OpenSolaris Developer’s Reference

June 14th, 2005
## Contents

1 Introduction  \hfill 2  
  1.1 Overview  \hfill 2  
  1.2 Getting Help  \hfill 2  
  1.3 Quick Start  \hfill 3  

2 Prerequisites  \hfill 11  
  2.1 Hardware  \hfill 11  
  2.2 Operating Environment Packages  \hfill 14  
  2.3 Obtaining and Installing Compilers  \hfill 15  
  2.4 Obtaining and Installing Custom Tools  \hfill 15  
  2.5 Environment Variables  \hfill 15  

3 The Source Tree  \hfill 17  
  3.1 Retrieving the Sources  \hfill 17  
  3.2 Tour of Your Workspace  \hfill 17  
  3.3 Using Your Workspace  \hfill 40  
  3.4 Keeping Your Workspace in Sync  \hfill 55  

4 Building OpenSolaris  \hfill 56  
  4.1 Environment Variables  \hfill 56  
  4.2 Using nightly and bldenv  \hfill 58  
  4.3 Using Make  \hfill 59  
  4.4 Build Products  \hfill 61  

5 Installing and Testing OpenSolaris  \hfill 65  
  5.1 Installation Overview  \hfill 65  
  5.2 Using Cap-Eye-Install to Install Kernels  \hfill 66  
  5.3 Using BFU to Install ON  \hfill 69  
  5.4 Test Suites  \hfill 74  

6 Integration Procedure  \hfill 76  
  6.1 Submission Procedure  \hfill 76  
  6.2 Code Review Process  \hfill 76  
  6.3 Putback Procedures  \hfill 76  


## CONTENTS

6.4 Licensing Requirements ............................................. 77

7 Best Practices and Requirements .................................. 78
  7.1 Compatibility and Visibility Constraints ....................... 78
  7.2 Style Guide .......................................................... 89
  7.3 Testing Guidelines .................................................. 93
  7.4 Using Lint ............................................................ 93
  7.5 Tips and Suggestions .............................................. 95

A For Further Reading ..................................................... 96

B About Opensolaris.org ................................................ 97

C Glossary ................................................................. 98

D MANUAL INDEX ......................................................... 101

E CONCEPT INDEX ....................................................... 103
Chapter 1

Introduction

Welcome to OpenSolaris!

This guide is intended to provide developers with comprehensive information about working with OpenSolaris tools and sources. This chapter provides a brief overview of what OpenSolaris is and provides pointers to additional information.

1.1 Overview

This chapter serves as an introduction both to OpenSolaris and to the guide itself. Because much of the information in the guide will change over time, we always recommend reading the latest version. Section 1.2 covers where you can find the current guide and other documents, and additional resources including answers to common questions, ways to contact other developers, and information about the sources. While we strongly recommend reading the parts of this guide that are relevant to your interests, it is not necessary in order to participate. See section 1.3 "Quick Start" for five ways you can start working with OpenSolaris right now. Section 1.4 details the typographic conventions used in the rest of this guide.

1.2 Getting Help

The OpenSolaris web site offers numerous ways to find answers to your questions. A set of technical and non-technical frequently asked questions is maintained at §???. A larger and more in-depth set of technical documentation is maintained at §???.

If your question is not answered in the FAQ or in any of the documentation, or you don’t know where to look, you can ask on one of the mailing lists. There are many mailing lists available; for a current list and to sign up, see §???.

If you are having difficulty using the OpenSolaris web site itself, please send a message to webmaster@opensolaris.org. Note that this address should not be used for technical questions about OpenSolaris itself.
CHAPTER 1. INTRODUCTION

Finally, if you have any questions or comments about this document, please send a message to wesolows@eng.sun.com or join the OpenSolaris docs mailing list at §??.

You can always find the latest version of this document at §??.

1.3 Quick Start

This section bypasses most of the important material you’ll find later in this document and makes a lot of assumptions about your development and build environment. The intent is to offer a set of step-by-step instructions that will help you begin development in less than an hour, even if you have no previous Solaris development experience. Although this procedure will work for most people, it cannot work for all. There are dozens of tools and utilities used in developing OpenSolaris, and most have at least a few options and arguments. This tremendous flexibility exists for a reason: each developer has unique needs. If you encounter difficulty in following these instructions, don’t give up! Each instruction lists the relevant sections you can read to learn more about the commands and procedures used in that step. If all else fails, please see 1.2 Getting Help to learn about the many resources available to the OpenSolaris developer community.

1.3.1 Example Environment

The instructions in the remainder of section 1.3 have been tested and verified to work. Because it is impossible to test on every imaginable hardware, software, and policy configuration, our testing was limited to the environments described here. While we expect these instructions will be broadly applicable, the more similar your environment is, the more likely you will be able to use them without modification.

1.3.1.1 x86 Environment We assume that the entire system is dedicated to OpenSolaris. Our test machine is a dual Intel Pentium 4 Xeon 2.8 GHz with hyperthreading enabled, 512MB memory, a standard ATA controller, and a Broadcom 57xx ethernet adapter. The machine has 2 40GB ATA disks, partitioned as follows:

Both c0d0p0 and c1d0p0 have a single partition which assigns the entire disk to Solaris.

c0d0s0 contains a 35GB UFS filesystem mounted on /.
c0d0s1 contains a 2GB swap space.
c0d0s8 is a 1MB boot slice.

c1d0s0 contains a 37GB UFS filesystem mounted on /aux0.

Your system does not need to be identical or even similar to this one in order for it to work; this configuration is merely an example of a supported system.
Please see chapter 2 for more information about the requirements for installing, developing, and building OpenSolaris.

1.3.1.2 SPARC Environment  Our SPARC test environment is a Sun Ultra 60 workstation with 2 450MHz UltraSPARC-II CPUs and 512MB memory. The machine has a single 18GB SCSI disk, partitioned as follows:

- `c0t0d0s0` contains a 8GB UFS filesystem mounted on `/`.
- `c0t0d0s1` contains a 2GB swap space.
- `c0t0d0s7` contains a 7GB UFS filesystem mounted on `/aux0`.

Your system does not need to be identical or even similar to this one in order for it to work; this configuration is merely an example of a supported system. Please see chapter 2 for more information about the requirements for installing, developing, and building OpenSolaris.

1.3.2 Installing a Binary Distribution

You will have to install a complete Solaris distribution, because you will need drivers, package tools, and other components that aren’t in the Operating System and Networking components released with OpenSolaris. Either a normal Solaris release or a Solaris Express release should work. If your interest is limited to some user-level applications, such an installation will normally be sufficient. However, if you want to install your own kernels or core libraries, you may need to install the newer binary archives provided with the source as well. In that case, the normal procedure will be to install the most recent available release; this will minimize the number of configuration file conflicts you will have to resolve. Both parts of this process are documented here; all users and developers will need to perform the steps described in 1.3.2.1 Installing an Official Solaris or Solaris Express Build, while only developers interested in using the very latest bits will need to read 1.3.2.3 Upgrading to the Latest Bits.

1.3.2.1 Installing an Official Solaris or Solaris Express Build  The Solaris installation procedure is well documented at ???. While many of the default options presented in the interactive installation procedure are acceptable for creating a development or build system, you should take note of the following:

**Locales are not needed.**

You do not need to install any locales unless you need them for other reasons. Selecting the C (POSIX) locale as default is recommended.

**Follow local policies,**

Network configuration, time zone, use of IPv6 and Kerberos, name service, and other environment-specific parameters must be set according to local policy. OpenSolaris development is not directly affected by them.
CHAPTER 1. INTRODUCTION

Reserve space for sources (and optionally builds).

To install the sources, you will need at least 300MB of free space (more is recommended). If you also wish to build OpenSolaris, you will need several GB. It is recommended that you create a separate local filesystem to store these sources and objects as well as user home directories and other standalone data if appropriate to your configuration; our example system mounts this filesystem at /aux0. See 2.1 Hardware for more detailed information.

Install the Entire Distribution package cluster.

If you do not have enough disk space to do so, you will need to upgrade to a disk at least 9GB in size. You may also be able to install the Development package cluster instead, but this has not been tested and may prevent you from building successfully. See 2.2 Operating Environment Packages for more information on software requirements.

1.3.2.2 Installing Required Tools All required compilers and other tools are provided in a single tarball for each supported platform. You can find tools tarballs for SPARC and x86 at §???. These tools must be installed into /opt as follows:

```
# cd /opt
# bzip2 -dc /path/to/opensolaris-build-<date>.sparc.tar.bz2 | tar xf -
```

You will need to adjust the above filename as appropriate for your system.

1.3.2.3 Upgrading to the Latest Bits WARNING! The steps described in this section are optional for advanced developers only and are not required to view and edit the sources. Performing this process unnecessarily will result in reduced system manageability and exposes you to the greater risks associated with the use of development software. If in doubt, SKIP THIS STEP. See 5.3 Using BFU to Install ON for more details on the risks and benefits of this process.

If you wish to build OpenSolaris, you may in some cases be required to update all or part of your system’s software before doing so. Such requirements are listed for each build in the Flag Day notices at §???: in most cases your existing installation will be sufficient to build and use the latest sources. In general, it is both safer and more effective to use the official upgrade process to install a more recent Solaris, Solaris Express, or OpenSolaris build than to use the BFU upgrade process; the official upgrade upgrades all of your system software, while BFU upgrades only a few parts. Unless you need to use the very latest bits, you should skip this step.

Before proceeding, please read 5.3 Using BFU to Install ON in its entirety. The remainder of this section provides an example usage of bfu(1), but you must understand how BFU works and how to resolve conflicts before you can
CHAPTER 1. INTRODUCTION

successfully use BFU. We assume here that you have installed a sufficiently recent OpenSolaris or Solaris Express system in 1.3.2.1 Installing an Official Solaris or Solaris Express Build; in this case BFU archives corresponding to that build are available and can be installed first to aid in conflict resolution. It is strongly recommended that you begin this process only if your installation is sufficiently recent that such BFU archives are available. In particular, BFUing a system older than Solaris 10 build 74 (FCS is OK) will fail and may render your system unbootable.

First, download the BFU archives for your system architecture from §??.

Then, unpack the archives into a temporary directory of your choice. In this example, we will use /var/tmp/bfu to store the archives, <build> to mean the currently installed build ('uname -v'), and <latest> to be the build you want to install.

```bash
# mkdir /var/tmp/bfu
# cd /var/tmp/bfu
# bzip2 -dc /path/to/opensolaris-bfu-<build>.sparc.tar.bz2 | tar xf -
# bzip2 -dc /path/to/opensolaris-bfu-<latest>.sparc.tar.bz2 | tar xf -
```

This will create the directories /var/tmp/bfu/sparc/<build> and /var/tmp/bfu/sparc/<latest>. In each directory will be a set of compressed CPIO archives; these are used by bfu(1) to upgrade your system binaries. Next, exit the window system, log in on the console as root, and issue the command:

```bash
# /opt/onbld/bin/bfu /var/tmp/bfu/sparc/<build>
```

Your system will be re-installed to the same version. You can ignore all conflicts, since your existing installed configuration files are known to work correctly for this build. You must now reboot.

```bash
# reboot
```

When your system reboots, you should once again log in on the console as root. This time, you will upgrade to the latest build:

```bash
# /opt/onbld/bin/bfu /var/tmp/bfu/sparc/<latest>
```

Your system will be upgraded. You may see warnings about upgrading non-ON packages; if you have not already done so, you will need to upgrade these before you can safely use BFU. If BFU fails because it tries to upgrade a package that is not available, please send a message to opensolaris-discuss@opensolaris.org describing the message and the circumstances.

After BFU completes, you must resolve conflicts in configuration files **before** you reboot. See 5.3.2 Resolving Conflicts for instructions. Finally, reboot your system. As it comes up, note the new kernel version. The OpenSolaris bits on your system have been upgraded.
1.3.3 Creating a Source Workspace

Suppose you are using /aux0/testws as your workspace.

First, cd to /aux0/testws and unpack the sources and encumbered binaries, e.g.,

```bash
$ mkdir /aux0/testws
$ cd /aux0/testws
$ gzcat src.tar.gz | tar xf -
$ gzcat closed-bins.i386.tar.gz | tar xf -
```

The sources will unpack into "usr/src" and the binaries will unpack into "closed/root_<mach>".

1.3.4 Performing a Basic Build

First, Create an environment file to guide tools like nightly(1) and bldenv(1).

- Copy in the template environment file

  Copy /aux0/testws/usr/src/tools/env/opensolaris.sh to /aux0/testws. It doesn’t have to go in /aux0/testws, but that’s a convenient place to put it. Nor do you have to keep the name opensolaris.sh, but that’s the name we’ll use in these notes.

  ```bash
  $ cp /aux0/testws/usr/src/tools/env/opensolaris.sh /aux0/testws
  ```

- Make required changes

  Using your favorite text editor, make the following changes:

  Change GATE to the name of the top-level directory (e.g., "testws").
  Change CODEMGR_WS to the top-level path (e.g., "/aux0/testws").
  Change STAFFER to your login.

- (optional) Customize VERSION.

  For now, this is only useful on SPARC systems; it is the string that "uname -v" will print. On x86 systems, VERSION is baked into one of the encumbered binaries. (We expect VERSION to be customizable on x86 before the public Launch.)

Then, To build a complete set of BFU archives, cd to /aux0/testws, utter

```bash
$ nightly ./opensolaris.sh &
```

and find something else to work on for a few hours. You can monitor the build’s progress using ptree(1). nightly(1) will send mail to $MAILTO when it has finished. The mail will have an overview of the build results. A copy of the mail text and a more detailed log file will be available in the workspace (e.g., /aux0/testws/log/log.XXXX/nightly.log, where XXXX is the build date in MMDD format). The log file is also available (in a slightly different location) during the build; to monitor the progress of your build in real time, you can do:
$ tail -f /aux0/testws/log/nightly.log

The location of the log is controlled by the LOGFILE and ATLOG variables; see nightly(1) for more details.

If your build fails, you can correct the problem, then use the '-i' option to nightly to run an incremental build, bypassing the initial clobber. See the nightly(1) manual and 4 Building OpenSolaris for more information.

To build a specific component, first use bldenv(1) to set up various environment variables, then cd to the subtree that you want to build. For example:

$ cd /aux0/testws
$ bldenv ./opensolaris.sh
[status information from bldenv]
$ cd usr/src/cmd/vi
$ dmake all

To build the kernel, run dmake(1) from usr/src/uts.

See 4 Building OpenSolaris for more information on the build process and tools. Once you have successfully completed a build, see 5 Installing and Testing OpenSolaris for more information on what to do with it.

1.3.5 Mini-Projects

These activities are intended to help developers gain familiarity with the OpenSolaris tools and environment while at the same time making useful contributions toward meeting their own needs and those of the community. We hope that by engaging in some of these mini-projects, new developers will be exposed to OpenSolaris from a variety of directions and will find inspiration for new improvements in the large body of existing work. The project suggestions are roughly ordered from simplest to most complex; less experienced developers should find the first several projects to be a good introduction to OpenSolaris, while more experienced developers may wish to take on some of the more challenging tasks. Of course, all of these are only suggestions; you are always free to work on any part of the system you wish. We ask only that you utilize the mailing lists to coordinate your activities with others to reduce duplication of effort.

1. Start a blog

Many developers keep notes, bookmarks, pointers to useful documentation, and other historical materials in a notebook or journal. As you become more familiar with OpenSolaris, your notes and experiences along the way will be valuable to you and to others in the community. You can create your own blog at ?? and start writing right away! Suggestions for topics include any observations you make, difficulties you encounter, or ideas you dream up for improving OpenSolaris. Writing about your ideas and experiences in your blog forms a focal point for wider community involvement and discussion, filing of bugs or RFEs, or the creation of a new development project.
2. Fix a simple bug

We include here several examples of as-yet unfixed bugs; this list will
be updated frequently, but you should consult the bug-tracking system
first before beginning work on any of these issues. Many are low-priority
issues that have remained unfixed over a long period of time; they are
not especially difficult to fix, and their effects tend to be annoying rather
than catastrophic. Eventually, we will identify bugs which are easily fixed
with searchable keywords, so that new developers can obtain up to date
starting points.

You can view information about these bugs at §7.2.1.

4567280  sckmdrv:_init could be simplified
4403708  panic in ndi_dc_allocdl()
4687825  *cron* /etc/cron.d/FIFO should move to /var/run/CRONFIFO
4349394  *troff* multiple buffer overflows in troff options processing
5016179  *sed* sed is almost 2x slower than ucb equivalent
4416487  /etc/saf/zsmon/_pid and /etc/saf/_cmdpipe should move to
          /var/run

3. Enhance cstyle(1)

Improve the automated C/C++ style checking tools cstyle(1) and hdrchk(1)
to implement more of the style guide requirements. For example, cstyle(1)
could be improved to detect poor declarations of initialized variables. See
7.2.1 Automated Style Tools for more information.

4. Clean up and modernize code

Make more commands and libraries lint-clean. Although the entire ker-
nel (uts/...) lints without warnings, many commands and libraries are
excluded from linting because they contain lint errors. Correcting these
errors is usually not difficult; start by enabling lint (see 3.3.3 Modifying
the Build System) and re-linting the command or library in question. In
some cases it may be difficult or impossible to fix a particular lint error,
and that error must be disabled instead. You can discuss such situations
on opensolaris-discuss.

5. Simplify Install(1) usage

Improve Install(1)'s `-i' option to use official platform names in addition
to code names when building kernel packages for specific platforms. See
5.2 Using Cap-Eye-Install to Install Kernels for more information.

6. Fix something that bothers you

Search for and fix a bug in a program you use frequently, or that you have
noticed yourself. You can search for bugs at §7.2.1. Many bugs may already
be undergoing fixes, so you should avoid duplication of effort by joining
the opensolaris-bugs list and sending a message about the bug you want
to work on.
7. Implement BFU conflict resolution

Add automatic conflict resolution to BFU. Such a feature would automatically merge whitespace or other trivial changes following a BFU, and could present an interface like filemerge(1) or Gentoo’s etc-update(1) (see §??). 5.3.2 Resolving Conflicts has more information about BFU conflict resolution.

1.4 Conventions

TBD
Chapter 2

Prerequisites

This chapter describes the hardware and software requirements for developing, building, and testing OpenSolaris. Most of these requirements can easily be met on any Solaris or OpenSolaris installation. Depending on your interests, you may need to test your work with hardware or software which is not readily available to you. If this is the case, please ask your project leader for assistance. Sun and other companies and organizations sponsor various facilities which may be available to OpenSolaris community developers. If you are a project leader, please contact community liaison@opensolaris.org for information about resources your project can use.

