NUT Quality Assurance and Build Automation Guide

Jim Klimov

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
2.8.3	2025-04-07	Some changes to docs and recipes, libusbclient API and functionality. Updated NUT SEMVER definition and added scripting around it. Groundwork for vendor-defined status and INSTCMD buzzwords like "ECO". Fixed some regressions and added improvements for certain new device series.	JK

Contents

Abstract

The aim of this document is to describe the different ways we ensure and maintain the source code quality of Network UPS Tools, its portability to various platforms, and non-regression.

Previous NUT releases may have included parts of this documentation in the developer guide or user manual. Most of this information can be applied to both automated testing environments and local development workflows.

Warning
This is a Work In Progress document.

1 NUT Quality Assurance

Recognizing the critical nature of NUT, the NUT Quality Assurance (NQA) effort has been established to improve NUT where necessary, and to maintain software quality as high as it should be.

NQA is present in many aspects and areas of NUT.

1.1 Documentation

The documentation toolchain uses AsciiDoc to output both HTML pages and manual pages (troff). This single point of control fills many gaps, suppresses many redundancies, and optimizes documentation management in general.

- The NUT website and HTML documentation are tested for W3C XHTML 1.1 and CSS compliance. This can be counter verified by clicking the W3C XHTML 1.1 and CSS icons, at the bottom of each page.

- Documentation source files are spell checked, using Aspell, both interactively (using *make spellcheck-interactive*) and automatically in NUT CI farm runs (using *make spellcheck*).

Note
A NUT dictionary is also available (as `docs/nut.dict` in NUT sources), providing a glossary of terms related to power devices and management, as well as partial terms, technical jargon and author names.

1.2 Source code

1.2.1 Use of standards

NUT promotes and uses many standards, such as:

- the variable names standard used in NUT,
- the coding rules and best practices for developers,
- the use of a software architecture limiting developments to the very minimum,
- the use of standard Free and OpenSource Software components, like:
  - the USB library,
  - the Net SNMP project,
  - the Neon library,
  - the OpenSSL library (to be replaced by NSS, which is more license compliant with NUT and can be FIPS 140 certified),
  - the TCP Wrappers library.

1.2.2 QA tools and metrics

NUT's quality is constantly monitored using many tools, like:

- a Revision Control System (Git) to track development and ease regression fixes.
- OBSOLETE: Buildbot for older take on multi-platform builds, and the
- new dedicated Jenkins incarnation of the NUT CI Farm with "legacy UI" for main branch and PRs, also accessible at the slower but slicker-looking Blue Ocean user interface for activity, branches and PRs, are all used to automate the compile/test cycle, using numerous platforms, target distributions, C/C++ revisions, compiler toolkits and make program implementations. Any build failure is caught early, and fixed quickly. Also we get to see if incoming pull requests (as well as Git branch HEADs) do not have code (or recipe) that is instantly faulty and can not build on one of the platforms we track even with relaxed warnings.
- a project portal with trackers for bugs, feature request, patches and tasks, currently at GitHub
- CodeQL (previously LGTM.COM) automatically checking C/C++ and Python code
- Static code analysis:
  - Coverity Scan overview of NUT
  - status: Coverity Scan Build Status
  - cppcheck as part of NUT CI farm builds and reports

NUT QA also relies on external tools and trackers, like:

- Clang
- the Debian QA tools, available through the NUT Package Tracking System:
  - Lintian general QA checks,
  - piuparts automates the installation, upgrade and removal testing processes.
- a runtime testing suite, which automates the inter-layer communication testing (driver — upsd — upsmon / clients), that is part of Ubuntu. The NUT testing script is available in the Ubuntu QA Regression Testing suite.

  It installs NUT packages, configures it with the dummy-ups driver, changes a few data points and checks that these are well propagated with upsc.
- similar approach is explored in NIT (NUT Integration Testing) suite, which is part of the codebase and automated with `make check-NIT`; it can also be added to default `make check` activities by running `configure --enable-check-NIT`
  - Note that developers updating components which directly impact NIT runs may benefit from `make check-NIT-devel` target, to rebuild the `upsd`, `dummy-ups`, `cppnit` and other programs used in the test as they iterate.
- Redhat / Fedora Bug tracker
- Black Duck Open Hub (formerly Ohloh.net) provides metrics on NUT source code base and activity.

1.2.3 Runtime quality

- NUT provides many security features (or `docs/security.txt` in NUT sources for up-to-date information) to ensure a maximum runtime security level.
- Packages use several Hardening methods to protect NUT binaries.

2 Code and recipe analysis

2.1 GNU Autotools distcheck

The Network UPS Tools project code base is managed by the GNU Build System colloquially known as "The Autotools", which include `autoconf`, `automake` and many other components. Some of their important roles are to generate the portable shell `configure` script to detect build environment capabilities and other nuances, and to help actually build the project with `Makefile` recipes (supported by many implementations of the standard POSIX `make` tool) generated from the `Makefile.am` templates by the `automake` tool.

Among the many standard recipes promulgated by the Autotools, `make dist` handles creation of archive files with a release (or other) snapshot of a software project, which "distribute" all the original or generated files needed to build and install that project on a minimally capable end-user system (should have a compiler, `make`, and dependency libraries/headers, but is not required to have autotools, manual page generation tools, etc.)

The `make distcheck` goal allows to validate that the constructed archive is in fact sufficient for such a build (includes all required files), and also that the code structure and its recipes properly support out-of-tree builds (as used on multi-platform and cross-build environments) without contaminating the source code directory structure.

NUT's root 'Makefile.am` defines the `DISTCHECK_FLAGS` eventually passed to the `configure` script executed as part of `distcheck` validation, and the default set of the flags requires to build everything. This in turn constrains the set of systems where this validation can be performed to build environments that have all dependency projects installed, have the documentation generation tools, etc. in order to make sure that for all files that are compiled or otherwise processed by the build routine, we actually distribute the sources (implicitly as calculated by programs' listed sources, or via explicit `EXTRA_DIST` and similar statements) regardless of features enabled or not to be built in the original run.

To avoid this constraint and allow the relevant `distcheck`-like validation to happen on environments without "everything and a kitchen sink" installed, further recipes are defined by NUT, such as:

- `distcheck-light`: does not **require** the optional features to be built, but just allow them (using `--with-all=auto --with-ssl=auto --with-doc=auto` etc. flags);
- `distcheck-light-man`: similar to the above, but require validation that manual pages can all be built (do not build PDF or HTML formats, though);
- `distcheck-fake-man`: for formal validation on systems without documentation processing tools used by NUT recipes, populate the distribution archive with "PLACEHOLDER" contents of missing pre-generated manual pages (such an archive SHOULD NOT be delivered to end-users as a fully functional release), so the validation of **recipes** around pre-built documentation installation can be performed;
- `distcheck-ci`: based on current build circumstances, dispatch to standard strict `distcheck` or to `distcheck-fake-man`.

Other recipes based on this concept are also defined, including:

- `distcheck-valgrind`: build whatever code we can, and do not waste time on documentation processing (`--with-all=auto --with-ssl=auto --with-doc=skip`), to run the NUT test programs (`make check` in the built environment) through the Valgrind memory-checking tool.

2.2 Valgrind checks

NUT sources include a helper script and a suppression file which allow developers and CI alike to easily run built programs through the popular Valgrind tool and check for memory leaks, un-closed file descriptors, and more.

One use-case to cover the population of NUT self-test programs (and the common code they pull in from NUT libraries and drivers) is automated as the `make distcheck-valgrind` goal.

2.2.1 VALGRIND resources

Helper script and suppression file to analyze NUT binaries.

Example use-case:

```
:; make -ks -j && LD_LIBRARY_PATH=`pwd`/clients/.libs \
    ./scripts/valgrind/valgrind.sh ./tools/nut-scanner/nut-scanner -DDDDDD -m auto
```

Note that the script is generated under `${top_builddir}` by `configure` from a template file located in `${top_srcdir}/scrip`
You might be able to run it directly, falling back to a `valgrind` program in your `PATH`, if any.

See also:

- Valgrind Suppression File How-to
  - Notably, add `--gen-suppressions=all --error-limit=no` to `valgrind` program options to generate suppression snippets

2.3 cppcheck

The root `Makefile.am` includes a recipe to run a special build of NUT analyzed by the `cppcheck` tool (if detected by `configure` script) and produce a `cppcheck.xml` report for further tools to use, e.g. visualize it by the Jenkins Warnings plugin.

2.4 Static analysis by compilers

As compilers like GCC and LLVM/CLANG evolve, so do their built-in code analyzers and warnings. In fact, this is a large part of the reasoning behind using a vast array of systems along with the compilers they have (many points of view on the same code discover different issues in it), and on another — behind the certain complexity pattern in NUT's own code base, where code recommended by one compiler seems offensive to another (so stacks of `pragma` expressions are used to quiesce certain warnings around certain lines).

2.4.1 Pre-set warning options

The options chosen into pre-sets that can be selected by `configure` script options are ones we use for different layers of CI tests.

Values to note include:

- `--enable-Werror(=yes/no)` — make warnings fatal;
- `--enable-warnings(=.../no)` — enable certain warning presets:
  - `gcc-hard`, `clang-hard`, `gcc-medium`, `clang-medium`, `gcc-minimal`, `clang-minimal`, `all` — actual definitions that are compiler-dependent (the latter just adds `-Wall` which may be relatively portable);
  - `hard`, `medium` or `minimal` — if current compiler is detected as CLANG or GCC, apply corresponding setting from above (or `all` otherwise);
  - `gcc` or `clang` — apply the set of options (regardless of detected compiler) with default "difficulty" hard-coded in `configure` script, to tweak as our codebase becomes cleaner;
  - `yes`/`auto` (also takes effect if `--enable-warnings` is requested without an `=ARG` part) — if current compiler is detected as CLANG or GCC, apply corresponding setting with default "difficulty" from above (or `all` otherwise).

Note that for backwards-compatibility reasons and to help filter out introduction of blatant errors, builds with compilers that claim GCC compatibility can enable a few easy warning presets by default. This can be avoided with an explicit argument to `--disable-warnings` (or `--enable-warnings=no`).

All levels of warnings pre-sets for GCC in particular do not enforce the `-pedantic` mode for builds with C89/C90/ANSI standard revision (as guesstimated by `CFLAGS` content), because nowadays it complains more about the system and third-party library headers, than about NUT codebase quality (and "our offenses" are mostly something not worth fixing in this era, such as the use of `__func__` in debug commands). If there still are practical use-cases that require builds of NUT on pre-C99 compiler toolkits, pull requests are of course welcome — but the maintainer team does not intend to spend much time on that.

Hopefully this warnings pre-set mechanism is extensible enough if we would need to add more compilers and/or "difficulty levels" in the future.

Finally, note that such pre-set warnings can be mixed with options passed through `CFLAGS` or `CXXFLAGS` values to your local `configure` run, but it is up to your compiler how it interprets the resulting mix.

2.5 Shell script checks

The `make shellcheck` recipe finds files which the `file` tool determines to be POSIX or Bourne-Again shell scripts, and runs them through respective interpreter's (`bash` or system `/bin/sh`) test mode to validate the syntax works.

Given that the `/bin/sh` implementation varies wildly on different systems (e.g. Korn shell, BASH, DASH and many others), this goal performed by CI on a large number of available platforms makes sure that the lowest-denominator syntax we use is actually understood everywhere.

Note
At a later time additional tests, perhaps using the `shellcheck` tool, can be introduced into the stack.

The `make shellcheck-nde` recipe calls `tests/nut-driver-enumerator-test.sh` to self-test the `scripts/upsdrvs`
against an array of `SHELL_PROGS` (e.g. a list of interpreters provided by specific CI agents), and make sure that shell-script based processing of `ups.conf` in various interpreters provides the exact spelling of expected results.

2.6 Documentation spelling checks

NUT recipes rely on the `aspell` tool (with `aspell-en` dictionary, default but different on numerous platforms), and a custom maintained dictionary file (specified in `Makefile` variables as `NUT_SPELL_DICT` — by default, it is `${top_srcdir}/docs/nut`
for additional words either unique to NUT or quite common but absent in older standard dictionaries on some systems. Operations are done according to `LANG` and `LC_ALL` values, both specified in `Makefile` variables as `ASPELL_ENV_LANG`, by default `en.UTF-8`.

The "nut-website" generation has similar recipes and relies on integration with those provided by the main NUT code base, but maintains its own custom dictionary for words only present in the website sources.

The root `Makefile.am` includes recipes which allow developers and maintainers to check spelling of all documentation (and/or update the custom dictionary), while recipes in numerous subdirectories (where `README.adoc` or other specific documentation files exist) have similar goals to check just their files.

The actual implementation of the goals is in `docs/Makefile.am`, and either calls the tool if it was detected by the `configure` script, or skips work.

