The Source Tree, part 2

Sachidananda Urs

Using Your Workspace

This section describes how to perform common operations on your workspace, such as editing files and modifying the build system.

Getting Ready to Work

Once you have brought over a workspace, you must set up both the workspace itself and your environment before you can safely use it.

First, you must set your environment variables appropriately so that the build tools will work properly. This must be done once in any shell which will run commands that affect the workspace. Although there are numerous variables which must be set, nearly all of them are set by bldenv(1), which accepts a nightly-style environment file. For example, if your workspace's environment file is in /aux0/testws/opensolaris.sh, you would need to run the following command in each shell from which you will run commands that affect the workspace:

```
$ bldenv /aux0/testws/opensolaris.sh
```

It may be tempting to run this command from your shell initialization scripts; however, if you have more than one workspace, it can be inconvenient and even dangerous, as you may inadvertently perform an operation on the wrong workspace.

Editing Files

You can use any text editor you like to edit source files. Although choice of editor is a personal matter, not all editors are equal. In particular, there are two types of editor behavior which are certain to cause problems. First, some editors such as pico and nano wrap lines longer than 72 characters or so. While this is fine for documents, it causes havoc in source files. If you must use such an editor, be sure to turn off line wrapping; otherwise your diffs will include large quantities of noise and the resulting file may not compile. Second, some editors, especially on non-Unix systems which may have different conventions, will change newline characters from '\n' to something else. If you must edit

sources on a foreign system, be absolutely certain your editor outputs Unix-style newlines. Source files containing foreign newline characters cannot be integrated into any OpenSolaris consolidation and may not compile.

When you edit source files, be sure that you are familiar with the style guide; Once again, not all editors are equal: some may offer the ability to set parameters that will help you to write conformant code, while others may enforce fixed settings which conflict with our style. Regardless of your choice of editor, you will need to be sure that your code conforms to the style guidelines.

Modifying the ON Build System

This section provides detailed, step-by-step instructions for common build-related tasks, such as adding or moving kernel modules and adding new libraries. You should also read the sections regarding makefile layout and operation to gain a better understanding of the overall build system.

Adding a New Kernel Module

The most common build-related operation developers need to perform is adding a new kernel module to the gate. In this section, we describe the process of adding build instructions for foofs, whose sources are located under usr/src/uts/common/fs/foofs. Adding related commands and libraries (in usr/src/cmd and usr/src/lib) is not covered here.

1. Create the usr/src/uts/common/fs/foofs directory and populate it with your sources.

2. Edit uts/*/Makefiles.files to define the set of objects. By convention the symbolic name of this set is of the form MODULE_OBJS, where MODULE is the module name (foofs). The files in each subtree should be defined in the Makefile.files in the root directory of that subtree. Note that they are defined using the += operator, so that the set can be accumulated across multiple makefiles. As example:

   ```
   FOOFS_OBJS +=foovfs.o foovno.o
   ```

   Each source file needs a build rule in the corresponding Makefile.rules file (compilation and linting). A typical pair of entries would be:

   ```
   $(OBJS_DIR)/%.o:       $(UTSBASE)/common/fs/foofs/%.c
         $(COMPILE.c) -o $@ $<
         $(CTFCONVERT_O)

   $(LINTS_DIR)/%.ln:     $(UTSBASE)/common/fs/foofs/%.c
         @($(LHEAD) $(LINT.c) $< $(LTAIL))
   ```

In this case, these are added to usr/src/uts/common/Makefile.rules. If the module being added is architecture-specific, they must instead be added to the appropriate architecture-specific Makefile.rules file.

3. Create build directories in the appropriate places. If the module can be built in a platform-independent way, this would be in the "instruction set architecture" directory (i.e.: sparc, intel). If not, these directories would be created for all appropriate "implementation architecture"-dependent directories (i.e.: sun4u). In this case, foofs is common code, so two build directories are needed: usr/src/uts/sparc/foofs and usr/src/uts/intel/foofs.

