT/usr/lib/doc/doc/man_templates*) to your workarea. See the next section, "Deciding Where to Put a New Manual Page in the Source Tree," for information about what to name the file and where to put it.

Edit the manual page as necessary. The content, format, and style of manual pages are described in some of the references listed in Appendix C, "For More Information."

Deciding Where to Put a New Manual Page in the Source Tree

This section covers these issues in deciding where to put the source file for a new manual page:

- What section should the manual page be in?
- What should the file name of the manual page be?
- What directory should the manual page source be in?

Manual Page Sections

Most of the manual page sections in use at SGI today are listed in Table 3-2 with a brief description of each one. To choose the correct section, review the descriptions below to see which one most closely matches your manual page and look at some of the manual pages in a section you are considering to see if your page is like them. Sometimes it's best to just ask someone who knows more about what's in each manual page section than you do!

Table 3-2 Manual Page Sections

Section	Content
1	Commands of general utility.
1C	Commands for communication with other systems.
1D	Demos.
1EXP	Explorer commands.
1G	Graphics utilities.
1L	Local SGI tools.
1M	Commands for system maintenance and administration.
1V	Video commands.
1X	Motif Window Manager, toolchest, and user interface compiler.
2	System calls.
3	CD, DAT, printer, compression, VideoCreator, filesystem, hardware inventory, and signal set manipulation routines.
3A	Audio library routines.
3B	4.3BSD system calls and library routines.
3C	These routines (plus those in Section 2 and Section 3S) constitute the Standard C Library.
3C++	C++ versions of some Math Library and other routines.
3E	Explorer routines.
3F	Fortran routines.

Table 3-2 (continued) Manual Page Sections

Section	Content
3G	The IRIS Graphics Library.
3iv	Inventor routines.
3L	The Sphere Library.
3M	Math Library routines.
3mv	Movie Library routines.
3N	The 4.3BSD Internet network library.
3P	POSIX parallel programming primitives.
3R	Remote Procedure Call routines.
3S	The Standard I/O Library.
3T	Parallel processing library.
3t	Terminal interface routines.
3V	Indigo Video Library.
3W	Font Manager routines.
3X	X library routines and various specialized libraries.
3x	ViewKit routines.
3X11	X11 library routines.
3Xt	Xt library routines.
3Y	Remote Procedure Call and NIS support routines.
4	File format descriptions.
4E	Explorer definitions.
4F	FlexFAX database and format descriptions.
5	Miscellaneous facilities such as macro packages, character set tables, etc.
6D	Demos and image tools.

Table 3-2 (continued) Manual Page Sections

Section	Content
7	Descriptions of special files (hardware peripherals and device drivers) and STREAMS stuff.
7F	Protocol family descriptions.
7M	Descriptions of controllers and drivers.
7P	Protocol descriptions.
8	ToolTalk topics.

Manual Page File Names

The name of your manual page should follow the current conventions. Manual page source file names have this form:

name.*section*

where:

name is the first (or only) name listed before the \- in the line after the .SH NAME line. When there is more than one page with a particular name in a section (case is ignored), append an underscore (_) and an explanatory suffix. For example, *mail* and *Mail* become *mail_att* and *mail_bsd*. The possible suffixes are: *_att*, *_bsd*, *_mips*, and *_sun*.

section is a section number from the first column of Table 3-2 except that section number letters, if any, are always lower case. For example, *bbox2* in section 3G is *bbox2.3g*.

Source Tree Manual Page Directories

This section contains information about currently used directory structures for manual page source files that should help you figure out where to put new manual pages.

Beginning with IRIX 5.0, manual page source is distributed across the source tree so that it is closer to the software is documents. Figure 3-1 shows some examples in the tree *bonnie:/proj/irix5.2/isms*.

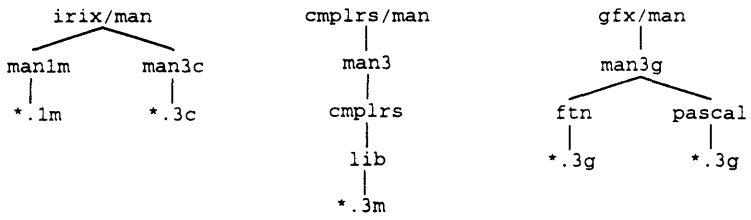


Figure 3-1 Manual Page Source Tree Hierarchies

Each manual page directory hierarchy usually follows these rules:

- There is a directory called *man* at the "top" of the manual page hierarchy.
- There is a directory called *manXX* where *XX* stands for a section number (from Table 3-2 except with lower case letters) somewhere in the hierarchy between each manual page source file and the *man* directory.

Creating a New Makefile

There are two types of Makefiles that you may have to create:

- Makefiles in directories that contain manual page source files
- Makefiles that are in a manual page tree, but contain directories rather than source files

Creating the second type of Makefile is simple:

1. Copy the file *$(ROOT)/usr/lib/doc/doc/manpages/Makefile.top* to a file called *Makefile* in the correct directory in your workarea.
2. Edit the SUBDIRS variable in *Makefile* so that it contains the names of the subdirectories in this directory.

Follow these steps to create a new Makefile in a manual page source directory:

1. Copy the file *$ROOT/usr/lib/doc/doc/man_templates/Makefile* to the correct directory in your workarea.
2. Edit the variables in *Makefile*. The variables are:

   IDB_PATH — This variable is set to the full path name of the directory that the manual pages in this directory should be installed in on users' workstations. This variable is not used if this information is specified in an IDB file. See Chapter 5, "Specifying IDB Information for Installation," for more information.

   LANGOPT — This variable should be set when the .Op macro is used and language-specific manual pages are desired or when the manual page source is from Atria Software, Inc. The possible values of LANGOPT are `C` (C++), `a` (Ada), `c` (C), `f` (Fortran), `p` (Pascal), and `atria` (Atria Software, Inc.). When the .Op macro is not used or when all language-specific information is included, LANGOPT should be commented out or set to `null`.