For each checked file, a `*-spellchecked` touch-file is created in respective `${builddir}`, so it is not re-checked until the source document, the custom dictionary, or the `Makefile` recipe is updated.

The ecosystem of `Makefile.am` files includes the following useful recipes:

- `spellcheck`: passively check that all used words are in some dictionary known to this system, or report errors for unknown words;

- `spellcheck-interactive`: actively check the documents, and for unknown words start the interactive mode of `aspell` so you can either edit the source text (replace typos with suggested correct spelling), update the custom dictionary, or ignore the hit (to rephrase the paragraph later, etc.)

Note
This recipe can update the timestamp of the custom dictionary file, causing all documents to become fair game for re-checks of their spelling. * `spellcheck-sortdict`: make sure the custom dictionary file is sorted alphanumerically (helpful in case of manual edits) and the word count in the heading line is correct (helpful in case of manual edits or git branch merges).

The root `Makefile.am` also provides some aids for maintainers:

- `spellcheck-interactive-quick`: runs "passive" `spellcheck` in parallel `make` mode, and only if it errors out — runs `spellcheck-interactive`;
- `spellcheck-report-dict-usage`: prepares a `nut.dict.usage-report` file to validate that words used in a custom dictionary are actually present in any NUT documentation source file.

3 Test automation

3.1 The ci_build.sh script

This script was originally introduced (following ZeroMQ/ZProject example) to automate CI builds, by automating certain scenarios driven by exported environment variables to set particular `configure` options and `make` some targets (chosen by the `BUILD_TYPE` envvar). It can also be used locally to avoid much typing to re-run those scenarios during development.

Developers can directly use the scripts involved in CI builds to fix existing code on their workstations or to ensure support for new compilers and C standard revisions, e.g. save a local file like this to call the common script with pre-sets:

```
$ cat _fightwarn-gcc10-gnu17.sh
#!/bin/sh

BUILD_TYPE=default-all-errors \
CFLAGS="-Wall -Wextra -Werror -pedantic -std=gnu17" \
CXXFLAGS="-Wall -Wextra -Werror -std=gnu++17" \
CC=gcc-10 CXX=g++-10 \
    ./ci_build.sh
```

... and then execute it to prepare a workspace, after which you can go fixing bugs file-by-file running a `make` after each save to confirm your solutions and uncover the next issue to address :-)

Helpfully, the NUT CI farm build logs report the configuration used for each executed stage, so if some build combination fails — you can just scroll to the end of that section and copy-paste the way to reproduce an issue locally (on an OS similar to that build case).

Note that while spelling out sets of warnings can help in a quest to fix certain bugs during development (if only by removing noise from classes of warnings not relevant to the issue one is working on), there is a reasonable set of warnings which NUT codebase actively tries to be clean about (and checks in CI), detailed in the next section.

For the `ci_build.sh` usage like above, one can instead pass the setting via `BUILD_WARNOPT=...`, and require that all emitted warnings are fatal for their build, e.g.:

```
$ cat _fightwarn-clang9-gnu11.sh
#!/bin/sh

BUILD_TYPE=default-all-errors \
BUILD_WARNOPT=hard BUILD_WARNFATAL=yes \
CFLAGS="-std=gnu11" \
```

```
CXXFLAGS="-std=gnu++11" \
CC=clang-9 CXX=clang++-9 CPP=clang-cpp \
    ./ci_build.sh
```

Finally, for refactoring effort geared particularly for fighting the warnings which exist in current codebase, the script contains some presets (which would evolve along with codebase quality improvements) as `BUILD_TYPE=fightwarn-gcc`, `BUILD_TYPE=fightwarn-clang` or plain `BUILD_TYPE=fightwarn`:

```
:; BUILD_TYPE=fightwarn-clang ./ci_build.sh
```

As a rule of thumb, new contributions must not emit any warnings when built in GNU99 mode with a `minimal` "difficulty" level of warnings. Technically they must survive the part of test matrix across the several platforms tested by NUT CI and marked in project settings as required to pass, to be accepted for a pull request merge.

Developers aiming to post successful pull requests to improve NUT can pass the `--enable-warnings` option to the `configure` script in local builds to see how that behaves and ensure that at least in some set-up their contribution is viable. Note that different compiler versions and vendors (gcc/clang/...), building against different OS and third-party dependencies, with different CPU architectures and different language specification revisions, might all complain about different issues — and catching this in as diverse range of set-ups as possible is why we have CI tests.

It can be beneficial for serial developers to set up a local BuildBot, Travis or a Jenkins instance with a matrix test job, to test their local git repository branches with whatever systems they have available.

- https://github.com/networkupstools/nut/issues/823

While `autoconf` tries its best to provide portable shell code, sometimes there are builds of system shell that just fail under stress. If you are seeing random failures of `./configure` script in different spots with the same inputs, try telling `./ci_build.sh` to loop configuring until success (instead of quickly failing), and/or tell `./configure` to use another shell at least for the system call-outs, with options like these:

```
:; SHELL=/bin/bash CONFIG_SHELL=/bin/bash CI_SHELL_IS_FLAKY=true \
   ./ci_build.sh
```

3.2 Test programs in NUT codebase

FIXME: Write the chapter text.

For now, investigate files in the `tests/` directory contents in NUT sources.

3.3 NUT Integration Testing suite (aka NIT)

This suite aims to simplify running `upsd`, a `dummy-ups` driver and a few clients to query them, as part of regular `make check` routine or separately with existing binaries (should not impact any existing installation data, processes or communications).

Warning
Current working directory when starting the script should be the location where it may create temporary data (e.g. the `BUILDDIR`).

See also The NUT testing script available in the Ubuntu QA Regression Testing suite and Debian packaging recipe doing a similar job with NUT installed from packages and configuring it via files in standard path names.

A sandbox prepared by this script can be used for `upsmon` testing:

```
:; make check-NIT-sandbox-devel &

# Wait for sandbox, e.g. test that "${NUT_CONFPATH}/NIT.env-sandbox-ready"
# file appeared; then source the envvars, e.g.:
:; sleep 5 ; while ! [ -e ./tests/NIT/tmp/etc/NIT.env-sandbox-ready ] ; do sleep 1 ; done
:; . ./tests/NIT/tmp/etc/NIT.env

# Prepare upsmon.conf there, e.g.:
:; printf 'MINSUPPLIES 1\nPOWERDOWNFLAG "%s/killpower"\nSHUTDOWNCMD "date >> \"%s/nut- ↩
    shutdown.log\""\nMONITOR "%s@127.0.0.1:%s" 1 "%s" "%s" primary\n' \
    "$NUT_STATEPATH" "$NUT_STATEPATH" 'dummy' "$NUT_PORT" \
    'dummy-admin' "${TESTPASS_UPSMON_PRIMARY}" \
    > "${NUT_CONFPATH}/upsmon.conf"
```

The `nit.sh` script supports a lot of environment variables to tune its behavior, notably `NIT_CASE`, `NUT_PORT`, `NUT_STATEPATH` and `NUT_CONFPATH`, but also many more. See its sources, as well as the top-level `Makefile.am` recipe and the `./tests/NIT/tmp` file generated during a test run, for more details and examples about the currently supported tunables.

4 Continuous Integration (NUT CI farm) technologies

4.1 CI Farm configuration notes

Note
This chapter contains information about NUT CI farm setup tricks that were applied at different times by the maintainer team to ensure regular builds and tests of the codebase. Whether these are used in daily production today or not, similar setup should be possible locally on developer and contributor machines.

4.2 Multiple FOSS CI providers and technologies

While there are many FOSS-friendly CI offerings, they are usually (and reasonably) focused on the OS market leaders — offering recent releases of Linux, Windows and MacOS build agents, and sometimes a way to "bring your own device" to cover other systems. The NUT CI farm does benefit from those offerings as well, using GitHub Actions with CodeQL for Linux code quality inspection, AppVeyor CI for Windows, and CircleCI for MacOS, to name a few.

But on the other hand, being a massively multi-platform effort (and aiming to support older boxes that are still alive even if their vendors and/or distro versions are not), a comprehensive NUT CI approach requires many machines running uncommon operating systems. This is where custom virtual machines help, and more so — a core set of those hosted in the cloud and dedicated to the project, rather than only some resources intermittently contributed by community members which come and go.

Note
Community-provided builders running on further systems are also welcome, and the option is of course supported, as managed by the Jenkins-Dynamatrix effort which appeared due to such need, and runs the core NUT CI farm.

We have also had historic experience with FOSS CI providers (and community members' machines) disappearing, so having NUT CI farm goals covered by multiple independent implementations is also a feature beyond having yet another set of digital eyes looking at our code quality (which is also a goal in itself).

4.3 Jenkins is the way

- https://stories.jenkins.io/user-story/jenkins-is-the-way-for-networkupstools/
- https://github.com/jenkins-infra/stories/blob/main/src/user-story/jenkins-is-the-way-for-networkupstools/index.yaml

4.3.1 The jenkins-dynamatrix library

FIXME: Write the chapter text.

For now, see https://github.com/networkupstools/jenkins-dynamatrix sources (note the README and large comments at start of files may be obsolete, as of this writing — documenting the initial ideas, but the implementation might differ from that over time).

4.3.2 Jenkinsfile-dynamatrix cases in NUT sources

FIXME: Write the chapter text.

For now, see the `Jenkinsfile-dynamatrix` in the NUT sources (maybe only git), e.g. https://github.com/networkupstools/-nut/blob/master/Jenkinsfile-dynamatrix for the practical pipeline preparation and hand-off to library implementation.

4.3.3 Jenkins CI

Since mid-2021, the NUT CI farm is implemented by several virtual servers courteously provided originally by Fosshost and later by DigitalOcean.

These run various operating systems as build agents, and a Jenkins instance to orchestrate the builds of NUT branches and pull requests on those agents.

This is driven by `Jenkinsfile-dynamatrix` and a Jenkins Shared Library called jenkins-dynamatrix which prepares a matrix of builds across as many operating systems, bitnesses/architectures, compilers, make programs and C/C++ revisions as it can — based on the population of currently available build agents and capabilities which they expose as agent labels.

This hopefully means that people interested in NUT can contribute to the build farm (and ensure NUT is and remains compatible with their platform) by running a Jenkins Swarm agent with certain labels, which would dial into https://ci.networkupstools.org/ controller. Please contact the NUT maintainer if you want to participate in this manner.

The `Jenkinsfile-dynamatrix` recipe allows NUT CI farm to run different sets of build scenarios based on various conditions, such as the name of branch being built (or PR'ed against), changed files (e.g. C/C++ sources vs. just docs), and some build combinations may be not required to succeed.

For example, the main development branch and pull requests against it must cleanly pass all specified builds and tests on various platforms with the default level of warnings specified in the `configure` script. These are balanced to not run too many build scenarios overall, but just a quick and sufficiently representative set.

As another example, there is special handling for "fightwarn" pattern in the branch names to run many more builds with varying warning levels and more variants of intermediate language revisions, and so expose concerns deliberately missed by default warnings levels in "master" branch builds (the bar moves over time, as some classes of warnings become extinct from our codebase).

Further special handling for branches named like `fightwarn.*89.*` regex enables more intensive warning levels for a GNU89 build specifically (which are otherwise disabled as noisy yet not useful for supported C99+ builds), and is intended to help develop fixes for support of this older language revision, if anyone would dare.

Many of those unsuccessful build stages are precisely the focus of the "fightwarn" effort, and are currently marked as "may fail", so they end up as "UNSTABLE" (seen as orange bubbles in the Jenkins BlueOcean UI, or orange cells in the tabular list of stages in the legacy UI), rather than as "FAILURE" (red bubbles) for build scenarios that were not expected to fail and usually represent higher-priority problems that would block a PR.

Developers whose PR builds (or attempts to fix warnings) did not succeed in some cell of such build matrix, can look at the individual logs of that cell. Beside indication from the compiler about the failure, the end of log text includes the command which was executed by CI worker and can be reproduced locally by the developer, e.g.:

```
22:26:01  FINISHED with exit-code 2 cmd:  (
22:26:01  [ -x ./ci_build.sh ] || exit
22:26:01
22:26:01  eval BUILD_TYPE="default-alldrv" BUILD_WARNOPT="hard" \
    BUILD_WARNFATAL="yes" MAKE="make"  CC=gcc-10 CXX=g++-10 \
```

```
    CPP=cpp-10 CFLAGS='-std=gnu99 -m64' CXXFLAGS='-std=gnu++11 -m64' \
    LDFLAGS='-m64' ./ci_build.sh
22:26:01  )
```

or for autotools-driven scenarios (which prep, configure, build and test in separate stages — so for reproducing a failed build you should also look at its configuration step separately):

```
22:28:18  FINISHED with exit-code 0 cmd:  ( [ -x configure ] || exit; \
    eval  CC=clang-9 CXX=clang++-9 CPP=clang-cpp-9 CFLAGS='-std=c11 -m64' \
    CXXFLAGS='-std=c++11 -m64' LDFLAGS='-m64' time ./configure )
```

To re-run such scenario locally, you can copy the line from `eval` (but without the `eval` keyword itself) up to and including the executed script or tool, into your shell. Depending on locally available compilers, you may have to tweak the `CC`, `CXX` and `CPP` arguments; note that a `CPP` may be specified as `/path/to/CC -E` for GCC and CLANG based toolkits at least, if they lack a standalone preprocessor program (e.g. IntelCC).