2.1 Hardware

This section details hardware requirements for OpenSolaris. Because some projects may have additional requirements (for example, driver development obviously requires the device in question), this information is intended to be a guide rather than a definitive set of rules. It is possible, for example, to develop on a system which does not support OpenSolaris at all, transferring patches or other work to a remote environment for building and testing. For purposes of this section, however, we will assume that your development and build systems will run Solaris or OpenSolaris.

2.1.1 Development Environment

The simplest set of requirements applies to a development environment: you need only have sufficient space to store the source tree and enough CPU and memory to run text editors and tools such as cscope. The current source tree occupies approximately 300MB without source code management metadata, so you should budget at least that amount of space for each tree you wish to store (about 1.9GB is required for a tree with a full set of SCCS metadata). Note that if you plan to build the tree on the same system, you will need additional space; see section 2.1.2 below.
CHAPTER 2. PREREQUISITES

In general, any computer which will run Solaris 10 or OpenSolaris is adequate for this purpose. As of this writing, all SPARC systems with UltraSPARC-II or newer CPUs (that is, all CPUs which have faster clock rates than 200 MHz) are supported; the specific list of supported SPARC platforms for Solaris 10 can be found at §??. For x86 and amd64 (Opteron) systems, hardware support is somewhat more complex; you can find out more about x86 hardware compatibility at §??. Be sure to check the latest release notes for Solaris for information about current and future hardware support; these notes can be found at §??.

Note that OpenSolaris may at times support a somewhat different set of hardware from the latest Solaris release or update; the latest information about hardware support in OpenSolaris will be placed at §?? as it becomes available.

Table: Development system requirements

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Any supported CPU</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>128MB recommended</td>
</tr>
<tr>
<td><strong>Swap</strong></td>
<td>No requirement</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>300MB to 1.9GB per copy of the source tree (*)</td>
</tr>
</tbody>
</table>

(*) The total size will vary depending on the type of source management you use. With no source management data, a built tree requires approximately 3.5GB; CVS and SCCS add 50MB or more, depending on the amount of revision history.

2.1.2 Build Environment

Building a large, complex piece of software such as OpenSolaris is a memory-, compute-, and space-intensive process. Although it is possible to build OpenSolaris on almost any computer, it may take prohibitively long to do so on small configurations. It is especially important to have enough swap space if you will be using dmake to perform highly parallel builds. Inadequate swap will cause your build to fail. Following are the hardware requirements for a system on which you intend to build OpenSolaris:

Table: Build system requirements

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>300 MHz UltraSPARC or x86 CPU</td>
</tr>
<tr>
<td></td>
<td>AMD Opteron or UltraSPARC-III recommended</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>256MB minimum, 1GB recommended (+)</td>
</tr>
<tr>
<td><strong>Swap</strong></td>
<td>1GB minimum, 2GB recommended (+)</td>
</tr>
</tbody>
</table>
CHAPTER 2. PREREQUISITES

Storage 3.5 to 5GB per copy of the source tree (*) (**) 

On a minimal build system as described above, expect a full build to take at least 14 hours. Incremental builds will take somewhat less time; a good rule of thumb is one-third of the full build time.

This table shows some approximate full build times for several representative systems. These are current as of 24 January 2005; as more code as added and the number of encumbered components decreases, build times will increase. Of course, the actual time to build will vary from system to system as well.

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2</td>
<td>2x300MHz UltraSPARC-II</td>
<td>512M</td>
<td>5h9m</td>
</tr>
<tr>
<td>Generic</td>
<td>AMD Athlon64 3200</td>
<td>1GB</td>
<td>1h23m</td>
</tr>
</tbody>
</table>

The build system can take advantage of multiple CPUs or even distribute the work of compiling across multiple machines. See 4 Building OpenSolaris for more information about configuring parallel and distributed builds.

(*) The total size will vary depending on the type of source management you use. With no source management data, a built tree requires approximately 3.5GB; CVS and SCCS add 50MB or more, depending on the amount of revision history.

(+) If you use dmake(1) in parallel mode and run more than 4 jobs on the same machine, you may need additional memory and/or swap space.

(**) Compilers and tools are installed in /opt and will require an additional 300MB on x86 or 800MB on SPARC.

2.1.3 Test Environment

The requirements for testing your changes will vary greatly. Some changes do not affect architecture-specific code at all and can simply be tested by building and running a test suite on any supported system of each architecture. For example, if you have added a new STREAMS module to the network stack, it is probably sufficient to build and test on one x86 system, one amd64 system, and one SPARC system. In other cases, for example modifications to the x86 boot code, it will be necessary to test on the widest possible array of hardware to ensure you have not broken some obscure piece of functionality.

Some hardware variables you should consider testing include:

- Architecture: x86, amd64, SPARC
- Memory size: does your change work with 32MB? With 32GB?
- Multiprocessor versus uniprocessor
- Graphical console versus serial console versus system LOM console
- Diskless versus "normal" systems
Not all these variables will be applicable to your particular change, so you will need to consider, being as conservative as possible, what effects your changes could cause. In general, changes to core kernel code such as the VM subsystem or filesystem support require the broadest testing. Changes to add-on drivers or machine-specific code need only be tested on the relevant hardware in most cases. See Chapter V for more details.

Quite often you may want to make a change which you cannot completely test on the hardware available to you. If you find yourself in this situation, please contact us via ???. The OpenSolaris community includes several organizations, including Sun, who may be able to provide you with access to additional hardware for the purpose of testing your changes.

2.2 Operating Environment Packages

The rest of this chapter discusses the software needed to build and package OpenSolaris. If you are only interested in looking at the sources, editing them, and sending your patches to others, you can skip this material; no special packages are required to view and edit the sources. We do recommend, however, that if you are installing Solaris or OpenSolaris for the first time on a system you plan to use for development, you install the "Entire" cluster (SUNWCall) because this will save you a good deal of time if you later decide you want to build OpenSolaris.

We strongly recommend installing the "Entire" package cluster (SUNWCall) on all development, build, and test systems. Although systems with the Developer (SUNWCdev) package cluster installed may work, this configuration has not been tested and will not be evaluated or recommended. Additionally, you should obtain and install the SUNWonbld package, which contains prebuilt versions of the tools in /usr/src/tools needed to build OpenSolaris. If you prefer, you can instead use nightly(1) with the '-t' option to build these tools (see 4.2 Using nightly and bldenv), but installing the package is recommended to avoid dependency problems. SUNWonbld can be downloaded from ???. Finally, you will need to obtain and install the required Sun One Studio compilers, dmake, and other tools. You can obtain these tools from ???.

**NOTE for pilot participants:** Prior to launch, both SUNWonbld and the compilers are combined into a single standard tarball. There are no packages to install.

If you install and use a full installation (for example, Solaris or OpenSolaris CDs or DVDs), you will have a complete matching set of programs, libraries, and kernel components. However, if you later upgrade the OpenSolaris bits using a method such as BFU (see 5.3 Using BFU to Install ON) or Install (see 5.2 Using Cap-Eye-Install to Install Kernels), or want to build and copy in your own updated versions of a few files, you may need to install newer versions of one or more system packages first. When this happens, it is known as a Flag Day. You can find more information about Flag Days in 5.1.3 Flag Days and Other Hazards.
2.3 Obtaining and Installing Compilers

The compilers needed depend on the platform(s) on which you wish to build OpenSolaris. An x86 build of OpenSolaris requires both the Sun One Studio and GNU compilers. This is because 32-bit objects are built with cc and CC, while gcc is used for 64-bit objects. This may change in the future, depending on the relative performance, features, and availability of the two compiler suites.

Because compilers used to build OpenSolaris may require special patches, links to current Sun One Studio compiler binaries will always be maintained at the OpenSolaris web site. Since build 73, the required versions of the GNU C compiler and related utilities are shipped with Solaris in /usr/sfw. Using other versions of compilers, assemblers, linkers, make(1), or dmake(1) is not supported and may fail to build or create binaries that do not function correctly. Therefore, you must use the compilers from §?? to ensure a successful build. From time to time the required tools will be updated; we will post notices and make newer versions available as appropriate.

2.4 Obtaining and Installing Custom Tools

In addition to the general development tools and compilers, a set of custom tools called "onbld" is also required. This includes scripts and programs that handle compiler and assembler compatibility, §?? data, ABI auditing, installation of §?? bits, and more. Many of these utilities are platform-specific; therefore, we deliver a version of the SUNWonbld package for all supported platforms (currently x86 and SPARC). Note that these tools are built from sources in the usr/src/tools subdirectory of the ON source gate. Refer to the Tour of Your Workspace in section 3.2 for more information about location of the sources. If you have the sources available, you can get by without installing SUNWonbld; however, you may encounter dependency problems when bootstrapping your ON installation. You can obtain this package from §??.

2.5 Environment Variables

In order to use the source code management and build tools successfully, you will need to set several environment variables correctly. Most of these variables are either workspace-specific (see section 3.3.1) or build-specific (see section 4.1). However, in order to successfully use the installed tools, you will need to set your PATH environment variable correctly. This should normally be done as part of your login scripts so that it will be done automatically.

You can always get the latest information about recommend PATH components from the gate’s public/README file. You can find a copy of this file via the Internet at §??.
PATH is used by your shell and other programs to determine where to find programs. This functionality has nothing to do with OpenSolaris, but in order to find the various programs needed by the build tools, PATH requires several elements in addition to the standard system directories. In general, these are:

```
/opt/SUNWspro/bin
/opt/onbld/bin
/opt/onbld/`uname -p`/bin
```

If you have installed your compilers or onbld tools in nonstandard locations (which is not recommended), you will need to modify these locations accordingly. Also, you must be certain to include `/usr/ccs/bin` in your PATH, and even more specifically, it must come before `/usr/ucb` if that directory is included in your PATH. As an example of putting all this together, a working PATH might look like:

```
$ echo $PATH
/usr/bin:/usr/sbin:/usr/ccs/bin:/usr/dt/bin:/usr/openwin/bin: \ 
/opt/onbld/bin:/opt/onbld/bin/sparc:/opt/sfw/bin:/usr/sfw/bin
```

See the gate’s public/README file available at §?? for details.

Including `/usr/ucb` in your PATH is not recommended, and the build process does not require any programs located there. If you must include it for some other reason, be sure it is the last component. Otherwise, including `/usr/ucb` will cause your build to fail, especially if it is before `/usr/sbin`.

Note that the paths to some tools are explicitly set in `usr/src/Makefile.master` and other makefiles, so changing your PATH may not by itself have the effect you intend. Specifically, compiler paths are fixed. If you need to change this, you must override the make macros specifying the compiler locations. You can learn more about this in section 4.1.

In addition to PATH, you may find it helpful to add `/opt/onbld/man` and `/opt/SUNWspro/man` to your MANPATH. This will make it easier to read the manual pages associated with the many utilities discussed in this document. Alternately, you will need to add `-M /opt/onbld/man` or `-M /opt/SUNWspro/man` to your invocation of `man(1)` in order to view these pages.
Chapter 3

The Source Tree

This chapter discusses obtaining, managing, and modifying the OpenSolaris source tree.

3.1 Retrieving the Sources

There are two different ways to obtain the sources: a tarball containing only the sources, and a public read-only CVS repository. Both contain the same set of sources, and both are updated daily.

3.1.1 Obtaining Sources via Tarball

You can obtain the tarball at §??. If you plan to build the source, and wish to produce complete installation sets, you will also need the encumbered binaries, which you can obtain at §??. Be sure to obtain the appropriate binaries for your platform. Once you have done so, create your workspace as follows:

```
$ mkdir /path/to/workspace
$ cd /path/to/workspace
$ bzip2 -dc /path/to/tarball/opensolaris-src-<ver>.tar.bz2 | tar xf -
$ bzip2 -dc /path/to/tarball/opensolaris-bins-<ver>.{x86,sparc}.tar.gz | tar xf -
```

3.1.2 Obtaining Sources via Anonymous CVS

TBD

3.2 Tour of Your Workspace

After creating and populating your workspace, you can begin to investigate the contents of the source tree. This section describes the layout and contents of your §?? workspace.
CHAPTER 3. THE SOURCE TREE

3.2.1 Organization

Initially, your workspace will have only one directory: usr. Once you have completed a build, you may have several additional directories, including archives, log, packages, and proto. We include here a diagram showing all these directories and their subdirectories. We also include the build subdirectories, which contain object files and are described in greater detail in subsections 2-4 below.

If you have done a nightly(1) build, you will also have a log directory at the top level. It will contain the complete output from the nightly(1) build process. See 4.2 Using nightly and bldenv for more information on nightly(1) and the log files it generates.

All sources are found under usr/src. This includes both the sources used to build the §?? consolidation and sources for tools and other peripheral utilities needed to build but not shipped as part of Solaris. Because it includes only ON sources, it does not contain Java, the windowing system, or packaging and installation tools. Because of contractual obligations, it generally also does not include code from third-party hardware vendors. The usr/src directory has several subdirectories which are described here.

**cmd**

This directory contains sources for the executable programs and scripts that are part of ON. It includes all the basic commands, daemons, startup scripts, and related data. Most subdirectories are named for the command or commands they provide; however, there are some exceptions listed here.

**cmd/Adm**

Miscellaneous key system files, such as crontabs and data installed in /etc.

**cmd/cmd-crypto**

Basic cryptographic utilities, such as elfsign and kcfd.

**cmd/cmd-inet**

Network commands and daemons, including the Berkeley r-commands, PPP, telnet, the inetd super-server, and other network-related utilities.

**cmd/fs.d**

Utilities for checking, mounting, unmounting, and analyzing filesystems.

**cmd/netfiles**

IP port definitions and name service switch configuration files installed into /etc.

**cmd/ptools**

Utilities for manipulating and observing processes; these are based on proc(4) and libproc interfaces.
CHAPTER 3. THE SOURCE TREE

**cmd/sgs**
Software Generation System. This directory contains binary utilities, such as ld(1), ar(1), and mcs(1), and development tools such as lex(1), yacc(1), and m4(1). Note that this directory also includes several libraries and headers used by these tools.

**common**
Files which are common among cmd, lib, stand, and uts. These typically include headers and sources to basic libraries used by both the kernel and user programs.

**head**
Userland header files (kernel headers are in uts/). Note that only libc headers should be stored here; other libraries should have their headers in their own subdirectories under lib/.

**lib**
Libraries. Most subdirectories are named for the library whose sources they contain or are otherwise self-explanatory.

**pkgdefs**
Contains one subdirectory for each package generated from the ON sources. Each subdirectory contains packaging information files; see pkginfo(4), depend(4), prototype(4), pkgmap(1), and pkgproto(1) for more information about the contents of these files.

**prototypes**
Sample files showing format and copyright notices.

**psm**
Platform-specific modules. Currently this contains only OBP and most of the boot code.

**stand**
Standalone environment code. This is used for booting; for example, code for reading from UFS and the network is here.

**tools**
Development tools and sources. See README.tools for more information about each tool; the file should be updated as tools are added or removed.

**ucbcmd**
Commands and daemons installed into /usr/ucb (for SunOS 4.x compatibility).
CHAPTER 3. THE SOURCE TREE

ucbhead

Header files installed into /usr/ucb (for SunOS 4.x compatibility).

ucblib

Libraries installed into /usr/ucb (for SunOS 4.x compatibility).

uts

Kernel sources are here (UTS == UNIX Time Sharing). There are numerous subdirectories of uts which are of interest:

uts/adb

adb contained the obsolete kernel debugger macros; it is no longer supported, and this directory is now empty. Use mdb(1) instead, and write mdb dcmds instead of adb macros.

uts/common

All platform-independent kernel sources. Nearly all of the Solaris kernel is here; only a few small parts are architecture-dependent.

uts/common/c2

Auditing code to support the C2 U.S. government security standard.

uts/common/conf

System configuration parameters.

uts/common/contract

Code to support process contracts. See contract(4) and libcontract(3LIB) for more information on process contracts.

uts/common/cpr

CheckPoint-and-Resume support. This implements suspend and resume functionality.

uts/common/crypto

Kernel cryptographic framework. See kcfd(1M) and cryptoadm(1M) for more information.

uts/common/ctf

Code for handling Compact C Type Format data.

uts/common/des

Implements the old Data Encryption Standard. This is used by KCF.

uts/common/disp

Dispatcher, thread handling, and scheduling classes.
CHAPTER 3. THE SOURCE TREE

uts/common/dtrace
   CPU-independent dtrace(7D) kernel support.

uts/common/exec
   Code for handling userland binary executable types (a.out, ELF, etc).

uts/common/fs
   Filesystems.

uts/common/ssapi
   Generic Security Services API.

uts/common/inet
   IP networking subsystem, including IPv6.

uts/common/io
   I/O subsystem. Most of the code in this directory is device drivers (and
   pseudo-device drivers).

uts/common/ipp
   IP policy framework; includes QoS and other traffic management.

uts/common/kmdb
   Kernel modular debugger. See kmdb(1).

uts/common/krtd
   Kernel runtime linker/loader. This is responsible for handling loadable
   modules and symbol resolution; it is analogous to ld.so.1, and shares code
   with it.

uts/common/ktli
   Kernel TLI (Transport Layer Interface).

uts/common/net
   Header files; most are shipped in /usr/include/net.

uts/common/netinet
   Header files; most are shipped in /usr/include/netinet.

uts/common/nfs
   Network File System headers shipped in /usr/include/nfs.

uts/common/os
   Core operating system implementation. This includes such varied aspects
   as privileges, zones, timers, the DDI/DKI interfaces, and high-level locking
   mechanisms.
uts/common/pcmcia

PCMCIA I/O subsystem and drivers.

uts/common/rpc

Remote Procedure Call subsystem used by services such as NFS and NIS.

uts/common/rpcsvec

Generated RPC header files shipped in /usr/include/rpcsvec.

uts/common/sys

Header files shipped in /usr/include/sys. These same headers are used to build the kernel as well (if the \_KERNEL preprocessor symbol is defined).

uts/common/syscall

System call implementations. Most syscalls are implemented in files matching their names. Note that some system calls are implemented in os/ or other subdirectories instead.

uts/common/tnf

Old tracing subsystem, not related to dtrace(7D).

uts/common/vm

Virtual memory subsystem.

uts/common/zmod

Compression/decompression library.

uts/i86pc

Architecture-dependent files for x86 machines. The architecture-dependent directories (i86pc, sun, sun4, sun4u) all have a set of subdirectories similar to common/ above.

uts/intel

ISA-dependent, architecture-independent files for x86 machines. Note that all architecture-independent source files are built into objects in this hierarchy.

uts/sfmmu

Code specific to the SpitFire memory management unit (UltraSPARC).

uts/sparc

ISA-dependent, architecture-independent files for SPARC machines. Note that all architecture-independent source files are built into objects in this hierarchy.
uts/sun
Sources common to all Sun implementations. Currently this contains a small number of device drivers and some headers shipped in /usr/include/sys.

uts/sun4
Sources common to all sun4* machine architectures.

uts/sun4u
Architecture-dependent sources for the sun4u architecture. Each system implementation has a subdirectory here:

* blade Serverblade1
* chalupa Sun-Fire-V440
* cherrystone Sun-Fire-480R
* daktari Sun-Fire-880
* darwin Ultra-5_10
* enchilada Sun-Blade-2500
* ents Sun-Fire-V250
* excalibur Sun-Blade-1000
* fjlite UltraAX-i2
* grover Sun-Blade-100
* javelin Ultra-250
* littleneck Sun-Fire-280R
* lw2plus Netra-T4
* lw8 Netra-T12
* makaha UltraSPARC IIE-NetraCT-40, UltraSPARC IIE-NetraCT-60
* montecarlo UltraSPARC I1I-NetraCT
* mxpu Sun-Fire-V240
* quasar Ultra-80
* serengeti Sun-Fire
* snowbird Netra-CP2300
* starcat Sun-Fire-15000
* starfire Ultra-Enterprise-10000
* sunfire Ultra-Enterprise
* taco Sun-Blade-1500
* tazmo Ultra-4

3.2.2 Object Directories
There are two basic strategies that can be used in the creation of object files and finished binaries (executables and libraries):
(a) place objects in a dedicated directory hierarchy parallel to the sources
(b) place objects in the same directories as the sources they are built from
OpenSolaris actually uses each of these approaches in different parts of the tree. Strategy (a) must be used for all kernel code and many libraries, is preferred for new code, and will be described in detail here. There are several legacy
variations on strategy (a) as well as instances in which strategy (b) is still used; the majority of these are in the cmd hierarchy. The entire uts hierarchy has been converted to strategy (a) as described below, and you will see this same approach used throughout much of the rest of the tree.

