oi-userland Makefile targets and variables

Deprecation warning

The page has been replaced by documentation in oi-userland: https://github.com/OpenIndiana/oi-userland/blob/oi/hipster/doc/makefile-targets.txt and https://github.com/OpenIndiana/oi-userland/blob/oi/hipster/doc/makefile-variables.txt

General notes

You should not need to use root privileges to run any userland operations. If something only appears to work because you're running as root, there's a good chance that something's broken that isn't being fixed by hammering it with privileges. If in doubt, please mail to oi-dev@openindiana.org or reach us on IRC.

Directory structure

The top level directory of the workspace is $(WS_TOP), which is the same thing as the root directory of the repository.

Build infrastructure directories

Build infrastructure lives in the following directories: transforms, make-rules, and tools.

transforms

This directory contains transform rules for packaging metadata. Packaging is covered on a separate page, so we will not address the particulars here.

make-rules

The make-rules are a series of files that need to be referenced by **include** statements in component Makefiles. Normally the first line of the Makefile is an include of make-rules/shared-macros.mk, which is referenced by a relative path (e.g. include ../../make-rules/shared-macros.mk). This file defines $(WS_TOP), so subsequent includes can use that in the path expansion (e.g. include $(WS_TOP)/make-rules/shared-targets.mk; relative paths such as include ../../make-rules/shared-targets.mk are also acceptable). One normally has to include shared-targets.mk, ips.mk, and depend.mk further down in the Makefile. We will return later to the conventional ordering of the Makefile.

tools

This directory contains various tools used by the build system. Tools are covered by a separate page, so we will address them on this page only in terms of variables used to invoke them and generalities of where that needs to be done.

$(WS_REPO)

This is the top-level directory for IPS package publication. This is automatically defined as $(WS_TOP/$(MACH)/repo, where $(MACH) is the value of uname -p.

Component directory structure

Components live under the components top-level directory. Most components exist as immediate child directories (e.g. component/links), but some related components are nested another level down (e.g. components/openssl/openssl-1.0.0 and components/openssl/openssl-fips).

$(COMPONENT_DIR)

The directory in which a component lives is $(COMPONENT_DIR). It contains the component Makefile, its packaging manifest(s), and, optionally, a license file, a patches directory, and an augments directory. This directory is automatically defined and should not be changed. It is assumed to be the current working directory in which all Makefile targets are invoked.

$(SOURCE_DIR)

Source is downloaded and extracted to this subdirectory of $(COMPONENT_DIR). It is automatically defined as $(COMPONENT_DIR) /$(COMPONENT_SRC) and should not be changed.

$(PATCH_DIR)

Patches are stored in and automatically applied from this directory. It is automatically defined as $(COMPONENT_DIR)/patches. See "patch stage" under "Targets" below for more information.

$(BUILD_DIR)

Builds happen under this directory, which is automatically defined as $(COMPONENT_DIR)/build and should not be changed.

$(BUILD_DIR_32), $(BUILD_DIR_64)

$(BUILD_DIR_32) is automatically defined as $(BUILD_DIR)/$(MACH32) and $(BUILD_DIR_64) as $(BUILD_DIR)/$(MACH64). Neither should be changed. They are used to separate the 32- and 64-bit build trees for each platform. For SPARC systems, MACH32 is sparcv7 and MACH64 is sparcv9. For x86 systems, MACH32 is i386 and MACH64 is amd64.

$(PROTO_DIR), $(PROTO*DIR)

Built content is installed into the prototype directory before it is packaged. $(PROTO_DIR) is automatically defined as $(BUILD_DIR)/prototype/$(MACH), where $(MACH) is either $(MACH32) or $(MACH64). Which values are used depends on how the build and install targets, discussed below, are defined. These values should not be changed. Need to provide all $(PROTO*DIR) variables and their values.

$(MANGLED_DIR)

Content that needs to be mangled for packaging is staged in this directory, which is automatically defined as $(PROTO_DIR)/mangled.

Targets

Targets are invoked with the Makefile (e.g. gmake build). Because they are defined by the Makefile, they are generally executed in the same directory, and although Makefiles can potentially be executed from elsewhere using the -f <path to Makefile> operand, this isn't recommended, as a number of variables necessary to their execution are defined in terms of the current working directory.

Top-level targets ($(WS_TOP))

The following rules run the equivalent component target for all components:

- download
- prep
- build
- install
- clean
- clobber
- test
- env-check
- env-prep
- REQUIRED_PACKAGES

prep

The prep rule runs the bass-o-matic tool for all components. See the tools documentation for further information.

check-environment

This rule confirms that all necessary build tools have been installed.

setup

This rule sets up the publication repo ($(WS_REPO)) and pkglint-cache for packages.

Component targets

There are eight basic targets for userland component Makefiles: prep, build, install, check, publish, sample-manifest, clean, and clobber. Except for clean and clobber, these steps are stateful, with a dot file prerequisite created when they are completed successfully ($(SOURCE_DIR)/.prep, $(BUILD_DIR)/%/.built, $(BUILD_DIR)/%/.installed, $(BUILD_DIR)/%/.tested, $(BUILD_DIR)/%/.publish, where % is $(MACH32), $(MACH64), or both, depending or how build, install, check, or publish has been defined, as explained below). sample-manifest uses the generated manifest to keep state.