Note
While NUT recipes do not currently recognize a separate `CXXCPP`, it would follow similar semantics.

Some further details about the NUT CI farm workers are available in Prerequisites for building NUT on different OSes (or `docs/config-prereqs.txt` in NUT sources for up-to-date information) and Custom NUT CI farm build agents: LXC multi-arch containers (or `docs/ci-farm-lxc-setup.txt` in NUT sources for up-to-date information) documentation.

4.4 AppVeyor CI

Primarily used for building NUT for Windows on Windows instances provided in the cloud — and so ensure non-regression as well as downloadable archives with binary installation prototype area, intended for enthusiastic testing (proper packaging to follow). NUT for Windows build-ability was re-introduced soon after NUT 2.8.0 release.

This relies on a few prerequisite packages and a common NUT configuration, as coded in the `appveyor.yml` file in the NUT codebase.

4.5 CircleCI

Primarily used for building NUT for MacOS on instances provided in the cloud, and so ensure non-regression across several Xcode releases.

This relies on a few prerequisite packages and a common NUT configuration, as coded in the `.circleci/config.yml` file in the NUT codebase.

4.6 Travis CI

See the `.travis.yml` file in project sources for a detailed list of third party dependencies and a large matrix of `CFLAGS` and compiler versions last known to work or to not (yet) work on operating systems available to that CI solution.

Note
The cloud Travis CI offering became effectively defunct for open-source projects in mid-2021, so the `.travis.yml` file in NUT codebase is not actively maintained.
Local private deployments of Travis CI are possible, so if anybody does use it and has updated markup to share, they are welcome to post PRs.

The NUT project on GitHub had integration with Travis CI to test a large set of compiler and option combinations, covering different versions of gcc and clang, C standards, and requiring to pass builds at least in a mode without warnings (and checking the other cases where any warnings are made fatal).

4.7 CodeQL

(Earlier this role was performed by LGTM.com) Run GitHub Actions for static analysis of C, C++ and Python code and recipes, to produce suggestions based on common coding flaws and best-practice security patterns.

5 Continuous Integration (NUT CI farm) build agent preparation

5.1 Custom NUT CI farm build agents: VMs on DigitalOcean

This section details Installation of VMs on Digital Ocean.

5.2 Setting up the non-standard VM farm for NUT CI on DigitalOcean

Since 2023 the Network UPS Tools project employs virtual machines, hosted and courteously sponsored as part of FOSS support program by DigitalOcean, for a significant part of the NUT CI farm based on a custom Jenkins setup.

Use of complete machines, virtual or not, in the NUT CI farm allows our compatibility and non-regression testing to be truly multi-platform, spanning various operating system technologies and even (sometimes emulated) CPU architectures.

To that extent, while it is easy to deploy common OS images and manage the resulting VMs, there are only so many images and platforms that are officially supported by the hosting as general-purpose "DigitalOcean VPS Droplets", and work with other operating systems is not easy. But not impossible, either.

In particular, while there is half a dozen Linux distributions offered out of the box, official FreeBSD support was present earlier but abandoned shortly before NUT CI farm migration from the defunct Fosshost.org considered this hosting option.

Still, there were community reports of various platforms including *BSD and illumos working in practice (sometimes with caveats), which just needed some special tinkering to run and to manage. This chapter details how the NUT CI farm VMs were set up on DigitalOcean.

5.2.1 Design trade-offs

Note that some design choices were made because equivalent machines existed earlier on Fosshost.org hosting, and filesystem content copies or ZFS snapshot transfers were the least disruptive approach (using ZFS wherever possible also allows to keep the history of system changes as snapshots, easily replicated to offline storage).

It is further important to note that DigitalOcean VMs in recovery mode apparently must use the one ISO image provided by DigitalOcean. At the time of this writing it was based on Ubuntu 18.04 LTS with ZFS support — so the ZFS pools and datasets on VMs that use them should be created **AND** kept with options supported by that version of the filesystem implementation.

Another note regards pricing: resources that "exist" are billed, whether they run or not (e.g. turned-off VMs still reserve CPU/RAM to be able to run on demand, dormant storage for custom images is used even if they are not active filesystems, etc.)

As of this writing, the hourly prices are applied for resources spawned and destroyed within a calendar month. After a monthly-rate total price for the item is reached, that is applied instead.

OS images

Some links will be in OS-specific chapters below; further reading for this effort included:

- https://www.digitalocean.com/blog/custom-images
- https://ptribble.blogspot.com/2021/04/running-tribblix-on-digital-ocean.html — notes on custom image creation, may involve https://github.com/illumos/metadata-agent
- https://bsd-cloud-image.org/ — A collection of pre-built *BSD cloud images

According to the fine print in the scary official docs, DigitalOcean VMs can only use "custom images" in one of a number of virtual HDD formats, which should carry an ext3/ext4 filesystem for DigitalOcean addons to barge into for management.

In practice, uploading other images (OpenIndiana Hipster "cloud" image, OmniOS, FreeBSD) from your workstation or by providing an URL to an image file on the Internet (see links in this document for some collections) sort of works. While the upload status remained "pending", a VM could often be made with it soon... but in other cases you have to wait a surprisingly long time, some 15-20 minutes, and additional images suddenly become "Uploaded".

- The initial theory was that we exceeded some limit and after ending the setups with one custom image, it can be nuked and then another used in its place; in practice this seems to be not true — just the storage information refresh (perhaps propagation from cache to committed) can lag.
- Note that your budget would be invoiced for storage of custom images too. If you use stock ISOs once, it makes sense to remove them later.
- There is also an option to use pre-installed operating systems, so you can dynamically create and destroy VMs with minimal work after creation (e.g. with Jenkins cloud plugins to spawn workers according to labels); in this case you may want to retain (eventually update) the golden image.
- It may be that not **all** non-standard images are supported, but those with `cloud-init` or similar tools (see https://www.digitalocean.co blog/custom-images for details).

Networking

FIXME: Private net, DO-given IPs

One limitation seen with "custom images" is that IPv6 is not offered to those VMs.

Generally all VMs get random (hopefully persistent) public IPv4 addresses from various subnets. It is possible to also request an interconnect VLAN for one project's VMs co-located in same data center and have it attached (with virtual IP addresses) to an additional network interface on each of your VMs: it is supposed to be faster and free (regarding traffic quotas).

- For the Jenkins controller which talks to the world (and enjoys an off-hosting backup at a maintainer's home server) having substantial monthly traffic quota is important.
- For the set of builders hosted on DigitalOcean, which would primarily talk to the controller in the common VLAN — not so much (just OS upgrades? maybe GitHub?)

One more potential caveat: while DigitalOcean provides VPC network segments for free inter-communications of a group of droplets, it assigns IP addresses to those and does not let any others be used by the guest. This causes some hassle when importing a set of VMs which used different IP addresses on their inter-communications VLAN originally (on another hosting).

Common notes for illumos VMs

The original OpenIndiana Hipster and OmniOS VMs were configured with the https://github.com/jimklimov/illumos-splitroot-scripts methodology and scripting, so there are quite a few datasets dedicated to their purposes instead of a large one.

There are known issues about VM reboot:

- Per https://www.illumos.org/issues/14526 and personal and community practice, it seems that "slow reboot" for illumos VMs on QEMU-6.x (and on DigitalOcean) misbehaves and hangs, ultimately the virtual hardware is not power-cycled.
- A power-off/on cycle through UI (and probably REST API) does work.
- It took about 2 hours for `rebooting...` to take place in fact. At least, the machine would not be stuck for eternity in case of unattended crashes.
- Other kernels (Linux, BSD, ...) are not impacted by this, it seems.

Wondering if there are QEMU HW watchdogs on DigitalOcean that we could use...

5.2.2 Using the DigitalOcean Recovery ISO

As noted above, for installation and subsequent management DigitalOcean's recovery ISO must be used when booting the VM, which is based on Ubuntu and includes ZFS support. It was used a lot both initially and over the years, so deserves a dedicated chapter.

To boot into the recovery environment, you should power off the VM (see the Power left-menu item in DigitalOcean web dashboard, and "Turn off Droplet"), then go into the Recovery menu and select "Boot from Recovery ISO" and power on the Droplet. When you are finished with recovery mode operations, repeat this routine but select "Boot from Hard Drive" instead.

Note
Sometimes you might be able to change the boot device in advance (it takes time to apply the setting change) and power-cycle the VM later.

The recovery live image allows to install APT packages, such as `mc` (file manager and editor) and `mbuffer` (to optimize `zfs-send`/`zfs-recv` traffic). When the image boots, it offers a menu which walks through adding SSH public keys (can import ones from e.g. GitHub by username).

Note that if your client system uses `screen`, `tmux` or `byobu`, the new SSH connections would get the menu again. To get a shell right away, interactive or for scripting like `rsync` and `zfs recv` counterparts, you should `export TERM=vt220` from your `screen` session (the latter proved useful in any case for independence of the long replication run from connectivity of my laptop to Fosshost/DigitalOcean VMs).

- SSH keys can be imported with a `ssh-import-id-gh` helper script provided in the image:

```
#recovery# ssh-import-id-gh jimklimov
2023-12-10 21:32:18,069 INFO Already authorized ['2048',
    'SHA256:Q/ouGDQn0HUZKVEIkHnC3c+POG1r03EVeRr81yP/TEoQ',
    'jimklimov@github/10826393', '[RSA]']
...
```

- More can be pasted into `~/.ssh/authorized_keys` later;
- The real SSH session is better than the (VNC-based web-wrapped) Rescue Console, which is much less responsive and also lacks mouse and copy-paste integration with your browser;
- On your SSH client side (e.g. in the `screen` session on original VM which would send a lot of data), you can add non-default (e.g. one-time) keys of the SSH server of the recovery environment with:

```
#origin# eval `ssh-agent`
#origin# ssh-add ~/.ssh/id_rsa_custom_key
```

Make the recovery userland convenient:

```
#recovery# apt install mc mbuffer
```

- `mc`, `mcview` and `mcedit` are just very convenient to manage systems and to manipulate files;
- ZFS send/receive traffic is quite bursty, with long quiet times as it investigates the source or target pools respectively, and busy streaming times with data.

  Using an `mbuffer` on at least one side (ideally both to smooth out network latency) is recommended to have something useful happen when at least one of the sides has the bulk data streaming phase.

5.2.3 OpenIndiana

Helpful links for this part of the quest:

- https://openindiana.org/downloads/ ⇒ see https://dlc.openindiana.org/isos/hipster/20231027/OI-hipster-cloudimage.img.gz — OI distro-provided cloud images (detailed at https://www.openindiana.org/announcements/openindiana-hipster-2023-04-announcemer release notes, though not at later ones)

DO-NUT-CI-OI VM creation

Initial attempt, using the OpenIndiana cloud image ISO:

The OI image could be loaded... but that's it — the logo is visible on the DigitalOcean Recovery Console, as well as some early boot-loader lines ending with a list of supported consoles. I assume it went into the `ttya` (serial) console as one is present in the hardware list, but DigitalOcean UI does not make it accessible and I did not find quickly if there are any REST API or SSH tunnel into serial ports.

Note
The web console did not come up quickly enough after a VM (re-)boot for any interaction with the early seconds of ISO image loader's uptime, if it even offers any.

It **probably** booted and auto-installed, since I could see an `rpool/swap` twice the size of VM RAM later on, and the `rpool` occupied the whole VM disk (created with auto-sizing).

The VM can however be rebooted with a (DO-provided) Recovery ISO, based at that time on Ubuntu 18.04 LTS with ZFS support — which was sufficient to send over the existing VM contents from original OI VM on Fosshost. See above about booting and preparing that environment.

DO-NUT-CI-OI VM OS transfer

As the practically useful VM already existed at Fosshost.org, and a quick shot failed at making a new one from scratch, in order to only transfer local zones (containers), a decision was made to transfer the whole ZFS pool via snapshots using the Recovery ISO.

