```
************************************************************
   24661 Fri May 17 18:03:04 2013
new/usr/src/pkg/Makefile
3664 pkg/Makefile openssl logic should be removed
************************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

#
# Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
#

include $(SRC)/Makefile.master
include $(SRC)/Makefile.buildnum

#
# Make sure we're getting a consistent execution environment for the
# embedded scripts.
#
SHELL= /usr/bin/ksh93

#
# To suppress package dependency generation on any system, regardless
# of how it was installed, set SUPPRESSPKGDEP=true in the build
# environment.
#
SUPPRESSPKGDEP= false

#
# Comment this line out or set "PKGDEBUG=" in your build environment
# to get more verbose output from the make processes in usr/src/pkg
#
PKGDEBUG= @

#
# Cross platform packaging notes
#
# By default, we package the proto area from the same architecture as
# the packaging build.  In other words, if you're running nightly or
# bldenv on an x86 platform, it will take objects from the x86 proto
# area and use them to create x86 repositories.
#
# If you want to create repositories for an architecture that's
# different from $(uname -p), you do so by setting PKGMACH in your
# build environment.
#
# For this to work correctly, the following must all happen:
#
```

```
#   1. You need the desired proto area, which you can get either by
#      doing a gatekeeper-style build with the -U option to
#      nightly(1), or by using rsync.  If you don't do this, you will
#      get packaging failures building all packages, because pkgsend
#      is unable to find the required binaries.
#   2. You need the desired tools proto area, which you can get in the
#      same ways as the normal proto area.  If you don't do this, you
#      will get packaging failures building onbld, because pkgsend is
#      unable to find the tools binaries.
#   3. The remainder of this Makefile should never refer directly to
#      $(MACH).  Instead, $(PKGMACH) should be used whenever an
#      architecture-specific path or token is needed.  If this is done
#      incorrectly, then packaging will fail, and you will see the
#      value of $(uname -p) instead of the value of $(PKGMACH) in the
#      commands that fail.
#   4. Each time a rule in this Makefile invokes $(MAKE), it should
#      pass PKGMACH=$(PKGMACH) explicitly on the command line.  If
#      this is done incorrectly, then packaging will fail, and you
#      will see the value of $(uname -p) instead of the value of
#      $(PKGMACH) in the commands that fail.
#
# Refer also to the convenience targets defined later in this
# Makefile.
#
PKGMACH=        $(MACH)

#
# ROOT, TOOLS_PROTO, and PKGARCHIVE should be set by nightly or
# bldenv.  These macros translate them into terms of $PKGMACH, instead
# of $ARCH.
#
PKGROOT.cmd=    print $(ROOT) | sed -e s:/root_$(MACH):/root_$(PKGMACH):
PKGROOT=        $(PKGROOT.cmd:sh)
TOOLSROOT.cmd=  print $(TOOLS_PROTO) | sed -e s:/root_$(MACH):/root_$(PKGMACH):
TOOLSROOT=      $(TOOLSROOT.cmd:sh)
PKGDEST.cmd=    print $(PKGARCHIVE) | sed -e s:/$(MACH)/:/$(PKGMACH)/:
PKGDEST=        $(PKGDEST.cmd:sh)

EXCEPTIONS= packaging

PKGMOGRIFY= pkgmogrify

#
# Always build the redistributable repository, but only build the
# nonredistributable bits if we have access to closed source.
#
# Some objects that result from the closed build are still
# redistributable, and should be packaged as part of an open-only
# build.  Access to those objects is provided via the closed-bins
# tarball.  See usr/src/tools/scripts/bindrop.sh for details.
#
REPOS= redist

#
# The packages directory will contain the processed manifests as
# direct build targets and subdirectories for package metadata extracted
# incidentally during manifest processing.
#
# Nothing underneath $(PDIR) should ever be managed by SCM.
#
PDIR= packages.$(PKGMACH)

#
# The tools proto must be specified for dependency generation.
# Publication from the tools proto area is managed in the
# publication rule.
```

```
#
$(PDIR)/developer-build-onbld.dep:= PKGROOT= $(TOOLSROOT)

PKGPUBLISHER= $(PKGPUBLISHER_REDIST)