3.2.2.1 General Strategy for Kernel Objects First, each platform-independent module has zero or one build directory per architecture. An architecture in this case can be a machine (sun4u, i86pc) or a processor architecture (intel, sparc). The path to this location is always /usr/src/uts/<platform>/<module>. The module name in this case is what’s found in /kernel/drv or a similar location, and in the case of device drivers or STREAMS modules should always match the name of the man page describing that driver or module.

The only files normally present in this directory for a clean tree are makefiles. After a build, these directories contain one or more of obj32, obj64, debug32, and debug64 directories. These directories contain the object files and finally linked modules that are later installed into the kernel directory and other locations in the prototype.

"Implementation architecture"-independent modules are produced in individual directories (one per module) under the "instruction-set architecture" directory (i.e.: sparc). Similarly, "implementation architecture"-dependent modules are produced in individual directories under the "implementation architecture" directory (i.e.: sun4, sun4u, i86pc).

Platform-dependent modules (including "unix") may be built more than once in different locations. Platform-dependent modules are discussed in greater detail in <section>.

The sources are not contained in these build directories.

3.2.2.2 General Strategy for Command/Library Objects Most libraries and some commands and daemons are built using makefiles very similar to those used to build the kernel. Accordingly, intermediate objects, shared objects, libraries, and executables are built by small makefile fragments and placed in dedicated ISA-specific subdirectories.

Other commands’ build systems place objects in the same directories as the sources. See sections 3.2.2.3, 3.2.3.2, and 3.2.5 for more information on how commands are built.

3.2.2.3 Exceptions in cmd and lib Most of the cmd tree is directly based on the original System V Release 4 source, which uses strategy (b) described above. Since most commands and daemons do not need to provide both 32- and 64-bit versions, or do anything special when building for different architectures, this strategy is adequate and appropriate for most commands and has been applied even to new subdirectories. In situations in which architecture-dependent build options are needed or multiple ISA versions of a program must be delivered, this strategy is unworkable and the more general approach of multiple per-ISA
object directories must be used instead. This latter approach is similar to the
approach used for kernel modules.

The lib hierarchy is somewhat simpler; nearly all subdirectories must use
per-ISA object file locations and makefiles.

A few directories do not appear to follow any rule or pattern, such as
cmd/cmd-inet and cmd/agents. These are primarily historical artifacts of Sun’s
internal project organization.

3.2.3 Makefile Layout

This discussion is intended to provide a step-by-step explanation of what targets
exist and how they are built by the makefiles. I ignore the platform-specific
module architecture because it is unlikely to be of significant interest except
to hardware support engineers. The three main subtrees of interest are the
kernel (uts), commands and daemons (cmd), and libraries (lib). The next three
subsections cover these three subtrees in turn. There are also a handful of
makefiles which apply to all builds:

**usr/src/Makefile**

This is the top-level makefile. It drives builds for various targets in each
subdirectory. It is aware of the specific targets that need to be built in
each subdirectory in order to perform a complete build, and itself knows
how to create a skeleton proto area for later use by install and install.h
targets.

**usr/src/Makefile.lint**

All linting from the top level is driven by this makefile. It contains long
lists of directories known to be lint-clean and contains simple recursive
rules for rebuilding each subdirectory’s lint target. The actual linting is
driven by the lower-level makefiles.

**usr/src/Makefile.master**

**usr/src/Makefile.master.64**

These two makefiles contain generic definitions, such as build and instal-
lation tools locations, template macros for compilers, linkers, and other
tools to be used by other makefiles in defining rules, and global definitions
such as the ISA and machine names that apply to this build. Make-
file.master.64 contains definitions specific to 64-bit builds that override
the generic definitions.

**usr/src/Makefile.msg.targ**

Common targets for building message catalogues are defined here. Mes-
sage catalogues provide translations of messages for g11n purposes.
CHAPTER 3. THE SOURCE TREE

usr/src/Makefile.psm
This makefile defines the installation locations for platform-specific modules. These are analogous to the other kernel module install locations /kernel and /usr/kernel (see section 3.2.6 below for more information on kernel module installation).

usr/src/Makefile.psm.targ
Installation target definitions for platform-specific modules are defined here. This instructs the build system how to install files into the directories defined by Makefile.psm.

usr/src/Targetdirs
This is a set of definitions for the owner, group, and permissions of each directory that will be created by the installation process. It also contains information about special symbolic links to be installed for some 64-bit library versions.

3.2.3.1 Kernel Makefile Layout  The driving makefile for any module is located in the leaf directory (build directory) where the module and its component objects are built. After a 'make clobber' operation, the makefile should be the only file remaining in that directory. There are two other types of makefiles in the tree: suffixed and non-suffixed. Common definitions and rules needed by all leaf makefiles are contained in the suffixed makefiles; these are included by leaf makefiles. Non-suffixed makefiles generally invoke multiple lower-level makefiles with the same target so that many modules can be built with a single make invocation.

uts/Makefile
uts/sparc/Makefile
uts/sun4u/Makefile
uts/intel/Makefile
uts/intel/ia32/Makefile
uts/i86pc/Makefile

These makefiles generally are cognizant of the components made in sub-directories and invoke makefiles in those sub-directories to perform the actual build. Some targets (or pseudo-targets) may be directly built at this level (such as the cscope databases).

uts/Makefile.uts
Contains common definitions for all possible architectures.

uts/Makefile.targ
Contains common targets for all possible architectures.
uts/common/Makefile.files
uts/sun/Makefile.files
uts/sparc/Makefile.files
uts/sun4/Makefile.files
uts/sun4u/Makefile.files
uts/intel/Makefile.files
uts/intel/ia32/Makefile.files
uts/i86pc/Makefile.files

These makefiles are divided into two sections. The first section defines the object lists which comprise each module. The second section defines the appropriate header search paths and other machine-specific global build parameters.

uts/common/Makefile.rules
uts/sun/Makefile.rules
uts/sparc/Makefile.rules
uts/sun4/Makefile.rules
uts/sun4u/Makefile.rules
uts/intel/Makefile.rules
uts/intel/ia32/Makefile.rules
uts/intel/amd64/Makefile.rules
uts/i86pc/Makefile.rules

The files provide build rules (targets) which allow make to function in a multiple directory environment. Each source tree below the directory containing the makefile has a build rule in the file.

uts/sun4/Makefile.sun4
uts/sun4u/Makefile.sun4u
uts/intel/Makefile.intel
uts/intel/ia32/Makefile.ia32
uts/i86pc/Makefile.i86pc

These makefiles contain the definitions specific (defaults) to the obvious "implementation architecture". These rules can be overridden in specific leaf node makefiles if necessary.
CHAPTER 3. THE SOURCE TREE

uts/sun4u/unix/Makefile

uts/i86pc/unix/Makefile

Main driving makefile for building unix.

uts/sun4u/MODULE/Makefile (for MODULE in cgsix, cpu, kb, ...)

Main driving makefile for building MODULE.

uts/sun4u/genunix/Makefile

uts/i86pc/genunix/Makefile

Main driving makefile for building genunix.

Issuing the command ‘make’ in the uts directory will cause all supported, modularized kernels and modules to be built.

Issuing the command ‘make’ in a uts/ARCHITECTURE directory (i.e.: uts/sparc) will cause all supported, ”implementation architecture”-independent modules for ARCHITECTURE to be built.

Issuing the command ‘make’ in a uts/MACHINE directory (i.e.: uts/sun4u) will cause that kernel and all supported, ”implementation architecture”- dependent modules for MACHINE to be built.

The makefiles are verbosely commented. It is desired that they should stay this way.

3.2.3.2 Command Makefile Layout

Most command and daemon subdirectories follow one of two general layout rules, depending on where object files will be located (see sections 3.2.2.2 and 3.2.2.3). For ISA-dependent programs, the layout is similar to that used by the kernel. Programs which do not need ISA-dependent build behavior use a simplified makefile layout. In the description here, we use the example of a generic command called ”foocmd” whose sources are located in usr/src/cmd/foocmd. The makefiles relevant to building foocmd are:

usr/src/cmd/Makefile

Top-level driving makefile for all commands/daemons. This is a simple recursive makefile which is aware of which subdirectories should be built and will cause the given target to be rebuilt in each of them.

usr/src/cmd/Makefile.cmd

This makefile defines the installation directories and rules for installing executables into them.

usr/src/cmd/Makefile.cmd.64

Additional definitions specific to 64-bit builds are provided here.
CHAPTER 3. THE SOURCE TREE

**usr/src/cmd/Makefile.cmd.bsm**
This specialty makefile is used only by auditstat and dminfo. It provides some generic boilerplate rules.

**usr/src/cmd/Makefile.targ**
Basic target definitions for clobber, lint, and installation.

**usr/src/cmd/foocmd/Makefile**
Driving makefile for foocmd. Normally defines PROG but otherwise contains only boilerplate definitions and targets. This is almost always copied from another similar makefile. If foocmd does not require ISA-dependent build behavior, rules will normally be specified directly, including those for the install target. If foocmd does require ISA-dependent build behavior, this makefile will instead define SUBDIRS to include the ISA-specific versions that must be built, and define targets recursively. This will usually leave the install target definition for each ISA makefile and cause a link to $(ISAEXEC) to be created. See section 3.2.5 for more information on platform dependencies and $(ISAEXEC).

**usr/src/cmd/foocmd/Makefile.com**
Defines PROG, OBJS, SRCS, and includes Makefile.cmd. May also contain additional flags for compilation or linking. This makefile normally defines targets for all, $(PROG), clean, and lint; this portion is usually generic and would be copied from another similar makefile.

**usr/src/cmd/foocmd/*/Makefile**
ISA-specific makefiles, which may define additional ISA-specific flags or targets, and will generally include its own install target to install the ISA-specific program(s) it builds.

### 3.2.3.3 Library Makefile Layout
Most library subdirectories follow the same general layout, which is similar to the command layout. Unlike commands, most libraries are built for both 32- and 64-bit architecture variants, so ISA-specific directories will almost always be present. Therefore, the overall build structure for each library is similar to that of the kernel. See 3.3.3.3 Adding New Libraries for more detailed information on library makefiles.

### 3.2.4 Makefile Operation
This section describes in detail how make dependencies and rules are built up, and how each individual makefile contributes to the build process. Once you understand how the makefiles work, modifying existing build rules and creating new ones are mainly an exercise in copying existing files and making minor modifications to them. We begin with the kernel makefiles, then address the very similar command and library makefiles.
3.2.4.1 Kernel Makefile Operation  There are two parts to the make dependencies: a set of targets and a set of rules.

The rules are contained in the Makefile.rules files in each directory under uts. They cover how to compile a source file in some other directory (common/io for example) into an object file in the build directory described above. They also describe which sources are needed to build a given object file. The order in which they are specified determines which source file is needed if there are multiple source files matching the same basename. The key to all this is understanding that all rules and targets are static and module-independent, but variables set in each module’s build directory (leaf) makefile define the module-specific build information.

The targets are of two kinds: the first are ”make targets,” generic directives you can give make like ”all” or ”install” which are executed in each build directory. The build directory makefiles in turn include other makefiles that contain rules for building objects and modules. The second type of target is more interesting - they are the actual module names (directories, in fact) that must be built. These variables are called KMODS and XMODS.

If KMODS contains, for example, ”aac ata asy”, then for each of the aac, ata, and asy build directories, the make system will cd to that directory and rerun make with whatever target (all, install, etc) is given. The rules for that module are derived from all the included makefiles; the crucial variables to look for are of the form XXX.OBJS, where XXX is the module name. More than one makefile may contain definitions adding to each of these object lists, because building a module may require different objects on each platform.

XMODS is used only for building export versions of the commercial Solaris product; it is not used in building OpenSolaris.

So for example we see:

```
common/Makefile.files:ASY_OBJS +=  asy.o
```

This indicates that the asy.o object is needed to build the ASY module. The makefile in the <platform>/asy directory will reference ASY_OBJS and thus know what to build. So in intel/asy/Makefile we see:

```
MODULE = asy
OBJECTS = $(ASY_OBJS:%=$(OBJS_DIR)/%)
```

So, what good is this? Plenty. Because our ALL_TARGET is also defined:

```
ALL_TARGET = $(BINARY) $(SRC_CONFILE)
```

Makefile.uts defines BINARY as follows:

```
BINARY = $(OBJS_DIR)/$(MODULE)
```

And OBJS_DIR is the individual module build directory I described at the beginning. So, make knows we need to build usr/src/uts/intel/asy/obj32/asy. Makefile.targ defines the rules for doing so, too:
We're almost there - OBJECTS have to come from somewhere. We find this defined in intel/asy/Makefile:

```
OBJECTS = $(ASY_OBJS:%=$(OBJS_DIR)/%)
```

And that's that. We have a module (asy) that has a BINARY (obj32/asy) which depends on OBJECTS made from ASY_OBJS. The rules for each entry in ASY_OBJS is in one of the Makefile.rules files; they look something like this:

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

Each object is built in accordance with these rules. The rule for $(BINARY) then links them into the module.

We can now put it all together. The extensive use of macros and local per-module definitions provides much greater flexibility, but in the example given, the equivalent makefile fragment would look very familiar:

```
KMODS += asy

all: $(KMODS)

asy: obj32/asy.o
  $(LD) -r $(LDFLAGS) -o $@ obj32/asy.o
...

obj32/asy.o: ../../../common/io/asy.c
  $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@
...
```

### 3.2.4.2 Command Makefile Operation

Command makefiles vary widely, as described above. Therefore it is impossible to provide a "representative example" of a command makefile. However, understanding the material in sections 3.2.4.1 and 3.3.3.3 will provide you with the necessary knowledge to work with command makefiles. As always, if you are looking for information about a specific command, it is usually best to use the mailing lists and other community resources.

### 3.2.4.3 Library Makefile Operation

Section 3.3.3.3 describes library makefile operation in detail.
3.2.5 ISA Dependencies

Most code in OpenSolaris is generic and platform- and ISA-independent; that is, it is portable code that can be compiled and run on any system for which a suitable compiler is available. However, even portable code must sometimes deal with specific attributes of the system on which it runs, or must be provided compiled for multiple supported architectures so that users can run or link with the architecture-specific version of their choosing.

There are two distinct situations in which multiple architecture-specific builds must be shipped: commands that, by their nature, must have visibility into the system’s hardware characteristics, and libraries, which must be built separately for each architecture a developer may wish to target. Each case is discussed in its own subsection below.

3.2.5.1 ISA-Dependent Commands  Commands (normally executable programs or daemons) usually do not depend on the ISA on which they are executed; that is, they will be built and installed in the same way regardless of the ISA they are built for. Some commands, however, mainly those which operate on other processes or manipulate binary objects, must behave differently depending on the ISA(s) supported by the system and/or their targets. The /usr/lib/isaexec wrapper program provides this behavior in the following way:

First, one or more ISA-specific programs are installed into ISA-specific directories in the system. These are subdirectories of those directories which ordinarily contain executable programs, such as /usr/bin. On SPARC systems, ISA subdirectories might include /usr/bin/sparcv7 and /usr/bin/sparcv9 to contain 32- and 64-bit binaries, respectively. Other ISAs may be supported by other systems; you can find out which your system supports by using isalist(1).

Next, a hard link is installed into the ordinary binary directory (in the example above, /usr/bin) targeting /usr/lib/isaexec. This will cause isaexec to be invoked with the name of the ISA-dependent program the user has executed.

When this happens, isaexec will select the most appropriate ISA-specific program installed in the first step and exec(2) it. This is transparent, so a user will not notice that a separate program has been invoked.

A similar mechanism (/usr/lib/platexec) exists for platform-specific programs; these are much less common even than ISA-specific programs.

Because it needs to interact both with user processes and kernel data structures, dtrace(1) is an example of a command that must provide multiple ISA-specific versions.