   OPTHEADER — This variable specifies a string to be printed in the header of each manual page in the directory. For example,

   ```
   OPTHEADER = CASEVision/ClearCase
   ```

   Most manual pages should not contain this optional header field.

   OPTFOOTER — This variable specifies a string to be printed in the footer of each manual page in the directory. For example,

   ```
   OPTFOOTER = Beta Draft
   ```

   Most manual pages should not contain this optional footer field.

ODDEVEN	This variable controls the type of headers and footers on printed manual pages (only one type of header and footer is used in online manual pages). When ODDEVEN is not set, or is set to `-rt1`, all pages have the same headers and footers. When ODDEVEN is set to `-rt2`, odd and even pages have different headers and footers.
PAGENUM	This variable is used when preparing printed manual pages that are sequentially numbered. When this variable is set to `-n`*N*, each printed page starts at page *N*. This variable has no effect on online manual pages.

Example 3-3 shows an example of a new Makefile.

Example 3-3 Example Makefile for a Manual Page Source Directory

```
#ident  "$Revision: 1.10 $"

#
# Makefile for a manual page source directory.
# See $(ROOT)/usr/lib/doc/doc/make_targets for a list of available make
# targets.
#

#       IDB_PATH is the directory that (formatted) manual pages in this
#       directory are installed in.
IDB_PATH=/usr/catman/u_man/cat1

include $(ROOT)/usr/include/make/mandefs
include $(ROOT)/usr/include/make/manrules

#       To build language-specific versions of manual pages (source must use
#       the .Op macro), uncomment LANGOPT and replace null with a language
#       (C, a, c, f, or p).  When building manual pages from Atria, uncomment
#       LANGOPT and replace null with atria.
#LANGOPT = null

#       To specify a string to be included in each manual page header, set
#       OPTHEADER to the string (no quoting required).  Normally, this
#       variable is not set.
#OPTHEADER =

#       To specify a string to be included in each manual page footer, set
#       OPTFOOTER to the string (no quoting required).  Normally, this
#       variable is not set.
#OPTFOOTER =

#       To build printed manual pages that have headers and footers formatted
#       for odd and even pages, set ODDEVEN to -rt2.  For generic headers
#       and footers, don't set ODDEVEN at all, or set it to -rt1.
#ODDEVEN = -rt1

#       To build a printed manual page with a starting page number other
#       than 1, set PAGENUM to -nN where N is the starting page number.
#PAGENUM = -n1
```

Creating Books of Manual Pages

By default, each manual page begins on page one. To print a set of manual pages that are sequentially numbered, use these techniques instead:

- Concatenate the files (pages) to be printed in order into a single, temporary file. The suffix of the temporary file name should be "*.n*" where *n* is any valid section number (it doesn't have to be the same as the section number(s) of the pages).
- If any of the files use *eqn*, *tbl*, or *pic*, put the two special comment lines at the beginning of the temporary file (see *sgiman*(5)).
- To do get headers and footers that are set up for even and odd pages, set the variable ODDEVEN to "-rt2" in the Makefile.
- To specify the page number for the first page, set the variable PAGENUM in the Makefile. Be sure to comment out PAGENUM when you are done.
- To specify a string to appear in each page header, set the variable OPTHEADER.
- To specify a string to appear in each page footer, set the variable OPTFOOTER.

Building Release Notes

This chapter describes various procedures that you might need while working on release notes. The sections are:

- "Formatting, Previewing, and Printing Release Notes"
- "Converting an Old Set of Release Notes"
- "Deciding Where to Put New Release Notes"
- "Using the Release Notes Templates for New Release Notes"

Formatting, Previewing, and Printing Release Notes

Many Makefile targets are available for release notes. Just give the *make* commands shown below.

To format files for printing and preview them with *xpsview* or *ghostview*:

`make front.xp`
Format the front pages—cover and credits—and display them with *xpsview*.

`make contents.xp`
Format the table of contents, list of figures, and list of tables, and display them with *xpsview*.

`make ch`*x*`.xp` Format chapter *x* or appendix *x* and display it with *xpsview*.

`make front.gv` Format the front pages—cover and credits—and display them with *ghostview*.

`make contents.gv`
Format the table of contents, list of figures, and list of tables, and display them with *ghostview*.

make ch*x*.gv Format chapter *x* or appendix *x* and display it with *ghostview.*

make book.gv Format the entire manual and display it with *ghostview* (due to bugs in *xpsview,* an *xpsview* analog is not available).

To print release notes (and format them if they haven't been formatted already):

make *name*.**p** Print one or all files where *name* is `front`, `contents`, ch*x*, or `book`.

To format the printed release notes and create PostScript (*.ps*) files (*.ps* files are created automatically by the commands above):

make *name*.**ps**
Create PostScript files where *name* is `front`, `contents`, ch*x*, or `book`.

To create the online version of the release notes, use these commands:

make Format the online version of the release notes (uses *nroff* instead of *troff* and different macros; doesn't include front pages or table of contents) compress them, and create *.z* files.

make default Same as `make`.

make install Make the online release notes as above and install them into */usr/relnotes.*

make relnotes
Make the online release notes as above and display them with *more*(1) or whatever tool $MANPAGER is set to.

A target for errata sheets is available, too:

make errata.*suffix*
Make an errata sheet from *errata.x.* The suffixes are: `.xp` (display with *xpsview*), `.gv` (display with *ghostview*), `.p` (print), and `.ps` (create PostScript file).

There are three additional useful targets:

make spell Spell check all release notes files with *spell*(1).

make check	Check all of the release notes files to see if any of them still contain words in double angle brackets.
make clean	Remove all generated files except *.ps* and *.z*.
make clobber	Remove all files that can be recreated.