#
# To get these defaults, manifests should simply refer to $(PKGVERS).
#
PKGVERS_COMPONENT= 0.$(RELEASE)
PKGVERS_BUILTON= $(RELEASE)
PKGVERS_BRANCH= 0.$(ONNV_BUILDNUM)
PKGVERS= $(PKGVERS_COMPONENT),$(PKGVERS_BUILTON)-$(PKGVERS_BRANCH)

#
# The ARCH32 and ARCH64 macros are used in the manifests to express
# architecture-specific subdirectories in the installation paths
# for isaexec'd commands.
#
# We can't simply use $(MACH32) and $(MACH64) here, because they're
# only defined for the build architecture.  To do cross-platform
# packaging, we need both values.
#
i386_ARCH32= i86
sparc_ARCH32= sparcv7
i386_ARCH64= amd64
sparc_ARCH64= sparcv9

OPENSSL =       /usr/bin/openssl
OPENSSL10.cmd = $(OPENSSL) version | $(NAWK) '{if($$2<1){print "\043";}}'
OPENSSL10_ONLY  =       $(OPENSSL10.cmd:sh)

#
# macros and transforms needed by pkgmogrify
#
# If you append to this list using target-specific assignments (:=),
# be very careful that the targets are of the form $(PDIR)/pkgname.  If
# you use a higher level target, or a package list, you'll trigger a
# complete reprocessing of all manifests because they'll fail command
# dependency checking.
#
PM_TRANSFORMS= common_actions publish restart_fmri facets defaults \
        extract_metadata
PM_INC= transforms manifests

PKGMOG_DEFINES= \
        i386_ONLY=$(POUND_SIGN) \
        sparc_ONLY=$(POUND_SIGN) \
        OPENSSL10_ONLY=$(OPENSSL10_ONLY) \
        $(PKGMACH)_ONLY= \
        ARCH=$(PKGMACH) \
        ARCH32=$($(PKGMACH)_ARCH32) \
        ARCH64=$($(PKGMACH)_ARCH64) \
        PKGVERS_COMPONENT=$(PKGVERS_COMPONENT) \
        PKGVERS_BUILTON=$(PKGVERS_BUILTON) \
        PKGVERS_BRANCH=$(PKGVERS_BRANCH) \
        PKGVERS=$(PKGVERS)

PKGDEP_TOKENS_i386= \
        'PLATFORM=i86hvm' \
        'PLATFORM=i86pc' \
        'PLATFORM=i86xpv' \
        'ISALIST=amd64' \
        'ISALIST=i386'
PKGDEP_TOKENS_sparc= \
        'PLATFORM=sun4u' \
        'PLATFORM=sun4v' \
```

```
        'ISALIST=sparcv9' \
        'ISALIST=sparc'
PKGDEP_TOKENS= $(PKGDEP_TOKENS_$(PKGMACH))

#
# The package lists are generated with $(PKGDEP_TYPE) as their
# dependency types, so that they can be included by either an
# incorporation or a group package.
#
$(PDIR)/osnet-redist.mog := PKGDEP_TYPE= require
$(PDIR)/osnet-incorporation.mog:= PKGDEP_TYPE= incorporate

PKGDEP_INCORP= \
        depend fmri=consolidation/osnet/osnet-incorporation type=require

#
# All packaging build products should go into $(PDIR), so they don't
# need to be included separately in CLOBBERFILES.
#
CLOBBERFILES= $(PDIR) proto_list_$(PKGMACH)

#
# By default, PKGS will list all manifests.  To build and/or publish a
# subset of packages, override this on the command line or in the
# build environment and then reference (implicitly or explicitly) the all
# or install targets.
#
MANIFESTS :sh= (cd manifests; print *.mf)
PKGS= $(MANIFESTS:%.mf=%)
DEP_PKGS= $(PKGS:%=$(PDIR)/%.dep)
PROC_PKGS= $(PKGS:%=$(PDIR)/%.mog)