4. In each build directory, create a makefile. This can usually be accomplished by copying a Makefile from a parallel directory and editing the following lines (in addition to comments).

```
MODULE          =foofs
OBJECTS         =$(FOOFS_OBJS:%=$(OBJS_DIR)/%)
LINTS           =$(FOOFS_OBJS:%.o=$(LINTS_DIR)/%.ln)
ROOTMODULE      =$(ROOT_FS_DIR)/$(MODULE)
```

   - replace directory part with the appropriate installation directory name (see Makefile.uts)

   If a custom version of modstubs.o is needed to check the undefineds for this routine, the following lines need to appear in the makefile (after the inclusion of Makefile.plat (i.e.: Makefile.sun4u)).

```
MODSTUBS_DIR = $(OBJS_DIR)
$(MODSTUBS_O) := AS_CPPFLAGS += -DFOOFS_MODULE
```

   - replace "-DFOOFS_MODULE" with the appropriate flag for the modstubs.o assembly.

```
CLEANFILES+= $(MODSTUBS_O)
```

5. Edit the parent Makefile.mach (i.e.: Makefile.sparc, Makefile.intel) to know about the new module:

```
FS_KMODS        += foofs
```

Note that if your module only applies to a subset of the supported architectures, you will only need to perform this step for the makefiles that are used for those architectures.

Making a Kernel Module Architecture-Independent

In some cases, a module which was specific to a particular implementation architecture is adapted to be more general, often because the hardware it supports has become available on more platforms. Once the module itself is able to be used across multiple platforms, its build parameters should be updated to reflect this. Its location in the source tree will also need to change.

1. Create the build directory under the appropriate "instruction set architecture" build directory (i.e.: sparc/MODULE).
2. Move the makefile from the "implementation architecture" build directory (i.e.: sun4u/MODULE) to the directory created above. Edit this makefile to reflect the change of parent (trivial: comments, paths and includes).
3. Edit the "implementation architecture" directory makefile (i.e.: Makefile.sun4u) to **not** know about this module and edit the "instruction set architecture" directory makefile (i.e.: Makefile.sparc) to know about it.
4. Since the install locations may have changed (as well as the set of systems on which these files are installed) you may also need to adjust the package manifests in usr/src/pkg/manifests to reflect such changes.

Adding New Libraries

This section describes the overall layout and operation of the library makefiles and provides detailed step-by-step instructions for adding new libraries and enhancing existing library makefiles. This is based very closely on the file usr/src/lib/README.Makefiles and will be updated from time to time to match its contents.

Your library should consist of a hierarchical collection of Makefiles:

lib/<library>/Makefile
: This is your library's top-level Makefile. It should contain rules for building any ISA-independent targets, such as installing header files and building message catalogs, but should defer all other targets to ISA-specific Makefiles.

lib/<library>/Makefile.com
: This is your library's common Makefile. It should contain rules and macros which are common to all ISAs. This Makefile should never be built explicitly, but instead should be included (using the make include mechanism) by all of your ISA-specific Makefiles.

lib/<library>/<isa>/Makefile
: These are your library's ISA-specific Makefiles, one per ISA (usually sparc and i386, and sometimes sparcv9 and ia64). These Makefiles should include your common Makefile and then provide any needed ISA-specific

rules and definitions, perhaps overriding those provided in your common Makefile. To simplify their maintenance and construction, $(SRC)/lib has a handful of provided Makefiles that yours must include; the examples provided throughout the document will show how to use them. Please be sure to consult these Makefiles before introducing your own custom build macros or rules.

lib/Makefile.lib
This contains the bulk of the macros for building shared objects.

lib/Makefile.lib.64
This contains macros for building 64-bit objects, and should be included in Makefiles for 64-bit native ISAs.

lib/Makefile.rootfs
This contains macro overrides for libraries that install into /lib (rather than /usr/lib).

lib/Makefile.targ
This contains rules for building shared objects. The remainder of this document discusses how to write each of your Makefiles in detail, and provides examples from the libinetutil library.

The Library Top-level Makefile

As described above, your top-level library Makefile should contain rules for building ISA-independent targets, but should defer the building of all other targets to ISA-specific Makefiles. The ISA-independent targets usually consist of:

install_h
Install all library header files into the proto area. Can be omitted if your library has no header files.

check
Check all library header files for hdrchk compliance. Can be omitted if your library has no header files.