Converting an Old Set of Release Notes

To use the tools and macros in the *man_** ISMs for release notes, you may have to do a little conversion work. You can tell if release notes have already been converted by looking at the Makefile. A converted Makefile has this include line.

```
include $(ROOT)/usr/include/make/relnotesrules
```

The tasks are:

1. Update the Makefile.

   Replace the old Makefile with a copy of the template Makefile from *$ROOT/usr/lib/doc/doc/rel_templates*. Edit the template as described in step 4 of the section "Using the Release Notes Templates for New Release Notes" in this chapter.

2. Update the cover sheet and credits pages.

   Copy *front.x* from *$ROOT/usr/lib/doc/doc/rel_templates* and fill in the "blanks" as instructed in the comments. The old *cover.x* and *credits.x* files won't be used in the new version of the release notes so remove them from the source tree as described in the section "Removing Files From the Source Tree" in Appendix B.

3. Convert *Product.macros* if present.

   Some release notes used a file called *Product.macros* that contained string definitions. To continue to use that file, put this line near the beginning of each chapter and appendix file that relies on those string definitions:

   ```
   .so Product.macros
   ```

   There are two alternatives you might want to use instead:

   - You can replace calls to the strings with the actual text. The calls have the form *(*xx* where *xx* is one of Pc, PD, Pn, RN, or Ve.

- You can move the string definitions from *Product.macros* to *front.x* and delete the file *Product.macros*.

4. Update the font change commands in the chapters and appendices.

   Old release notes may inadvertently specify Times fonts rather than Palatino fonts. You can quickly fix them with the program *$ROOT/usr/local/bin/cvtfonts*. See the section "Converting Fonts in an Old Manual Page" in Chapter 3 for more information.

Deciding Where to Put New Release Notes

Release notes source is usually placed in a directory called *relnotes*, which is usually either a sibling of a *man* directory or a a subdirectory of *man*. Figure 4-1 shows an example of the directory structure for Fortran release notes.

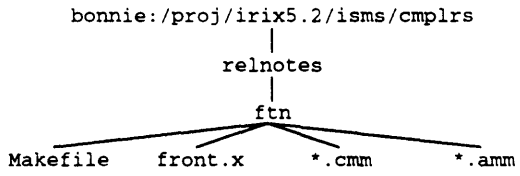


Figure 4-1 Release Notes Hierarchy for Fortran

Using the Release Notes Templates for New Release Notes

The subsystem *man_eoe.sw.doc* contains a full set of release notes templates and a README file that describes the files. It also contains a template file for an errata sheet for a manual. These files are installed in *$ROOT/usr/lib/doc/doc/rel_templates*.

The basic procedure for developing a new set of release notes is:

1. Set up a workarea if you don't already have one (see the section "Setting Up a Workarea" in Appendix B).
2. Copy the files in *$ROOT/usr/lib/doc/doc/rel_templates* to the appropriate directory in your workarea (see the section "Deciding Where to Put New Release Notes" in this chapter).
3. Delete any files that you don't need for your release notes. For example, release notes often don't include appendices (**.amm* files) and you don't need *errata.x* when you are doing release notes.
4. Edit the Makefile:
   - Edit the lists of *.cmm* and *.amm* files to match the chapters and appendices in your release notes.
   - Edit the setting of RELPROD so that it is set to *product*, as in *product.image.subsystem.*

   Example 4-1 shows an example of a release notes Makefile.

Example 4-1 Release Notes Makefile Example

```
# ident $Revision: 1.10 $
#
# Makefile for NFS release notes.
# See $(ROOT)/usr/lib/doc/doc/make_targets for a list of available make
# targets.
#

#       Edit the CMM and AMM macros to contain the list of chapters and
#       appendices in this book.  Do not use abbreviations (i.e. ch[1-8].cmm).
CMM     = ch1.cmm ch2.cmm ch3.cmm ch4.cmm ch5.cmm ch6.cmm
AMM     =

#       Set RELPROD to the short installation name for your product --
#       your release notes subsystem will be called RELPROD.man.relnotes
RELPROD = nfs

include $(ROOT)/usr/include/make/relnotesrules
```

5. Edit the file *front.x* according to the directions in the file.
6. Edit the chapter files.

7. Preview and print the release notes as necessary (see the section "Formatting, Previewing, and Printing Release Notes" in this chapter).
8. Use ptools commands as necessary during your development to check in your source files. See Appendix B, "Using Ptools," for information about ptools commands.

Chapter 5

Specifying IDB Information for Installation

Two pieces of information must be specified in order for your manual pages or release notes to be included in the software distribution for a product and installed properly on users' systems:

- The name of the subsystem that the manual page or release notes belong in.
- The directory where the manual page or release notes should be installed.

The sections in this chapter provide the information you need to choose and specify the subsystem and installation directory. The sections are:

- "Conventions for Subsystem Names"
- "Directories for Installed Manual Pages"
- "Directories for Installed Release Notes"
- "Specifying Subsystems and Installation Directories"

Conventions for Subsystem Names

By convention:

- Release notes for a product are put into a subsystem called *product.man.relnotes* ("man" and "relnotes" are literal). It contains only release notes.
- Manual pages are put into subsystems called *product.man.subsystem* ("man" is literal). They contain only manual pages.
- No manual pages are included in subsystems that contain software.

Subsystem names have this form:

product . image . subsystem

product is not a name that you pick; ask others on your project what *product* you should use if you don't know.

If there is just one manual page subsystem for *product*, then *subsystem* is often the same as *product* (*product.man.product*). If there is to be more than one manual page subsystem, ideally the manual page subsystems correspond to software (*product.sw.subsystem*) subsystems. In this case, *subsystem* matches the software *subsystem* name for example *eoe2.sw.X11* and *eoe2.man.X11*. If there isn't a close correspondence between manual page and software subsystems, choose descriptive *subsystem* names.

Directories for Installed Manual Pages

Figure 5-1 shows the directories that formatted online manual pages (.z files) should be installed in.

