Tour of your ON workspace

Sachidananda Urs

Abstract

This document explains the source layout of the Solaris/illumos operating system. The original webpage that explained the below process is removed by Oracle and is no longer available. I have copied the content and removed the unnecessary bits.

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 ON workspace.

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.

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. Nightly and buildenv is explained in another document.

All sources are found under usr/src. This includes both the sources used to build the ON 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 may not include all 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.

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.

pkg
: Contains a manifests subdirectory with one manifest file for each package generated from the ON sources. The rest of the files here are used to drive package publication based on those manifests. See pkg/README.pkg for more information.

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).

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. Note: This directory does not exist on illumos, this entry is kept purely for historical reasons.

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.

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/gssapi
: 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/krtld
: 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/rpcsvc
Generated RPC header files shipped in /usr/include/rpcsvc.

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 IIi-NetraCT
• mpxu	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

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

ON 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.

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. The sources are not contained in these build directories.

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.

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.

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 installation 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. Makefile.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. Message catalogues provide translations of messages for g11n purposes.

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.

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 subdirectories 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.

Makefile.*.shared

These makefiles provide settings that are shared between the open-source makefiles and the closed-source makefiles that are internal to Sun.

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.

Command Makefile Layout

Most command and daemon subdirectories follow one of two general layout rules, depending on where object files will be located. 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.

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.

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.

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.

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.

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 ON.

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:

```
$(BINARY):      $(OBJECTS)
        $(LD) -r $(LDFLAGS) -o $@ $(OBJECTS)
        $(CTFMERGE_UNIQUIFY_AGAINST_GENUNIX)
        $(POST_PROCESS)
        $(ELFSIGN_MOD)
```

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 $@
        ...
```

Command Makefile Operation

Command makefiles vary widely, as described above. Therefore it is impossible to provide a “representative example” of a command makefile. 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.

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.

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 examined, and the second specifies (indirectly) the location in which the program 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 programs 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:

```
$(ROOT)/usr/sbin/dtrace --> $(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.

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:

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

DLIBSRCS += errno.d \
            io.d \
            procfs.d \
            regs.d \
            sched.d
            signal.d \
            unistd.d
...

ROOTDLIBDIR = $(ROOT)/usr/lib/dtrace
ROOTDLIBDIR64 = $(ROOT)/usr/lib/dtrace/64
...
$(ROOTDLIBDIR):
        $(INS.dir)


$(ROOTDLIBDIR64): $(ROOTDLIBDIR)
        $(INS.dir)


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

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

$(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 LIBSRCS 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 of system libraries is not supported by OpenSolaris-based distributions, 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:

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

pics/%.o: ../$(MACH)/%.s
        $(COMPILE.s) -o $@ $$<$
        $(POST_PROCESS_O)
```

```
%.o: ../common/%.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:

```
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 is available online.

Understanding Kernel Module Installation

The install target causes binaries and headers to be installed into a hierarchy called the prototype or proto area. 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:

```
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_CONFILE)
```

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)
```

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:

```
SUBDIR64_sparc = sparcv9
SUBDIR64_i386  = amd64
SUBDIR64       = $(SUBDIR64_$(MACH))
```

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

```
ROOT_KERN_DIR_32        = $(ROOT_MOD_DIR)
ROOT_DRV_DIR_32         = $(ROOT_MOD_DIR)/drv
ROOT_DTRACE_DIR_32      = $(ROOT_MOD_DIR)/dtrace
ROOT_EXEC_DIR_32        = $(ROOT_MOD_DIR)/exec
...


ROOT_KERN_DIR_64        = $(ROOT_MOD_DIR)/$(SUBDIR64)
ROOT_DRV_DIR_64         = $(ROOT_MOD_DIR)/drv/$(SUBDIR64)
ROOT_DTRACE_DIR_64      = $(ROOT_MOD_DIR)/dtrace/$(SUBDIR64)
ROOT_EXEC_DIR_64        = $(ROOT_MOD_DIR)/exec/$(SUBDIR64)
...
```

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

```
ROOT_KERN_DIR   = $(ROOT_KERN_DIR_$(CLASS))
ROOT_DRV_DIR    = $(ROOT_DRV_DIR_$(CLASS))
ROOT_DTRACE_DIR = $(ROOT_DTRACE_DIR_$(CLASS))
ROOT_EXEC_DIR   = $(ROOT_EXEC_DIR_$(CLASS))
...
```

There are similar definitions for USR_XXX_DIR based in $(ROOT)/usr/kernel. See usr/src/uts/Makefile.uts for the complete list of possible ROOT_ and USR_ installation directories.

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

```
$(ROOT_MOD_DIR)/%: $(OBJS_DIR)/% $(ROOT_MOD_DIR) FRC
        $(INS.file)

$(ROOT_DRV_DIR)/%: $(OBJS_DIR)/% $(ROOT_DRV_DIR) FRC
        $(INS.file)
...
```

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

\$(ROOT_CONFFILE) is the module's configuration file (most commonly used for drivers), transformed into the proto area. The CONFFILE variables are defined in usr/src/uts/Makefile.uts:

```
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.

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.

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.

Source Files not Included

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

The source could be subject to third-party rights. Sun's lawyers won't let anyone talk about this. Not even a little bit. They do love explaining it themselves, though, so you should contact Sun's legal department and ask there. We can't help you. Sorry. The consolidation may not yet have been released. The OpenSolaris code base does not yet include all consolidations in Solaris, and some consolidations are still incomplete. To accommodate fully functional builds for ON, even though some sources are missing, a set of closed binaries are available, and the build system has been modified to make use of them. The makefile variable CLOSED_BUILD controls whether the build system will use the prebuilt closed binaries or look for the closed sources.

CLOSED_BUILD is typically used in makefile lines that build up the list of modules or subdirectories that are to be built. Makefile lines that start with CLOSED_BUILD, e.g.,

```
$(CLOSED_BUILD)DCSUBDIRS += $(CLOSED)/cmd/pax
```

are for use only when the closed sources are present.

Sometimes a separate closed module list is built up, rather than adding to the open module list. This approach is used extensively in the kernel, as in

```
$(CLOSED_BUILD)CLOSED_DRV_KMODS += chxge
```

Because this approach requires additional variables, the first approach is usually preferred. The second approach is used when the list needs to contain individual component names, rather than paths. For example, the kernel makefiles use the module names to construct a series of "-l" clauses for linting; that would be a lot messier if paths were used instead of module names.

You shouldn't normally need to set CLOSED_BUILD. Instead, bldenv(1) and nightly(1) set the environment variable CLOSED_IS_PRESENT, and the ON makefiles use this to set CLOSED_BUILD.

The kernel provides an additional challenge, which is that many makefile rules and declarations are practically the same for both the open and closed source trees. To avoid duplication of code, the common text was put in shared makefiles (e.g., Makefile.sun4u.shared). When used for open code, the makefile variable $(UTSTREE) is set to $(UTSBASE). For closed code, it is set to $(UTSCLOSED).