**cmd/dtrace** contains two makefiles: Makefile and Makefile.com. It also contains a single source file, dtrace.c, and four ISA directories: amd64, i386, sparc, and sparcv9. Each contains a single makefile. The top-level makefiles should look very familiar after our tour of the uts makefiles; Makefile is mainly boilerplate, customized only to specify that the program produced is ISA-specific and therefore a link to the isaexec program needs to be installed into /usr/sbin in the proto area:
install: $(SUBDIRS)
    -$(RM) $(ROOTUSRSBINPROG)
    -$(LN) $(ISAEXEC) $(ROOTUSRSBINPROG)

Otherwise, it simply leaves all building and installation to the makefiles in
the ISA subdirectories. Makefile.com is also boilerplate; it specifies the name of
the program to build and the objects from which it must be built:

    PROG = dtrace
    OBJS = dtrace.o

It then transforms the objects into sources:

    SRCS = $(OBJS:%.o=../%.c)

and includes the generic top-level Makefile.cmd fragment to pick up build
rules.

Compilation and linkage flags are specified:

    CFLAGS += $(CCVERBOSE)
    CFLAGS64 += $(CCVERBOSE)
    LDLIBS += -ldtrace -lproc

and the remainder consists of a handful of generic build rules. One feature
is noteworthy: an extra ../ is prepended to all paths in this fragment. This
is because it is always included by makefiles interpreted from the ISA-specific
build directory, which is one level deeper than Makefile.com itself.

The makefiles in the ISA-specific build directories are even simpler; they
each consist of only two lines:

    include ../Makefile.com
    install: all $(ROOTUSRSBINPROG32)

The first line picks up all common definitions in the makefile we just ex-
amined, and the second specifies (indirectly) the location in which the pro-
gram is to be installed. To complete the picture, we need to briefly examine
usr/src/cmd/Makefile.cmd, which defines the various locations in which pro-
grams can be installed. Since we have not defined ROOTUSRSBINPROG32, it
must be defined there, and indeed it is:

    ROOTUSRSBIN= $(ROOT)/usr/sbin
    ROOTUSRSBINPROG=$(PROG:%=$(ROOTUSRSBIN)/%)
    ROOTUSRSBIN32= $(ROOTUSRSBIN)/$(MACH32)
    ROOTUSRSBINPROG32= $(PROG:%=$(ROOTUSRSBIN32)/%)

Note that the program is actually installed in /usr/sbin/$(MACH32), which
is not normally in a user’s search path. But remember that we also created a
link from $(ROOTUSRSBINPROG) to $(ISAEXEC); expanding these variables
shows that we will actually install two separate files:
CHAPTER 3. THE SOURCE TREE

$(ROOT)/usr/sbin/dtrace \rightarrow $(ROOT)/usr/lib/isaexec
$(ROOT)/usr/sbin/$(MACH32)/dtrace (our executable program)

Since the isaexec program is responsible for selecting an ISA-specific program from among several options depending on the system’s supported instruction sets, we can see that a typical system will have one or more dtrace(1) binaries installed and that when dtrace(1) is run by the user, isaexec will select the appropriate one from those installed in the ISA-specific subdirectories of /usr/sbin.

As we might expect, other ISA-specific makefiles under cmd/dtrace install their binaries into ROOTUSRSBIN64, also defined in usr/src/cmd/Makefile.cmd.

3.2.5.2 ISA-Dependent Libraries

Libraries, unlike commands, must almost always be built for all supported ISAs and have separate ISA-specific versions installed on, at minimum, every system that supports that ISA. In practice, libraries for each ISA in the system’s processor family are installed. This allows developers on that system to write programs using any supported ISA and link them with the appropriate system libraries (it is not possible, for example, to link 32-bit object files and 64-bit libraries, nor vice versa).

Libraries use a build system very similar to that used by commands which must provide multiple ISA-specific executable programs. We examine here the makefiles used by dtrace(1)’s library counterpart libdtrace(3LIB) and how they support building and installing multiple ISA-specific versions of the dtrace library.

lib/libdtrace contains two makefiles: Makefile and Makefile.com. It also contains a directory for common code and four ISA directories: amd64, i386, sparc, and sparcv9. Each ISA directory contains a single makefile and in some cases some ISA-specific sources. The top-level Makefile should once again look very familiar; is very similar to the uts and cmd makefiles and is mainly boilerplate. There are a few customizations to add a yydebug target and to tell top-level makefiles that lex and yacc generated files should not have cross-references built against them.

yydebug := TARGET = yydebug
....
lint yydebug: $(SUBDIRS)

and

XRDIRS = common i386 sparc sparcv9
XRDEL = dt_lex.c dt_grammar.c Makefile*

Otherwise, it simply leaves all building and installation to the makefiles in the ISA subdirectories. Makefile.com is somewhat more interesting. It first specifies the name of the library to build (in static format; shared library names are translated as we shall see below), and the library version. These are common to all library makefiles. Next, we see long lists of source files used to build the library, and an OBJECTS specification. This particular library also has some
unique characteristics, including the D runtime init code, built as a separate object file (drti.o), and a number of D "header" files which are installed into a separate directory /usr/lib/dtrace. You can see these unique features here:

```make
DRTISRC = drti.c
DRTIOBJ = $(DRTISRC:.c=.o)
```

```make
DLIBSRSCS += \
  errno.d \ 
  io.d \ 
  procfs.d \ 
  regs.d \ 
  sched.d \ 
  signal.d \ 
  unistd.d
```

```make
ROOTDLIBDIR = $(ROOT)/usr/lib/dtrace
ROOTDLIBDIR64 = $(ROOT)/usr/lib/dtrace/64
```

```make
$(ROOTDLIBDIR):
  $(INS.dir)
```

```make
$(ROOTDLIBDIR64):
  $(ROOTDLIBDIR)
```

```make
$(ROOTDLIBDIR)/%.d: ../common/%.d
  $(INS.file)
```

```make
$(ROOTDLIBDIR)/%.d: ../$(MACH)/%.d
  $(INS.file)
```

```make
$(ROOTDLIBDIR)/%.d: %.d
  $(INS.file)
```

Note that some of the D "headers" are in turn generated from other files; this makefile includes rules for performing these transformations.

Once the LIBSRSCS and OBJECTS have been defined, the generic library makefile fragment Makefile.lib is included to provide target and directory definitions.

Binary objects are built in the pics/ subdirectory. pics stands for position-independent code(s), referring to the fact that dynamic shared libraries are built in such a way as to be loaded at any address. This is an historical artifact; static linking is not supported in OpenSolaris, so the non-PIC objects which would have been used to build an archive-style library are not built. The rules for building are given as:

```make
pics/%.o: ../../../$(MACH)/%.c
  $(COMPILE.c) -o $@ $<
```

$(POST_PROCESS_O)
Note that because this makefile fragment is included by each ISA’s makefile, the objects produced will be placed relative to that makefile, in each ISA directory. Once again, this also means that included makefile fragments include an additional ../ component.

The libdtrace Makefile.com also contains a large number of custom rules to accommodate building a lexer and parser from lex(1) and yacc(1) definitions; these sources are then compiled specially using the custom rules and are removed when the clean or clobber target is executed.

The makefiles in the ISA-specific build directories are once again very simple. The SPARC-specific makefile, for example, defines additional assembler flags, specifies the directory where the library interface map is located, and then includes the common makefile:

```
ASFLAGS += -D_ASM -K PIC -P
MAPDIR = ../spec/sparc
include ../Makefile.com
```

The makefile may also add a few ISA-specific sources to the build list; for example, sparc/Makefile includes:

```
SRCS += dt_asmsubr.s
OBJECTS += dt_asmsubr.o
```

And once again, a separate install target is provided to indicate the directories where these specific libraries and headers should be installed:

```
install yydebug: all $(ROOTLIBS) $(ROOTLINKS) $(ROOTLINT) \ $(ROOTDLIBS) $(ROOTDOBJS)
```

To support building a separate set of 64-bit objects, sparcv9/Makefile includes slightly different definitions and a different install target:

```
MAPDIR = ../spec/sparcv9
include ../Makefile.com
include ../../Makefile.lib.64
CPPFLAGS += -D_ELF64
```

```
install yydebug: all $(ROOTLIBS64) $(ROOTLINKS64) $(ROOTLINT64) \ $(ROOTDLIBS) $(ROOTDOBJS64)
```
Note the additional inclusion of Makefile.lib.64 and the additional flag for 64-bit ELF targets. Also, the 64-bit-specific versions of the installation files are used, which as we can see in Makefile.lib cause them to be installed into `/usr/lib/sparcv9`:

```makefile
ROOTLIBDIR= $(ROOT)/usr/lib
ROOTLIBDIR64= $(ROOT)/usr/lib/$(MACH64)
... 
ROOTLIBS= $(LIBS:%=$(ROOTLIBDIR)/%)
ROOTLIBS64= $(LIBS:%=$(ROOTLIBDIR64)/%)
```

The linker (`ld(1)`) and runtime loader (`ld.so.1(1)`) know which of these directories to search for libraries depending on the bitness of the objects being linked or the program being run, respectively. For more information about 32-versus 64-bit linking, consult the Solaris Linkers and Libraries Guide, which can be found at §??.

### 3.2.6 Understanding Kernel Module Installation

The install target causes binaries and headers to be installed into a hierarchy called the prototype or proto area (see section 4.4.1 for more information on the prototype area’s contents and purpose). The root of the proto area is defined by the environment variable ROOT. This variable is used in defining `ROOT_MOD_DIR` and `USR_MOD_DIR` in `usr/src/uts/Makefile.uts`:

```makefile
ROOT_MOD_DIR = $(ROOT)/kernel
USR_MOD_DIR = $(ROOT)/usr/kernel
```

All modules are installed below one of these directories. The exact subdirectory used depends on the definition of `ROOTMODULE` in the module build directory, which is discussed below.

Additional files to be installed may be given by the `INSTALL_TARGET` definition in this makefile. For example, the asy makefile (`usr/src/uts/intel/asy/Makefile`) specifies:

```
INSTALL_TARGET = $(BINARY) $(ROOTMODULE) $(ROOT_CONFFILE) $(ROOT_ITUFILE)
```

Since this is a common set of installation targets, let’s look at how each component is derived.

`$(BINARY)` is simply the module itself; we’ve discussed its dependencies and rules in detail above. Including it here requires it to be fully built before any installation is attempted.

`$(ROOTMODULE)` is a name transform on the name of the module, relocating it into the prototype area. For example, `usr/src/uts/intel/asy/Makefile` defines:

```
ROOTMODULE = $(ROOT_DRV_DIR)/$(MODULE)
```
CHAPTER 3. THE SOURCE TREE

ROOT_DRV_DIR requires a bit of additional explanation; it is one of many module installation subdirectories defined in Makefile.uts. Because the ending location of binary kernel modules depends on the machine type, specifically its 32 versus 64 bitness, there are really several definitions. First, the subdirectory definition for each 64-bit machine type:

\[
\begin{align*}
\text{SUBDIR64}_{\text{sparc}} &= \text{sparcv9} \\
\text{SUBDIR64}_{\text{i386}} &= \text{amd64} \\
\text{SUBDIR64} &= \$(\text{SUBDIR64}_{\$(MACH)})
\end{align*}
\]

Next, the 32- and 64-bit installation directories for each type of module:

\[
\begin{align*}
\text{ROOT_KERN_DIR}_{32} &= \$(\text{ROOT_MOD_DIR}) \\
\text{ROOT_DRV_DIR}_{32} &= \$(\text{ROOT_MOD_DIR})/\text{drv} \\
\text{ROOT_DTRACE_DIR}_{32} &= \$(\text{ROOT_MOD_DIR})/\text{dtrace} \\
\text{ROOT_EXEC_DIR}_{32} &= \$(\text{ROOT_MOD_DIR})/\text{exec} \\
\ldots
\end{align*}
\]

\[
\begin{align*}
\text{ROOT_KERN_DIR}_{64} &= \$(\text{ROOT_MOD_DIR})/\$(\text{SUBDIR64}) \\
\text{ROOT_DRV_DIR}_{64} &= \$(\text{ROOT_MOD_DIR})/\text{drv}/\$(\text{SUBDIR64}) \\
\text{ROOT_DTRACE_DIR}_{64} &= \$(\text{ROOT_MOD_DIR})/\text{dtrace}/\$(\text{SUBDIR64}) \\
\text{ROOT_EXEC_DIR}_{64} &= \$(\text{ROOT_MOD_DIR})/\text{exec}/\$(\text{SUBDIR64}) \\
\ldots
\end{align*}
\]

And finally the selection of either the 32- or 64-bit directory based on the actual bitness of this build:

\[
\begin{align*}
\text{ROOT_KERN_DIR} &= \$(\text{ROOT_KERN_DIR}_{\$(CLASS)}) \\
\text{ROOT_DRV_DIR} &= \$(\text{ROOT_DRV_DIR}_{\$(CLASS)}) \\
\text{ROOT_DTRACE_DIR} &= \$(\text{ROOT_DTRACE_DIR}_{\$(CLASS)}) \\
\text{ROOT_EXEC_DIR} &= \$(\text{ROOT_EXEC_DIR}_{\$(CLASS)}) \\
\ldots
\end{align*}
\]

There are similar definitions for \text{USR}XXX_DIR based in \$(ROOT)/usr/kernel. See \text{usr}/src/uts/Makefile.uts for the complete list of possible \text{ROOT}, and \text{USR}, installation directories.

The rules for installing files into each of these directories are given in \text{usr}/src/uts/Makefile.targ:

\[
\begin{align*}
\$(\text{ROOT_MOD_DIR})/\%: & \quad \$(\text{OBJJS_DIR})/\% \quad \$(\text{ROOT_MOD_DIR}) \text{ FRC} \\
& \quad \$(\text{INS.file}) \\
\$(\text{ROOT_DRV_DIR})/\%: & \quad \$(\text{OBJJS_DIR})/\% \quad \$(\text{ROOT_DRV_DIR}) \text{ FRC} \\
& \quad \$(\text{INS.file}) \\
\ldots
\end{align*}
\]

These rules install the file currently located in \$(\text{OBJJS_DIR}), described in detail above, into the appropriate directory in the proto area.

\$(\text{ROOT_CONFFILE}) is the module’s configuration file (most commonly used for drivers), transformed into the proto area. The CONFFILE variables are defined in \text{usr}/src/uts/Makefile.uts:
CHAPTER 3. THE SOURCE TREE

40

CONFFILE = $(MODULE).conf
SRC_CONFFILE = $(CONF_SRCDIR)/$(CONFFILE)
ROOT_CONFFILE_32 = $(ROOTMODULE).conf
ROOT_CONFFILE_64 = $(ROOTMODULE:%/$(SUBDIR64)/$(MODULE)=%/$(MODULE)).conf
ROOT_CONFFILE = $(ROOT_CONFFILE_$(CLASS))

Each module’s Makefile defines CONF_SRCDIR to specify the location of the configuration file that should be installed on this platform. The asy makefile defines:

CONF_SRCDIR = $(UTSBASE)/common/io

which causes usr/src/uts/common/io/asy.conf to be installed in the proto area as /kernel/drv/asy.conf. Note that the configuration file installation directory does not depend on whether this is a 32- or 64-bit build.

$(ROOT ITUFILE) is an not used by the standard build.

Note that although it is confusing, installation targets are always named ROOTMODULE, ROOT_CONFFILE, and so on, even if they are actually installed into one of the USR_ directories. This simplifies the higher-level makefiles.

3.2.7 Finding Sources for a Module

As we’ve seen, the sources are derived from $(XXX OBJS) and the included rules. Now, here’s the tricky part - finding where those sources are. The easiest way by far is to run make from the module’s object directory and observe the compilation commands that are run. The path to each source file will be included in the output. This is the simplest and recommended way to find which source files are needed to build a module. However, if this does not work (perhaps the makefile itself is buggy) or if you want to gain greater understanding, the rest of this section describes logical processes for finding source files.

Since you know the basenames (they are simply the XXX OBJS entries with .o replaced by .c) you can use a simple

$ find usr/src/uts -name asy.c

Suppose you find two source files with the same name - which is actually used? There are two ways to figure this out. The lazy way is to use make -d to show you the exact dependencies being used. Alternatively, you could reason it out, starting with the fact that not all source files are used on your platform (i.e., usr/src/uts/sparc/* is not used by x86 builds), and then looking at the specific order of the rules in each Makefile.rules. The more specific rules are listed first and take precedence over later rules (in fact they all add a dependency to the specific object file in question, but since the rules all use $<, only the first dependency is actually used).

Non-kernel source files are easier to locate, since the cmd and lib directories are generally organized with one subdirectory for each library or command, named accordingly. The major exceptions are cmd/sgs, which contains
the entire binary tools subsystem, and cmd/cmd-inet, which contains most of
the Berkeley networking commands and daemons as well as some additional
network-related utilities, laid out in a BSD-style directory structure.

3.2.8 Source Files not Included

Some sources used to build the branded Solaris product are not included in
OpenSolaris. There are three main reasons for this:

The source is encumbered.

Encumbered sources are those to which Sun has limited rights that do
not include distribution in source form. Many encumbered sources will be
replaced over time.

Law or regulation prevents us from distributing it.

Restrictions on distribution of Open Source code implementing cryptogra-
phy and other technology has been relaxed in recent years; however, some
sources are still restricted. This is especially true during the pilot phase,
because our prerelease license is not Open Source. Most of these sources
are expected to be available under an Open Source license at public launch.

Consolidations other than §?? are not yet included.

OpenSolaris does not yet include code from consolidations other than ON
(Operating System and Networking). These consolidations include, for
example, the X window system, CDE, the Java virtual machine and com-
piler tools, and the graphics device drivers and X server modules. More
of these consolidations may be added in the future.

To accommodate fully functional builds even though some sources are miss-
ing, a set of encumbered binaries are available, and the build system has been
modified to make use of them. However, some makefiles may inadvertently refer
to these encumbered files; if you notice any instances of this you should file a
bug. Finally, some sources may be included but depend on other sources which
are not included; in these cases, neither will be built even though a subset of
the sources is available for inspection.

3.3 Using Your Workspace

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

3.3.1 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:

```bash
$ 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.

### 3.3.2 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 OpenSolaris and may not compile.

When you edit source files, be sure that you are familiar with the style guide; see section 7.2 for more details. 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.

### 3.3.3 Modifying the 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.

#### 3.3.3.1 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.

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

1) 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:

```
$(OBJ_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. See section 3.2.3 for more
information about specific makefiles.