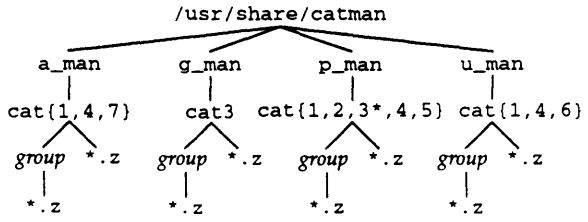


Figure 5-1 Directory Hierarchy for Installed Manual Pages

The key points of Figure 5-1 to note are:

- All released manual pages go in */usr/share/catman*.
- */usr/share/catman* is divided, for historical reasons, into *a_man*, *g_man*, *p_man*, and *u_man*: administrative manual pages, graphics manual pages, programmers' manual pages, and users' manual pages, respectively.

- Because subsection distinctions (3c *vs* 3m, for instance) are sometimes lost name collisions can result. To prevent name collisions, groups of manual pages are sometimes put in a subdirectory called *group. group* is any descriptive word, but it cannot contain periods (.).

If you know a little about the way the *man* command finds manual pages, you know that by default the *man* command looks in other locations as well as */usr/share/catman* for manual pages. These locations are not recommended for the following reasons:

- */usr/man* is not recommended because, by convention, it is for unformatted manual pages (manual page source files). Making unformatted manual pages available to users is not recommended because it opens several cans of worms: macros, tools, fonts, and licensing. Discussion of these issues is beyond the scope of this guide.
- */usr/catman* is the "old" directory for installed manual pages. Starting with IRIX 5.0, */usr/share/catman* has replaced */usr/catman*.
- */usr/share/catman/local* is not recommended because of the widespread practice of NFS-mounting manual pages. Manual pages for internal stuff, which might logically be installed there is often not installed on manual page servers. Since users who mount */usr/share/catman* can't install into */usr/share/catman/local* on the server or on their workstation, they are stuck. Use */usr/local/catman* instead.

Directories for Installed Release Notes

Release notes files are always installed in a directory called */usr/relnotes/productname. productname* is a name you pick (*/usr/relnotes* is literal) and is the argument that users give to the *relnotes*(1) or *grelnotes*(1) commands when they want to view your release notes. Usually *productname* is the same as *product* (see the section "Conventions for Subsystem Names" in this chapter), but sometimes it's an abbreviated version of *product* that is formed by truncating *product* at an underscore.

Specifying Subsystems and Installation Directories

To specify the subsystem and installation directory for manual pages or release notes files, follow these steps:

1. Set the variable IDB_PATH (in the Makefile that contains the manual page or release notes source files) to the full path name of the directory that these manual pages should be installed in on users' workstations.

   An example of the setting of IDB_PATH in a Makefile is:

   ```
   IDB_PATH=/usr/share/catman/a_man/cat1
   ```

2. Create or modify an IDB file. An IDB file is named *idb* or *idb.man* and is in a directory called *build* that is somewhere in your source tree (probably close to the top of the hierarchy for your product).

   Each file in a subsystem must have a line in the IDB file. A complete discussion of the line is beyond the scope of this guide, but for manual pages and release notes the syntax for each file is typically:

   `f 444 root sys` *install_path source_tree_path idb_tag*

   where:

   install_path is the directory that the manual page or release notes file should be installed in on a user's workstation, but without an initial `/`.

   source_tree_path is the path name of the formatted, compressed version (.z file) of the manual page or release notes file in the source tree. The path name is relative to the parent of the *build* directory.

   idb_tag is the IDB tag. The IDB tag is a name that specifies which subsystem the manual page should be included in. The name can be the subsystem name or a synonym for the subsystem name. More information about IDB tags appears in the *Engineer's Guide to Packaging Software for inst* (to get more information about this guide, see Appendix C, "For More Information").

3. In the spec file for your product, *build/spec*, include one or more manual page subsystems and specify the *idb_tags* that are included in each subsystem. For more information, see the *Engineer's Guide to Packaging Software for inst*.

Troubleshooting

This appendix lists some common error messages and explains what to do about them.

No RCS

```
Error # 7 EXECV_FAILED
```

eoe2.sw.rcs is not installed. Install it (see the section "Installing Required Subsystems" in Chapter 2) to solve the problem.

WORKAREA not set

```
p_tupdate: ERROR 99 -> The location of the workarea couldn't
be determined.
p_tupdate: ERROR WORKAREA environment variable is not set
p_tupdate: ERROR 142 -> The workarea is not setup correctly.
```

Your WORKAREA environment variable is not set. See "Setting Up Automatic Environment Variable Setting" in Chapter 2 for one technique.

Census Database is Empty

```
p_tupdate: WARNING The census database is empty.
workarea/census
```

This message appears the first time you give a *p_tupdate*(1) command in a workarea. You can safely ignore it.

Source Tree Problems

```
p_tupdate: WARNING The census database is empty.
workarea/census
p_tupdate: The source tree location specified does not exist
or is not readable link_dir/
p_tupdate: ERROR 201 -> The source machine initialization of
the source_tree failed
```

This group of messages appears when *src_tree* is not exported for auto-mounting. If this is the case, the command **`ls /hosts/src_host/src_tree`** prints the error message `src_host:src_tree: No such file or directory`. This problem needs to be fixed by the system administrator for *src_host*.

```
p_tupdate: ERROR 201 -> The source machine initialization of
the source_tree failed
```

This message appears when network problems prevent your workstation from accessing *src_host*, or your *.workarea* file is incorrect. To confirm that accessing *src_host* is the problem, use the *ping*(1M) command:

```
ping -c 5 src_host
```

If your workstation is successfully accessing *src_host*, you'll get about eight lines of output immediately and a prompt. If output doesn't appear fairly quickly and/or you get an error message, you know you have a problem.

If you can access *src_host*, examine *.workarea* and verify that everything is correct.

,v File Doesn't Exist

```
p_tupdate: ERROR 17 -> RCS ,v file doesn't exist for file file
```

file doesn't exist or it exists, but there is no RCS history for it.