Some of these targets also have stages or virtual targets, distinct groups of actions that are sufficiently involved that we document them in their own right. These are documented with the target of which they are a stage.

prep, build, and install must be completed in sequence, and install is a prerequisite for publish, sample-manifest, and check.

It is important to note that the Makefile itself is a prerequisite of the prep stage, which is a prequisite for all rules other than clean and clobber. If you update the Makefile, it will require all stateful targets to be performed again. If you know what you are doing (i.e. you are only modifying variables or targets applicable only to the target you need to invoke again), you make touch the relevant state files for the steps you don't need to repeat.

Many of the component targets have related definitions for their environment and actions, which are covered in subsequent sections.

prep

The prep step downloads the source using $(FETCH), extracts it using $(UNPACK), and applies patches (see patch stage). Normally you do not have to define or directly invoke this target, as it is a prerequisite to the build target, and all of the actions to complete are already defined in the core Makefiles.

Some components have to be assembled from multiple archives (git, for example, provides its man pages in a separate archive at the time of this writing). The mandatory variables listed below tell prep how to retrieve and extract a single archive, but these can be extended to define an arbitrary further number by adding _1, _2, etc. to set of the variable names.

Mandatory variables: $(COMPONENT_SRC), $(COMPONENT_ARCHIVE), $(COMPONENT_ARCHIVE_URL), $(COMPONENT_ARCHIVE_HASH). Optional variables: $(FETCH) (defaults to userland-fetch), $(UNPACK) (defaults to userland-unpack), $(COMPONENT_SRC$(1)), $(COMPONENT_ARCHIVE$(1)), $(COMPONENT_ARCHIVE_URL$(1)), $(COMPONENT_ARCHIVE_HASH$(1)).

patch stage

Any file ending in .patch (or $(PATCH_PATTERN), if it is defined) in $(PATCH_DIR) is automatically applied to $(SOURCE_DIR) when it is extracted. Alternatively, all patches can be defined using the $(PATCHES) variable, which is otherwise expanded in terms of the variables just described. It is very important to note that patches are normally applied against the source rather than the build directory, meaning that patches must normally be applicable to all builds. For recommendations on dealing with this, please see the section on target environment. $(PATCH_LEVEL) is used to for -p argument to gpatch, which is used to apply patches. It's default setting is 1 (meaning that the diffs used to generate the patches were generated in $(COMPONENT_DIR) rather than $(SOURCE_DIR)).

The patch stage is stateful via the automatic expansion of the $(STAMPS) variable to files that are touched for successful application of individual $(PATCHES) and then $(SOURCE_DIR)/.patched for the set.

build

The build target causes a build tree to be created, build to be configured, and source to be compiled, where relevant. The build step is one of the most variable, so the explanation here tries to cover the majority of cases without dealing with all possible exceptions. The simplest build rules are defined using existing targets: $(BUILD_32) or $(BUILD_64), causing the desired combination of 32- and 64-bit binaries to be built. $(BUILD_32) builds in $(BUILD_DIR_32) and $(BUILD_64) builds in $(BUILD_DIR_64). build itself is undefined by default.

The build target does not strictly apply to all components: some components consist entirely of pre-built content or data that needs to be installed and packaged. build is, however, a prerequisite for install (and thus publish), so it needs to be short-circuited. The simplest way to do this is to define it as the completion of the prep step:

```
build: $(SOURCE_DIR)/.prep
```

configure stage

configure has a set of defaults that should be right for most configure/GNU auto*-based systems that prepare the distributed Makefiles (what makes Makefile.in a suitable Makefile for each platform). There are a number of variables applicable to this stage: $(CONFIGURE_SCRIPT), $(CONFIGURE_OPTIONS). Conventionally, $(CONFIGURE_SCRIPT) is invoked with the installation directory information passed as the $(CONFIGURE_OPTIONS) arguments, and the compile environment is passed via the environment (see CONFIGURE_ENV section below under "Target environment").

One case in which you may wish to use $(CONFIGURE_OPTIONS) is if you're building GNU/FSF executables which are already exist in illumos, where you want to install them in /usr/gnu/bin and choose between them using shell customization such as $PATH, functions, or aliases. Thus:

```
CONFIGURE_OPTIONS += --bindir=$(GNUBIN)
CONFIGURE_OPTIONS += --sbindir=$(GNUSBINDIR)
```

$(CONFIGURE_OPTIONS) should be treated a list and added to using "+=" rather than assigned (and thus clobbered) using "=".

configure is stateful via $(BUILD_DIR_32)/.configured and/or $(BUILD_DIR_64)/.configured.

Optional variables: $(ACLOCAL), $(AUTOMAKE), $(AUTORECONF), $(CONFIG_SHELL), $(CONFIGURE_SCRIPT), $(CONFIGURE_OPTIONS).

install

The install target causes the built content to be installed into $(DESTDIR), which is automatically defined at $(PROTO_DIR). The automatically defined target $(INSTALL_32) install from the 32-bit build and $(INSTALL_64) from 64-bit. install itself is undefined by default.

