Building ON

Sachidananda Urs

Abstract

This document discusses two commonly-used ways to build ON: - 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. The instructions in this document apply to ON and similar consolidations, including SFW. Other consolidations may have substantially different build procedures, which should be incorporated here.

Environment Variables

This section describes a few of the environment variables that affect all ON builds, regardless of the build method.

CODEMGR_WS This variable 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. Originally, CODEMGR_WS was defined by TeamWare. Over time, ON's build tools have come to depend on it, so we continue to use it with Mercurial.

SRC This variable must be set to the root of the ON 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:
        MACH=`/usr/bin/uname -p`
Bad:
        MACH=sparc
```

ROOT Root of the proto area for the build. The makefiles direct the installation of header files and libraries to this area and direct references to these files by builds of commands and other targets. It should be expressed in terms of $CODEMGR_WS. 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 This points to the top of the installed compiler tree. People outside Sun will normally set this to /opt/SUNWspro. You can see how this works by looking at usr/src/Makefile.master. But if you need to override the default, you should do so via the environment variable. Note that opensolaris.sh already sets SPRO_ROOT to /opt/SUNWspro.

SPRO_VROOT The 'V' stands for version. At Sun, multiple versions of the compilers are installed under $SPRO_ROOT to support building older sources. The compiler itself is expected to be in $SPRO_VROOT/bin/cc, so you will most likely need to set this variable to /opt/SUNWspro. Note that opensolaris.sh has this variable already set to this value.

GNU_ROOT The GNU C compiler is used by default to build the 64-bit kernel for amd64 systems. 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.

__GNUC, __GNUC64 These variables control the use of gcc. __GNUC controls the use of gcc to build i386 and sparc (32-bit) binaries, while __GNUC64 controls the use of gcc to build amd64 and sparcv9 (64-bit) binaries. Setting these variables to the empty value enables the use of gcc to build the corresponding binaries. Setting them to '#' enables Studio as the primary compiler. The default settings use Studio, with gcc invoked in parallel as a 'shadow' compiler (to ensure that code remains warning and error clean).

CLOSED_IS_PRESENT This variable tells the ON makefiles whether to look for the closed source tree. Normally this is set automatically by nightly(1) and bldenv(1).

ON_CLOSED_BINS If the closed source tree is not present, this variable points to the tree of unpacked closed binaries.

Using nightly and bldenv

There are many steps in building any consolidation; ON's build process entails creation of the proto area, compiling and linking binaries, generating lint libraries and linting the sources, building 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 processes. 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 log is normally located in $CODEMGR_WS/log/log.YYYY-MM-DD.HH:MM/nightly.log, where YYYY-MM-DD.HH:MM is a timestamp, 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 report 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 effects your changes, if any, have 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’
```

Note the ‘<’ - this means this output was for the previous build. If the output is prefaced with ‘>’, it is associated with the most recent build. In this way you will be able to see whether you have corrected all problems or introduced new ones.

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.

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. 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.

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
ON\_CRYPTO\_BINS
MAKEFLAGS
```

DEBUG and Non-DEBUG Builds

The ON sources contain additional debugging support and self-checks (assertions) that can be enabled by doing a DEBUG build. This makes the binaries larger and slower, but unless you are running on older hardware, you are unlikely to notice the difference. Benchmarking should only be done with non-DEBUG builds.

Non-DEBUG binary deliverables (e.g., closed binaries) are usually tagged with "-nd" in the file name. If you are unsure as to which you have, the kernel will say when it boots if it is a DEBUG kernel. Non-DEBUG closed binaries unpack into closed/root_$MACH-nd; DEBUG closed binaries unpack into closed/root_$MACH. In the source tree, non-DEBUG kernel objects live in "objNN" (e.g., "obj64" or "obj32") directories, while DEBUG kernel objects live in "debugNN" directories.

When building with nightly(1), you can specify a DEBUG build, a non-DEBUG build, or both, by setting the appropriate flags in
NIGHTLY_OPTIONS. Note that "D" turns on the DEBUG build, while "F" turns off the non-DEBUG build.

By default, nightly(1) will reuse the proto area for both the DEBUG and non-DEBUG builds. (This is done to save space.) You can tell nightly(1) to put the non-DEBUG build in a separate proto area by setting the MULTI_PROTO environment variable to "yes". In that case, the (default) DEBUG proto area will be $CODEMGR_WS/proto/root_$MACH, and the (default) non-DEBUG proto area will be $CODEMGR_WS/proto/root_$MACH-nd. Be careful about extracting binaries–especially kernel modules–by hand from the proto area. It doesn't always work to mix DEBUG and non-DEBUG kernel modules.

When building with bldenv(1), you can build either DEBUG binaries or non-DEBUG binaries, but not both at the same time. bldenv(1) is not smart enough to look at NIGHTLY_OPTIONS to determine the build type; see CR 6414851 for details. Instead, bldenv(1) defaults to a non-DEBUG build. Use "bldenv -d" to get a DEBUG build.

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;

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 some other distributions. 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 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 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.

Build Products

A fully-built source tree is not very useful by itself; the binaries, scripts, and configuration files that make up the 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. Alternately, the Install program can be used to construct a kernel installation archive directly from a fully-built usr/src/uts.

Below sections describes the proto area, the most basic collection of built objects and the first to be generated by the build system. BFU archives, which are used to upgrade a system with a full OpenSolaris-based distribution installation to the latest ON bits. The construction of the package tree for ON deliverables. The construction of deliverables that can be posted on opensolaris.org.

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. Normally nightly(1) and bldenv(1) will set ROOT

for you. The usual value for ROOT is $CODEMGR_WS/proto/root_$MACH, though it may be different for non-DEBUG builds.

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.

If you're checking a current ON workspace, then you can simply build the protocmp make target in usr/src/pkg:

```
$ bldenv opensolaris.sh
$ cd $SRC/pkg
$ dmake protocmp
```

Otherwise, you may invoke protocmp directly.

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

```
protocmp [-gupGUPlmsLv] [-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 SVr4 packaging directory 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 or package manifest, 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.