Couldn't Load Shell

```
Couldn't load Shell: No such file or directory
```

make wants to use *$TOOLROOT/bin/sh*, but it doesn't exist. You can work around this problem by creating a link:

```
mkdir $TOOLROOT/bin
ln -s /bin/sh $TOOLROOT/bin
```

Troff Can't Open File

troff: Can't open /usr/lib/font/devpsc/*x*.out; line *n*, file /usr/tmp/mmdoc*nnnnn*

This message is caused by an incorrect font change specification. The most likely cause is a typo in your manual page or a release notes file. You typed \f*x* when you meant to type something like \f2*x* or \fP*x*. To find your error search for "\f*x*" where *x* is the character that appears in the error message. The line number *n* given in the message is meaningless and the file */usr/tmp/mmdocnnnnn* has been deleted by the time you see the message.

troff: can't open file //usr/lib/doc/macros/origin.*name*; line *n*, file /usr/tmp/mmdoc*nnn*

The manual page you are trying to format contains a line of the form:

```
.so /pubs/tools/origin.name
```

This line is a left-over from a period when SGI manual pages included an ORIGIN section. The line should be removed.

Appendix B

Using Ptools

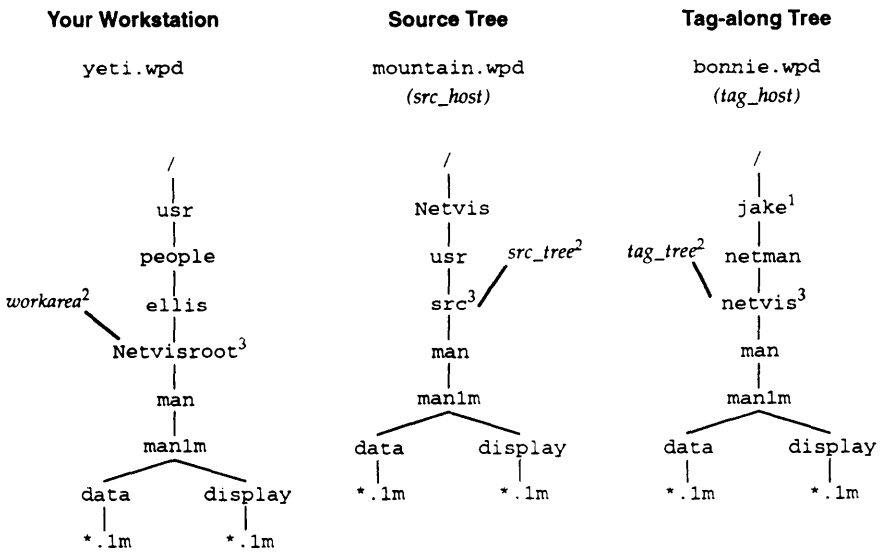
This appendix explains how to use *ptools*. It contains these sections:

- "Setting Up a Workarea"
- "Populating Your Workarea"
- "Making Files Writable"
- "Refreshing Your Workarea and Integrating Your Changes with Others' Changes"
- "Checking in Changed Files"
- "Cleaning Up Your Workarea"
- "Other Useful Ptools Commands"
- "Removing Files From the Source Tree"

Setting Up a Workarea

A *workarea* is a directory that you use for all of your changes to a particular *source tree*. In some cases, you may be able to use this workarea to make identical changes to a second source tree, called a tag-along tree, as well. The revision control tools used at SGI are called ptools and they enable you to copy files from the source tree to your workarea and back. You can install ptools from *dist.wpd:/sgi/ptools/new*. The subsystem you need is *ptools.sw.ptools_local*.

Figure B-1 shows an example directory hierarchy on a source tree, the corresponding hierarchy on a tagalong tree, and some directories on a user's workstation. These hierarchies are used in the examples in this section.



[1]This is a fictional hierarchy for illustration purposes.

[2]Use full pathname for variable.

[3]Directory hierarchies below this point must be equivalent for directories of interest to you.

Figure B-1 Workarea, Source Tree, and Tag-along Tree Directory Hierarchies

To set up a workarea, follow these steps:

1. Find out the location of the source tree you want to modify, *src_host:src_tree*. In the example in Figure B-1, the NetVisualyzer source tree is located at *mountain.wpd:/Netvis/usr/src*.
2. Find out the name of the newsgroup that you are to post your *TAKE* messages to.
3. Check to see if you have an account on *src_host* by giving this command on your workstation:

   rsh *src_host* **date**

You should see one line of output that contains the date, for example:

```
Tue Nov 10 11:35:19 PST 1992
```

If the date is not displayed, you don't have an account and you should skip to step 7.

4. If the output from step 3 contained more than the date line, your *.cshrc* file on *src_host* displays text when it is read by the shell and must be modified so that it doesn't display anything. An example of the procedure is:

   rlogin *src_host*
   vi .cshrc
   modify the file so there is no output, save the file, and exit
   logout

5. To verify that you have a *.rhosts* file on *src_host* that enables you to access *src_host* without supplying a password, give this command on your workstation:

   rsh *src_host* **cat .rhosts**

   If the output is either one of these lines:

   ```
   +
   ```

   ```
   + userid
   ```

   where *userid* is your login name, then you have access to *src_host* without a password, and you should skip to step 8.

6. To give yourself access to *src_host* without supplying a password, edit or create the file *src_host:~userid/.rhosts* with your favorite editor. For example:

   rlogin *src_host*
   vi .rhosts

   Add this line:

   ```
   + userid
   ```

   Save the file, exit the editor, logout from *src_host*, and skip to step 8.

7. Run *nacct* on *src_host* to set up an account for yourself (*userid*) on *src_host*:

   rsh guest@*src_host* **/usr/local/bin/nacct** *userid*

   If this command fails in any way, contact the system administrator of *src_host* for help in setting up an account on *src_host*. Repeat step 3 to verify that you have a valid account.