If building for both 32- and 64-bit, there are a number of ways to approach merging the two. One way is to invoke the install rule for one and then use post-install actions (see COMPONENT_POST_INSTALL_ACTION below in the "Target actions" section) for the rest of the merge. For example, if you are building a component that includes both libraries and executables, you may wish to have only 32-bit executables but libraries for both:

```
build: $(BUILD_32) $(BUILD_64)

install: $(INSTALL_32)

COMPONENT_POST_INSTALL_ACTION += \
     $(CP) $(BUILD_DIR_64)$(USRLIBDIR64)/* $(PROTOUSRLIBDIR64)
```

publish

The publish target takes all the packaged manifest files (*.p5m) under $(COMPONENT_DIR) and runs them through pkgmogrify, pkgdepend, pkgmangle, and pkglint, which are documented in full detail on a separate page dedicated to package manifests. If all these steps are successful, the package will be published to $(WS_REPO).

sample-manifest

This target takes the contents of $(PROTO_DIR) and generates a manifest for each $(MACH) value as $(BUILD_DIR)/manifest-$(MACH)-generated.p5m. This rule is useful beyond generating an initial manifest. Components may add or remove files over time, so it is useful to generate a sample-manifest every time and compare it to the canonical(s) in $(COMPONENT_DIR). This target will also store a copy of generated sample manifest in manifests/ directory. This eases determining what changed between package versions.

test

The test target is used to run a component's bundled test suite, if applicable. test can be defined as either $(TEST_32), $(TEST_64), both, or $(NO_TEST), which is a successful no-op for components that do not have a test suite. $(TEST_32) tests what's in $(BUILD_DIR_32) and $(TEST_64) tests what using $(COMPONENT_TEST_TARGET), which is defined as "check" by default.

clean

The clean target recursively deletes $(BUILD_DIR) and $(PROTO_DIR).

clobber

clobber defines clean as a prerequisite and adds to that $(RM) $(CLOBBER_PATHS). $(CLOBBER_PATHS) is automatically defined as $(COMPONENT_ARCHIVE$(1)).

env-prep

env-prep installs missing package dependencies defined in REQUIRE_PACKAGES variable in component Makefile. This target needs privileges to install packages.

env-check

env-check checks the build environment for missing build dependencies.

REQUIRED_PACKAGES

REQUIRED_PACKAGES tries to guess build dependencies and adds this information into component's Makefile. Usually, contributor should double check and add any missing package.

Target actions

Actions are the conventional steps that are used for targets, which can be replaced or extended to deal with individual components. Their names should make self-evident what they do: allow additional actions to be performed before the main activity of configure, build, install or test. The below provides a few tips and tricks as examples.

Note that actions are effectively a list, so you'll normally want to add to them using a "+=" assignment rather than a "=". You should also end them with a ";" as a shell separator.

$(COMPONENT_PRE_CONFIGURE_ACTION)

A very common pre-configure action is to use the cloney tool (documented more fully on the userland tools page) to generate a complete tree of links from $(SOURCE_DIR) to $(BUILD_DIR).

```
COMPONENT_PRE_CONFIGURE_ACTION = \
     $(CLONEY) $(SOURCE_DIR) $(@D)
```

$(COMPONENT_POST_CONFIGURE_ACTION)

$(COMPONENT_PRE_BUILD_ACTION)

$(COMPONENT_POST_BUILD_ACTION)

$(COMPONENT_PRE_INSTALL_ACTION)

$(COMPONENT_POST_INSTALL_ACTION)

Post-install actions are often used to clean up the prototype directory after it's been populated by the component's own Makefile. For example, a component may use a man page with macros not supported by illumos, so we have to substitute those with re-writes:

```
COMPONENT_POST_INSTALL_ACTION += $(RM) $(PROTOUSRSHAREMAN1DIR)/* && \
     $(CP) $(COMPONENT_DIR)/augments/man/man1/* $(PROTOUSRSHAREMAN1DIR) ;
```

$(COMPONENT_PRE_TEST_ACTION)

$(COMPONENT_POST_TEST_ACTION)

Target environment

$(ENV)

$(CONFIGURE_ENV)

$(COMPONENT_BUILD_ENV)

This sets the environment variables set during the build phase. If the build process invokes configure for a needed sub-module, then any needed variables such as CC and CFLAGS should be set here as well.

$(COMPONENT_INSTALL_ENV)

Compiler/linker variables

Variants

Variants are a way of providing different builds in the same package. These can be used for variant packaging (covered on the package manifest authoring page) or to produce multiple package or binaries with different configure options.

Library linting

Python setup.py

Ant

Recommended practices

If you update a Makefile or packaging without performing a complete rebuild, it is recommended that, before you submit changes, you clobber your build and confirm that it is reproducible.

If you patch source, it is recommended that you report the issue requiring patching to the upstream so that they can either accept the patch or suggest workarounds or alternative solutions.

Generate a sample-manifest each time you update a component to a new version and compare it to canonical manifest(s) to see if different files or directories are being populated into the prototype directories, requiring updates to the canonical(s).