For a current ON workspace, you can generate a protolist corresponding to the package manifests by building the protolist make target in usr/src/pkg:

```
$ bldenv opensolaris.sh
$ cd $SRC/pkg
$ dmake protolist
```

This will generate the file $SRC/pkg/protolist_‘uname -p‘.

You can generate protolists from a proto area with the protolist command, which also 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) for more information on automating proto comparison as part of your automatic build.

In addition to the protolist and protocmp build targets described above, as a shortcut to using protolist and protocmp, you can use the ‘checkproto’ command found in the developer/opensolaris/onbld 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 developer/opensolaris/onbld package.

BFU Archives

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

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 above 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 above section for more information on automating BFU archive construction as part of your automatic build.

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

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

Packages

The -p option to nightly will package the files that were installed into your proto area. If your workspace contains the directory usr/src/pkgdefs, this will result in SVr4 packages. If instead it contains usr/src/pkg, it will produce an IPS package repository. In either case, this is done automatically by the main makefile's all and pkg_all targets (see usr/src/Makefile) as part of a successful build. SVr4 packages or IPS package repositories are placed in $CODEMGR_WS/packages. See nightly(1) and section above for more information on automating package construction as part of your automatic build.

OpenSolaris Deliverables

A project–including the ON gate–may wish to post tarballs on opensolaris.org for the benefit of other developers or users. nightly(1) will generate tarballs for closed binaries (if the project has access to the closed source) signed cryptographic binaries. This section explains how to use nightly(1) to produce these tarballs. It also sketches what nightly(1) does to make this work, and it explains project team responsibilities, especially when introducing new code.

Generating Deliverables

To generate tarballs that can be posted on opensolaris.org, add 'O' (upper case oh) to NIGHTLY_OPTIONS. When nightly(1) finishes, you will have these files in your workspace top-level directory, assuming that you are doing both DEBUG and non-DEBUG builds:

```
README.opensolaris
on-closed-bins-nd.$MACH.tar.bz2
on-closed-bins.$MACH.tar.bz2
on-crypto.$MACH.tar.bz2
on-crypto-nd.$MACH.tar.bz2
```

These files are ready to post.

How It Works

This section describes how the deliverables listed in above section are generated.

The README is created from a template and two files from opensolaris.org (the current issues list and the installation quick-start instructions).

To help external developers produce builds that are as functional as possible, we include closed binaries that correspond to the given source snapshot. External developers can then bake these binaries into their own builds.

The first step in producing the closed binaries is to create a shadow proto area that contains only closed binaries. This is done as part of the normal build–when a closed makefile installs a file into the proto area, it installs a second copy into the closed proto area. The second step is to filter out binaries that cannot be included in the tarball.

The crypto tarball contains signed cryptographic binaries from one of two sources. If the build is a "signing" build (usually an ON gate build), the binaries come from the workspace's proto area. These binaries have been signed with a special key and certificate that let the binaries be run anywhere. If the build is not a signing build, the workspace's binaries can only be run inside Sun/Oracle. In that case, nightly copies a tarball from a signing build from the ON gate.

Some open source licenses require that a copy of the license be included with binary deliveries. For this reason nightly(1) includes an aggregated list of licenses in the closed-bins and crypto tarballs. The list of license files is extracted from the package manifest files, then filtered to eliminate licenses that don't apply to the binaries in question.

Most license files are static (that is, they are checked in like any source file). A few are derived dynamically from other files in the source tree. Each license file has a corresponding "description" file (e.g., THIRDPARTYLICENSE.descrip). This file contains the one-line description that identifies the software covered by the license.

Team Responsibilities

If you are updating or introducing new code, there are three things you need to look at to support opensolaris.org deliveries: third-party licenses, confidential code (internal teams adding source to the closed tree), and crypto code.

For third-party license files, the basic requirement is to keep the bookkeeping straight. So when you update code that already has a third-party license, review the license text for changes (e.g., new copyright years). If there are changes, make sure they are propagated as needed. In a few cases, you will have nothing to do, because the build just drops in the license file that we get from upstream. In most cases, you will need to copy the changes to the static third-party license file that lives in the source directory. In the cases where the license text is dynamically extracted, you can do a "make install" and then review the generated license file, to make sure that the extraction code is still working.

If you add new code that has a third-party license, review the instructions in usr/src/README.license-files.

If you delete code that has a third-party license, check what license(s) cover any remaining code in the package. If there is nothing left that is covered by a particular license, then you should remove that license from the package's manifest.

If you are on a Sun/Oracle-internal team and have to deal with confidential code, determine whether the resulting deliverables (e.g., binaries) can be included in the closed binaries tarball. Contact tonic-iteam@sun.com if you need help with this. If the file cannot be included, add it to the exclusion list in usr/src/tools/scripts/bindrop.sh.

If you are adding a new location in the source tree for crypto code, make sure the new location is listed in usr/src/tools/scripts/mktpl.pl (look for the declaration of $iscrypto).