8. Start *p_setup*(1) to begin creating your workarea directory and two files, *.workarea* and *census*:

   p_setup

9. Choose `Create a new ptools workarea` from the menu:

   `Please enter your choice -> 1`

10. Choose `Modify source tree selection`:

    `Please enter your choice -> 2`

11. Choose `Specify a source machine and tree not in this list` from the menu:

    `Please enter your choice -> 1`

12. Specify *src_host* and *src_tree*:

    `Specify a source machine and tree:` *src_host*`:`*src_tree*

    For the example shown in Figure B-1, *src_host*:*src_tree* is *mountain.wpd:/Netvis/usr/src*.

13. Look at this line of output:

    `New workarea will be create at` *default_workarea*

    If you do not like the directory name chosen by ptools for your workarea directory, *default_workarea*, (*/usr/people/ellis/Netvisroot* in this example) specify a different directory name this way:

    `Please enter your choice -> 3`

    `Enter location for the new ptools workarea`
    `[`*default_workarea*`]` *workarea*

    where *workarea* is a full pathname. You can use ~ syntax to abbreviate your home directory.

14. Choose `Create workarea now using settings shown above` from the menu:

    ```
    Please enter your choice -> 1
    ```

15. Exit *p_setup*:

    ```
    Please enter your choice -> Q
    ```

16. Open the new *.workarea* file for editing, for example:

    vi ~/*workarea*/.workarea

    The file contains four resource settings and some comments.

    ```
    workarea.sm_machine : src_host
    workarea.sm_location : src_tree
    workarea.sm_transport_method : nfs_ro
    workarea.sm_nfs_mount_point : src_tree
    ```

17. Change the setting of workarea.sm_nfs_mount_point so that it looks like this:

    ```
    workarea.sm_nfs_mount_point : /hosts/src_host/src_tree
    ```

18. Add this line to the end of *.workarea*:

    census_db.output_format : 2

    This resource setting specifies that you want relative path names in the *census file* rather than absolute pathnames. If *src_tree* changes in the future, it will be easier to update your workarea with this setting.

19. Add these lines to the end of *.workarea*:

    p_bugpost.bug_groups : 1
    p_bugpost.bug_groups.1 : *newsgroup*

    where *newsgroup* is the newsgroup that you are supposed to post TAKE messages to (from step 2 in this section).

20. If you want to modify a tagalong tree, *tag_host:tag_tree*, as well as the source tree, add these two lines to the end of *.workarea*:

    ```
    p_finalize.option.t : on
    ```
    p_finalize.option.t.value : *tag_host:tag_tree*

    In the example, *tag_host*:*tag_tree* is *bonnie.wpd:/jake/netman/netvis*.

21. Save *.workarea* and exit the editor. For the example shown in Figure B-1, *.workarea* now contains:

```
# All these resources are described in the p_resources(1) man page

# machine = name of source machine (required)
workarea.sm_machine : mountain.wpd

# sm_location = top of source tree you'll be working with (required)
workarea.sm_location : /Netvis/usr/src

# sm_transport_method = how to access source tree from this machine (optional)
workarea.sm_transport_method : nfs_ro

# sm_nfs_mount_point = where to access source tree from this machine (optional)
workarea.sm_nfs_mount_point : /hosts/mountain.wpd/Netvis/usr/src

census_db.output_format : 2
p_bugpost.bug_groups : 1
p_bugpost.bug_groups.1 : sgi.bugs.netvis
p_finalize.option.t : on
p_finalize.option.t.value : bonnie.wpd:/jake/netman/netvis
```

Populating Your Workarea

When you populate your workarea, you put copies of files in the source tree (or just symbolic links to files) into your workarea in a directory structure that matches the directory structure of the source tree. You can copy as many or as few files from the source tree as you want. You need to populate your workarea only if you are modifying files that are already in the source tree; this step is not done when you create new files.

To populate your work area by copying files, use the *p_tupdate* command. To populate your workarea by linking files, use *p_link*(1). Some examples of useful *p_tupdate* and *p_link* commands are:

p_tupdate Populate your current directory with copies of the most recently checked in versions of all files in the directory on the source tree equivalent to your current directory, including files in source tree subdirectories.

p_link Same as **p_tupdate** except symbolic links instead of copies are made (this saves disk space and is useful when you know that you won't be modifying files).

p_tupdate *file ...*
Get a copy of one or more specific files from the source tree.

p_link *file ...*
Same as **p_tupdate** *file* except a symbolic link instead of a copy is made (this saves disk space and is useful when you know that you won't be modifying files).

p_tupdate -m Populate your current directory as with **p_tupdate**, and put each of the files into *modify state* (make them writable) in preparation for updating. (You don't need to give the *p_modify* command described in the next section, "Making Files Writable," when you give this command.)

p_tupdate -m *file ...*
Get a copy of one or more files and put them in modify state. (You don't need to give the *p_modify* command described in the next section, "Making Files Writable," when you give this command.)

Making Files Writable

If you copy files to your workarea using *p_link* or *p_tupdate* without the **-m** option, the copies are not writable. Use the *p_modify* command to make a file writable:

p_modify *file*

Once a file is writable (also known as "in *modify state*"), you can edit it using your favorite text editor. Later, after you've completed and checked your changes, you should check the changed file into the source tree with *p_finalize*(1) (see the section "Checking in Changed Files" in this appendix).

Refreshing Your Workarea and Integrating Your Changes with Others' Changes

Ptools makes it possible for several people to work on a source file simultaneously. One result of this is that other people may check new versions of files that you have copies of into the source tree. If you made symbolic links with *p_link*, you see the new version automatically. If you use *p_tupdate*, your copy is not updated and you must *refresh your workarea*. When you refresh your workarea, you get new copies of changed files.