2) 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

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

```
MODULE = foofs
OBJECTS = $(FOOFS_OBJS:%=$(OBJ_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`).)
CHAPTER 3. THE SOURCE TREE

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

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

CLEANFILES += $(MODSTUBS_O)

4) 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.

Any additional questions can be easily answered by looking at the many existing examples or by using the mailing lists and fora at §??.

3.3.3.2 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.

0) Create the build directory under the appropriate "instruction set architecture" build directory (i.e.: sparc/MODULE).

1) 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).

2) 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.

3) 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 prototypes in usr/src/pkgdefs to reflect that such changes.

3.3.3.3 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:
CHAPTER 3. THE SOURCE TREE

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

\[
\begin{align*}
\text{install\_h: } & \text{(ROOTHDRS)} \\
\text{check: } & \text{(CHECKHDRS)}
\end{align*}
\]

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:

1. 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.

2. Providing a common "descend into SUBDIRS" rule:

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

```
FRC:
```

3. 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.

4. 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):

```makefile
include ../Makefile.lib

HDRS = libinetutil.h
HDHDR = common
SUBDIRS = $(MACH)
$(BUILD64)SUBDIRS += $(MACH64)
all := TARGET = all
clean := TARGET = clean
clobber := TARGET = clobber
install := TARGET = install
lint := TARGET = lint

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 §??, 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
CHAPTER 3. THE SOURCE TREE

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 COMPILER.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 \section{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.
3.4 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 with the main gate is often a good compromise.

3.4.1 Updating from the Parent

TBD (CVS)

3.4.2 Resolving Conflicts

TBD (CVS has no useful filemerge)

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 gate has pushed out to CVS) and resubmit your source patch. This reduces the chance that your change will break other changes being made to the same area of code.
Chapter 4

Building OpenSolaris

This chapter discusses two commonly-used ways to build OpenSolaris, nightly(1)/bldenv(1) and make(1)/dmake(1). The former provides a high degree of automation and fine-grained control over each step in a full or incremental build of the entire workspace. Using make(1) or dmake(1) directly provides much less automation but allows you to build individual components more quickly. Section 4.1 is common to both methods; 4.2 describes nightly(1) and bldenv(1), and 4.3 describes the use of low-level make(1) targets. Finally, section 4.4 describes what results from a full build and how these intermediate products can be used.

4.1 Environment Variables

This section describes a few of the environment variables that affect all OpenSolaris builds, regardless of the build method. See 4.2 Using nightly and bldenv for information on environment variables and files that affect nightly(1) and bldenv(1).

**CODEMGR_WS**

This variable is normally used by TeamWare (Sun’s internal source code management system). Although a workspace that does not use TeamWare does not need this variable, you will occasionally see it referenced when setting other variables. It should be set to the root of your workspace. It is highly recommended to use bldenv(1) to set this variable as it will also set several other important variables at the same time. See section 3.4.1 for more information on bldenv(1).

**SRC**

This variable must be set to the root of the source tree within your workspace; that is, `{CODEMGR_WS}/usr/src. It is used by numerous makefiles and by nightly(1). This is only needed if you are building. bldenv(1) will set this variable correctly for you.
MACH

The instruction set architecture of the machine as given by 
uname -p, e.g. 
sparc, i386. This is only needed if you are building. 
bldenv(1) will set this 
variable correctly for you; it should not be changed. If you prefer, you 
can also set this variable in your dot-files, and use it in defining PATH 
and any other variables you wish. If you do set it manually, be sure not 
to set it to anything other than the output of `/usr/bin/uname -p` on the 
specific machine you are using:

Good:

\[
\text{MACH} = '/usr/bin/uname -p' 
\]

Bad:

\[
\text{MACH} = \text{sparc} 
\]

ROOT

Root of the proto area for the build. The makefiles direct the instal-
lation of header files and libraries to this area and direct references to 
these files by builds of commands and other targets. It should be ex-
pressed in terms of $CODEMGR_WS. See section 4.4.1 for more informa-
tion on the proto area. If bldenv(1) is used, this variable will be set to 
$\{CODEMGR_WS\}/proto/root$\{MACH\}.

PARENT_ROOT

PARENT_ROOT is the proto area of the parent workspace. This can be 
used to perform partial builds in children by referencing already-installed 
files from the parent. Setting this variable is optional.

MAKEFLAGS

This variable has nothing to do with OpenSolaris; however, in order for 
the build to work properly, make(1) must have access to the contents 
of the environment variables described in this section. Therefore the 
MAKEFLAGS environment variable should be set and contain at least 
"e". bldenv(1) will set this variable for you; it is only needed if you are 
building. It is possible to dispense with this by using 'make -e' if you are 
building using make(1) or dmake(1) directly, but use of MAKEFLAGS is 
strongly recommended.

SPRO_ROOT

By default, it is expected that a working compiler installation exists in 
/ws/onnv-tools/SUNWspro/SOS8 (this path is used internally at Sun). 
This variable can be set to override that location; since you probably do 
not have compilers installed in the default location, you will need to set
this, normally to /opt/SUNW$pro. You can see how this works by looking
at usr/src/Makefile.master; if you need to override the default, however,
you should do so via the environment variable.

GNU_ROOT

The GNU C compiler is used to build the 64-bit kernel for amd64 sys-
tems. By default, if building on an x86 host, the build system assumes
there is a working amd64 gcc installation in /usr/sfw. Although it is not
recommended, you can use a different gcc by setting this variable to the
gcc install root. See usr/src/Makefile.master for more information.

4.2 Using nightly and bldenv

There are many steps in building OpenSolaris: creation of the proto area, com-
pling and linking binaries, generating lint libraries and linting the sources, build-
ging packages and BFU archives, and verifying headers, packaging, and proto area
changes. Fortunately, a single utility called nightly(1) automates all these steps
and more. It is controlled by a single environment file, the format of which
is shared with bldenv(1). This section describes what nightly(1) does for you,
what it does not, and how to use it.

nightly(1) can automate most of the build and source-level checking pro-
cesses. It builds the source, generates BFU archives, generates packages, runs
lint(1), does syntactic checks, and creates and checks the proto area. It does
not perform any runtime tests such as unit, functional, or regression tests; you
must perform these separately, ideally on dedicated systems.

Despite its name, nightly(1) can be run manually or by cron(1M) at any
time; you must run it yourself or arrange to have it run once for each build you
want to do. nightly(1) does not start any daemons or repetitive background
activities: it does what you tell it to do, once, and then stops.

After each run, nightly(1) will leave a detailed log of the commands it ran and
their output; this is normally located in $CODEMGR_WS/log/log.XXXX/nightly.log,
where XXXX is the date in MMDD format, but can be changed as desired. If
such a log already exists, nightly(1) will rename it for you.

In addition to the detailed log, you (actually, the address specified in the
MAILTO environment variable) will also receive an abbreviated completion re-
port when nightly(1) finishes. This report will tell you about any errors or
warnings that were detected and how long each step took to complete. It will
list errors and warnings as differences from a previous build (if there is one);
this allows you to see what effect your changes, if any, has had. It also means
that if you attempt a build and it fails, and you then correct the problems and
rebuild, you will see information like:

< dmake: Warning: Command failed for target ‘yppasswd’
< dmake: Warning: Command failed for target ‘zcons’
< dmake: Warning: Command failed for target ‘zcons.o’
< dmake: Warning: Command failed for target ‘zdump’
CHAPTER 4. BUILDING OPENSOLARIS

4.2.1 Options

nightly(1) accepts a wide variety of options and flags that control its behavior. Many of these options control whether nightly(1) performs each of the many steps it can automate for you. These options may be specified in the environment file or on the command line; options specified on the command line take precedence. See nightly(1) for the complete list of currently accepted options and their effect on build behavior.

4.2.2 Using Environment Files

nightly(1) reads a file containing a set of environment definitions for the build. This file is a simple shell script, but normally just contains variable assignments. All variables described in section 2.5 Environment Variables and below in 4.2.3 Variables can be set in the nightly(1) environment file; however, common practice is to use the developer, or gatekeeper, or opensolaris environment files, as appropriate, and modify one of them to meet your needs. The name of the resulting environment file is then passed as the final argument to nightly(1). The sample environment files are available in usr/src/tools/env.

4.2.3 Variables

Although any environment variables can be set in a nightly(1) environment file, this section lists those which are used directly by nightly(1) to control its operation and which are commonly changed. The complete list of variables and options is found in nightly(1).

- NIGHTLY_OPTIONS
- CODEMGR_WS
- CLONE_WS
- STAFFER
- MAILTO
- MAKEFLAGS

4.3 Using Make

Although nightly(1) can automate the entire build process, including source-level checks and generation of additional build products, it is not always necessary. If you are working in a single subdirectory and wish only to build or
lint that subdirectory, it is usually possible to do this directly without relying on nightly(1). This is especially true for the kernel, and if you have not made changes to any other part of your workspace, it is advantageous to build and install only the kernel during intermediate testing. See section 5.2 for more information on installing test kernels.

You will need to set up your environment properly before using make(1) directly on any part of your workspace. You can use bldenv(1) to accomplish this; see 3.4.1 Getting Ready to Work and 4.2 Using nightly and bldenv for more information on this command.

Because the makefiles use numerous make(1) features, some versions of make will not work properly. Specifically, you cannot use BSD make or GNU make to build your workspace. The dmake(1) included in the OpenSolaris tools distribution will work properly, as will the make(1) shipped in /usr/ccs/bin with Solaris and OpenSolaris. If your version of dmake is older (or, in some cases, newer), it may fail in unexpected ways. While both dmake(1) and make(1) can be used, dmake(1) is normally recommended because it can run multiple tasks in parallel or even distribute them to other build servers. This can improve build times greatly.

The entire uts directory allows you to run make commands in a particular build subdirectory (see sections 3.3.2 though 3.3.6) to complete only the build steps you request on only that particular subdirectory. Most of the cmd and parts of the lib subdirectories also allow this; however, some makefiles have not been properly configured to regenerate all dependencies. All subdirectories which use the modern object file layout (see section 3.3.2) should generally work without any problems.

There are several valid targets which are common to all directories; to find out which ones apply to your directory of interest and which additional targets may be available, you will need to read that directory’s makefile.

Here are the generic targets:

- **all**
  Build all derived objects in the object directory.
- **install**
  Install derived objects into the proto area defined by ${ROOT}.
- **install_h**
  Install header files into the proto area defined by ${ROOT}.
- **clean**
  Remove intermediate object files, but do not remove ”complete” derived files such as executable programs, libraries, or kernel modules.
- **clobber**
  Remove all derived files.
- **check**
  Perform source-specific checks such as source and header style conformance.
- **lint**
  Generate lint libraries and run all appropriate lint passes against all code which would be used to build objects in the current directory.
CHAPTER 4. BUILDING OPENSOLARIS

4.4 Build Products

A fully-built source tree is not very useful by itself; the binaries, scripts, and configuration files that make up the OpenSolaris system are still scattered throughout the tree. The makefiles provide the install targets to produce a proto area and package tree, and other utilities can be used to build additional conglomerations of build products in preparation for integration into a full Wad Of Stuff build or installation on one or more systems for testing and further development. The nightly(1) program can automate the generation of each build product. Alternatively, the Install program (see section 5.2) can be used to construct a kernel installation archive directly from a fully-built/usr/src/uts.

Section 4.4.1 describes the proto area, the most basic collection of built objects and the first to be generated by the build system. Section 4.4.2 describes BFU archives, which are used to upgrade a system with a full Solaris/OpenSolaris installation to the latest ?? bits. Section 4.4.3 describes the construction of the OpenSolaris package tree for ON deliverables.

4.4.1 Proto Area

The install target causes binaries and headers to be installed into a hierarchy called the prototype or proto area. Since everything in the proto area is installed with its usual paths relative to the proto area root, a complete proto area looks like a full system install of the ON bits. However, a proto area can be constructed by arbitrary users, so the ownership and permissions of its contents will not match those in a live system. The root of the proto area is defined by the environment variable ROOT, which defaults to /proto. If you use bldenv(1) to set up your environment, ROOT defaults instead to ${CODEMGR_WS}/proto/root,${MACH}.

The proto area is useful if you need to copy specific files from the build into your live system. It is also compared with the parent’s proto area and the packaging information by tools like protocmp and checkproto to verify that only the expected shipped files have changed as a result of your source changes.

protocmp does not include a man page. Its invocation is as follows:

```
protocmp [-gupGPlmsLv] [-e <exception-list> ...] -d <protolist|pkg dir> [-d <protolist|pkg dir> ...] [<protolist|pkg dir>...]<root>
```

Where:

- `-g` : don’t compare group
- `-u` : don’t compare owner
- `-p` : don’t compare permissions
- `-G` : set group
- `-U` : set owner
- `-P` : set permissions
- `-l` : don’t compare link counts
- `-m` : don’t compare major/minor numbers
-s : don’t compare symlink values
-d <protolist|pkg dir>
  proto list or packaging to check
-e <file>: exceptions file
-L : list filtered exceptions
-v : verbose output

If any of the -[GUP] flags are given, then the final argument must be the
proto root directory itself on which to set permissions according to the packaging
data specified via -d options.

A protolist is a text file with information about each file in a proto area, one
per line. The information includes: file type (plain, directory, link, etc.), full
path, link target, permissions, owner uid, owner gid, i-number, number of links,
and major and minor numbers. You can generate protolists with the protolist
command, which does not include a man page. Its invocation is as follows:

$ protolist <protoroot>

where protoroot is the proto area root (normally $ROOT). Redirecting its
output yields a file suitable for passing to protocmp via the -d option or as the
final argument.

The last argument to protocmp always specifies either a protolist or proto
area root to be checked or compared. If a -d option is given with a protolist file as
its argument, the local proto area will be compared with the specified reference
protolist and lists of files which are added, missing, or changed in the local proto
area will be provided. If a -d option is given with a package definitions directory
as its argument, the local proto area will be checked against the definitions
provided by the package descriptions and information about discrepancies will
be provided.

The exceptions file (-e) specifies which files in the proto area are not to
be checked. This is important, since otherwise protocmp expects that any files
installed in the proto area which are not part of any package represent a package
definition error or spurious file in the proto area.

This comparison is automatically run as part of your nightly(1) build so long
as the -N option is not included in your options. See nightly(1) and section 4.2
for more information on automating proto comparison as part of your automatic
build.

As a shortcut to using protolist and protocmp, you can use the ‘checkproto’
command found in the SUNWonbld package. This utility does not include a
man page, but has a simple invocation syntax:

$ checkproto [-X] <workspace>

The exception files and packaging information will be selected for you, and
the necessary protolists will be generated automatically. Use of the -X option,
as for nightly(1), will instruct checkproto to check the contents of the realmode
subtree, which normally is not built.
You can find the sources for protolist, protocmp, and checkproto in usr/src/tools.
The resulting binaries are included in the SUNWonbld package.

4.4.2 BFU Archives

BFU archives are cpio-format archives (see cpio(1) and archives(4)) used by bfu(1) to install ?? binaries into a live system. The actual process of using bfu(1) is described in greater detail in the man page and in section 5.1.2.

BFU archives are built by mkbfu, which does not include a man page. Its invocation is as follows:

$ mkbfu [-f filter] [-z] proto-dir archive-dir

The -f option allows you to specify a filter program which will be applied to the cpio archives. This is normally used to set the proper permissions on files in the archives, as discussed in greater detail below.

The -z option causes the cpio archives to be compressed with gzip(1). If both -f and -z are given, the compression will occur after the filter specified by -f is applied.

proto-dir is the proto area root described in section 4.4.1 and normally given by the ROOT environment variable.

archive-dir is the directory in which the archives should be created. If it does not exist it will be created for you.

However, 'mkbfu' is rarely used. Instead, 'makebfu' offers a much simpler alternative. It has no man page, but the invocation is simple:

$ makebfu [filename]

If an argument is given, it refers to an environment file suitable for nightly(1) or bldenv(1). Otherwise, 'makebfu' assumes that bldenv(1) or equivalent has already been used to set up the environment appropriately.

'makebfu' is a wrapper around 'mkbfu' that feeds package and environment information to 'cpiotranslate' to construct an appropriate filter for the archives. This filter's purpose is to set the correct mode, owner, and group on files in the archives. This is needed to allow builds run with ordinary user privileges to produce BFU archives with the correct metadata. Without root privileges, there is no way for the build user to set the permissions and ownership of files in the proto area, and without filtering, the cpio archives used by BFU would be owned by the build user and have potentially incorrect permissions. 'cpiotranslate' uses package definitions to correct both. See usr/src/tools/protocmp/cpiotranslate.c to learn how this works.

Each archive contains one subset of the proto area. Platform-specific directories are broken out into separate archives (this can simplify installing on some systems since only the necessary platform-specific files need to be installed), as are /, /lib, /sbin, and so on. The exact details of the files included in the archives can be found only by reading the latest version of mkbfu.
BFU archives are built automatically as part of your nightly(1) build if -a is included in your options. Also, if -z is included in your options, it will be passed through to mkbfu. See nightly(1) and section 4.2 for more information on automating BFU archive construction as part of your automatic build.

mkbfu and makebfu are Korn shell scripts included in the SUNWonbld package. Their sources are located in usr/src/tools/scripts.

cpiotranslate is a C program included in the SUNWonbld package. Its source is located in usr/src/tools/protocmp/cpiotranslate.c.

4.4.3 Packages

Ordinary OpenSolaris packages can be built from the files installed into the proto area (see section 4.4.1). This is done automatically by the main makefile's all and pkg_all targets (see usr/src/Makefile) as part of a successful build. The definitions located in usr/src/pkgdefs are used to build all packages; each subdirectory contains the package definitions and a makefile fragment used to build that package. Built packages are rooted in usr/src/packages and are in the standard format. See pkgmk(1) for more information on how packages are built.

Packages are built automatically as part of your nightly(1) build if -p is included in your options. See nightly(1) and section 4.2 for more information on automating package construction as part of your automatic build.

OpenSolaris packages are similar to those delivered by other operating systems; they contain the files to be installed, metadata about those files, scripts to perform tasks during installation and removal, and information about the package itself. Although the exact data formats used may differ, all the general concepts are the same. However, OpenSolaris packages tend to be coarser-grained than those that make up most Linux distributions. For example, SUNWcsu contains a large portion of the core system commands and kernel components; a typical Linux distribution might deliver equivalent functionality in ten or more different packages. Most OpenSolaris packages are separated based on the files' location, whether in the root filesystem or the /usr subdirectory. This is primarily to accommodate installation for diskless clients, which may have a shared /usr located on a file server.

pkgmk(1) is part of the SUNWpkgcmdsu package. Its sources are not part of the §?? consolidation.
Chapter 5

Installing and Testing OpenSolaris

This chapter describes several flexible methods for installing your OpenSolaris bits. Please note that because OpenSolaris does not yet include all the programs needed for a working system, you must have an existing full install (typically Solaris or Solaris Express) before you can perform these procedures successfully.