First, following up from the first experiment above: I can import the ZFS pool created by cloud-OI image into the Linux Recovery CD session:

- Check known pools:

```
#recovery# zpool import
   pool: rpool
     id: 7186602345686254327
  state: ONLINE
 status: The pool was last accessed by another system.
 action: The pool can be imported using its name or numeric identifier and the `-f' flag.
    see: http://zfsonlinux.org/msg/ZFS-8000-EY
 config:
        rpool ONLINE
          vda ONLINE
```

- Import without mounting (`-N`), using an alternate root if we decide to mount something later (`-R /a`), and ignoring possible markers that the pool was not unmounted so might be used by another storage user (`-f`):

```
#recovery# zpool import -R /a -N -f rpool
```

- List what we see here:

```
#recovery# zfs list
NAME                  USED  AVAIL  REFER  MOUNTPOINT
rpool                34.1G   276G   204K  /rpool
rpool/ROOT           1.13G   276G   184K  legacy
rpool/ROOT/c936500e  1.13G   276G  1.13G  legacy
rpool/export          384K   276G   200K  /export
rpool/export/home     184K   276G   184K  /export/home
rpool/swap           33.0G   309G   104K  -
```

The import and subsequent inspection above showed that the kernel core-dump area was missing, compared to the original VM… so adding per best practice:

- Check settings wanted by the installed machine for the `rpool/dump` dataset:

```
#origin# zfs get -s local all rpool/dump
NAME        PROPERTY                       VALUE                          SOURCE
rpool/dump  volsize                        1.46G                          local
rpool/dump  checksum                       off                            local
rpool/dump  compression                    off                            local
rpool/dump  refreservation                 none                           local
rpool/dump  dedup                          off                            local
```

- Apply to the new VM:

```
#recovery# zfs create -V 2G -o checksum=off -o compression=off \
    -o refreservation=none -o dedup=off rpool/dump
```

To receive ZFS streams from the running OI into the freshly prepared cloud-OI image, it wanted the ZFS features to be enabled (all were disabled by default) since some are used in the replication stream:

- Check what is there initially (on the new VM):

```
#recovery# zpool get all
NAME   PROPERTY                       VALUE                          SOURCE
rpool  size                           320G                           -
rpool  capacity                       0%                             -
rpool  altroot                        -                              default
rpool  health                         ONLINE                         -
rpool  guid                           7186602345686254327            -
rpool  version                        -                              default
rpool  bootfs                         rpool/ROOT/c936500e            local
rpool  delegation                     on                             default
rpool  autoreplace                    off                            default
rpool  cachefile                      -                              default
rpool  failmode                       wait                           default
rpool  listsnapshots                  off                            default
rpool  autoexpand                     off                            default
rpool  dedupditto                     0                              default
rpool  dedupratio                     1.00x                          -
rpool  free                           318G                           -
rpool  allocated                      1.13G                          -
rpool  readonly                       off                            -
rpool  ashift                         12                             local
rpool  comment                        -                              default
rpool  expandsize                     -                              -
rpool  freeing                        0                              -
rpool  fragmentation                  -                              -
rpool  leaked                         0                              -
rpool  multihost                      off                            default
rpool  feature@async_destroy          disabled                       local
rpool  feature@empty_bpobj            disabled                       local
rpool  feature@lz4_compress           disabled                       local
rpool  feature@multi_vdev_crash_dump  disabled                       local
rpool  feature@spacemap_histogram     disabled                       local
rpool  feature@enabled_txg            disabled                       local
rpool  feature@hole_birth             disabled                       local
rpool  feature@extensible_dataset     disabled                       local
rpool  feature@embedded_data          disabled                       local
rpool  feature@bookmarks              disabled                       local
rpool  feature@filesystem_limits      disabled                       local
```

```
rpool  feature@large_blocks           disabled                       local
rpool  feature@large_dnode            disabled                       local
rpool  feature@sha512                 disabled                       local
rpool  feature@skein                  disabled                       local
rpool  feature@edonr                  disabled                       local
rpool  feature@userobj_accounting     disabled                       local
```

- Enable all features this pool knows about (list depends on both ZFS module versions which created the pool and which are running now):

```
#recovery# zpool get all | grep feature@ | awk '{print $2}' | \
    while read F ; do zpool set $F=enabled rpool ; done
```

On the original VM, stop any automatic snapshot services like ZnapZend or `zfs-auto-snapshot`, and manually snapshot all datasets recursively so that whole data trees can be easily sent over (note that we then remove some snaps like for `swap`/`dump` areas which otherwise waste a lot of space over time with blocks of obsolete swap data held by the pool for possible dataset rollback):

```
#origin# zfs snapshot -r rpool@20231210-01
#origin# zfs destroy rpool/swap@20231210-01&
#origin# zfs destroy rpool/dump@20231210-01&
```

On the receiving VM, move existing cloudy `rpool/ROOT` out of the way, if we would not use it anyway, so the new one from the original VM can land (for kicks, we can `zfs rename` the cloud-image's boot environment back into the fold after replication is complete). Also prepare to maximally compress the received root filesystem data, so it does not occupy too much in the new home (this is not something we write too often, so slower `gzip-9` writes can be tolerated):

```
#recovery# zfs rename rpool/ROOT{,x} ; \
    while ! zfs set compression=gzip-9 rpool/ROOT ; do sleep 0.2 || break ; done
```

Send over the data (from the prepared `screen` session on the origin server); first make sure all options are correct while using a dry-run mode, e.g.:

```
### Do not let other work of the origin server preempt the replication
#origin# renice -n -20 $$

#origin# zfs send -Lce -R rpool/ROOT@20231210-01 | mbuffer | \
    ssh root@recovery "mbuffer | zfs recv -vFnd rpool"
```

- Then remove `-n` from `zfs recv` after initial experiments confirm it would receive what you want and where you want it, and re-run.

With sufficiently large machines and slow source hosting, expect some hours for the transfer.

- I saw 4-8Mb/s in the streaming phase for large increments, and quite a bit of quiet time during enumeration of even almost-empty regular snapshots made by ZnapZend — low-level work with ZFS metadata has a cost.

Note that one of the benefits of ZFS (and the non-automatic snapshots used here) is that it is easy to catch-up later to send the data which the original server would generate and write **during** the replication. You can keep it actually working until the last minutes of the migration.

After the large initial transfers complete, follow-up with a pass to stop the original services (e.g. whole `zones` either from OS default grouping or as wrapped by https://github.com/jimklimov/illumos-smf-zones scripting) and replicate any new information created on origin server during this transfer (and/or human outage for the time it would take you to focus on this task again, after the computers were busy for many hours...)

Note
The original VM had ZnapZend managing regular ZFS snapshots and their off-site backups. As the old machine would no longer be doing anything of consequence, keep the service there disable and also turn off the tunnel to off-site backup — this serves to not confuse your remote systems as an admin. The new VM clone would just resume the same snapshot history, poured to the same off-site backup target.

- `rsync` the `rpool/boot/` from old machine to new, which is a directory right in the `rpool` dataset and has boot-loader configs; update `menu.lst` for GRUB boot-loader settings;
- run `zpool set bootfs=...` to enable the transplanted root file system;
- `touch reconfigure` in the new rootfs (to pick up changed hardware on boot);
- be ready to fiddle with `/etc/dladm/datalink.conf` (if using virtual links, etherstubs, etc.), as well as `/etc/hostname*`, `/etc/defaultrouter` etc.
- revise the loader settings regarding the console to use (should be `text` first here on DigitalOcean) — see in `/boot/solaris/boote` and/or `/boot/defaults/loader.conf`
- reboot into production mode to see if it all actually "works" :)

If the new VM does boot correctly, log into it and:

- Revive the `znapzend` retention schedules: they have a configuration source value of `received` in ZFS properties of the replica, so are ignored by the tool. See `znapzendzetup list` on the original machine to get a list of datasets to check on the replica, e.g.:

```
:; zfs get -s received all rpool/{ROOT,export,export/home/abuild/.ccache,zones{,-nosnap}} ↩
    \
    | grep znapzend | while read P K V S ; do zfs set $K="$V" $P & done
```

- re-enable `znapzend` and `zones` SMF services on the new VM;
- check about `cloud-init` integration services; the `metadata-agent` seems buildable and installable, it logged the SSH keys on console after service manifest import (details elaborated in links above).

DO-NUT-CI-OI VM preparation as build agent

As of this writing, the NUT CI Jenkins controller runs on DigitalOcean — and feels a lot snappier in browsing and SSH management than the older Fosshost.org VMs. Despite the official demise of the platform, they were alive and used as build agents for the newly re-hosted Jenkins controller for over a year until somebody or something put them to rest: the container with the old production Jenkins controller was set to not-auto-booting, and container with worker was attached to the new controller.

The Jenkins SSH Build Agent setups involved here were copied on the controller (as XML files) and then updated to tap into the different "host" and "port" (so that the original definitions can in time be used for replicas on DO), and due to trust settings — the `~jenkins/.ssh/known_hosts` file on the new controller had to be updated with the "new" remote system fingerprints. Otherwise, the migration went smooth.

Similarly, existing Jenkins swarm agents from community PCs had to be taught the new DNS name (some had it in `/etc/hosts`), but otherwise connected OK.

5.2.4 OmniOS

Helpful links for this part of the quest:

- https://omnios.org/download ⇒ see https://downloads.omnios.org/media/lts/omnios-r151046.cloud.vmdk (LTS) or https://downloads.o media/stable/omnios-r151048.cloud.vmdk (recent stable) or daily "bloody" images like https://downloads.omnios.org/media/-bloody/omnios-bloody-20231209.cloud.vmdk

DO-NUT-CI-OO VM preparation

Added replicas of more existing VMs: OmniOS (relatively straightforward with the OI image).

The original OmniOS VM used ZFS, so its contents were sent-received similarly to the OI VM explained above.

5.2.5 FreeBSD

Helpful links for this part of the quest:

- https://www.adminbyaccident.com/freebsd/how-to-upload-a-freebsd-custom-image-on-digitalocean/

DO-NUT-CI-FREEBSD VM preparation

Added replicas of more existing VMs: FreeBSD 12 (needed to use a seed image, tried an OpenIndiana image first but did not cut it — the ZFS options in its `rpool` were too new, so the older build of the BSD loader was not too eager to find the pool).

The original FreeBSD VM used ZFS, so its contents were sent-received similarly to the OI VM explained above.

- The (older version of?) FreeBSD loader rejected a `gzip-9` compressed `zroot/ROOT` location, so care had to be taken to first disable compression (only on the original system's tree of root filesystem datasets). The last applied ZFS properties are used for the replication stream.

5.2.6 OpenBSD

Helpful links for this part of the quest:

- https://dev.to/nabbisen/custom-openbsd-droplet-on-digitalocean-4a9o — how to piggyback OpenBSD via FreeBSD images (no longer offered by default on DO)

DO-NUT-CI-OPENBSD VM creation

Added a replica of OpenBSD 6.5 VM as an example of relatively dated system in the CI farm, which went decently well as a `dd` stream of the local VM's vHDD into DO recovery console session:

```
#tgt-recovery# mbuffer -4 -I 12340 > /dev/vda

#src# dd if=/dev/rsd0c | time nc myHostingIP 12340
```

... followed by a reboot and subsequent adaptation of `/etc/myname` and `/etc/hostname.vio*` files.

I did not check if the DigitalOcean recovery image can directly mount BSD UFS partitions, as it sufficed to log into the pre-configured system.

One caveat was that it was originally installed with X11, but DigitalOcean web-console did not pass through the mouse nor advanced keyboard shortcuts. So `rcctl disable xenodm` (to reduce the attack surface and resource waste).

FWIW, `openbsd-7.3-2023-04-22.qcow2` "custom image" did not seem to boot. At least, no activity on display and the IP address did not go up.

5.2.7 Linux

Helpful links for this part of the quest:

- https://openzfs.github.io/openzfs-docs/Getting%20Started/Debian/Debian%20Bookworm%20Root%20on%20ZFS.html — first steps for moving our older Linux VM onto ZFS root

DO-NUT-CI-LINUX VM creation

Spinning up the Debian-based Linux builder (with many containers for various Linux systems) with ZFS, to be consistent across the board, was an adventure.

- DigitalOcean rescue CD is Ubuntu 18.04 based, it has an older ZFS version so instructions from https://openzfs.github.io/-openzfs-docs/Getting%20Started/Debian/Debian%20Stretch%20Root%20on%20ZFS.html have to be used particularly to `zpool create bpool` (with the dumbed-down options for GRUB to be able to read that boot-pool);
- For the rest of the system, https://openzfs.github.io/openzfs-docs/Getting%20Started/Debian/Debian%20Bookworm%20Root%20on% is relevant for current distro (Debian 12) and is well-written;
- Note that while in many portions the "MBR or (U)EFI" boot is a choice of either one command to copy-paste or another, the spot about installing GRUB actually requires both (MBR for disk to be generally bootable, and EFI to proceed with that implementation);
- If the (recovery) console with the final OS is too "tall" in the Web-UI, so the lower rows are hidden by the DO banner with IP address, and you can't see the commands you are typing, try `clear ; stty size` to check the current display size (was 128x48 for me) and `stty rows 45` to reduce it a bit. Running a full-screen program like `mc` helps gauge if you got it right.