You can find out about changes by reading the appropriate newsgroup or by periodically giving ptools commands to check for changes. Normally, you check for changes and incorporate any changes with one command. Some ptools commands that check for changes and incorporate them are:

p_tupdate Get copies of new files that have been added to the source tree (same as the *p_tupdate* command without arguments described in the section "Populating Your Workarea" in this appendix) and, for files already in your current directory and below, update them with new versions if your copy is not in modify state.

p_tupdate *file* ...
Check for changes to one or more specific files (they can't be in modify state).

p_integrate -m
Check for changes to all files that are in modify state and integrate any changes (see below for a discussion of integration).

p_integrate -m *file* ...
Check for changes to specific files that are in modify state. Integrate the changes.

The *p_integrate* **-m** command merges the changes the were made to the source tree version of a file with the changes you have made and leaves the file in modify state. It merges the changes on a line-by-line basis. If it finds conflicts—lines that have been changed by both you and the person who checked changes into the source tree—it marks the overlapping changes with the strings `<<<<<<<`, `=======`, and `>>>>>>>`. A copy of your file before the merge is always saved in the directory */usr/tmp/pbackup*.

Checking in Changed Files

When you are ready to check your changes into the source tree (and a tag-along tree if you specified it when you set up your workarea), give this command:

p_finalize -M *file* . . .

You'll be asked to provide a log message. Enter a short description of the changes you made. You'll also be asked if you want to post your changes to the newsgroup you specified when you set up the workarea. In most cases you should post your changes. An example of the dialog with *p_finalize* is:

```
p_finalize: Enter a message for man/Makefile
p_finalize: (~h for editor options)
Fixed typos in comments and added "doc" to SUBDIRS.
p_finalize: Message Complete
p_finalize: Putting file man/Makefile into jake.wpd:/jake/irix
Do you wish to post these changes to the bug system? (y or n) [y] y
Reply to file: [none]
p_finalize brings up an editor with a skeleton message you can edit
edit the subject and add a message and /usr/tmp/final.yourlogin
save the file and exit the editor when you are done
[S]end, [A]bort, [E]dit, [L]ist, [H]elp: [s] s
Remove /usr/tmp/final.ellis (y or n) [n] y
```

Two additional options of *p_finalize* can be used to speed up the checkin process:

-m *"message"*	Use *message* as the message for each file and don't prompt for messages.
-B	Do not post changes to the bug system.

If someone else has checked in changes to a file since you made your copy and you haven't integrated those changes (see the section "Refreshing Your Workarea and Integrating Your Changes with Others' Changes" in this appendix), *p_finalize* **-M** refuses to complete your check in. In this case you should follow these steps:

1. Check to see how your file is different from the version checked into the source tree:

   p_rdiff *file*

2. Give this command to integrate your changes with the source tree changes:

   p_integrate *file*

3. Check for any occurrences of the strings `<<<<<<<`, `=======`, and `>>>>>>>`. These strings mark changes that could not be integrated automatically. You must integrate these lines yourself.

4. Do any final checking of the merged file that you want to do.

5. Check in the file with this command:

   p_finalize *file*

Cleaning Up Your Workarea

There are several types of "clean up" of your workarea that you might want to do:

- Remove generated files.
- Remove files that you've copied or linked from the source tree, but not changed.
- Remove your whole workarea.

Commands that clean up a workarea are:

make clean Remove all generated files from a directory except .z files.

make clobber Remove all generated files from a directory. Generated files include .*ps* files and .z files.

p_purge *directory*
: Remove all files in *directory* that are copies of files in the source tree and not in modify state. *directory* is also removed if it becomes empty. For each file, *p_purge*(1) asks you to confirm that you want it removed. This command is more useful than simply using *rm*(1) to remove files when you are finished with a project because it tells you which files are still in modify state or another state. This indicates that you might have forgotten to check in some changes.

rm -rf *workarea*
: Remove a workarea.

Other Useful Ptools Commands

Additional ptools commands that you may find useful are listed below. See the *ptools*(1) manual page and the manual pages for individual ptools commands for more information.

p_rlog *file* — Display the log messages and dates of all revisions to *file*.

p_rdiff *file* — Display the differences between the version of *file* in your workarea and the version of *file* in the source tree in *diff*(1) format.

p_rdiff -g *file*
: Display the differences between the version of *file* in your workarea and the version of *file* in the source tree in *gdiff*(1) format.

p_error *errornumber*
: Display additional information about a ptools error.

p_state *file* ... — Report the state of *file*. The possible states are fetal, trunk, link, modify, and integrate.

p_check -w — Report the names of files in your workarea that were not copied from the source tree. This is useful when you want to quickly find out what new files you've added to the workarea and when you want to quickly identify the files that were created as a result of giving *make* commands.

p_purge *file* Remove the copy of the source file *file* from your workarea. This command fails if *file* is in modify state or has been locked.

p_integrate *file ...*
Lock files in that are already in modify state. This prevents anyone else from checking a new version into the source tree. Use *p_finalize* without the **-M** option to check in a locked file.

p_unintegrate *file*
Undo a *p_integrate*(1) command. This unlocks the file on the source tree. No changes are made to your copy of the file.

p_unmodify *file*
Undo a *p_modify* command. This makes *file* unwritable and takes it out of modify state. A copy of *file* is put in the directory */usr/tmp/pbackup*.

Removing Files From the Source Tree

When files on the source tree become obsolete (that file is no longer part of the source tree) the readable version of the file, say *file*, and its RCS history, *RCS/file,v* need to be "removed" from the source tree. The recommended way to "remove" a file from the source tree is to hide its RCS history in a directory called *old* and remove the readable file. For example:

File	Becomes
file	deleted
RCS/file,v	*RCS/old/file,v*

File permissions on the source tree may not allow you to make these changes yourself. If you can't make these changes, you'll need to contact someone who has root permission for your source tree and ask them to make the changes. It's easiest if you provide them with a list of commands that they can cut and paste, for example:

```
cd directory_on_src_host
rm file
mv RCS/file,v RCS/old/file,v
```

Appendix C

For More Information

Other resources for information about manual pages and release notes include the manual pages, books, and internal documentation listed below.