Additionally, some of the common testing procedures for the kernel and core userland components are covered. Although these tests are intended to cover as much of the system as possible, and to be flexible enough that additional tests can be written and added to the infrastructure, most testing is still done by project-specific test suites. When fixing bugs or adding new features, you are well-advised to contact the owner(s) of the code you are changing to obtain any existing tests. You should also contribute new tests that detect the bug you are fixing or verify the functionality of your new features.

5.1 Installation Overview

Other than manually copying specific files from your proto area into your live system, there are three main ways to install your bits on your system. Which one you use will depend on what you have changed: if you have changed only the kernel, see section 5.1.1 to learn about Install. If you have changed the kernel and userland components of §?? and your changes must be applied together, you must either hand-copy your userland changes before using Install, or use BFU; see section 5.1.2 for information on BFU.

To accommodate fully functional builds even though some sources are missing, a set of encumbered binaries is available, and the build system has been modified to make use of them. You will need to use the closed-bins.<platform>.tar.gz components along with your build products to build either Install or BFU archives that work. Please see 4 Building OpenSolaris and the latest Release Notes for more information on building and the use of the encumbered binaries.
CHAPTER 5. INSTALLING AND TESTING OPENSOLARIS

Each of these installation methods is progressively more complex and time-consuming, but upgrades a larger part of your system. Because most ON developers only modify the ON sources and are not responsible for integration testing, BFU is by far the most popular method for performing system upgrades. Developers working heavily on the kernel will often make use of Install during development and use BFU to keep current between testing phases.

Each method is described briefly in this section. Detailed instructions are provided in sections 5.2 and 5.3.

5.1.1 Cap-Eye-Install
Install (pronounced cap-eye-install) is used to update only the kernel and its associated modules on a specific system. It will place the new kernel in a nonstandard location and install only the platform-specific modules for your particular host. This allows you to test your changes without removing the normal kernel; if your new kernel does not boot or crashes, this makes recovery much easier.

5.1.2 BFU
BFU is used to update all \$?? bits, both kernel and userland. It is capable of updating some configuration files and is aware of the impact of the changes that have been made to OpenSolaris. BFU is more thorough than Install, and takes longer. Also, unlike Install, the new kernel will be installed over the existing one, so if it does not work properly you may have to boot from alternate media to recover.

5.1.3 Flag Days and Other Hazards
In some cases, you will need to install newer versions of one or more system packages before you will be able to use a new version of the \$?? bits. When this happens, it is known as a Flag Day. Flag Day notices will be posted at \$?? and will include instructions for building and/or installing the newer software you will need. It should be noted that this installation procedure is not guaranteed to interoperate cleanly with the standard packaging tools such as pkgadd(1M). In particular, use of BFU (see section 4.1.4) or ad-hoc replacement of Solaris components means that those components can no longer be updated using Solaris packages. If this is of concern to you, we recommend that you utilize exclusively Solaris Express for managing your system package installation, and build and test only against the source tree current at the time of the latest Express build. You can

5.2 Using Cap-Eye-Install to Install Kernels
First, you must have a workspace containing a built kernel. If you need more information on building kernels, see chapter 4 and especially section 4.3. Once
you have built a kernel, you need to make an Install tarball using the Install command.

The Install(1) utility creates a tar file that can be extracted into an appropriate machine’s root directory. It utilizes an existing built kernel tree and the kernel makefiles to determine the correct contents of the tar file. See Install(1) or the output of ‘Install -h’ for a complete list of options; only the most commonly-used options are described here.

The tar file constructed by Install is specific to an architecture, such as sun4u or i86pc. There are two ways to specify the architecture for which you want Install to create an archive. The first is to be in the architecture’s subdirectory (usr/src/uts/<arch>) when running Install; the second and preferred method is to use the -k <arch> option. Note that current releases OpenSolaris support only one architecture in each ISA: sun4u on SPARC and i86pc on x86/amd64.

Another setting which is usually specified is the ”glomname”, using the -G option. This is the name of the subdirectory in /platform/<arch>/ that the binaries will go into, and is generally of the form ”kernel.something”. If you don’t use a glomname, you will overwrite the current kernel and modules on the target machine, and are likely to make BFUing the machine later a more difficult task, as well as running the risk of having to boot from alternate media to fix your machine if the new kernel does not boot.

A simple invocation of Install might look like:

```
$ Install -G kernel.foo -k sun4u
... lots of spew ...
Creating tarfile /tmp/Install.username/Install.sun4u.tar
Install complete
```

You can now copy /tmp/Install.username/Install.sun4u.tar to your test machine and extract it in the root directory. Then, reboot the test machine and have it use the new kernel by running:

(SPARC)
```
# reboot -- 'kernel.foo/sparcv9/unix'
```

(AMD64)
```
# reboot -- 'kernel.foo/amd64/unix'
```

(x86)
```
# reboot -- 'kernel.foo/unix'
```

Note that you will need to use either this reboot syntax each time you wish to boot the test kernel, or use similar arguments to the bootloader or OBP. Otherwise, the normal kernel installed by BFU or the regular installation will be booted.
5.2.1 Caveats

Although Install is useful for developers who have changed only kernel code, it is of limited value for others. In particular, if a recent flag day notice indicates that newer kernels are incompatible with existing userland libraries or commands, Install cannot be used to test the updated kernel until you have upgraded your userland via BFU or some other mechanism such as the regular installation or upgrade procedure.

Like bfu (see section 5.3), Install is rather closely attached to its particular release, so you should use the current version from the gate matching the release you are building. Normally this is in the public/bin subdirectory of the gate; however, for installations outside Sun it is located in /opt/onbld/bin.

It is critical that Install users install the correct set of platform-specific modules, especially on SPARC systems. Failure to do so can result in an unbootable system. See section 5.2.2 below for more information on how platform-specific modules relate to Install.

One major advantage of Install over BFU is the ability to keep your existing kernel in place so that you can still boot if the test kernel proves toxic. We strongly recommend that if you use Install to test kernels, you take advantage of this feature and use distinct locations (see the -G option described in section 5.2.1 above) for each new kernel you test. Otherwise, you will likely have to boot from alternate media to repair your system following the installation of a bad kernel.

5.2.2 Platform-Specific Information

Ordinarily, Install does not generate archives with implementation-specific modules. If these archives are installed onto a system which requires the missing modules, the system may fail to boot or work properly. If you do this, you will need to boot from a known-working kernel and correct the problem.

An example symptom of the problem (on an Enterprise 3500):

SunOS Release on81 Version jrhyason_[we-vmstat]_05/15/01 64-bit
Copyright 1983-2001 Sun Microsystems, Inc. All rights reserved.
DEBUG enabled
obpsym: symbolic debugging is available.
Read 297063 bytes from misc/forthdebug
====> WARNING: consconfig: consconfig_util_openvp failed: err 6 vnode 0x2803c80
====> WARNING: consconfig: consconfig_util_link I_PLINK failed: error 22
    configuring IPv4 interfaces: hme0.
...
The console is gone!
To include the needed modules in your Install tarball, make sure to use

$ Install -i <implementation>
to include implementation-specific modules. But how do you know what sun4u implementation you need? First, obtain your machine’s "official" implementation name from the output of 'uname -i'. Then, in usr/src/uts/sun4u, run "grep IMPLEMENTED_PLATFORM */Make*" to see a list of implementations and the corresponding platform name reported by uname(1).

In the example above, the E3500 reports:

$ uname -i
SUNW,Ultra-Enterprise

And we see from the grep output:

$ grep IMPLEMENTED_PLATFORM */Make*
...
sunfire/Makefile:IMPLEMENTED_PLATFORM = SUNW,Ultra-Enterprise
...

In this case, the "-i sunfire" argument must be added to get the correct behavior.

Additionally, one of the easiest ways to get tripped up with Install wads comes from the fact that not all drivers are delivered by $??$. This has been particularly noticeable with x86, but it also happens with SPARC, especially framebuffer drivers. One way to work around this is to do:

# cd /platform/{sun4u,i86pc}
# mkdir glomname
# (cd kernel; tar cf - .) | (cd glomname; tar xf -)

and then install the Install glom image. This will copy your existing drivers to the new kernel installation, ensuring that the drivers which are not part of ON (and OpenSolaris) are available when you reboot.

5.3 Using BFU to Install ON

The Blindingly Fast Upgrade (or Bonwick-Faulkner Upgrade) is a process used to update $?? bits on a system. The ordinary Solaris upgrade procedure requires the complete $?? and takes at least 30 minutes (usually much longer). To save time, BFU uses a set of cpio(1) archives to directly overwrite the existing contents of the system. BFU often takes less than 10 minutes to run to completion, and if you are upgrading from a recent build, conflict resolution will usually take only a few minutes. Over the course of a year, using BFU can save dozens of hours of development time.

In order to use BFU, you will need to set three additional environment variables first. You can set these in your login dot-files, or on the command line. If you prefer, you could create a local wrapper for bfu(1) that sets them first. The environment variables are:
FASTFS
This should normally be set to /opt/onbld/bin/‘uname -p’/fastfs.

BFULD
This should normally be set to /opt/onbld/bin/‘uname -p’/bfULD.

GZIPBIN
This should normally be set to /usr/bin/gzip.

BFU is simple to use and normally takes only a single argument: a path to the set of archives you wish to install. For example, if your workspace is located in /home/jdoe/workspace, and you have completed a nightly(1) build, you would invoke bfu(1) as follows:

```
# bfu /home/jdoe/workspace/archives/‘uname -p’/nightly
```

Note that, since it modifies the system software installation, bfu(1) must always be run as root.

When bfu completes there’s no guarantee that the new commands and libraries are compatible with the currently running (old) kernel. Therefore, instead of exiting, bfu puts you into a subshell in which PATH=/tmp/bfubin and LD_LIBRARY_PATH=/tmp/bfulib. These directories contain the old versions of the commands and libraries commonly needed to resolve conflicts and reboot the system. They have also been modified to work with a saved copy of the old dynamic linker.

Note that you may receive warnings from BFU about being unable to copy files from “greenline.eng” or other systems or locations. In general, these warnings should be reported as bugs. However, at the time of this writing, they are harmless provided that your system is running at least Solaris 10 build 74 prior to your BFU attempt. See the latest release notes for any additional requirements and restrictions.

5.3.1 Caveats
Although it saves time, BFU is not a panacea. This section contains information about BFU’s drawbacks. You should carefully evaluate these drawbacks against the benefits and decide whether BFU is appropriate to your needs. In general, unless you are an active developer, we recommend that you do not use BFU.

BFU does not update package information. Therefore you will most likely be unable to install official Solaris patches or run a full system upgrade procedure on a system which has been BFU’d. The importance of this cannot be overemphasized: **IF YOU BFU YOUR SYSTEM, DO NOT ATTEMPT TO USE ”NORMAL” SYSTEM MANAGEMENT PROCEDURES TO UPDATE IT IN THE FUTURE. USE BFU OR REINSTALL**.

BFU does not update non-?? packages, even if newer versions of those packages are required in order to successfully install or run the version of ON
you are installing. You may need to update those packages before and/or after running BFU. To understand what package updates may be needed, consult §§?? for the full list of flag days between the build you are currently running and the build you wish to BFU to. Each flag day notice will instruct you as to what package updates, if any, are needed, and whether they must be completed before or after BFUing. It is critical that you read, understand, and follow these instructions exactly before running BFU. If you fail to do so you will almost certain brickify your system. See section 5.1.3 for more information about flag days.

Although the core functionality of BFU consists of the simple activity of unpacking cpio(1) archives into the running system, it also performs numerous other tasks related to the update of your system. Many of these tasks are specific to particular changes that have been made in the sources over a period of years. If your system has unusual characteristics, these additional updates can fail, which may result in a nonfunctional system. Because these updates vary greatly, it is impossible to know in advance which updates could fail, or what failure modes are possible. Although such failures are rare, they can occur. If you use BFU, you should follow the development of OpenSolaris to understand changes being made that could have an adverse effect on your system. If you have doubts as to how well BFU will update a particular aspect of your system following a major change to OpenSolaris, read bfu(1) and consult with the engineers who made the change.

When bfu finishes, it invokes ksh with a limited PATH. The PATH contains programs which have been specially modified to work regardless of what changes the BFU archives may contain. You can use these programs to resolve conflicts (see section 5.3.2). In particular, "reboot" works, but "init 6" does not. You can exit from this protected environment if you want, but it is not advisable unless you are sure that there have not been any "flag days" (synchronized kernel/userland changes) since your last bfu.

Never BFU in a window; always use the system console. If BFU or the system crashes in midstream, or you use the window system and it crashes or hangs, your system will be in an inconsistent state. You may be unable to boot. Therefore, you should ensure that as much of the system as possible is quiescent before starting a BFU, and be sure you have a copy of suitable system software media handy to reinstall if necessary.

Never BFU a production system. Production systems should always be updated to approved releases using the supported upgrade mechanism.

5.3.2 Resolving Conflicts

Every machine has several configuration files which get modified from the default installation; bfu keeps a list of these. This list is known as the conflicts database. BFU saves a copy of each configuration file it would overwrite under /bfu.child; likewise, it stores a copy of each such file from the cpio archives it is extracting under /bfu.parent, having moved the previous BFU’s /bfu.parent (if any) to /bfu.ancestor. We refer to the files saved under those directories as the child,
When the extraction is complete, BFU diffs the various versions of these configuration files and acts according to the following rules:

* If the file is unchanged (i.e., there is no difference between the child and the parent), there is nothing to do or report.
* If the file was accepted from the parent the previous run (i.e., the child is identical to the ancestor) then the parent is accepted automatically this time; such files are marked as "update:"
* If the file is the same as the beginning of the previous file, it is assumed that the user has added lines to the end (i.e., the child consists of the parent plus some added lines); the child version is restored; such files are marked as "restore:"
* If the file differs between the parent and the child, but the parent and ancestor are identical, then it is deemed an old conflict; such files are marked as "old:"
* Lastly, if the file differs between the parent and the child, and the parent and ancestor differ as well, then it is deemed a NEW conflict; such files are marked as "NEW:"

So now we know that a NEW conflict means a file which was already different from the default, where the default has been updated. To resolve this difference, whatever changes were introduced in the new build must be ported to the existing file on the system.

Although it is possible to blindly accept the parent, this will cause any customizations in the child to be lost. As it is very common for these files to have such customizations made automatically by class action scripts from non-

§

-§-

packages, this is usually a mistake, which can lead to hours of lost productivity. Therefore, there are two forms of conflict resolution which are discussed in the following two sections. They are automatic and manual conflict resolution.

5.3.2.1 Automatic Conflict Resolution Automatic conflict resolution is performed by a script called acr. This script is available in /opt/onbld/bin and in the usr/src/tools/scripts directory of the § workspace. In the usr/src/tools/scripts directory, the acr executable gets made from acr.sh if you do a nightly(1) build with the -t option. There is man page, acr.1, in /opt/onbld/man/man1 and in the usr/src/tools/scripts directory of the § workspace.

The standard way to use acr is to invoke it while still in the protected environment after a BFU is performed. In this mode there is no need to specify any command line parameters.

For more specialized applications, acr accepts one or two parameters. The first parameter is the name of an alternate root directory. The second parameter, if specified, is the name of the directory containing the archives.

acr uses a file called conflict_resolution.gz in the directory containing the BFU cpio archives. This conflict resolution file is constructed whenever nightly(1) creates archives. That is to say whenever nightly(1) is run with the -a option.

When acr runs, it lists the files that it is processing to standard out. Detailed
results are written to a file called `allresults` in a subdirectory of `/tmp`. When acr exits, it prints the full path name of the allresults file. The allresults should be examined to ensure that no errors occurred during acr processing. If errors occurred, you will need to resort to manual conflict resolution discussed in the next section.

### 5.3.2.2 Manual Conflict Resolution

Manual conflict resolution requires you to resolve each of the conflicts by hand. The general way to resolve conflicts is:

```
% diff /bfu.ancestor/$file /bfu.parent/$file
```

then manually apply the diffs to `/$file`. For many files, a short-cut can be employed. BFU drops you in `/bfu.conflicts` when it completes, and most changes are additions or modifications, so doing:

```
% diff $file /$file
```

and making sure the diffs all point to the right (i.e., ">") rather than "/") will do the trick. This will not work in all cases, however; some notable exceptions are detailed below.

**etc/name_to_major**

This file matches device names with major numbers; it is critical that each device have a unique major number. Diffing the ancestor with the parent is a good start – just remember that the name is important, but the major number may vary. Old devices which have been removed in the parent should be deleted, but if you’re not sure, just leave them alone as they are generally harmless. New devices should be added with an unused major number (if it’s available, use the one from the diffs). Never change the number for an existing entry unless you are sure you know what you are doing. If you are in doubt as to what changed, you can consult the history of the files directly from your *workspace* (e.g. `usr/src/uts/{intel|sparc}/os/name_to_major`).

Once finished, you can sanity-check your changes by running:

```
% sort -k1,1 /etc/name_to_major | sort -uc -k1,1
% sort -k2n /etc/name_to_major | sort -uc -k2n
```

These report the first duplicated device name and the lowest duplicated major number, respectively. Either of these indicates a problem that you must correct before rebooting. If there is no output, you’re all set. Note that the kernel will also warn you of such conflicts after you reboot, but by then it may be too late.

**etc/security/*_attr**

These files tend to get shuffled around quite a bit by class-action scripts, so the parent and child versions can differ wildly. For these, the shortcut described above is ineffective, but the more general diff of the ancestor with the parent works well.
To seed the BFU conflicts database properly before using BFU for the first time, you must BFU to the same build as you have installed using the standard Solaris or OpenSolaris installation process. In doing this, one can ignore all conflicts. On subsequent BFUs, only NEW conflicts need to be examined.

5.3.3 BFU and Zones

The contents of each zone are copied from the global zone when the zone is created. To ensure that all zones remain operational after BFUing, it is necessary to keep the zone contents consistent with those in the global zone. To do this, BFU will update each zone in turn once the global zone has been updated; however, because each zone may have configuration files that differ from the global zone’s, each zone has its own BFU conflict management directories. Unfortunately, it is not possible to BFU only a single zone. BFU affects the entire system including all zones. If you want to test userspace changes in a zone, you will need to copy files from your proto area into that zone manually.

It is safest to shut down all zones before BFUing your system. This will ensure that dependencies which may be violated by the files being installed will not break a zone.