DO-NUT-CI-LINUX VM OS transfer

After the root pool was prepared and the large tree of datasets defined to handle the numerous LXC containers, `abuild` home directory, and other important locations of the original system, `rsync -avPHK` worked well to transfer the data.

DO-NUT-CI-LINUX VM preparation as build agent

Numerous containers with an array of Linux distributions are used as either Jenkins SSH build agents or swarm agents, as documented in chapters about LXC containers.

5.3 Custom NUT CI farm build agents: LXC multi-arch containers

This section details configuration of LXC containers as build environments for NUT CI farm; this approach can also be used on developer workstations.

5.4 Setting up the multi-arch Linux LXC container farm for NUT CI

Due to some historical reasons including earlier personal experience, the Linux container setup implemented as described below was done with persistent LXC containers wrapped by LIBVIRT for management. There was no particular use-case for systems like Docker (and no firepower for a Kubernetes cluster) in that the build environment intended for testing non-regression against a certain release does not need to be regularly updated — its purpose is to be stale and represent what users still running that system for whatever reason (e.g. embedded, IoT, corporate) have in their environments.

5.4.1 Common preparations

- Example list of packages for Debian-based systems may include (not necessarily is limited to):

```
:; apt install lxc lxcfs lxc-templates \
    ipxe-qemu qemu-kvm qemu-system-common qemu-system-data \
    qemu-system-sparc qemu-system-x86 qemu-user-static qemu-utils \
    virt-manager virt-viewer virtinst ovmf \
    libvirt-daemon-system-systemd libvirt-daemon-system \
    libvirt-daemon-driver-lxc libvirt-daemon-driver-qemu \
    libvirt-daemon-config-network libvirt-daemon-config-nwfilter \
    libvirt-daemon libvirt-clients
```

```
# TODO: Where to find virt-top - present in some but not all releases?
# Can fetch sources from https://packages.debian.org/sid/virt-top and follow
# https://www.linuxfordevices.com/tutorials/debian/build-packages-from-source
# Be sure to use 1.0.x versions, since 1.1.x uses a "better-optimized API"
# which is not implemented by libvirt/LXC backend.
```

Note
This claims a footprint of over a gigabyte of new packages when unpacked and installed to a minimally prepared OS. Much of that would be the graphical environment dependencies required by several engines and tools.

- Prepare LXC and LIBVIRT-LXC integration, including an "independent" (aka "masqueraded) bridge for NAT, following https://wiki.debian.org/LXC and https://wiki.debian.org/LXC/SimpleBridge
  - For dnsmasq integration on the independent bridge (`lxcbr0` following the documentation examples), be sure to mention:
    * `LXC_DHCP_CONFILE="/etc/lxc/dnsmasq.conf"` in `/etc/default/lxc-net`
    * `dhcp-hostsfile=/etc/lxc/dnsmasq-hosts.conf` in/as the content of `/etc/lxc/dnsmasq.conf`
    * `touch /etc/lxc/dnsmasq-hosts.conf` which would list simple `name,IP` pairs, one per line (so one per container)
    * `systemctl restart lxc-net` to apply config (is this needed after setup of containers too, to apply new items before booting them?)
    * For troubleshooting, see `/var/lib/misc/dnsmasq.lxcbr0.leases` (in some cases you may have to rename it away and reboot host to fix IP address delegation)
- Install qemu with its `/usr/bin/qemu-*-static` and registration in `/var/lib/binfmt`
- Prepare an LVM partition (or preferably some other tech like ZFS) as `/srv/libvirt` and create a `/srv/libvirt/rootfs` to hold the containers
- Prepare `/home/abuild` on the host system (preferably in ZFS with lightweight compression like lz4 — and optionally, only if the amount of available system RAM permits, with deduplication; otherwise avoid it); account user and group ID numbers are `399` as on the rest of the CI farm (historically, inherited from OBS workers)
  - It may help to generate an ssh key without a passphrase for `abuild` that it would trust, to sub-login from CI agent sessions into the container. Then again, it may be not required if CI logs into the host by SSH using `authorized_keys` and an SSH Agent, and the inner ssh client would forward that auth channel to the original agent.

```
abuild$ ssh-keygen
# accept defaults

abuild$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
abuild$ chmod 640 ~/.ssh/authorized_keys
```

- Edit the root (or whoever manages libvirt) `~/.profile` to default the virsh provider with:

```
LIBVIRT_DEFAULT_URI=lxc:///system
export LIBVIRT_DEFAULT_URI
```

- If host root filesystem is small, relocate the LXC download cache to the (larger) `/srv/libvirt` partition:

```
:; mkdir -p /srv/libvirt/cache-lxc
:; rm -rf /var/cache/lxc
:; ln -sfr /srv/libvirt/cache-lxc /var/cache/lxc
```

  - Maybe similarly relocate shared `/home/abuild` to reduce strain on rootfs?

5.4.2 Setup a container

Note that completeness of qemu CPU emulation varies, so not all distros can be installed, e.g. "s390x" failed for both debian10 and debian11 to set up the `openssh-server` package, or once even to run `/bin/true` (seems to have installed an older release though, to match the outdated emulation?)

While the `lxc-create` tool does not really specify the error cause and deletes the directories after failure, it shows the pathname where it writes the log (also deleted). Before re-trying the container creation, this file can be watched with e.g. `tail -F /var/cache/lxc/.../debootstrap.log`

Note
You can find the list of LXC "template" definitions on your system by looking at the contents of the `/usr/share/lxc/templates/` directory, e.g. a script named `lxc-debian` for the "debian" template. You can see further options for each "template" by invoking its help action, e.g.:

```
:; lxc-create -t debian -h
```

Initial container installation (for various guest OSes)

- Install containers like this:

```
:; lxc-create -P /srv/libvirt/rootfs \
    -n jenkins-debian11-mips64el -t debian -- \
    -r bullseye -a mips64el
```

  - to specify a particular mirror (not everyone hosts everything — so if you get something like `"E: Invalid Release file, no entry for main/binary-mips/Packages"` then see https://www.debian.org/mirror/list for details, and double-check the chosen site to verify if the distro version of choice is hosted with your arch of choice):

```
:; MIRROR="http://ftp.br.debian.org/debian/" \
   lxc-create -P /srv/libvirt/rootfs \
    -n jenkins-debian10-mips -t debian -- \
    -r buster -a mips
```

  - ...or for EOLed distros, use the Debian Archive server.
    * Install the container with Debian Archive as the mirror like this:

```
:; MIRROR="http://archive.debian.org/debian-archive/debian/" \
   lxc-create -P /srv/libvirt/rootfs \
    -n jenkins-debian8-s390x -t debian -- \
    -r jessie -a s390x
```

    * Note you may have to add trust to their (now expired) GPG keys for packaging to verify signatures made at the time the key was valid, by un-symlinking (if appropriate) the debootstrap script such as `/usr/share/debootstrap/scripts/jessie` commenting away the `keyring /usr/share/keyrings/debian-archive-keyring.gpg` line and setting `keyring /usr/share/keyrings/debian-archive-removed-keys.gpg` instead. You may further have to edit `/usr/share/debootstrap/functions` and/or `/usr/share/lxc/templates/lxc-debian` to honor that setting (the `releasekeyring` was hard-coded in version I had installed), e.g. in the latter file ensure such logic as below, and re-run the installation:

```
...
    # If debian-archive-keyring isn't installed, fetch GPG keys directly
    releasekeyring="`grep -E '^keyring ' "/usr/share/debootstrap/scripts/$release" |  ↩
        sed -e 's,^keyring ,,' -e 's,[ #].*$,,'`" 2>/dev/null
    if [ -z $releasekeyring ]; then
        releasekeyring=/usr/share/keyrings/debian-archive-keyring.gpg
    fi
    if [ ! -f $releasekeyring ]; then
...
```

- ... Alternatively, other distributions can be used (as supported by your LXC scripts, typically in `/usr/share/debootstrap/sc` e.g. Ubuntu:

```
:; lxc-create -P /srv/libvirt/rootfs \
    -n jenkins-ubuntu1804-s390x -t ubuntu -- \
    -r bionic -a s390x
```

- For distributions with a different packaging mechanism from that on the LXC host system, you may need to install corresponding tools (e.g. `yum4`, `rpm` and `dnf` on Debian hosts for installing CentOS and related guests). You may also need to pepper with symlinks to taste (e.g. `yum => yum4`), or find a `pacman` build to install Arch Linux or derivative, etc. Otherwise, you risk seeing something like this:

```
root@debian:~# lxc-create -P /srv/libvirt/rootfs \
    -n jenkins-centos7-x86-64 -t centos -- \
    -R 7 -a x86_64

Host CPE ID from /etc/os-release:
'yum' command is missing
lxc-create: jenkins-centos7-x86-64: lxccontainer.c:
  create_run_template: 1616 Failed to create container from template
lxc-create: jenkins-centos7-x86-64: tools/lxc_create.c:
  main: 319 Failed to create container jenkins-centos7-x86-64
```

  Note also that with such "third-party" distributions you may face other issues; for example, the CentOS helper did not generate some fields in the `config` file that were needed for conversion into libvirt "domxml" (as found by trial and error, and comparison to other `config` files):

```
lxc.uts.name = jenkins-centos7-x86-64
lxc.arch = x86_64
```

  Also note the container/system naming without underscore in "x86_64" — the deployed system discards the character when assigning its hostname. Using "amd64" is another reasonable choice here.

- For Arch Linux you would need `pacman` tools on the host system, so see https://wiki.archlinux.org/title/Install_Arch_Linux_from_e for details. On a Debian/Ubuntu host, assumed ready for NUT builds per Prerequisites for building NUT on different OSes (or `docs/config-prereqs.txt` in NUT sources for up-to-date information), it would start like this:

```
:; apt-get update
:; apt-get install meson ninja-build cmake

# Some dependencies for pacman itself; note there are several libcurl builds;
# pick another if your system constraints require you to:
:; apt-get install libarchive-dev libcurl4-nss-dev gpg libgpgme-dev

:; git clone https://gitlab.archlinux.org/pacman/pacman.git
:; cd pacman

# Iterate something like this until all needed dependencies fall
# into line (note libdir for your host architecture):
:; rm -rf build; mkdir build && meson build --libdir=/usr/lib/x86_64-linux-gnu

:; ninja -C build
# Depending on your asciidoc version, it may require that `--asciidoc-opts` are
# passed as equation to a vale (not space-separated from it). Then apply this:
#   diff --git a/doc/meson.build b/doc/meson.build
#   -      '--asciidoc-opts', ' '.join(asciidoc_opts),
#   +      '--asciidoc-opts='+' '.join(asciidoc_opts),
# and re-run (meson and) ninja.

# Finally when all succeeded:
:; sudo ninja -C build install
:; cd
```

You will also need `pacstrap` and Debian `arch-install-scripts` package does not deliver it. It is however simply achieved:

```
:; git clone https://github.com/archlinux/arch-install-scripts
:; cd arch-install-scripts
:; make && sudo make PREFIX=/usr install
:; cd
```

It will also want an `/etc/pacman.d/mirrorlist` which you can populate for your geographic location from https://archlinux.org/mirrorlist/ service, or just fetch them all (don't forget to uncomment some `Server =` lines):

```
:; mkdir -p /etc/pacman.d/
:; curl https://archlinux.org/mirrorlist/all/ > /etc/pacman.d/mirrorlist
```

And to reference it from your host `/etc/pacman.conf` by un-commenting the `[core]` section and `Include` instruction, as well as adding `[community]` and `[extra]` sections with same reference, e.g.:

```
[core]
### SigLevel = Never
SigLevel = PackageRequired
Include = /etc/pacman.d/mirrorlist

[extra]
### SigLevel = Never
SigLevel = PackageRequired
Include = /etc/pacman.d/mirrorlist

[community]
### SigLevel = Never
SigLevel = PackageRequired
Include = /etc/pacman.d/mirrorlist
```

And just then you can proceed with LXC:

```
:; lxc-create -P /srv/libvirt/rootfs \
    -n jenkins-archlinux-amd64 -t archlinux -- \
    -a x86_64 -P openssh,sudo
```

In my case, it had problems with GPG keyring missing (using one in host system, as well as the package cache outside the container, it seems) so I had to run `pacman-key --init; pacman-key --refresh-keys` on the host itself. Even so, `lxc-create` complained about updating some keyring entries and I had to go one by one picking key servers (serving different metadata) like this:

```
:; pacman-key --keyserver keyserver.ubuntu.com --recv-key 6D1655C14CE1C13E
```

In the worst case, see `SigLevel = Never` for `pacman.conf` to not check package integrity (seems too tied into thinking that host OS is Arch)...