_msg
Build and install a message catalog. Can be omitted if your library has no message catalog. Of course, other targets are (such as 'cstyle') are fine as well, as long as they are ISA-independent.

The ROOTHDRS and CHECKHDRS targets are provided in lib/Makefile.lib to make it easy for you to install and check your library's header files. To use these targets, your Makefile must set the HDRS to the list of your library's header files to install and HDRDIR to the their location in the source tree. In addition, if your header files need to be installed in a location other than

$(ROOT)/usr/include, your Makefile must also set ROOTHDRDIR to the appropriate location in the proto area. Once HDRS, HDRDIR and (optionally) ROOTHDRDIR have been set, your Makefile need only contain

```
install_h: $(ROOTHDRS)

check: $(CHECKHDRS)
```

to bind the provided targets to the standard 'install_h' and 'check' rules.

Similar rules are provided (in $(SRC)/Makefile.msg.targ) to make it easy for you to build and install message catalogs from your library's source files.

To install a catalog into the catalog directory in the proto area, define the POFILE macro to be the name of your catalog, and specify that the _msg target depends on $(MSGDOMAINPOFILE). The examples below should clarify this.

To build a message catalog from arbitrarily many message source files, use the BUILDPO.msgfiles macro.

```
include ../Makefile.lib

POFILE =  libfoo.po
MSGFILES =  $(OBJECTS:%.o=%.i)


# ...


$(POFILE): $(MSGFILES)
           $(BUILDPO.msgfiles)


_msg: $(MSGDOMAINPOFILE)


include $(SRC)/Makefile.msg.targ
```

Note that this example doesn't use grep to find message files, since that can mask unreferenced files, and potentially lead to the inclusion of unwanted messages or omission of intended messages in the catalogs. As such, MSGFILES should be derived from a known list of objects or sources.

It is usually preferable to run the source through the C preprocessor prior to extracting messages. To do this, use the ".i" suffix, as shown in the above example. If you need to skip the C preprocessor, just use the native (.[ch]) suffix.

The only time you shouldn't use BUILDPO.msgfiles as the preferred means of extracting messages in when you're extracting them from shell scripts; in that case, you can use the BUILDPO.pofiles macro as explained below.

To build a message catalog from other message catalogs, or from source files that include shell scripts, use the BUILDPO.pofiles macro:

```
include ../Makefile.lib


SUBDIRS =  $(MACH)


POFILE=  libfoo.po
POFILES=  $(SUBDIRS:%=%/_%.po)


_msg :=  TARGET = _msg


# ...


$(POFILE): $(POFILES)
           $(BUILDPO.pofiles)


_msg: $(MSGDOMAINPOFILE)


include $(SRC)/Makefile.msg.targ
```

The Makefile above would work in conjunction with the following in its subdirectories' Makefiles:

```
POFILE =  _thissubdir.po
MSGFILES =  $(OBJECTS:%.o=%.i)


$(POFILE):  $(MSGFILES)
        $(BUILDPO.msgfiles)


_msg:  $(POFILE)


include $(SRC)/Makefile.msg.targ
```

Since this POFILE will be combined with those in other subdirectories by the parent Makefile and that merged file will be installed into the proto area via MSGDOMAINPOFILE, there is no need to use MSGDOMAINPOFILE in this Makefile (in fact, using it would lead to duplicate messages in the catalog).

When using any of these targets, keep in mind that other macros, like XGETFLAGS and TEXT_DOMAIN may also be set in your Makefile to override or augment the defaults provided in higher-level Makefiles.

As previously mentioned, you should defer all ISA-specific targets to your ISA-specific Makefiles. You can do this by:

Setting SUBDIRS to the list of directories to descend into:

```
SUBDIRS = $(MACH)
```

Note that if your library is also built 64-bit, then you should also specify

```
$(BUILD64)SUBDIRS += $(MACH64)
```

so that SUBDIRS contains $(MACH64) if and only if you're compiling on a 64-bit ISA.