After BFUing a system with zones other than the global zone, you will need to first resolve conflicts in the global zone, then resolve conflicts in each remaining zone before rebooting the system. Resolving BFU conflicts in a zone works exactly the same way as for the global zone; however, in many cases it will not be necessary to manually merge the file contents. Instead you are likely to find that the file should be identical to that in the global zone, and you can simply copy it over. After resolving conflicts in all zones, you must reboot your system before rebooting any zone.

5.4 Test Suites

TBD

5.4.1 STC
TBD

5.4.2 SVVS
TBD

5.4.3 Writing Tests
TBD

5.4.4 Coverage
TBD
5.4.5 Platform Testing

TBD
Chapter 6

Integration Procedure

TBD

6.1 Submission Procedure

TBD

6.2 Code Review Process

Code review is the "last defense against brokenness." It is a critical part of the development process: no change can be putback by anyone, no matter how knowledgeable or experienced, without first being reviewed. As a version nears release, code review requirements become more strict; more reviewers may be required. The code review process can take anywhere from a few minutes to many hours, depending on the type, size, and intrusiveness of the changes being made. Normally, developers will ask others to review their changes who are familiar with the code being modified.

The purpose of code review is to find subtle and non-obvious problems with the changes being made, not to absolve the implementor of responsibility for his or her changes. Specifically, developers must conduct their own self-review and testing before requesting a formal code review.

Code review process TBD.

6.3 Putback Procedures

TBD. At OpenSolaris launch all putbacks will occur within Sun.
6.4 Licensing Requirements

All code integrated into OpenSolaris by a Sun employee must contain the usual Sun copyright notice (see section 7.2.3). The code will then be licensed by Sun Microsystems both as part of the OpenSolaris project and to other customers via various Solaris and other licenses. Thus, licensing is not a concern for Sun employees.

In order to contribute code to OpenSolaris, community developers will need to enter into a contributor agreement. A contributor agreement allows us to ensure that contributors have sufficient rights to the code being contributed, and that any third party intellectual property issues are disclosed. It also allows us to defend the project should there be a legal dispute regarding the software at some time in the future. We’re currently in the process of composing a contributor agreement.

Contributors who add significant changes to an existing file or add new files should include a brief copyright notice similar to the standard Sun copyright notice. See section 7.2.3 and the contributor agreement for more information.
Chapter 7

Best Practices and Requirements

This chapter provides information on standards and procedures which must be followed by all developers. These rules are the same ones which Sun engineers have been required to follow when developing Solaris, and are the primary reason Solaris, and now OpenSolaris, is among the world’s best available software.

7.1 Compatibility and Visibility Constraints

OpenSolaris has a comprehensive set of rules governing interfaces. An interface is any aspect of the system that a user or developer can observe or use. Examples include functions and variables in libraries, the location of a particular executable program, the output content, format, and argument semantics of an executable program, and the location in the filesystem of a header file. There are thousands of interfaces provided by OpenSolaris, and each of them has certain visibility, documentation, and compatibility constraints.

7.1.1 Interface Stability Taxonomy

One of the attributes included in many Solaris man pages is ”Interface Stability.” The value of this attribute defines who may depend on the interface, and what expectations they may have for its future maintenance. The possible values and their explanations are described here; note that the Private levels will not appear in man pages shipped to customers.

In the absence of any evidence about an interface’s stability level, certain assumptions must be made. Following the general rule ”be strict in what you produce, and liberal in what you accept” would lead one to conclude that any documented interface is Stable, and undocumented interfaces are probably being used by others, but could be changed at any time without your knowledge (i.e. they are at once both Project Private and Unstable); as a consequence you
should neither change the interface nor depend on its current implementation. If you need to depend on or change an undocumented interface whose stability level cannot be determined, you need to find that interface’s owner, if any, and have the action approved by the relevant ARC. If the owner of the interface can be found, you will most likely want to promote the interface either to Open (public) status, or use one of the Contracted Private stability levels.

Following is a list of stability levels and their respective meanings. Additional information is available in the attributes(5) man page, including the precise public definitions of major, minor, and micro releases and additional information about Open interfaces.

**Stability Level: Standard**

<table>
<thead>
<tr>
<th>Specification</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompatible Change</td>
<td>major release (X.0)</td>
</tr>
<tr>
<td>Compatible Change</td>
<td>minor release (x.Y)</td>
</tr>
<tr>
<td>ARC review of Specs</td>
<td>A precise reference is normally recorded</td>
</tr>
<tr>
<td>Examples</td>
<td>POSIX, ANSI-C, ABI, SCD, SVID, XPG, X11, DKI, VMEbus, Ethernet, NFS protocol, DPS</td>
</tr>
</tbody>
</table>

Most of these interfaces are defined by a formal standard, and controlled by a standards organization. Incompatible changes to these interfaces are rare.

This stability classification can also apply to interfaces that have been adopted (without a formal standard) by an "industry convention" (X/Open, MIT X- Consortium, OMG), or even by a single-source (Adobe’s Display PostScript, Novell’s NetWare Protocols, Legato’s network backup protocols, Berkeley’s sendmail) if we expect that the de facto standard is unlikely to change incompatible.

If possible, there should still be a reference to a standard specification or reference system, although there may be cases where no such citation is possible. Customers are normally pointed to the same specification.

Support is only provided for specific version(s) of a standard, and support for a particular version does not guarantee that support will be provided for other versions. Sometimes bugs are corrected or interpretation is clarified in a standard; we may make incompatible changes to react to these, but will evaluate the impact of doing so and will announce a compatibility and migration strategy. (PSARC/1995/224’s Advisory Information section provides guidelines for implementing a preliminary draft of a new standard.)

Some standards lack bindings to a specific programming language; if the project team chose names, numbers, extensions, or other implementation-specific details, they should be called out for architectural review.

**Stability Level: Stable**
We publish the specification of these interfaces, typically as manual pages or other product documentation. We also tell customers we will remain compatible with them.

The intention of a Stable interface is to enable arbitrary third parties to develop applications to these interfaces, release them, and have confidence that they will run on all minor releases of the product (after the one in which the interface was introduced, and within the same major release), without having to change or recompile the applications. Even at a major release, incompatible changes are expected to be rare, and to have strong justifications. Stable interfaces are sometimes proposed to be industry Standards, as was the case with ToolTalk (now an X/Open standard).

An ARC should review and archive the specification, and adequate customer documentation must also exist.

These are interfaces whose specification is generally controlled by the Open-Solaris project, and does not rely on externally delivered projects.

**Stability Level: Evolving**

An Evolving interface is subject to incompatible change at a major or minor release, but we should expect to change an Evolving interface only carefully,
and probably slowly. As the interface evolves, we will make reasonable efforts
to ensure that all changes are source and binary compatible.

An ARC should review the interface specification (especially with respect
to ability to absorb expected evolution compatibly). Adequate customer docu-
mentation should also exist. The intention of an Evolving interface is to enable
ISV’s to exploit new technology, and it should be expected that they will ship
products that depend on these interfaces. As a result, any incompatible change
to an Evolving interface requires an assessment of potential customer impact,
and a notification and migration plan. Elements of such a plan might include:
* clearly setting customer expectations about the circumstances under which
  the interfaces might change.
* ensuring that all such changes are described in the release notes for the
  affected release.
* providing migration aids for binary compatibility and/or continued source
development.

A project’s intention to accept the risk of depending on an Evolving interface
must be evaluated and explicitly approved by an ARC.

NOTE: It will often be the case that interfaces declared to be Evolving will
later be reclassified as Stable or Standard. Nonetheless, foreseen promotion is
not a necessary attribute of the Evolving taxonomy level. An interface could be
classified Evolving and remain as such indefinitely.

**Stability Level: Unstable**

<table>
<thead>
<tr>
<th>Specification</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompatible Change</td>
<td>minor release (x.Y)</td>
</tr>
<tr>
<td>Compatible Change</td>
<td>micro release (x.y.Z)</td>
</tr>
<tr>
<td>ARC review of Specs</td>
<td>Yes</td>
</tr>
<tr>
<td>Examples</td>
<td>SUNW* package abbreviations, some config utils</td>
</tr>
</tbody>
</table>

Unstable interfaces are experimental or transitional. They are typically used
to give developers early access to new or rapidly changing technology, or to
provide an interim solution to a problem where a more general solution is an-
ticipated. No claims are made about either source or binary compatibility from
one minor release to the next.

The intention of an Unstable interface is that they be imported only by
prototypes and by products on the same CD (or whatever release medium)
as the interface implementation. A project’s intention to import an Unstable
interface should be discussed with the ARC early. The stability classification
of the interface – or a replacement interface – might be raised. The opinion
allowing any project to import an Unstable interface should explain why it is acceptable.

Any documentation for an Unstable interface must contain warnings that these interfaces are subject to change without warning and should not be used in unbundled products. In some situations, it may be appropriate to document Unstable interfaces in White Papers rather than in standard product documentation.

Given such caveats, customer impact need not be a factor when considering incompatible changes to an Unstable interface in a major or minor release. Nonetheless, when such changes are introduced, the changes should still be mentioned in the release notes for the affected release.

An ARC should review and archive the specification. Any proposed change to the interface must be ARC approved.

NOTE: If we choose to offer a draft standard implementation but state our intention to track the standard (or the portions we find technically sound or likely to be standardized), we set customer expectations for incompatible changes by classifying the interface Unstable. The interface should be reclassified Standard when standard is final. Such an intention could be encoded "Unstable->Standard".

**Stability Level: External**

<table>
<thead>
<tr>
<th>Specification</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompatible Change</td>
<td>micro release (x.y.z)</td>
</tr>
<tr>
<td>Compatible Change</td>
<td>micro release (x.y.z)</td>
</tr>
<tr>
<td>Arc review of Specs</td>
<td>A precise reference is normally recorded</td>
</tr>
<tr>
<td>Examples</td>
<td>OpenSSL</td>
</tr>
</tbody>
</table>

These interfaces are controlled by a body outside of Sun and the OpenSolaris project, but unlike Standard, it can not be asserted that an incompatible change to the interface would be exceedingly rare. In some cases it may not even be possible to clearly identify the controlling body. This classification is typically used for community software, Free Software, open source and the like.

Use of the External interface stability level allows freeware interfaces provided by Sun to quickly track the fluid, external specification. In many cases, this is preferred to providing additional stability to the interface, as it tends to track the expectations of the community. However, External interfaces should adhere to OpenSolaris standards in at least the following areas:

* Security, Authentication * Manual Page Section Numbering * File System Semantics (/usr may be read-only, /var is where all significant run-time growth occurs, ...)
All External interfaces should be labeled as such in all associated documentation and the consequence of using such interfaces should be explained either as part of that documentation or by reference. Default search paths should not lead to External interfaces - the user should be required to take some simple, explicit action to access them.

Shipping incompatible change in a patch should be strongly avoided. It is not strictly prohibited for the following two reasons:

* Since we are not in explicit control of the changes, we can not guarantee with reasonable assurance that an unidentified incompatibility isn’t present.

* A strong business case may exist for shipping a newer version as a patch if that newer version closes significant escalations.

In general, the intent of allowing change in a patch is to allow for change in Update Releases.

It should be noted that in some cases it will be preferable to apply a less fluid interface classification to an interface even if the controlling body is external to OpenSolaris. Use of the Unstable classification extends the stability commitment over micro/patch releases, allowing use of additional support models for software that depends upon these interfaces, at the potential cost of less frequent updates. However, care should be exercised because it will be difficult to differentiate External and Unstable as a classification for seemingly identical software. It is suggested to use External and behave (through contracts) as Unstable. Use of the Evolving classification promotes these interfaces to first class OpenSolaris interfaces, at the potential cost of diverging from the external specification. By using Evolving, we are essentially taking control of the interface, although it may liberally import from the external reference. Use of the Stable classification is not recommended.

**Stability Level: Contracted External**

This stability level is the same as External, except that a contract has been put in place between the provider and consumer of the interface. The contract describes special arrangements made for the stability of the interface. This can be used, for example, to place restrictions on how and when an interface may change if the normal rules for External do not satisfy the requirements of the consumer.

An ARC should review, approve, and archive a contract between the provider and consumer of the interface. Any change to the contract, the interface, or the specification requires reapproval.

**Stability Level: Obsolete**

<table>
<thead>
<tr>
<th>Specification</th>
<th>Open, along with warning of obsolescence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompatible Change</td>
<td>minor release (x.Y)</td>
</tr>
<tr>
<td>Compatible Change</td>
<td>By former classification, but unlikely</td>
</tr>
<tr>
<td>ARC review of Specs</td>
<td>Normally downgraded from a higher</td>
</tr>
</tbody>
</table>
An interface that is "deprecated" and/or no longer in general use. An existing interface may be downgraded from some other status (such as Stable or Standard) to Obsolete to encourage customers to migrate from that interface before it will be removed (or incompatibly changed).

In addition to reclassifying the interface Obsolete and documenting the new classification in customer documentation, a pro-active program to communicate to customers the change in commitment must precede the incompatible change or removal in a minor release. For some interfaces, the ARC may find such a communication program appropriate before removing an interface, even at a major release.

The standard program to communicate a change in commitment requires:

1. Demonstration of support by the Steering Committee responsible for the deliverable(s) containing the interface. Such support can be demonstrated by a change to strategy document or resolutions taken in meetings and documented in the minutes.

2. One year’s notice to the customer base and the Sun product development community of the intended obsolescence of the interface. This requirement ensures that no further commitments against the interface are created and gives those affected by future removal of the facility a chance to make alternative arrangements.

The year must elapse after the notice and prior to the delivery of a product that contains a change incompatible with the present status of the interface.

Acceptable means of customer notice includes letters to customers on support contracts, release notes or product documentation, or announcements to customer forums appropriate for the interface in question.

The notice of obsolescence is considered to be ”public” information in that it is available freely to the customers. It is not intended that this require specific actions to ”publish” the information, such as press releases or similar forms of publicity.

3. Where technically feasible, inclusion in the release where the interface is declared Obsolete of a warning mechanism if the interface is used. The mechanism should produce a message of the form ”The application uses interface which has been declared obsolete and may not be present in versions of released after [event]. Please notify your support person. See in [reference] for more information.” One suggested method is to use syslog(3), with a level of ”LOG_WARNING”. A method for turning off the warning message should also be provided. Common sense should apply in determining how often the warning should appear.

4. Information in the User Documentation that contains the following:

* An explanation of the meaning of Obsolete.
* An indication of the kinds of warning messages that may appear.
  * A suggesting that the customer ask their support person to contact the vendor of any application that causes such a warning to appear.
  * General instructions for turning off the warning messages.
  * A list of the Obsolete interfaces contained in this release, the earliest that they may disappear, the kind of warning that might appear, and the method for disabling the warning.

Proposals to downgrade an interface through this mechanism must be approved by an ARC before "core" documentation may be altered to identify the interface as Obsolete. Release notes may warn of the possibility of removal with either ARC or Steering Committee approval. A warning that the interface *may* be removed in a future release could be included without ARC approval, but the ARC may not deem such notice alone as sufficient notification to customers to "start the 1-year clock".

A follow-on project to perform the actual feature removal in a forthcoming minor or major release after the timeout period expires, requires architectural approval to ensure that the requirements of the obsolescence policy have been met. Provided they have been met, approval will be straightforward.

**Stability Level: Committed Private**

<table>
<thead>
<tr>
<th>Specification</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompatible Change</td>
<td>major release (X.0)</td>
</tr>
<tr>
<td>Compatible Change</td>
<td>micro release (x.y.Z)</td>
</tr>
<tr>
<td>ARC review of Specs</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Example**

UFS media format, Calendar Manager RPC protocol

For some otherwise-private interfaces, we must maintain compatibility from release to release, in order to meet the customer’s expectations for compatibility of the programs using these interfaces. However, we don’t want customers to depend on these interfaces directly, and we don’t want to directly expose these interfaces to customers. These interfaces are classified as Committed Private.

Our commitment is that a customer’s "normal" use of system facilities should not allow them to see any incompatible changes to these interfaces. Since these interfaces typically span machines by being embodied in media or protocols (and since customers cannot upgrade all their machines simultaneously), these interfaces can’t be changed with the freedom of a private interface. Yet, changes to the details of the interface can be dramatic, provided the commitment to the customer is maintained. In general, Committed Private interfaces should be versioned.
An ARC should review and archive the specification, and will at least assure that the interface can satisfy its purpose and support the evolution described in the previous paragraph. Any proposed change to or new dependency on the interface must be ARC approved.

**Stability Level: Contracted Committed Private**

This stability level is the same as Committed Private, except that a contract has been put in place between the provider and consumer of the interface. The contract describes special arrangements made for the stability of the interface. This can be used, for example, to allow exposure of the interface to a Sun Partner.

An ARC should review, approve, and archive a contract between the provider and consumer of the interface. Any change to the contract, the interface, or the specification requires reapproval.

**Stability Level: Sun Private**

* Specification  Closed
* Incompatible Change  minor release (x.Y)
* Compatible Change  micro release (x.y.Z)
* ARC review of Specs  Yes
* Example  trap 40 (gethrtime)

These are interfaces which one consolidation depends on and another consolidation provides. Changes to these interfaces must be coordinated among all providers and users of the interface. Some internal kernel interfaces are Sun Private interfaces.

Interfaces are occasionally made Sun Private in order to gain some experience with them before opening them up to wider use as Unstable or Stable interfaces. Making such interfaces Consolidation Private would be preferable, however, as evolution is then far easier.

Sun Private interfaces are strongly discouraged. Coordinating changes to these interfaces within a consolidation is usually feasible, but coordinating changes among different consolidations released asynchronously is extremely difficult. Interface versioning is advised for Sun Private interfaces.

An ARC will review and archive these interfaces, with special attention to how the interface could evolve, if necessary. Any proposed change to the interface must be ARC approved.

**Stability Level: Contracted Sun Private**

This stability level is the same as Sun Private, except that a contract has been put in place between the provider and consumer of the interface. The contract describes special arrangements made for the stability of the interface. This can be used, for example, to allow exposure of the interface to a partner.
An ARC should review, approve, and archive a contract between the provider and consumer of the interface. Any change to the contract, the interface, or the specification requires reapproval.