It seems that pre-fetching the package databases with `pacman -Sy` on the host was also important.

Initial container-related setup

- Add the "name,IP" line for this container to `/etc/lxc/dnsmasq-hosts.conf` on the host, e.g.:

```
jenkins-debian11-mips,10.0.3.245
```

> **Note**
> Don't forget to eventually `systemctl restart lxc-net` to apply the new host reservation!

- Convert a pure LXC container to be managed by LIBVIRT-LXC (and edit config markup on the fly — e.g. fix the LXC `dir:/` URL schema):

```
:; virsh -c lxc:///system domxml-from-native lxc-tools \
    /srv/libvirt/rootfs/jenkins-debian11-armhf/config \
    | sed -e 's,dir:/srv,/srv,' \
    > /tmp/x && virsh define /tmp/x
```

Note
You may want to tune the default generic 64MB RAM allocation, so your launched QEMU containers are not OOM-killed as they exceeded their memory `cgroup` limit. In practice they do not eat **that much** resident memory, just want to have it addressable by VMM, I guess (swap is not very used either), at least not until active builds start (and then it depends on compiler appetite and `make` program parallelism level you allow, e.g. by pre-exporting `MAXPARMAKES` environment variable for `ci_build.sh`, and on the number of Jenkins "executors" assigned to the build agent).

- It may be needed to revert the generated "os/arch" to `x86_64` (and let QEMU handle the rest) in the `/tmp/x` file, and re-try the definition:

```
:; virsh define /tmp/x
```

- Then execute `virsh edit jenkins-debian11-armhf` (and same for other containers) to bind-mount the common `/home/abuild` location, adding this tag to their "devices":

```
    <filesystem type='mount' accessmode='passthrough'>
      <source dir='/home/abuild'/>
      <target dir='/home/abuild'/>
    </filesystem>
```

- Note that generated XML might not conform to current LXC schema, so it fails validation during save; this can be bypassed with `i` when it asks. One such case was however with indeed invalid contents, the "dir:" schema removed by example above.

5.4.3 Shepherd the herd

- Monitor deployed container rootfs'es with:

```
:; du -ks /srv/libvirt/rootfs/*
```

(should have non-trivial size for deployments without fatal infant errors)

- Mass-edit/review libvirt configurations with:

```
:; virsh list --all | awk '{print $2}' \
   | grep jenkins | while read X ; do \
     virsh edit --skip-validate $X ; done
```

- ... or avoid `--skip-validate` when markup is initially good :)

- Mass-define network interfaces:

```
:; virsh list --all | awk '{print $2}' \
   | grep jenkins | while read X ; do \
     virsh dumpxml "$X" | grep "bridge='lxcbr0'" \
     || virsh attach-interface --domain "$X" --config \
        --type bridge --source lxcbr0 ; \
   done
```

- Verify that unique MAC addresses were defined (e.g. `00:16:3e:00:00:01` tends to pop up often, while `52:54:00:xx:xx:xx` are assigned to other containers); edit the domain definitions to randomize, if needed:

```
:; grep 'mac add' /etc/libvirt/lxc/*.xml | awk '{print $NF" "$1}' | sort
```

- Make sure at least one console device exists (end of file, under the network interface definition tags), e.g.:

```
    <console type='pty'>
      <target type='lxc' port='0'/>
    </console>
```

- Populate with `abuild` account, as well as with the `bash` shell and `sudo` ability, reporting of assigned IP addresses on the console, and SSH server access complete with envvar passing from CI clients by virtue of `ssh -o SendEnv='*' container-name`:

```
:; for ALTROOT in /srv/libvirt/rootfs/*/rootfs/ ; do \
    echo "=== $ALTROOT :" >&2; \
    grep eth0 "$ALTROOT/etc/issue" || ( printf '%s %s\n' \
        '\S{NAME} \S{VERSION_ID} \n \l@\b ;' \
        'Current IP(s): \4{eth0} \4{eth1} \4{eth2} \4{eth3}' \
        >> "$ALTROOT/etc/issue" ) ; \
    grep eth0 "$ALTROOT/etc/issue.net" || ( printf '%s %s\n' \
        '\S{NAME} \S{VERSION_ID} \n \l@\b ;' \
        'Current IP(s): \4{eth0} \4{eth1} \4{eth2} \4{eth3}' \
        >> "$ALTROOT/etc/issue.net" ) ; \
    groupadd -R "$ALTROOT" -g 399 abuild ; \
    useradd -R "$ALTROOT" -u 399 -g abuild -M -N -s /bin/bash abuild \
    || useradd -R "$ALTROOT" -u 399 -g 399 -M -N -s /bin/bash abuild \
    || { if ! grep -w abuild "$ALTROOT/etc/passwd" ; then \
            echo 'abuild:x:399:399::/home/abuild:/bin/bash' \
            >> "$ALTROOT/etc/passwd" ; \
            echo "USERADDed manually: passwd" >&2 ; \
         fi ; \
         if ! grep -w abuild "$ALTROOT/etc/shadow" ; then \
            echo 'abuild:!:18889:0:99999:7:::' >> "$ALTROOT/etc/shadow" ; \
            echo "USERADDed manually: shadow" >&2 ; \
         fi ; \
       } ; \
    if [ -s "$ALTROOT/etc/ssh/sshd_config" ]; then \
        grep 'AcceptEnv \*' "$ALTROOT/etc/ssh/sshd_config" || ( \
            ( echo "" ; \
              echo "# For CI: Allow passing any envvars:"; \
              echo 'AcceptEnv *' ) \
            >> "$ALTROOT/etc/ssh/sshd_config" \
        ) ; \
    fi ; \
   done
```

  Note that for some reason, in some of those other-arch distros `useradd` fails to find the group anyway; then we have to "manually" add them.

- Let the host know and resolve the names/IPs of containers you assigned:

```
:; grep -v '#' /etc/lxc/dnsmasq-hosts.conf \
   | while IFS=, read N I ; do \
    getent hosts "$N" >&2 || echo "$I $N" ; \
   done >> /etc/hosts
```

5.4.4 Further setup of the containers

See Prerequisites for building NUT on different OSes (or `docs/config-prereqs.txt` in NUT sources for up-to-date information) about dependency package installation for Debian-based Linux systems.

It may be wise to not install e.g. documentation generation tools (or at least not the full set for HTML/PDF generation) in each environment, in order to conserve space and run-time stress.

Still, if there are significant version outliers (such as using an older distribution due to vCPU requirements), it can be installed fully just to ensure non-regression — e.g. that when adapting `Makefile` rule definitions or compiler arguments to modern toolkits, we do not lose the ability to build with older ones.

For this, `chroot` from the host system can be used, e.g. to improve the interactive usability for a population of Debian(-compatible) containers (and to use its networking, while the operating environment in containers may be not yet configured or still struggling to access the Internet):

```
:; for ALTROOT in /srv/libvirt/rootfs/*/rootfs/ ; do \
    echo "=== $ALTROOT :" ; \
    chroot "$ALTROOT" apt-get install \
        sudo bash vim mc p7zip p7zip-full pigz pbzip2 git \
   ; done
```

Similarly for `yum`-managed systems (CentOS and relatives), though specific package names can differ, and additional package repositories may need to be enabled first (see Prerequisites for building NUT on different OSes (or `docs/config-prereqs.txt` in NUT sources for up-to-date information) for more details such as recommended package names).

Note that technically `(sudo) chroot ...` can also be used from the CI worker account on the host system to build in the prepared filesystems without the overhead of running containers as complete operating environments with any standard services and several copies of Jenkins `agent.jar` in them.

Also note that externally-driven set-up of some packages, including the `ca-certificates` and the JDK/JRE, require that the `/proc` filesystem is usable in the chroot environment. This can be achieved with e.g.:

```
:; for ALTROOT in /srv/libvirt/rootfs/*/rootfs/ ; do \
    for D in proc ; do \
      echo "=== $ALTROOT/$D :" ; \
      mkdir -p "$ALTROOT/$D" ; \
      mount -o bind,rw "/$D" "$ALTROOT/$D" ; \
    done ; \
   done
```

TODO: Test and document a working NAT and firewall setup for this, to allow SSH access to the containers via dedicated TCP ports exposed on the host.

Arch Linux containers

Arch Linux containers prepared by procedure above include only a minimal footprint, and if you missed the `-P pkg,list` argument, they can lack even an SSH server. Suggestions below assume this path to container:

```
:; ALTROOT=/srv/libvirt/rootfs/jenkins-archlinux-amd64/rootfs/
```

Let `pacman` know current package database:

```
:; grep 8.8.8.8 $ALTROOT/etc/resolv.conf || (echo 'nameserver 8.8.8.8' > $ALTROOT/etc/ ↩
    resolv.conf)
:; chroot $ALTROOT pacman -Syu
:; chroot $ALTROOT pacman -S openssh sudo
:; chroot $ALTROOT systemctl enable sshd
:; chroot $ALTROOT systemctl start sshd
```

This may require that you perform bind-mounts above, as well as "passthrough" the `/var/cache/pacman/pkg` from host to guest environment (in `virsh edit`, and bind-mount for `chroot` like for `/proc` et al above).

It is possible that `virsh console` would serve you better than `chroot`. Note you may have to first `chroot` to set the `root` password anyhow.

5.4.5 Troubleshooting

- Q: Container won't start, its `virsh console` says something like:

```
Failed to create symlink /sys/fs/cgroup/net_cls: Operation not permitted
```

  A: According to https://bugzilla.redhat.com/show_bug.cgi?id=1770763 (skip to the end for summary) this can happen when a newer Linux host system with `cgroupsv2` capabilities runs an older guest distro which only knows about `cgroupsv1`, such as when hosting a CentOS 7 container on a Debian 11 server.

  – One workaround is to ensure that the guest `systemd` does not try to "join" host facilities, by setting an explicit empty list for that:

```
:; echo 'JoinControllers=' >> "$ALTROOT/etc/systemd/system.conf"
```

  – Another approach is to upgrade `systemd` related packages in the guest container. This may require additional "backport" repositories or similar means, possibly maintained not by distribution itself but by other community members, and arguably would logically compromise the idea of non-regression builds in the old environment "as is".

- Q: Server was set up with ZFS as recommended, and lots of I/O hit the disk even when application writes are negligible

  A: This was seen on some servers and generally derives from data layout and how ZFS maintains the tree structure of blocks. A small application write (such as a new log line) means a new empty data block allocation, an old block release, and bubble up through the whole metadata tree to complete the transaction (grouped as TXG to flush to disk).

  – One solution is to use discardable build workspaces in RAM-backed storage like `/dev/shm` (`tmpfs`) on Linux, or `/tmp` (`swap`) on illumos hosting systems, and only use persistent storage for the home directory with `.ccache` and `.gitcache-dynamatrix` directories.

  – Another solution is to reduce the frequency of TXG sync from modern default of 5 sec to conservative 30-60 sec. Check how to set the `zfs_txg_timeout` on your platform.

5.5 Connecting Jenkins to the containers

To properly cooperate with the jenkins-dynamatrix project driving regular NUT CI builds, each build environment should be exposed as an individual agent with labels describing its capabilities.

5.5.1 Agent Labels

With the `jenkins-dynamatrix`, agent labels are used to calculate a large "slow build" matrix to cover numerous scenarios for what can be tested with the current population of the CI farm, across operating systems, `make`, shell and compiler implementations and versions, and C/C++ language revisions, to name a few common "axes" involved.

Labels for QEMU

Emulated-CPU container builds are CPU-intensive, so for them we define as few capabilities as possible: here CI is more interested in checking how binaries behave on those CPUs, **not** in checking the quality of recipes (distcheck, Make implementations, etc.), shell scripts or documentation, which is more efficient to test on native platforms.

Still, we are interested in results from different compiler suites, so specify at least one version of each.

Note
Currently the NUT `Jenkinsfile-dynamatrix` only looks at various `COMPILER` variants for `qemu-nut-builder` use-cases, disregarding the versions and just using one that the environment defaults to.