Providing a common "descend into SUBDIRS" rule:

```
spec $(SUBDIRS): FRC
        @cd $@; pwd; $(MAKE) $(TARGET)
FRC:
```

Providing a collection of conditional assignments that set TARGET appropriately:

```
all     := TARGET= all
clean   := TARGET= clean
clobber := TARGET= clobber
install := TARGET= install
lint    := TARGET= lint
```

The order doesn't matter, but alphabetical is preferable.

Having the aforementioned targets depend on SUBDIRS:

```
all clean clobber install: spec .WAIT \$(SUBDIRS)
lint: \$(SUBDIRS)
```

A few notes are in order here:

The 'all' target must be listed first; the others might as well be listed alphabetically.

The 'lint' target is listed separately because there is nothing to lint in the spec subdirectory.

The .WAIT between spec and $(SUBDIRS) is suboptimal but currently required to make sure that two different make invocations don't simultaneously

build the mapfiles. It will likely be replaced with a more sophisticated mechanism in the future.

As an example of how all of this goes together, here's libinetutil's top-level library Makefile (copyright omitted):

```
include ../Makefile.lib


HDRS= libinetutil.h
HDRDIR= common
SUBDIRS = $(MACH)
$(BUILD64)SUBDIRS += $(MACH64)


all     :=TARGET = all
clean   :=TARGET = clean
clobber :=TARGET = clobber
install :=TARGET = install
lint    := TARGET = lint


.KEEP_STATE:


all clean clobber install: spec .WAIT $(SUBDIRS)


lint:$(SUBDIRS)

install_h:$(ROOTHDRS)

check:$(CHECKHDRS)

$(SUBDIRS) spec: FRC
        @cd $@; pwd; $(MAKE) $(TARGET)


FRC:

include ../Makefile.targ
```

The Common Makefile

In concept, your common Makefile should contain all of the rules and definitions that are the same on all ISAs. However, for reasons of maintainability

and cleanliness, you're encouraged to place even ISA-dependent rules and definitions, as long you express them in an ISA-independent way (e.g., by using $(MACH), $(TRANSMACH), and their kin).

The common Makefile can be conceptually split up into four sections:

1. A copyright and comments section.
   Please see the prototype files in usr/src/prototypes for examples of how to format the copyright message properly. For brevity and clarity, this section has been omitted from the examples shown here.

2. A list of macros that must be defined prior to the inclusion of Makefile.lib.
   This section is conceptually terminated by the inclusion of Makefile.lib, followed, if necessary, by the inclusion of Makefile.rootfs (only if the library is to be installed in /lib rather than the default /usr/lib).

3. A list of macros that need not be defined prior to the inclusion of Makefile.lib
   (or which must be defined following the inclusion of Makefile.lib, to override or augment its definitions). This section is conceptually terminated by the .KEEP_STATE directive.

4. A list of targets.

The first section is self-explanatory. The second typically consists of the following macros:

LIBRARY
: Set to the name of the static version of your library, such as 'libinetutil.a'. You should always specify the '.a' suffix, since pattern-matching rules in higher-level Makefiles rely on it, even though static libraries are not normally built in ON, and are never installed in the proto area. Note that the LIBS macro (described below) controls the types of libraries that are built when building your library.

  If you are building a loadable module (i.e., a shared object that is only linked at runtime with dlopen(3dl)), specify the name of the loadable module with a '.a' suffix, such as 'devfsadm_mod.a'.

VERS
: Set to the version of your shared library, such as '.1'. You actually do not need to set this prior to the inclusion of Makefile.lib, but it is good practice to do so since VERS and LIBRARY are so closely related.

OBJECTS
: Set to the list of object files contained in your library, such as 'a.o b.o'. Usually, this will be the same as your library's source files (except with .o extensions), but if your library compiles source files outside of the library directory itself, it will differ. We'll see an example of this with libinetutil.

  The third section typically consists of the following macros:

LIBS

Set to the list of the types of libraries to build when building your library. For dynamic libraries, you should set this to '$(DYNLIB) $(LINTLIB)' so that a dynamic library and lint library are built. For loadable modules, you should just list DYNLIB, since there's no point in building a lint library for libraries that are never linked at compile-time.