#
# Track the synthetic manifests separately so we can properly express
# build rules and dependencies.  The synthetic and real packages use
# different sets of transforms and macros for pkgmogrify.
#
SYNTH_PKGS= osnet-incorporation osnet-redist
DEP_SYNTH_PKGS= $(SYNTH_PKGS:%=$(PDIR)/%.dep)
PROC_SYNTH_PKGS= $(SYNTH_PKGS:%=$(PDIR)/%.mog)

#
# Root of pkg image to use for dependency resolution
# Normally / on the machine used to build the binaries
#
PKGDEP_RESOLVE_IMAGE = /

#
# For each package, we determine the target repository based on
# manifest-embedded metadata.  Because we make that determination on
# the fly, the publication target cannot be expressed as a
# subdirectory inside the unknown-by-the-makefile target repository.
#
# In order to limit the target set to real files in known locations,
# we use a ".pub" file in $(PDIR) for each processed manifest, regardless
# of content or target repository.
#
PUB_PKGS= $(SYNTH_PKGS:%=$(PDIR)/%.pub) $(PKGS:%=$(PDIR)/%.pub)

#
# Any given repository- and status-specific package list may be empty,
# but we can only determine that dynamically, so we always generate all
# lists for each repository we're building.
#
# The meanings of each package status are as follows:
#
```

```
#       PKGSTAT         meaning
#       ----------      ----------------------------------------------------
#       noincorp        Do not include in incorporation or group package
#       obsolete        Include in incorporation, but not group package
#       renamed         Include in incorporation, but not group package
#       current         Include in incorporation and group package
#
# Since the semantics of the "noincorp" package status dictate that
# such packages are not included in the incorporation or group packages,
# there is no need to build noincorp package lists.
#
PKGLISTS= \
        $(REPOS:%=$(PDIR)/packages.%.current) \
        $(REPOS:%=$(PDIR)/packages.%.renamed) \
        $(REPOS:%=$(PDIR)/packages.%.obsolete)

.KEEP_STATE:

.PARALLEL: $(PKGS) $(PROC_PKGS) $(DEP_PKGS) \
        $(PROC_SYNTH_PKGS) $(DEP_SYNTH_PKGS) $(PUB_PKGS)

#
# For a single manifest, the dependency chain looks like this:
#
#       raw manifest (mypkg.mf)
#               |
#               | use pkgmogrify to process raw manifest
#               |
#       processed manifest (mypkg.mog)
#               |
#          *    | use pkgdepend generate to generate dependencies
#               |
#       manifest with TBD dependencies (mypkg.dep)
#               |
#          %    | use pkgdepend resolve to resolve dependencies
#               |
#       manifest with dependencies resolved (mypkg.res)
#               |
#               | use pkgsend to publish the package
#               |
#       placeholder to indicate successful publication (mypkg.pub)
#
# * This may be suppressed via SUPPRESSPKGDEP.  The resulting
#   packages will install correctly, but care must be taken to
#   install all dependencies, because pkg will not have the input
#   it needs to determine this automatically.
#
# % This is included in this diagram to make the picture complete, but
#   this is a point of synchronization in the build process.
#   Dependency resolution is actually done once on the entire set of
#   manifests, not on a per-package basis.
#
# The full dependency chain for generating everything that needs to be
# published, without actually publishing it, looks like this:
#
#       processed synthetic packages
#               |               |
#       package lists       synthetic package manifests
#               |
#       processed real packages
#           |           |
#       package dir     real package manifests
#
# Here, each item is a set of real or synthetic packages.  For this
# portion of the build, no reference is made to the proto area.  It is
# therefore suitable for the "all" target, as opposed to "install."
```

```
#
# Since each of these steps is expressed explicitly, "all" need only
# depend on the head of the chain.
#
# From the end of manifest processing, the publication dependency
# chain looks like this:
#
#               repository metadata (catalogs and search indices)
#                       |
#                       | pkg.depotd
#                       |
#               published packages
#                |              |
#                |              | pkgsend publish
#                |              |
#         repositories      resolved dependencies
#                |                      |
# pkgsend        |                      | pkgdepend resolve
# create-repository                     |
#                |              generated dependencies
#        repo directories               |
#                                       | pkgdepend
#                                       |
#                               processed manifests
#

ALL_TARGETS= $(PROC_SYNTH_PKGS) proto_list_$(PKGMACH)

all: $(ALL_TARGETS)