The reduced set of labels for QEMU workers looks like:

```
qemu-nut-builder qemu-nut-builder:alldrv
NUT_BUILD_CAPS=drivers:all NUT_BUILD_CAPS=cppunit
OS_FAMILY=linux OS_DISTRO=debian11 GCCVER=10 CLANGVER=11
COMPILER=GCC COMPILER=CLANG
ARCH64=ppc64le ARCH_BITS=64
```

Labels for native builds

For contrast, a "real" build agent's set of labels, depending on presence or known lack of some capabilities, looks something like this:

```
doc-builder nut-builder nut-builder:alldrv
NUT_BUILD_CAPS=docs:man NUT_BUILD_CAPS=docs:all
NUT_BUILD_CAPS=drivers:all NUT_BUILD_CAPS=cppunit=no
OS_FAMILY=bsd OS_DISTRO=freebsd12 GCCVER=10 CLANGVER=10
COMPILER=GCC COMPILER=CLANG
ARCH64=amd64 ARCH_BITS=64
SHELL_PROGS=sh SHELL_PROGS=dash SHELL_PROGS=zsh SHELL_PROGS=bash
SHELL_PROGS=csh SHELL_PROGS=tcsh SHELL_PROGS=busybox
MAKE=make MAKE=gmake
PYTHON=python2.7 PYTHON=python3.8
```

5.5.2 Generic agent attributes

- Name: e.g. `ci-debian-altroot--jenkins-debian10-arm64` (note the pattern for "Conflicts With" detailed below)
- Remote root directory: preferably unique per agent, to avoid surprises; e.g.: `/home/abuild/jenkins-nut-altroots/jenki`
  - Note it may help that the system home directory itself is shared between co-located containers, so that the `.ccache` or `.gitcache-dynamatrix` are available to all builders with identical contents
  - If RAM permits, the Jenkins Agent working directory may be placed in a temporary filesystem not backed by disk (e.g. `/dev/shm` on modern Linux distributions); roughly estimate 300Mb per executor for NUT builds.
- Usage: "Only build jobs with label expressions matching this node"
- Node properties / Environment variables:
  - `PATH+LOCAL` ⇒ `/usr/lib/ccache`

5.5.3 Where to run agent.jar

Depending on circumstances of the container, there are several options available to the NUT CI farm:

- Java can run in the container, efficiently (native CPU, different distro) ⇒ the container may be exposed as a standalone host for direct SSH access (usually by NAT, exposing SSH on a dedicated port of the host; or by first connecting the Jenkins controller with the host as an SSH Build Agent, and then calling SSH to the container as a prefix for running the agent; or by using Jenkins Swarm agents), so ultimately the build `agent.jar` JVM would run in the container. Filesystem for the `abuild` account may be or not be shared with the host.
- Java can not run in the container (crashes on emulated CPU, or is too old in the agent container's distro — currently Jenkins requires JRE 17+, but eventually will require 21+) ⇒ the agent would run on the host, and then the host would `ssh` or `chroot` (networking not required, but bind-mount of `/home/abuild` and maybe other paths from host would be needed) called for executing `sh` steps in the container environment. Either way, home directory of the `abuild` account is maintained on the host and shared with the guest environment, user and group IDs should match.
- Java is inefficient in the container (operations like un-stashing the source succeed but take minutes instead of seconds) ⇒ either of the above

Note
As time moves on and Jenkins core and its plugins get updated, support for some older run-time features of the build agents can get removed (e.g. older Java releases, older Git tooling). While there are projects like Temurin that provide Java builds for older systems, at some point a switch to "Jenkins agent on new host going into older build container" approach can become unavoidable. One clue to look at in build logs is failure messages like:

```
Caused by: java.lang.UnsupportedClassVersionError:
  hudson/slaves/SlaveComputer$SlaveVersion has been compiled by a more
  recent version of the Java Runtime (class file version 61.0), this version
  of the Java Runtime only recognizes class file versions up to 55.0
```

Using Jenkins SSH Build Agents

This is a typical use-case for tightly integrated build farms under common management, where the Jenkins controller can log by SSH into systems which act as its build agents. It injects and launches the `agent.jar` to execute child processes for the builds, and maintains a tunnel to communicate.

Methods below involving SSH assume that you have configured a password-less key authentication from the host machine to the `abuild` account in each guest build environment container. This can be an `ssh-keygen` result posted into `authorized_keys`, or a trusted key passed by a chain of ssh agents from a Jenkins Credential for connection to the container-hoster into the container. The private SSH key involved may be secured by a pass-phrase, as long as your Jenkins Credential storage knows it too. Note that for the approaches explored below, the containers are not directly exposed for log-in from any external network.

- For passing the agent through an SSH connection from host to container, so that the `agent.jar` runs inside the container environment, configure:
  - Launch method: "Agents via SSH"
  - Host, Credentials, Port: as suitable for accessing the container-hoster

    Note
    The container-hoster should have accessed the guest container from the account used for intermediate access, e.g. `abuild`, so that its `.ssh/known_hosts` file would trust the SSH server on the container.

  - Prefix Start Agent Command: content depends on the container name, but generally looks like the example below to report some info about the final target platform (and make sure `java` is usable) in the agent's log. Note that it ends with un-closed quote and a space char:

    ```
    ssh jenkins-debian10-amd64 '( java -version & uname -a ; getconf LONG_BIT; getconf  ↩
        WORD_BIT; wait ) &&
    ```

  - Suffix Start Agent Command: a single quote to close the text opened above:

```
'
```

- The other option is to run the `agent.jar` on the host, for all the network and filesystem magic the agent does, and only execute shell steps in the container. The solution relies on overridden `sh` step implementation in the `jenkins-dynamatrix` shared library that uses a magic `CI_WRAP_SH` environment variable to execute a pipe into the container. Such pipes can be `ssh` or `chroot` with appropriate host setup described above.

  Note
  In case of ssh piping, remember that the container's `/etc/ssh/sshd_config` should `AcceptEnv *` and the SSH server should be restarted after such configuration change.

  - Launch method: "Agents via SSH"
  - Host, Credentials, Port: as suitable for accessing the container-hoster
  - Prefix Start Agent Command: content depends on the container name, but generally looks like the example below to report some info about the final target platform (and make sure it is accessible) in the agent's log. Note that it ends with a space char, and that the command here should not normally print anything into stderr/stdout (this tends to confuse the Jenkins Remoting protocol):

```
echo PING > /dev/tcp/jenkins-debian11-ppc64el/22 &&
```

  - Suffix Start Agent Command: empty

- Node properties / Environment variables:

  - `CI_WRAP_SH` ⇒

```
ssh -o SendEnv='*' "jenkins-debian11-ppc64el" /bin/sh -xe
```

Using Jenkins Swarm Agents

This approach allows remote systems to participate in the NUT CI farm by dialing in and so defining an agent. A single contributing system may be running a number of containers or virtual machines set up following the instructions above, and each of those would be a separate build agent.

Such systems should be "dedicated" to contribution in the sense that they should be up and connected for days, and sometimes tasks would land.

Configuration files maintained on the Swarm Agent system dictate which labels or how many executors it would expose, etc. Credentials to access the NUT CI farm Jenkins controller to register as an agent should be arranged with the farm maintainers, and currently involve a GitHub account with Jenkins role assignment for such access, and a token for authentication.

The jenkins-swarm-nutci repository contains example code from such setup with a back-up server experiment for the NUT CI farm, including auto-start method scripts for Linux systemd and upstart, illumos SMF, and OpenBSD rcctl.

5.5.4 Sequentializing the stress

Running one agent at a time

Another aspect of farm management is that emulation is a slow and intensive operation, so we can not run all agents and execute builds at the same time.

The current solution relies on https://github.com/jimklimov/conflict-aware-ondemand-retention-strategy-plugin to allow co-located build agents to "conflict" with each other — when one picks up a job from the queue, it blocks neighbors from starting; when it is done, another may start.

Containers can be configured with "Availability ⇒ On demand", with shorter cycle to switch over faster (the core code sleeps a minute between attempts):

- In demand delay: `0`;
- Idle delay: `0` (Jenkins may change it to `1`);
- Conflicts with: `^ci-debian-altroot--.*$` assuming that is the pattern for agent definitions in Jenkins — not necessarily linked to hostnames.

Also, the "executors" count should be reduced to the amount of compilers in that system (usually 2) and so avoid extra stress of scheduling too many emulated-CPU builds at once.

Sequentializing the git cache access

As part of the `jenkins-dynamatrix` optional optimizations, the NUT CI recipe invoked via `Jenkinsfile-dynamatrix` maintains persistent git reference repositories that can be used to cache NUT codebase (including the tested commits) and so considerably speed up workspace preparation when running numerous build scenarios on the same agent.

Such `.gitcache-dynamatrix` cache directories are located in the build workspace location (unique for each agent), but on a system with numerous containers these names can be symlinks pointing to a shared location.

To avoid collisions with several executors updating the same cache with new commits, critical access windows are sequentialized with the use of Lockable Resources plugin. On the `jenkins-dynamatrix` side this is facilitated by labels:

```
DYNAMATRIX_UNSTASH_PREFERENCE=scm-ws:nut-ci-src
DYNAMATRIX_REFREPO_WORKSPACE_LOCKNAME=gitcache-dynamatrix:SHARED_HYPERVISOR_NAME
```

- The `DYNAMATRIX_UNSTASH_PREFERENCE` tells the `jenkins-dynamatrix` library code which checkout/unstash strategy to use on a particular build agent (following values defined in the library; `scm-ws` means SCM caching under the agent workspace location, `nut-ci-src` names the cache for this project);
- The `DYNAMATRIX_REFREPO_WORKSPACE_LOCKNAME` specifies a semi-unique string: it should be same for all co-located agents which use the same shared cache location, e.g. guests on the same hypervisor; and it should be different for unrelated cache locations, e.g. different hypervisors and stand-alone machines.

6 Prerequisites for building NUT on different OSes

This chapter aims to list packages with the tools needed on a freshly minimally deployed worker to build as many targets of NUT recipes as possible, mainly the diverse driver and documentation types.

NUT codebase generally should not depend on particular operating system or kernel technology and version, and with the operating systems listed below one can benefit from use of containers (jails, zones) to build and test against numerous OS distributions on one physical or virtual machine, e.g. to cover non-regression with older tool kits while taking advantage of new releases.

- For Linux systems, we have notes on Custom NUT CI farm build agents: LXC multi-arch containers (or `docs/ci-farm-lxc-setu` in NUT sources for up-to-date information)

Some of the below are alternatives, e.g. compiler toolkits (gcc vs. clang) or SSL implementations (OpenSSL vs. Mozilla NSS) — no problem installing both, at a disk space cost.

Note

Some NUT branches may need additional or different software versions that are not yet included into `master` branch dependencies, e.g. the DMF (Dynamic Mapping Files) sub-project needs LUA 5.1 for build and run-time, and some Python modules for build, e.g. using OS packaging or custom call to `pip install pycparser`.

In case your system still provides a Python 2.x environment (and for some reason you want to use it instead of Python 3.x), but does not anymore provide a `pip` nor `pycparser` packages for it, you may need to use an external bootstrap first, e.g.:

```
# Fetch get-pip.py for python 2.7
:; curl https://bootstrap.pypa.io/pip/2.7/get-pip.py --output get-pip.py
:; python2 get-pip.py
:; python2 -m pip --version
:; python2 -m pip install pycparser
```

More packages and/or system setup may be needed to actually run NUT with all features enabled; chapters below concern just with building it.

6.1 General call to Test the ability to configure and build

Check out from git, generate files and configure to tailor to your build environment, and build some tests:

```
:; mkdir -p nut && cd nut && \
    git clone https://github.com/networkupstools/nut/ -b master .
:; ./autogen.sh && \
    ./configure --with-doc=all --with-all --with-cgi && \
    make all && make check && make spellcheck
```

You can toggle some `configure` options to check different dependency variants, e.g. `--with-ssl=nss` vs. `--with-ssl=opens`

For reproducible runs of various pre-sets of configuration during development, take a look at `ci_build.sh` script and different `BUILD_TYPE` (and other) environment variable settings that it supports. A minimal run with it is just to call the script, e.g.:

```
:; mkdir -p nut && cd nut && \
   git clone https://github.com/networkupstools/nut/ -b fightwarn .
:; ./ci_build.sh
```

Note
To build older releases, such as "vanilla" NUT 2.7.4 and older, you may need to address some nuances:

- Ensure that `python` in `PATH` points to a python-2.x implementation (`master` branch is fixed to work with python 2 and 3)
- Ensure that `bash` is your user and maybe system shell (or ensure the generated `configure` script gets interpreted by it)
- Generally you may have better results with GNU Make newer than 3.81 than with other make implementations; however, builds are regularly tested by CI with Sun dmake and BSD make as well, so recipes should not expect GNU-only syntax and constructs to work
- Parallel builds should be okay in current development version and since NUT 2.8.0 (is a bug to log and fix, if not), but they may be failure-prone in 2.7.4 and earlier releases

---

For intensive rebuilds, `ccache` is recommended. Note that instructions below detail how to provide its directory with symlinks as `/usr/lib/ccache` which is not the default case in all OS distributions. Recent versions of the NUT `ci_build.sh` script allow to override the location by using the `CI_CCACHE_SYMLINKDIR` environment variable, which is cumbersome and only recommended for build agents with immutable system areas, etc.