If your library needs to be built as a static library (typically to be used in other parts of the build), you should set LIBS to '$(LIBRARY)'. However, you should do this only when absolutely necessary, and you must *never* ship static libraries to customers.

ROOTLIBDIR (if your library installs to a nonstandard directory)

Set to the directory your 32-bit shared objects will install into with the standard $(ROOTxxx) macros. Since this defaults to $(ROOT)/usr/lib ($(ROOT)/lib if you included Makefile.rootfs), you usually do not need to set this.

ROOTLIBDIR64 (if your library installs to a nonstandard directory)

Set to the directory your 64-bit shared objects will install into with the standard $(ROOTxxx64) macros. Since this defaults to $(ROOT)/usr/lib/$(MACH64) ($(ROOT)/lib/$(MACH64) if you included Makefile.rootfs), you usually do not need to set this.

SRCDIR

Set to the directory containing your library's source files, such as '../common'. Because this Makefile is actually included from your ISA-specific Makefiles, make sure you specify the directory relative to your library's <isa> directory.

SRCS (if necessary)

Set to the list of source files required to build your library. This defaults to $(OBJECTS:%.o=$(SRCDIR)/%.c) in Makefile.lib, so you only need to set this when source files from directories other than SRCDIR are needed. Keep in mind that SRCS should be set to a list of source file **pathnames**, not just a list of filenames.

LINTLIB-specific SRCS (required if building a lint library)

Set to a special "lint stubs" file to use when constructing your library's lint library. The lint stubs file must be used to guarantee that programs that link against your library will be able to lint clean. To do this, you must conditionally set SRCS to use your stubs file by specifying 'LINTLIB := SRCS= $(SRCDIR)/$(LINTSRC)' in your Makefile. Of course, you do not need to set this if your library does not build a lint library.

LDLIBS

Appended with the list of libraries and library directories needed to build your library; minimally "-lc". Note that this should *never* be set, since

that will inadvertently clear the library search path, causing the linker to look in the wrong place for the libraries.

Since lint targets also make use of LDLIBS, LDLIBS **must** only contain -l and -L directives; all other link-related directives should be put in DYNFLAGS (if they apply only to shared object construction) or LDFLAGS (if they apply in general).

MAPDIR

Set to the directory in which your library mapfile is built. If your library builds its mapfile from specfiles, set this to '../spec/$(TRANSMACH)' (TRANSMACH is the same as MACH for 32-bit targets, and the same as MACH64 for 64-bit targets).

MAPFILE (required if your mapfile is under source control)

Set to the path to your library mapfile. If your library builds its mapfile from specfiles, this need not be set. If you set this, you must also set DYNFLAGS to include '-M $(MAPFILE)' and set DYNLIB to depend on MAPFILE.

SPECMAPFILE (required if your mapfile is generated from specfiles)

Set to the path to your generated mapfile (usually '$(MAPDIR)/mapfile'). If your library mapfile is under source control, you need not set this. Setting this triggers a number of features in higher-level Makefiles:

- Your shared library will automatically be linked with '-M $(SPECMAPFILE)'.
- A 'make clobber' will remove $(SPECMAPFILE).
- Changes to $(SPECMAPFILE) will cause your shared library to be rebuilt.
- An attempt to build $(SPECMAPFILE) will automatically cause a 'make mapfile' to be done in MAPDIR.

CPPFLAGS (if necessary)

Appended with any flags that need to be passed to the C preprocessor (typically -D and -I flags). Since lint macros use
CPPFLAGS, CPPFLAGS **must** only contain directives known to the C preprocessor. When compiling MT-safe code, CPPFLAGS **must** include -D_REENTRANT. When compiling large file aware code, CPPFLAGS **must** include -D_FILE_OFFSET_BITS=64.

CFLAGS

Appended with any flags that need to be passed to the C compiler. Minimally, append '$(CCVERBOSE)'. Keep in mind that you should add any C preprocessor flags to CPPFLAGS, not CFLAGS.

CFLAGS64 (if necessary)

Appended with any flags that need to be passed to the C compiler when compiling 64-bit code. Since all 64-bit code is compiled $(CCVERBOSE), you usually do not need to modify CFLAGS64.