#
# This will build the directory to contain the processed manifests
# and the metadata symlinks.
#
$(PDIR):
        @print "Creating $(@)"
        $(PKGDEBUG)$(INS.dir)

#
# This rule resolves dependencies across all published manifests.
#
# We shouldn't have to ignore the error from pkgdepend, but until
# 16012 and its dependencies are resolved, pkgdepend will always exit
# with an error.
#
$(PDIR)/gendeps: $(DEP_SYNTH_PKGS) $(DEP_PKGS)
        -$(PKGDEBUG)if [ "$(SUPPRESSPKGDEP)" = "true" ]; then \
                print "Suppressing dependency resolution"; \
                for p in $(DEP_PKGS:%.dep=%); do \
                        $(CP) $$p.dep $$p.res; \
                done; \
        else \
                print "Resolving dependencies"; \
                pkgdepend -R $(PKGDEP_RESOLVE_IMAGE) resolve \
                    -m $(DEP_SYNTH_PKGS) $(DEP_PKGS); \
                for p in $(DEP_SYNTH_PKGS:%.dep=%) $(DEP_PKGS:%.dep=%); do \
                        if [ "$$(print $$p.metadata.*)" = \
                             "$$(print $$p.metadata.noincorp.*)" ]; \
                        then \
                                print "Removing dependency versions from $$p"; \
                                $(PKGMOGRIFY) $(PKGMOG_VERBOSE) \
                                    -O $$p.res -I transforms \
                                    strip_versions $$p.dep.res; \
                                $(RM) $$p.dep.res; \
                        else \
                                $(MV) $$p.dep.res $$p.res; \
                        fi; \
                done; \
        fi
        $(PKGDEBUG)$(TOUCH) $(@)

install: $(ALL_TARGETS) repository-metadata

repository-metadata: publish_pkgs
        @print "Creating repository metadata"
        $(PKGDEBUG)for r in $(REPOS); do \
                /usr/lib/pkg.depotd -d $(PKGDEST)/repo.$$r \
                        --add-content --exit-ready; \
        done

#
# Since we create zero-length processed manifests for a graceful abort
# from pkgmogrify, we need to detect that here and make no effort to
# publish the package.
#
# For all other packages, we publish them regardless of status.  We
# derive the target repository as a component of the metadata-derived
# symlink for each package.
#
publish_pkgs: $(REPOS:%=$(PKGDEST)/repo.%) $(PDIR)/gendeps .WAIT $(PUB_PKGS)

#
# Before publishing, we want to pull the license files from $CODEMGR_WS
# into the proto area.  This allows us to NOT pass $SRC (or
# $CODEMGR_WS) as a basedir for publication.
#
$(PUB_PKGS): stage-licenses

#
# Initialize the empty on-disk repositories
#
$(REPOS:%=$(PKGDEST)/repo.%):
        @print "Initializing $(@F)"
        $(PKGDEBUG)$(INS.dir)
        $(PKGDEBUG)pkgsend -s file://$(@) create-repository \
                --set-property publisher.prefix=$(PKGPUBLISHER)

#
# rule to process real manifests
#
# To allow redistributability and package status to change, we must
# remove not only the actual build target (the processed manifest), but
# also the incidental ones (the metadata-derived symlinks).
#
# If pkgmogrify exits cleanly but fails to create the specified output
# file, it means that it encountered an abort directive.  That means
# that this package should not be published for this particular build
# environment.  Since we can't prune such packages from $(PKGS)
# retroactively, we need to create an empty target file to keep make
# from trying to rebuild it every time.  For these empty targets, we
# do not create metadata symlinks.
#
# Automatic dependency resolution to files is also done at this phase of
# processing.  The skipped packages are skipped due to existing bugs
# in pkgdepend.
#
# The incorporation dependency is tricky: it needs to go into all
# current and renamed manifests (ie all incorporated packages), but we
# don't know which those are until after we run pkgmogrify.  So
# instead of expressing it as a transform, we tack it on ex post facto.
#
# Implementation notes:
```