**Stability Level: Consolidation Private**

<table>
<thead>
<tr>
<th>Specification</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompatible Change</td>
<td>micro release (x.y.Z) or &quot;jumbo patch&quot;</td>
</tr>
<tr>
<td>Compatible Change</td>
<td>micro release (x.y.Z) or &quot;jumbo patch&quot;</td>
</tr>
<tr>
<td>ARC review of Specs</td>
<td>Not necessary</td>
</tr>
<tr>
<td>Examples</td>
<td>libdeskset, kernel nameslists</td>
</tr>
</tbody>
</table>

These are interfaces internal to the consolidation that one piece of a consolidation depends on and another piece of the same consolidation provides. Changes to these interfaces must be coordinated among all providers and users of the interface. Many internal kernel interfaces are Consolidation Private interfaces.

Generally these are interfaces that have proven convenient for building the consolidation, but which change often enough that we’re not willing to document them for external use nor to commit to their stability. libdeskset is an example of such an interface. Though the libkvm API is Public, the undocumented names that can be accessed through that interface are Consolidation or Project Private.

An ARC may review and archive these interfaces, or may leave the consolidation to monitor their own internal commitments. If a Consolidation Private interface is reviewed by the ARC, ask that ARC if they want to review later changes to that interface.

Importing the interface by any project outside the Consolidation would require negotiating a "contract" with the interface providers. An ARC must review and approve the classification change to Contracted Consolidation Private and the terms of the contract.

**Stability Level: Contracted Consolidation Private**

This stability level is the same as Consolidation Private, except that a contract has been put in place between the provider and consumer of the interface. The contract describes special arrangements made for the stability of the interface. This can be used, for example, to allow exposure of the interface to a specific consumer in a different consolidation.

An ARC should review, approve, and archive a contract between the provider and consumer of the interface. Any change to the contract, the interface, or the specification requires reapproval.

**Stability Level: Project Private**
CHAPTER 7. BEST PRACTICES AND REQUIREMENTS

Project Private interfaces usually occur when a project must communicate between its components across a boundary in the system. For instance, Metamucil includes several new ioctls to perform operations on UFS filesystems. The Metamucil ufsdump program uses these ioctls. The ioctls are private interfaces since they are intended to be used only by the Metamucil product. If the Metamucil product needs to change these ioctls in the future, they can do so without coordinating with any other projects, since no other projects may use these ioctls. Likewise, the nfssys system call is used to communicate between the kernel- and user-level portions of NFS.

Project Private interfaces also occur in libraries where one module needs to call a private routine in another module in the same library.

Also, Project Private interfaces may be provisional or in transition. The uadmin cpu control functions are Project Private because they will change form before appearing as Standard interfaces, and in the meantime we don’t want anyone depending on them.

Sadly, many kernel procedures are Project Private interfaces (instead of Internal interfaces) because they are visible to dynamically loaded kernel modules.

Once an interface is classified Project Private by an ARC, changes to that interface need not be ARC approved.

Any use of the interface from outside the project would involve negotiating a "contract" with the interface providers; an ARC must review and approve the classification change to Contracted Project Private and the terms of the contract.

**Stability Level: Contracted Project Private**

This stability level is the same as Project Private, except that a contract has been put in place between the provider and consumer of the interface. The contract describes special arrangements made for the stability of the interface. This can be used, for example, to allow exposure of the interface to a specific consumer in a different consolidation.

An ARC should review, approve, and archive a contract between the provider and consumer of the interface. Any change to the contract, the interface, or the specification requires reapproval.
CHAPTER 7. BEST PRACTICES AND REQUIREMENTS

7.1.2 Library Considerations

This section will cover specfiles and versioning. See /shared/ON/general_docs/libspec-rules.ps, scoping-rules.fm, and spectrans/ for sources.

7.1.2.1 Versioning TBW

7.1.2.2 Specfiles TBW

7.1.2.3 Scoping and Visibility TBW

7.2 Style Guide

Like many projects, OpenSolaris enforces a coding style on contributed code, regardless of its source. This coding style is very similar to that used by the Linux kernel, BSD systems, and many other non-GNU projects (the GNU project uses its own unique coding style). This style is described in detail at §??, although some elements in the style guide are rather dated, especially as they relate to K&R C versus ANSI (now ISO) C. You also should examine the files in/usr/src/prototypes; these provide examples of the correct general layout and style for most types of source files.

7.2.1 Automated Style Tools

Two tools for checking many elements of the coding style are available as part of the OpenSolaris distribution. These tools are cstyle(1) for verifying compliance of C code with most style guidelines, and hdrchk(1) for checking the style of C and C++ headers. Note that these tools are not perfect; there are style mistakes that cannot be caught by any reasonable tool, and others that cannot be caught by the particular implementations. Improving the accuracy and completeness of these tools is an ongoing task and enhancements of all kinds are welcome.

All headers are expected to pass 'hdrchk'. All C files and headers are expected to pass 'cstyle -P -p'. These tools produce no output and exit with status code 0 if the file(s) checked meet the style requirements.

7.2.2 Style Examples

A few common examples of bad style which are not currently caught by the tools include:

* Mixing declaration of initialized and uninitialized variables on the same line; for example:

    int a, b = 16, c = -4, d;

Instead, uninitialized variables may be declared one or more per line, and each initialized variable should be declared on its own line:
int a, d;
int b = 16;
int c = -4;

or

int a;
int b = 16;
int c = -4;
int d;

* Inconsistent use of acceptable styles. In the above example, two acceptable ways to declare variables are presented. Good style dictates that whichever is used by used everywhere within a file; however, the tools do not check for such consistency.

* Incorrect placement and use of braces around if/else constructs. The heuristic test for this is not run by default, and is not completely reliable. For simplicity’s sake, here is the correct format for if/else in which both alternatives are compound statements:

```c
if (x != SOME_CONSTANT) {
    do_stuff();
    return (x);
} else {
    do_other_stuff();
    x = OTHER_CONSTANT;
}
```

If neither alternative is a compound statement, use:

```c
if (x != SOME_CONSTANT)
    do_stuff();
else
    do_other_stuff();
```

Finally, if only one alternative is a compound statement, both alternatives should have braces, formatted as shown:

```c
if (x != SOME_CONSTANT) {
    do_stuff();
    return (x);
} else {
    do_other_stuff();
}
```

Note the placement of braces; they should surround an ”else” and only the closing brace should be on a line by itself.

* Use of comments associated with conditional compilation directives. It is good style to include trailing comments after #else and #endif directives describing the condition it references; for example:
CHAPTER 7. BEST PRACTICES AND REQUIREMENTS

```c
#ifdef __FOO
...
#else /* !_FOO */
...
#endif /* !_FOO */
```

The style tools do not check for the presence, usefulness, or correctness of these trailing comments, but you should normally include them, especially if the intervening code blocks are lengthy or the tests are part of a complicated set of nested preprocessor conditionals.

* Incorrect guards in header files. Guard names should be derived from the header name, but hdrchk does not check this. For example, a header installed in `<sys/scsi/foo_impl.h>` should have a guard as follows:

```c
#ifndef _SYS_SCSI_FOO_IMPL_H
#define _SYS_SCSI_FOO_IMPL_H
...
#endif /* _SYS_SCSI_FOO_IMPL_H */
```

However, hdrchk verifies neither the actual guard token name nor the comment following `#endif`.

* Comment style is only partially checked. For example, the correct style for block comments is:

```c
/**
 * Some comment here.
 * More here.
 */
```

cstyle(1) can detect some common errors, such as enclosing the entire block comment in a box of asterisks or other characters. However, it cannot detect all such errors; for example:

```c
/**
 * Some comment here.
 * We conserve asterisks even though it’s harder to read.
 * Our comment is nonconforming.
 */
```

Correct indentation of comments is also unchecked; block and single-line comments within functions should be indented at the same level as the code they document. Trailing comments should be aligned with other trailing comments in related code. None of these style guidelines is checked by the tools. It probably goes without saying that the contents of comments aren’t checked in any way. Please, be sure to follow the comment content guidelines in the style guide.
7.2.3 Non-Formatting Considerations

cstyle(1) and hdrchk(1) will, in addition to several code-formatting guidelines, verify compliance with SCCS keyword and copyright requirements. All files must be under SCCS control, and each must have a set of keywords near the top (see the prototypes for the exact order of file contents). In general, the keywords should have the following format in C files and headers:

```
#pragma ident "@(#)devref-ext.pod 1.4 05/03/22 SMI" /* optional comment */
```

Note that this string actually contains several embedded tabs; showing these tabs as \t, the keywords look like:

```
#pragma ident\t"@(#)devref-ext.pod\t1.4\t05/03/22 SMI\t/\t/* optional comment */
```

Files which are not C (or C++) implementations or headers should dispense with the "pragma" portion and just use #ident as follows:

```
#ident "@(#)devref-ext.pod\t1.4\t05/03/22 SMI\t/\t/* optional comment */
```

In addition, each file must contain a statement of copyright at the very top. The acceptable text and formats for these copyrights are shown in the example files in usr/src/prototypes. When a source file is significantly updated, the year or range of years in the copyright should be replaced with just the current year at the time of modification. Significant updates do not include formatting changes. However, if you prefer not to think about whether your change constitutes a significant update or if you don't trust your judgment, erring on the side of always updating the year is acceptable. Since the prototypes assume that Sun is always the sole copyright holder, you will need to change the copyright statement slightly; for example, if a file initially contains:

```
/*
 * Copyright (c) 1992-1998 by Sun Microsystems, Inc.
 * All rights reserved.
 */
```

you should replace this notice with:

```
/*
 * Copyright 2005 Sun Microsystems, Inc.
 * All rights reserved. Use is subject to license terms.
 */
```

and, if you are a contributor not employed by Sun, you should also add:

```
/*
 * Copyright 2005 J. Random Hacker
 * All rights reserved. Use is subject to license terms.
 */
```

or a similar statement of your copyright. Do not combine copyright notices, and do not remove the existing copyright notices unless specifically instructed to do so.
7.2.4 Integration Requirements

Before submitting any changes for code review prior to inclusion in OpenSolaris, you should self-review your code for stylistic correctness (as well as semantic correctness!). Running the cstyle(1) and/or hdrchk(1) tools is part of this process, but you should also be familiar with the entire style guide so that you will be able to use correct style even in elements that the tools do not check. All new code must be cstyle- or hdrchk-clean. If you are modifying existing code which does not conform to the style guide, your changes should be conformant, and at worst your changes should not increase the number of cstyle or hdrchk warnings.

7.3 Testing Guidelines

TBD

7.4 Using Lint

The entire Solaris kernel and many libraries and commands are completely lint clean, both pass1 and pass2. It is important that we maintain this cleanliness as yet another tool to insure a high quality release. Although not all non-kernel code is lint-clean, new code should be, and all new commands and libraries should be entirely lint-clean. Lint-clean code should indicate to the build system that it should be linted. For an example of code in transition to lint-cleanliness, see usr/src/cmd/cmd-inet/usr.sbin/Makefile and the associated code.

Before checking in your changes use the following steps (on both SPARC and x86) to check for any new errors:

```bash
% cd $SRC/uts
% ( make && make lint ) > ERRS 2>&1
% grep "warning:" ERRS

*ANY* warning messages must be fixed!
```

Note that you can also use nightly(1) to generate lint output. If you instruct nightly(1) to run lint, which you should always do by adding 'l' to your NIGHTLY_OPTIONS or explicitly using the '-l' command line option, it will produce a listing of any "lint noise" as part of its final output if your changes have introduced lint warnings; it should be empty. This can be much easier than looking through a log file as described above, and includes any lint warnings from both kernel and non-kernel code which is intended to be lint-clean.

In the build system, linting is fairly similar to a "normal" build, but it has an additional complication. In order to get meaningful output from lint pass2, all the modules must be linted together. This is accomplished by each module being responsible to produce its own pass1 output (file.in, one per .c/.s file). It is also responsible for placing the lint-library (llib-lMODULE) in the uts/MACHINE/lint-libs directory. The final full lint is accomplished by the
makefile in the uts/MACHINE directory by linting all the lint-libraries against each other.

Note that, to keep lint happy about functions defined in assembly only, there are also C prototypes in the .s files. For example:

```c
#if defined(lint)
int
blort(int, int)
{ return 0 }
#else /* lint */
ENTRY(blort)
ld [%i0],....
....
SET_SIZE(blort)
#endif /* lint */
```

Here are some additional rules for keeping code lint-clean and avoiding lint problems:

* NEVER EVER use /* LINTLIBRARY */ in OpenSolaris kernel source!

* Modification of header files, either directly or as a result of a merge, does not always cause the lint libraries to be rebuilt. This may result in seemingly impossible errors of a function having two different declarations. Be sure to run "make clean.lint" after any merge or modification of a header file.

* When calling a function with a return value but not using it, place a (void) cast in front. Common functions are sprintf and strcpy. Many times ignoring a function return value can hide error conditions. Functions that always have the values ignored might need to consider having their specification redeclared void, although of course this normally does not apply to public interfaces.

* Format strings for long integers use one of "%ld", %lx", or "%lu"

* Format strings for unsigned integers use one of "%u" or "%x"

* Format strings for long long integers use "%lld" or "%llx"

* Format strings for pointers should use "%p", with the pointer cast to (void *). Using %d or %x and casting to an (int) will break in a 64-bit kernel.

* For code that is supposed to work for either ILP32 or LP64, there are some macros that you can use in the format string, so that you don’t need to #ifdef longs versus ints. See <sys/int_fmtio.h>.

* Use full ANSI/ISO prototypes in function declarations, as the kernel is always compiled with _STDC_ defined. New code should not use K&R-style declarations.

* Make sure machine dependent function declarations are consistent across platforms.

* Be sure you are linting against the headers in your proto area and not the installed headers on your build machine.
7.5 Tips and Suggestions

TBD - We welcome community feedback and suggestions.
Appendix A

For Further Reading

TBD depending on which documents are aggregated here versus maintained separately.
Appendix B

About Opensolaris.org

TBD
Appendix C

Glossary

This glossary lists terms which are in common use within Sun but may be unfamiliar to other Open Source and Free Software developers. While we expect OpenSolaris will in time develop its own unique nomenclature, you may see other community members, especially Sun engineers, using these terms.

**BFU** Blindingly Fast Upgrade a.k.a. Bonwick/Faulkner Upgrade. This is a way to upgrade the subset of a system’s binaries that are delivered by the consolidation that uses cpio(1) archives instead of packages to improve speed. This requires manual resolution of configuration file conflicts and can be hazardous; therefore it is recommended only for developers who have read and understand 5.3 Using BFU to Install ON. BFU is implemented by ’mkbfu’, ’makebfu’, ’cpiotranslate’, and bfu(1).

**binary patch** Collection of updated binary objects distributed to customers other than when official releases are made. Binary patches may correct an urgent security problem or address a specific bug specific to a major customer. Binary patches will not be issued for OpenSolaris, since the sources and latest binary builds will include all such fixes.

**brickify** To render a system unbootable or otherwise unusable (a brick). Causes can be hardware or software, but in this document it usually means improper installation or installation of broken bits; in these cases the term ”warmbrick” is also used. Recovering from this condition usually entails booting from alternate media.

**CTF** Compact ANSI-C Type Format. CTF is a debugging information format similar to a subset of DWARF or STABS, but more efficient. The information is used by mdb, dtrace, and other facilities within Solaris.
**consolidation**  A set of related software components developed and delivered together. An example is the § consolidation, which consists of the kernel, libraries, and basic utility and server programs in OpenSolaris. Other consolidations deliver the windowing system, development tools, application servers, and so on.

**encumbered**  Sources licensed from third parties which are used to build Solaris and the OpenSolaris binary distributions but not distributed in source form because Sun’s license does not include the right to redistribute them. Over time, most or all encumbered code will be replaced by unencumbered code, which will be added to OpenSolaris.

**gate**  The main or official workspace for a project or consolidation. This workspace is managed by the gatekeeper for the project or consolidation, who performs regular builds, backs out incorrect or non-conforming changes, and is responsible for either integrating the sources into a parent gate or delivering regular builds to the §, as appropriate. After completing implementation and review, a developer will putback his or her changes to an appropriate gate. A gate is “golden source,” a shared resource expected to be usable at all times.

**lint**  lint(1) is a utility used to perform various checks on source code. All new code in OpenSolaris must be “lint-clean,” meaning that lint’s checks on the code do not result in any warnings. This process of checking with lint(1) is known as linting.

**Nevada**  Nevada is the current (as of 2005) development version of Solaris following Solaris 10. OpenSolaris is based on the Solaris Nevada sources. The name Nevada is used instead of 10.1 or 11 because it is not yet known whether this will be a micro or minor release; the initial assumption is that it will be a micro release; therefore the RELEASE is 5.10.1. However, if we decide to allow certain incompatible changes during development, Nevada will become a minor release and RELEASE will be 5.11. See attributes(5) and 7.1.1 Interface Stability Taxonomy for more information about the difference between release types.

**ON**  The consolidation which delivers the Solaris or OpenSolaris kernel, filesystems, some drivers and other modules, basic commands, daemons and servers, libraries, and system headers. Also known as OS/Net or OS/Networking.

**project**  A collection of features and/or bug fixes which is extensive enough to require its own implementation team, gate, and plans. Examples of projects are dtrace, Janus, and a port to a new hardware platform. Normal individual enhancements and bug fixes are self-contained and worked on by no more than one or two developers; these do not require the infrastructure associated with projects.
putback  After all changes are checked in, tested, reviewed, and approved, a developer integrates his or her changes into an appropriate gate. The word putback is used to refer to the set of changes itself and the act of integrating it into the gate.

Solaris Next  The generic term occasionally used to refer to a new development version of Solaris, before it is known what type of release it will be. As of early 2005, Solaris Next would refer to Nevada.

source patch  A set of diffs which describes a source code change. Many Open Source development efforts refer to this simply as a patch; however, see also binary patch above.

Tonic  Internal code name for the OpenSolaris project.

workspace  A workspace includes a full source tree as well as metadata such as log files and version control information.

WOS  "Wad Of Stuff," referring to the integration of all consolidations that make up the Solaris binary distribution shipped to customers.
Appendix D

MANUAL INDEX

Install(1)
ar(1)
archives(4)
as(1)
bfu(1)
cc(1)
contract(4)
cpio(1)
cryptoadm(1M)
cstyle(1)
depend(4)
dmake(1)
dtrace(1)
dtrace(7D)
exec(2)
gzip(1)
hdrchk(1)
isalist(1)
kcf(1M)
kmdm(1)
ld(1)
lex(1)
libcontract(3LIB)
libdtrace(3LIB)
make(1)
mcs(1)
nightly(1)
pkginfo(4)
pkgmap(1)
pkgmk(1)
pkgproto(1)
proc(4)
prototype(4)
yacc(1)
Appendix E

CONCEPT INDEX