COPTFLAG (if necessary)

Set to control the optimization level used by the C compiler when compiling 32-bit code. You should only set this if absolutely necessary, and it should only contain optimization-related settings (or -g).

COPTFLAG64 (if necessary)

Set to control the optimization level used by the C compiler when compiling 64-bit code. You should only set this if absolutely necessary, and it should only contain optimization-related settings (or -g).

LINTFLAGS (if necessary)

Appended with any flags that need to be passed to lint(1) when linting 32-bit code. You should only modify LINTFLAGS in rare instances where your code cannot (or should not) be fixed.

LINTFLAGS64 (if necessary)

Appended with any flags that need to be passed to lint(1) when linting 64-bit code. You should only modify LINTFLAGS64 in rare instances where your code cannot (or should not) be fixed.

Of course, you may use other macros as necessary.
The fourth section typically consists of the following targets:

- all
  Build all of the types of the libraries named by LIBS. Must always be the first real target in common Makefile. Since the higher-level Makefiles already contain rules to build all of the different types of libraries, you can usually just specify

all: $(LIBS)
though it should be listed as an empty target if LIBS is set by your ISA-specific Makefiles (see above).

- lint
  Use the 'lintcheck' rule provided by lib/Makefile.targ to lint the actual library sources. Historically, this target has also been used to build the lint library (using LINTLIB), but that usage is now discouraged. Thus, this rule should be specified as

lint: lintcheck
Conspicuously absent from this section are the 'clean' and 'clobber' targets. These targets are already provided by lib/Makefile.targ and thus should not be

provided by your common Makefile. Instead, your common Makefile should list any additional files to remove during a 'clean' and 'clobber' by appending to the CLEANFILES and CLOBBERFILES macros.

Once again, here's libinetutil's common Makefile, which shows how many of these directives go together. Note that Makefile.rootfs is included to cause libinetutil.so.1 to be installed in /lib rather than /usr/lib:

```
LIBRARY =libinetutil.a
VERS =.1
OBJECTS =octet.o inetutil4.o ifspec.o

include ../../Makefile.lib
include ../../Makefile.rootfs

LIBS =$(DYNLIB) $(LINTLIB)
SRCS =$(COMDIR)/octet.c $(SRCDIR)/inetutil4.c
      $(SRCDIR)/ifspec.c
$(LINTLIB):=SRCS = $(SRCDIR)/$(LINTSRC)
LDLIBS +=-lsocket -lc

SRCDIR =../common
COMDIR =$(SRC)/common/net/dhcp
MAPDIR =../spec/$(TRANSMACH)
SPECMAPFILE =$(MAPDIR)/mapfile

CFLAGS +=$(CCVERBOSE)
CPPFLAGS +=-I$(SRCDIR)

.KEEP_STATE:

all: $(LIBS)

lint: lintcheck

pics/\%.o: $(COMDIR)/\%.c
$(COMPILE.c) -o $@ $$<$
$(POST_PROCESS_O)

include ../../Makefile.targ
```

Note that for libinetutil, not all of the object files come from SRCDIR. To support this, an alternate source file directory named COMDIR is defined, and the source files listed in SRCS are specified using both COMDIR and SRCDIR. Additionally, a special build rule is provided to build object files from the sources in COMDIR; the rule uses COMPILE.c and POST_PROCESS_O so that any changes to the compilation and object-post-processing phases will be automatically picked up.

The ISA-Specific Makefiles

As the name implies, your ISA-specific Makefiles should contain macros and rules that cannot be expressed in an ISA-independent way. Usually, the only rule you will need to put here is 'install', which has different dependencies for 32-bit and 64-bit libraries. For instance, here are the ISA-specific Makefiles for libinetutil:

sparc/Makefile:

```
include ../Makefile.com

install: all $(ROOTLIBS) $(ROOTLINKS) $(ROOTLINT)


sparcv9/Makefile:

include ../Makefile.com
include ../../Makefile.lib.64

install: all $(ROOTLIBS64) $(ROOTLINKS64)


i386/Makefile:

include ../Makefile.com

install: all $(ROOTLIBS) $(ROOTLINKS) $(ROOTLINT)
```

Observe that there is no .KEEP_STATE directive in these Makefiles, since all of these Makefiles include libinetutil/Makefile.com, and it already has a .KEEP_STATE directive. Also, note that the 64-bit Makefile also includes Makefile.lib.64, which overrides some of the definitions contained in the higher level Makefiles included by the common Makefile so that 64-bit compiles work correctly.