```
#
# - The first $(RM) must not match other manifests, or we'll run into
#   race conditions with parallel manifest processing.
#
# - The make macros [ie $(MACRO)] are evaluated when the makefile is
#   read in, and will result in a fixed, macro-expanded rule for each
#   target enumerated in $(PROC_PKGS).
#
# - The shell variables (ie $$VAR) are assigned on the fly, as the rule
#   is executed.  The results may only be referenced in the shell in
#   which they are assigned, so from the perspective of make, all code
#   that needs these variables needs to be part of the same line of
#   code.  Hence the use of command separators and line continuation
#   characters.
#
# - The extract_metadata transforms are designed to spit out shell
#   variable assignments to stdout.  Those are published to the
#   .vars temporary files, and then used as input to the eval
#   statement.  This is done in stages specifically so that pkgmogrify
#   can signal failure if the manifest has a syntactic or other error.
#   The eval statement should begin with the default values, and the
#   output from pkgmogrify (if any) should be in the form of a
#   variable assignment to override those defaults.
#
# - When this rule completes execution, it must leave an updated
#   target file ($@) in place, or make will reprocess the package
#   every time it encounters it as a dependency.  Hence the "touch"
#   statement to ensure that the target is created, even when
#   pkgmogrify encounters an abort in the publish transforms.
#

.SUFFIXES: .mf .mog .dep .res .pub

$(PDIR)/%.mog: manifests/%.mf
        @print "Processing manifest $(<F)"
        @env PKGFMT_OUTPUT=v1 pkgfmt -c $<
        $(PKGDEBUG)$(RM) $(@) $(@:%.mog=%.nodepend) \
                $(@:%.mog=%.lics) $(PDIR)/$(@F:%.mog=%).metadata.* $(@).vars
        $(PKGDEBUG)$(PKGMOGRIFY) $(PKGMOG_VERBOSE) $(PM_INC:%= -I %) \
                $(PKGMOG_DEFINES:%=-D %) -P $(@).vars -O $(@) \
                $(<) $(PM_TRANSFORMS)
        $(PKGDEBUG)eval REPO=redist PKGSTAT=current NODEPEND=$(SUPPRESSPKGDEP) \
                `$(CAT) -s $(@).vars`; \
        if [ -f $(@) ]; then \
                if [ "$$NODEPEND" != "false" ]; then \
                         $(TOUCH) $(@:%.mog=%.nodepend); \
                fi; \
                $(LN) -s $(@F) \
                        $(PDIR)/$(@F:%.mog=%).metadata.$$PKGSTAT.$$REPO; \
                if [ \( "$$PKGSTAT" = "current" \) -o \
                     \( "$$PKGSTAT" = "renamed" \) ]; \
                        then print $(PKGDEP_INCORP) >> $(@); \
                fi; \
                print $$LICS > $(@:%.mog=%.lics); \
        else \
                $(TOUCH) $(@) $(@:%.mog=%.lics); \
        fi
        $(PKGDEBUG)$(RM) $(@).vars

$(PDIR)/%.dep: $(PDIR)/%.mog
        @print "Generating dependencies for $(<F)"
        $(PKGDEBUG)$(RM) $(@)
        $(PKGDEBUG)if [ ! -f $(@:%.dep=%.nodepend) ]; then \
                pkgdepend generate -m $(PKGDEP_TOKENS:%=-D %) $(<) \
                        $(PKGROOT) > $(@); \
        else \
                $(CP) $(<) $(@); \
        fi

#
# The full chain implies that there should be a .dep.res suffix rule,
# but dependency generation is done on a set of manifests, rather than
# on a per-manifest basis.  Instead, see the gendeps rule above.
#

$(PDIR)/%.pub: $(PDIR)/%.res
        $(PKGDEBUG)m=$$(basename $(@:%.pub=%).metadata.*); \
        r=$${m#$(@F:%.pub=%.metadata.)+(?).}; \
        if [ -s $(<) ]; then \
                print "Publishing $(@F:%.pub=%) to $$r repository"; \
                pkgsend -s file://$(PKGDEST)/repo.$$r publish \
                    -d $(PKGROOT) -d $(TOOLSROOT) \
                    -d license_files -d $(PKGROOT)/licenses \
                    --fmri-in-manifest --no-index --no-catalog $(<) \
                    > /dev/null; \
        fi; \
        $(TOUCH) $(@);