Manual Pages

Manual pages that contain useful information for manual page and release notes writers are:

- *man*(5)

  man(5) describes macros that can be used in manual page source files. It describes the manual page macros that ship with SGI's DWB software option which are not exactly the same macros as SGI uses to format the manual pages we ship with our products. The *sgiman*(5) manual page describes the differences.

- *sgiman*(5)

  sgiman(5) describes the differences between the manual page macros described in *man*(5) and the manual page macros used at SGI to build released manual pages.

- *mm*(5)

  mm(5) describes the mm macros, which can be used in release notes source files.

- *intro*(1)

  This manual page describes the syntax of the SYNOPSIS section of manual pages.

- *eqn*(1), *nroff*(1), *pic*(1), *psroff*(1), *tbl*(1), *troff*(1)

  These DWB commands format manual pages and release notes.

- *cvtfonts*(1)

  cvtfomts(1) describes the *cvtfonts* program that is included in *man_eoe.man.manpages*. This program is used to convert requests for Times fonts, if any, in manual page or release notes source to requests for Palatino fonts.

- *ghostview*(1), *xpsview*(1)

  These viewers are used to preview the printed versions of manual pages and release notes online.

- *man*(1), *gman*(1), *xman*(1)

  These manual pages describe several tools that can be used to display online manual pages.

- *relnotes*(1), *grelnotes*(1)

  These commands display online release notes.

- *ptools*(1), *p_tupdate*(1), *p_modify*(1), *p_finalize*(1), etc.

  These manual pages describe ptools in general and specific ptools commands.

- *ismtools*(1L)

  This manual page explains the tools and procedures for creating new ISMs.

Books

Books that contain useful information for manual page and release notes writers are:

- UNIX System V Documenter's Workbench Software Release 2.0 Technical Discussion and Reference Manual

  This DWB book from AT&T contains references guides for *nroff, troff, tbl* and the mm macros.

- UNIX System V Documentor's Workbench Software Release 2.0 User's Guide

  This DWB book from AT&T contains tutorials for *nroff, troff, tbl, eqn, pic* and the mm macros.

SGI Internal Documentation

SGI internal documents that contain useful information for manual page and release notes writers are:

- *Engineer's Guide to Packaging Software for inst*

  This internal SGI document, available via the *handbook*(1L) command, explains all the details of creating software distributions.

- Ptools Class Notes

  The ptools class notes document is available from the *handbook*(1L) command.

Glossary

census file

Each workarea directory, *workarea*, contains a census file. It lists the files that you have copied from the source tree and their current state. It is created and maintained by ptools commands.

modify state

Each file you copy from a source tree to your workarea always has a current state. In modify state, your copy of a file is writable, and others may check new versions of that file into the source tree. When you use ptools to copy a file from a source tree to your workarea, by default your copy is initially mode 444 (not writable or executable called trunk state). To make a file writable, put it into modify state by making the initial copy with `p_tupdate -m` or by giving the command `p_modify`. Files can be in several other states (link and integrate for instance), but most of the time when you are working on manual pages and release notes your files are in modify state.

ptools

Ptools is a set of revision control tools that enable people to get copies of files from source trees and check in modifications to files. The tool names start with "p_" and are installed in the directory */usr/local/bin/ptools*. The tools are designed so that several people can work on a file simultaneously with only brief periods when files are locked and others are prevented from checking in changes (this is called integrate state).

refresh your workarea

When you refresh your workarea, you get new versions (if any) of files from the source tree. For files that are not in modify state, use the *p_tupdate* command to refresh your workarea. For files that are in modify state, you must use the *p_integrate* command. *p_integrate* automatically merges changes made on the source tree with changes you have made.

ROOT

ROOT is an environment variable whose value is set to a directory. This directory is the parent of directories that contain macro files and include files. ROOT makes it possible to have one set of macros and include files installed in the default locations on your workstation, say */usr/lib*, and a different set of the same tools installed in a second location, say */d2/usr/lib*. ROOT is typically prepended to a full pathname when it is used in a Makefile, so if ROOT is /, you would get the tools in *//usr/lib* (*/usr/lib*) and if TOOLROOT is /d2, you would get the tools in */d2/usr/lib*.

source tree

A source tree is a directory hierarchy on a particular workstation that contains the source for a particular software release (either already released or to be released in the future). People use ptools to copy (check in) new versions of files to source trees. Released software is built from the files in a source tree.

TAKE

A TAKE (or TAKE message) is a posting to a newsgroup that informs others that you have modified one or more files in the source tree. The message also closes the bug report that prompted the changes. TAKE messages are semi-automatically created when you check in source files.

TOOLROOT

TOOLROOT is an environment variable whose value is set to a directory. This directory is the parent of subdirectories that contain build tools on your workstation. TOOLROOT makes it possible to have one set of tools installed in the default locations on your workstation, say */usr/sbin*, and a different set of the same tools installed in a second location, say */d2/usr/sbin*. TOOLROOT is typically prepended to the standard path of a tool when it is used in a Makefile, so if TOOLROOT is /, you would get the tools in *//usr/sbin* (*/usr/sbin*) and if TOOLROOT is /d2, you would get the tools in */d2/usr/sbin*.

workarea

A workarea is a directory in your home directory whose name typically ends in "root". This directory is where you do development that will eventually be checked into a particular source tree. You create a workarea using *p_setup*. It contains two special files called *census* and *.workarea*. In a workarea, you create a directory hierarchy that matches a portion of the directory hierarchy of a source tree. While working in a workarea, the environment variables

WORKAREA, ROOT, and TOOLROOT must be set properly.

.workarea

A *.workarea* file is created in a workarea directory by *p_setup* and is required for correct operation of the ptools commands. It contains information about the location of the source tree for this workarea, how you access the source tree, and your preferences for various ptools characteristics.

WORKAREA

WORKAREA is an environment variable whose value is set to the pathname of a workarea directory. While you are working in a workarea, the WORKAREA environment variable must be set to the full pathname of the workarea directory.