CTF Data in Libraries

By default, all position-independent objects are built with CTF data using ctfconvert, which is then merged together using ctfmerge when the shared object is built. All C-source objects processed via ctfmerge need to be processed via ctfconvert or the build will fail. Objects built from non-C sources (such as assembly or C++) are silently ignored for CTF processing.

Filter libraries that have no source files will need to explicitly disable CTF by setting CTFMERGE_LIB to ":"; see libw/Makefile.com for an example.

More Information

Other issues and questions will undoubtedly arise while you work on your library's Makefiles. To help in this regard, a number of libraries of varying

complexity have been updated to follow the guidelines and practices outlined in this document:

lib/libdhcputil
: Example of a simple 32-bit only library.

lib/libdhcpagent
: Example of a simple 32-bit only library that obtains its sources from multiple directories.

lib/ncad_addr
: Example of a simple loadable module.

lib/libipmp
: Example of a simple library that builds a message catalog.

lib/libdhcpsvc
: Example of a Makefile hierarchy for a library and a collection of related pluggable modules.

lib/lvm
: Example of a Makefile hierarchy for a collection of related libraries and pluggable modules. Also an example of a Makefile hierarchy that supports the _dc target for domain and category specific messages.

Keeping Your Workspace in Sync

Over time, other developers will put back their work into the main gate, and your workspace will become out of date with respect to these changes. By keeping your workspace in sync as much as possible, you will save time in two ways.

First, you will have less merging to do when you are ready to put back, which will reduce or eliminate conflict resolution and simplify testing.

Second, the risk of missing semantic changes in other parts of the code will be reduced. If an interface you use is changed, you want to know about it as soon as possible so you will not continue to implement your features on the basis of a deprecated or nonexistent interface. This will save much work and grief; you do not want to be preparing to put back and only then discover that your code no longer compiles because a crucial interface has been removed.

Keeping your workspace in sync is not difficult, and should be done as often as practical. However, there is one drawback, which may limit the frequency at which you choose to sync your workspace. Pulling in changes unrelated to your work exposes you to the risk of breakage caused by those changes. If an unrelated change breaks your workspace, you could waste time trying to identify the cause of breakage by assuming it is in your changes rather than someone else's. Also, it is possible that an unrelated change will render your workspace unable to boot or perform any useful work at all, which would make it impossible

for you to test your changes until the original bugs are fixed. For these reasons, you will want to carefully consider your synchronization process to minimize risk while keeping as up to date as possible. Merging with recent, tested build snapshots on a biweekly basis rather than merging daily with the gate is often a good compromise.

Until the main gate is accessible to all developers, the Sun engineer who is carrying the fix will have to do any final merges with the internal gate. If you have submitted a change that causes this merge to get too hairy, you should be ready to answer questions about the change and provide assistance during the merge. In extreme cases, the Sun engineer may ask that you do the merge externally (i.e., after the next snapshot has been released) and resubmit your source patch. This reduces the chance that your change will break other changes being made to the same area of code.

Workspace Updates with Mercurial

You can use the command "hg pull -u" to synchronize your workspace with the gate. If you haven't made any changes in your workspace, this will pull down any new changesets and update your working directory to the last one.

External developers may wish to specify a revision with "hg pull", to match up a source changeset with closed binaries and signed crypto.

Internal developers should remember to update both their open and closed repositories.

If you have made changes in your workspace, bringing down changes from the gate will create a branch in your workspace. The last changeset in each branch is called a "head", so you will see a message from Mercurial about the creation of a new head. You should merge these branches before proceeding, using "hg merge". See the Mercurial wiki for an overview of the merge process.

If you have nightly(1) configured to do a pull (the default for internal developers), nightly(1) will attempt an automated merge if one is needed. The build will error out if the merge fails. Even if the automated merge succeeds, be sure to review the changes before committing them.