#
# rule to build the synthetic manifests
#
# This rule necessarily has PKGDEP_TYPE that changes according to
# the specific synthetic manifest.  Rather than escape command
# dependency checking for the real manifest processing, or failing to
# express the (indirect) dependency of synthetic manifests on real
# manifests, we simply split this rule out from the one above.
#
# The implementation notes from the previous rule are applicable
# here, too.
#
$(PROC_SYNTH_PKGS): $(PKGLISTS) $$(@F:%.mog=%.mf)
        @print "Processing synthetic manifest $(@F:%.mog=%.mf)"
        $(PKGDEBUG)$(RM) $(@) $(PDIR)/$(@F:%.mog=%).metadata.* $(@).vars
        $(PKGDEBUG)$(PKGMOGRIFY) $(PKGMOG_VERBOSE) -I transforms -I $(PDIR) \
                $(PKGMOG_DEFINES:%=-D %) -D PKGDEP_TYPE=$(PKGDEP_TYPE) \
                -P $(@).vars -O $(@) $(@F:%.mog=%.mf) \
                $(PM_TRANSFORMS) synthetic
        $(PKGDEBUG)eval REPO=redist PKGSTAT=current ‘$(CAT) -s $(@).vars‘; \
        if [ -f $(@) ]; then \
                $(LN) -s $(@F) \
                        $(PDIR)/$(@F:%.mog=%).metadata.$$PKGSTAT.$$REPO; \
        else \
                $(TOUCH) $(@); \
        fi
        $(PKGDEBUG)$(RM) $(@).vars

$(DEP_SYNTH_PKGS): $$(@:%.dep=%.mog)
        @print "Skipping dependency generation for $(@F:%.dep=%)"
        $(PKGDEBUG)$(CP) $(@:%.dep=%.mog) $(@)

clean:

clobber: clean
        $(RM) -r $(CLOBBERFILES)

#
# This rule assumes that all links in the $PKGSTAT directories
# point to valid manifests, and will fail the make run if one
# does not contain an fmri.
#
# We do this in the BEGIN action instead of using pattern matching
# because we expect the fmri to be at or near the first line of each input
```

```
# file, and this way lets us avoid reading the rest of the file after we
# find what we need.
#
# We keep track of a failure to locate an fmri, so we can fail the
# make run, but we still attempt to process each package in the
# repo/pkgstat-specific subdir, in hopes of maybe giving some
# additional useful info.
#
# The protolist is used for bfu archive creation, which may be invoked
# interactively by the user.  Both protolist and PKGLISTS targets
# depend on $(PROC_PKGS), but protolist builds them recursively.
# To avoid collisions, we insert protolist into the dependency chain
# here.  This has two somewhat subtle benefits: it allows bfu archive
# creation to work correctly, even when -a was not part of NIGHTLY_OPTIONS,
# and it ensures that a protolist file here will always correspond to the
# contents of the processed manifests, which can vary depending on build
# environment.
#
$(PKGLISTS): $(PROC_PKGS)
        $(PKGDEBUG)sdotr=$(@F:packages.%=%); \
        r=$${sdotr%.+(?)}; s=$${sdotr#+(?).}; \
        print "Generating $$r $$s package list"; \
        $(RM) $(@); $(TOUCH) $(@); \
        $(NAWK) ’BEGIN { \
                if (ARGC < 2) { \
                        exit; \
                } \
                retcode = 0; \
                for (i = 1; i < ARGC; i++) { \
                        do { \
                                e = getline f < ARGV[i]; \
                        } while ((e == 1) && (f !~ /name=pkg.fmri/)); \
                        close(ARGV[i]); \
                        if (e == 1) { \
                                l = split(f, a, "="); \
                                print "depend fmri=" a[l], \
                                        "type=$$(PKGDEP_TYPE)"; \
                        } else { \
                                print "no fmri in " ARGV[i] >> "/dev/stderr"; \
                                retcode = 2; \
                        } \
                } \
                exit retcode; \
        }’ ‘find $(PDIR) -type l -a \( $(PKGS:%=-name %.metadata.$$s.$$r -o) \
                -name NOSUCHFILE \)‘ >> $(@)

#
# rules to validate proto area against manifests, check for safe
# file permission modes, and generate a faux proto list
#
# For the check targets, the dependencies on $(PROC_PKGS) is specified
# as a subordinate make process in order to suppress output.
#
makesilent:
        @$(MAKE) -e $(PROC_PKGS) PKGMACH=$(PKGMACH) \
                SUPPRESSPKGDEP=$(SUPPRESSPKGDEP) > /dev/null

#
# The .lics files were created during pkgmogrification, and list the
# set of licenses to pull from $SRC for each package.  Because
# licenses may be duplicated between packages, we uniquify them as
# well as aggregating them here.
#
license-list: makesilent
        $(PKGDEBUG)( for l in ‘cat $(PROC_PKGS:%.mog=%.lics)‘; \
                do print $$l; done ) | sort -u > $@
```

```
#
# Staging the license and description files in the proto area allows
# us to do proper unreferenced file checking of both license and
# description files without blanket exceptions, and to pull license
# content without reference to $CODEMGR_WS during publication.
#
stage-licenses: license-list FRC
	$(PKGDEBUG)$(MAKE) -e -f Makefile.lic \
		PKGDEBUG=$(PKGDEBUG) LICROOT=$(PKGROOT)/licenses \
		`$(NAWK) '{ \
			print "$(PKGROOT)/licenses/" $$0; \
			print "$(PKGROOT)/licenses/" $$0 ".descrip"; \
		}' license-list` > /dev/null;

protocmp: makesilent
	@validate_pkg -a $(PKGMACH) -v \
		$(EXCEPTIONS:%=-e $(CODEMGR_WS)/exception_lists/%) \
		-m $(PDIR) -p $(PKGROOT) -p $(TOOLSROOT)

pmodes: makesilent
	@validate_pkg -a $(PKGMACH) -M -m $(PDIR) \
		-e $(CODEMGR_WS)/exception_lists/pmodes

check: protocmp pmodes

protolist: proto_list_$(PKGMACH)

proto_list_$(PKGMACH): $(PROC_PKGS)
	@validate_pkg -a $(PKGMACH) -L -m $(PDIR) > $(@)

$(PROC_PKGS): $(PDIR)

#
# This is a convenience target to allow package names to function as
# build targets.  Generally, using it is only useful when iterating on
# development of a manifest.
#
# When processing a manifest, use the basename (without extension) of
# the package.  When publishing, use the basename with a ".pub"
# extension.
#
# Other than during manifest development, the preferred usage is to
# avoid these targets and override PKGS on the make command line and
# use the provided all and install targets.
#
$(PKGS) $(SYNTH_PKGS): $(PDIR)/$$(@:%=%.mog)

$(PKGS:%=%.pub) $(SYNTH_PKGS:%=%.pub): $(PDIR)/$$(@)

#
# This is a convenience target to resolve dependencies without publishing
# packages.
#
gendeps: $(PDIR)/gendeps

#
# These are convenience targets for cross-platform packaging.  If you
# want to build any of "the normal" targets for a different
# architecture, simply use "arch/target" as your build target.
#
# Since the most common use case for this is "install," the architecture
# specific install targets have been further abbreviated to elide "/install."
#
i386/% sparc/%:
	$(MAKE) -e $(@F) PKGMACH=$(@D) SUPPRESSPKGDEP=$(SUPPRESSPKGDEP)
```

```
i386 sparc: $$(@)/install

FRC:

# EXPORT DELETE START
XMOD_PKGS= \
	BRCMbnx \
	BRCMbnxe \
	SUNWadpu320 \
	SUNWibsdpib \
	SUNWkdc \
	SUNWlsimega \
	SUNWwbint \
	SUNWwbsup

EXPORT_SRC: CRYPT_SRC
	$(RM) $(XMOD_PKGS:%=manifests/%.mf)
	$(RM) Makefile+
	$(SED) -e "/^# EXPORT DELETE START/,/^# EXPORT DELETE END/d" \
		< Makefile > Makefile+
	$(MV) -f Makefile+ Makefile
	$(CHMOD) 444 Makefile
# EXPORT DELETE END
```