6.2 Build prerequisites to make NUT from scratch on various Operating Systems

6.2.1 Debian 10/11/12/13

Being a popular baseline among Linux distributions, Debian is an important build target. Related common operating systems include Ubuntu and customized distros for Raspberry Pi, Proxmox, as well as many others.

For some (newer) distributions, `apt` or other front-ends may be preferable to `apt-get`, but otherwise the commands are equivalent.

The package list below should largely apply to those as well, however note that some well-known package names tend to differ. A few of those are noted below.

Note
While Debian distros I've seen (8 to 11) provide a "libusb-dev" for libusb-0.1 headers, the binary library package name is specifically versioned package by default of the current release (e.g. "libusb-0.1-4"), while names of both the library and development packages for libusb-1.0 must be determined with:

```
:; apt-cache search 'libusb.*1\.0.*'
```

yielding e.g. "libusb-1.0-0-dev" (string name was seen with different actual package source versions on both Debian 8 "Jessie" and Debian 11 "Buster").

FUN NOTE

For development on the road (or a native ARM build) you can use the Termux project on Android. It provides a sufficiently Debian-like operating environment for all intents and purposes, but you may have to use their `pkg` wrapper instead of `apt` tooling directly, and `ldd` may be in a package separate from `binutils`, but otherwise the Debian/Ubuntu oriented lists of packages below apply.

You would need at least a couple of gigabytes available on the internal phone storage though, especially if using `ccache` or setting up cross builds.

The Termux distribution as of this writing seems to lack cppunit packages but you can build it from https://www.freedesktop.org/wiki/Software/cppunit/ e.g. `--with-prefix=${HOME}/nut-deps/usr` and `make install` there. A custom `PKG_CONFIG_PATH=${HOME}/nut-deps/usr/lib/pkgconfig` would find it during NUT build.

Debian-like package installations commonly start with an update of metadata about recently published package revisions:

```
:; apt-get update

:; apt-get install \
    ccache time \
    git perl curl \
    make autoconf automake libltdl-dev libtool binutils \
    valgrind \
    cppcheck \
    pkg-config \
    gcc g++ clang

# To debug eventual core dump files, or trace programs with an IDE like
# NetBeans (perhaps remotely), you may want the GNU Debugger program:
:; apt-get install \
    gdb

# NOTE: Older Debian-like distributions may lack a "libtool-bin"
:; apt-get install \
    libtool-bin

# See comments below, python version and package naming depends on distro
:; apt-get install \
    python

# NOTE: For python, you may eventually have to specify a variant like this
# (numbers depending on default or additional packages of your distro):
#   :; apt-get install python2 python2.7 python-is-python2
# and/or:
#   :; apt-get install python3 python3.9
# You can find a list of what is (pre-)installed with:
#   :; dpkg -l | grep -Ei 'perl|python'
#
# For localization maintenance (currently in Python NUT-Monitor app),
# provide an `msgfmt` implementation, e.g.:
#   :; apt-get install gettext
#
# To install the Python NUT-Monitor app, you may need some modules.
# Ideally, they would be packaged, named according to major Python version:
#   :; apt-get install python3-pyqt5
#
# If not packaged for your distro:
#   :; apt-get install pip
# For Python3:
#   :; python3 -m pip install PyQt5 configparser
```

```
# For spell-checking, highly recommended if you would propose pull requests:
:; apt-get install \
    aspell aspell-en

# For other doc types (man-page, PDF, HTML) generation - massive packages (TEX, X11):
:; apt-get install \
    asciidoc source-highlight python3-pygments dblatex

# For CGI graph generation - massive packages (X11):
:; apt-get install \
    libgd-dev

# Debian 13 serves metadata needed for NUT to find where to install the
# service unit files in a separate development package; earlier distro
# releases did not seem to require it explicitly:
:; apt-get install \
    systemd-dev

# Optionally for sd_notify integration:
:; apt-get install \
    libsystemd-dev

# NOTE: Some older Debian-like distributions, could ship "libcrypto-dev"
# and/or "openssl-dev" instead of "libssl-dev" by its modern name
# and may lack a libgpiod2 + libgpiod-dev altogether
:; apt-get install \
    libcppunit-dev \
    libssl-dev libnss3-dev \
    augeas-tools libaugeas-dev augeas-lenses \
    libusb-dev libusb-1.0-0-dev \
    libi2c-dev \
    libmodbus-dev \
    libsnmp-dev \
    libpowerman0-dev \
    libfreeipmi-dev libipmimonitoring-dev \
    libavahi-common-dev libavahi-core-dev libavahi-client-dev
# For libneon, see below

# NOTE: Older Debian-like distributions may lack a "libgpiod-dev"
# Others above are present as far back as Debian 7 at least
:; apt-get install \
    libgpiod-dev

# NOTE: Some distributions lack a lua*-dev and only offer the base package
:; apt-get install lua5.1
:; apt-get install lua5.1-dev || true

:; apt-get install \
    bash dash ksh busybox
```

Alternatives that can depend on your system's other packaging choices:

```
:; apt-get install libneon27-dev
# ... or
:; apt-get install libneon27-gnutls-dev
```

Over time, Debian and Ubuntu had different packages and libraries providing the actual methods for I2C; if your system lacks the `libi2c` (and so fails to `./configure --with-all`), try adding the following packages:

```
:; apt-get install build-essential git-core libi2c-dev i2c-tools lm-sensors
```

For cross-builds (note that not everything supports multilib approach, limiting standard package installations to one or another implementation; in that case local containers each with one ARCH may be a better choice, with `qemu-user-static` playing a role to "natively" run the other-ARCH complete environments):

```
:; apt-get install \
    gcc-multilib g++-multilib \
    crossbuild-essential \
    gcc-10:armhf gcc-10-base:armhf \
    qemu-user-static
```

Note
For Jenkins agents, also need to `apt-get install openjdk-21-jdk-headless`. You may have to ensure that `/proc` is mounted in the target chroot (or do this from the running container).

6.2.2 CentOS 6 and 7

CentOS is another popular baseline among Linux distributions, being a free derivative of the RedHat Linux, upon which many other distros are based as well. These systems typically use the RPM package manager, using directly `rpm` command, or `yum` or `dnf` front-ends depending on their generation.

For CI farm container setup, prepared root filesystem archives from http://download.proxmox.com/images/system/ worked sufficiently well.

Prepare CentOS repository mirrors

For CentOS 7 it seems that not all repositories are equally good; some of the software below is only served by EPEL (Extra Packages for Enterprise Linux), as detailed at:

- https://docs.fedoraproject.org/en-US/epel/
- https://www.redhat.com/en/blog/whats-epel-and-how-do-i-use-it
- https://pkgs.org/download/epel-release

You may have to specify a mirror as the `baseurl` in a `/etc/yum.repos.d/...` file (as the aged distributions become less served by mirrors), such as:

- https://www.mirrorservice.org/sites/dl.fedoraproject.org/pub/epel/7/x86_64/

```
# e.g. for CentOS7 currently:
:; yum install https://download-ib01.fedoraproject.org/pub/epel/7/x86_64/Packages/e/epel- ↩
    release-7-14.noarch.rpm

# And edit /etc/yum.repos.d/epel.repo to uncomment and set the baseurl=...
# lines, and comment away the mirrorlist= lines (if yum hiccups otherwise)
```

For systemd support on CentOS 7 (no equivalent found for CentOS 6), you can use backports repository below:

```
:; curl https://copr.fedorainfracloud.org/coprs/jsynacek/systemd-backports-for-centos-7
    > /etc/yum.repos.d/systemd-backports-for-centos-7.repo
```

For CentOS 6 (the oldest I could try) the situation is similar, with sites like https://www.getpagespeed.com/server-setup/how-to-fix-yum-after-centos-6-went-eol detailing how to replace `/etc/yum.repos.d/` contents (you can wholesale rename the existing directory and populate a new one with `curl` downloads from the article), and additional key trust for EPEL packages:

```
:; yum install https://dl.fedoraproject.org/pub/archive/epel/6/x86_64/epel-release-6-8. ↩
    noarch.rpm
```

Set up CentOS packages for NUT

Instructions below apply to both CentOS 6 and 7 (a few nuances for 6 commented). For newer distributions, `dnf` may be preferable to `yum`, but otherwise the commands are equivalent.

General developer system helpers mentioned in Custom NUT CI farm build agents: LXC multi-arch containers (or `docs/ci-farm-lx` in NUT sources for up-to-date information):

```
:; yum update

:; yum install \
    sudo vim mc p7zip pigz pbzip2 tar
```

To have SSH access to the build VM/Container, you may have to install and enable it:

```
:; yum install \
    openssh-server openssh-clients

:; chkconfig sshd on
:; service sshd start

# If there are errors loading generated host keys, remove mentioned files
# including the one with .pub extension and retry with:
#:; service sshd restart
```

Note
Below we request to install generic `python` per system defaults. You may request specifically `python2` or `python3` (or both): current NUT should be compatible with both (2.7+ at least).

Note
On CentOS, `libusb` means 0.1.x and `libusbx` means 1.x.x API version (latter is not available for CentOS 6).

Note
On CentOS, it seems that development against libi2c/smbus is not supported. Neither the suitable devel packages were found, nor i2c-based drivers in distro packaging of NUT. Resolution and doc PRs are welcome.

```
:; yum install \
    ccache time \
    file \
    git perl curl \
    make autoconf automake libtool-ltdl-devel libtool \
    valgrind \
    cppcheck \
    pkgconfig \
    gcc gcc-c++ clang

# To debug eventual core dump files, or trace programs with an IDE like
# NetBeans (perhaps remotely), you may want the GNU Debugger program:
:; yum install \
    gdb

# See comments below, python version and package naming depends on distro
:; yum install \
    python
```

```
# NOTE: For python, you may eventually have to specify a variant like this
# (numbers depending on default or additional packages of your distro):
#   :; yum install python-2.7.5
# and/or:
#   :; yum install python3 python3-3.6.8
# You can find a list of what is (pre-)installed with:
#   :; rpm -qa | grep -Ei 'perl|python'
# Note that CentOS 6 includes python-2.6.x and does not serve newer versions

# For spell-checking, highly recommended if you would propose pull requests:
:; yum install \
    aspell aspell-en

# For other doc types (man-page, PDF, HTML) generation - massive packages (TEX, X11):
:; yum install \
    asciidoc source-highlight python-pygments dblatex

# For CGI graph generation - massive packages (X11):
:; yum install \
    gd-devel

# Optionally for sd_notify integration (on CentOS 7+, not on 6):
:; yum install \
    systemd-devel

# NOTE: "libusbx" is the CentOS way of naming "libusb-1.0" (not in CentOS 6)
# vs. the older "libusb" as the package with "libusb-0.1"
:; yum install \
    cppunit-devel \
    openssl-devel nss-devel \
    augeas augeas-devel \
    libusb-devel libusbx-devel \
    i2c-tools \
    libmodbus-devel \
    net-snmp-devel \
    powerman-devel \
    freeipmi-devel \
    avahi-devel \
    neon-devel
#?# is python-augeas needed? exists at least...
#?# no (lib)i2c-devel ...
#?# no (lib)ipmimonitoring-devel ... would "freeipmi-ipmidetectd" cut it at least for run- ↩
    time?
#?# no (lib)gpio(d)-devel - starts with CentOS 8 (or extra repositories for later minor  ↩
    releases of CentOS 7)

# Some NUT code related to lua may be currently limited to lua-5.1
# or possibly 5.2; the former is default in CentOS 7 releases...
:; yum install \
    lua-devel

:; yum install \
    bash dash ksh
```

Note

`busybox` is not packaged for CentOS 7 release; a static binary can be downloaded if needed. For more details, see https://unix.stackexchange.com/questions/475584/cannot-install-busybox-on-centos

CentOS packaging for 64-bit systems delivers the directory for dispatching compiler symlinks as `/usr/lib64/ccache`. You

can set it up same way as for other described environments by adding a symlink `/usr/lib/ccache`:

```
:; ln -s ../lib64/ccache/ "$ALTROOT"/usr/lib/
```

Note
For Jenkins agents, also need to install JDK 17 or newer, which is not available for CentOS 6 nor 7 directly (in distribution packaging). Alternative packaging, such as Temurin from the Adoptium project, is possible (checked for at least CentOS 7), see their instructions for specific details. This may require updated library package versions as dependencies from the OS distribution, so you may also have to make sure that your `/etc/yum.repos.d/*` files (certainly `CentOS-Base.repo`, maybe also `CentOS-fasttrack.repo` and/or `CentOS-CR.repo`) to use e.g.

```
baseurl=https://vault.centos.org/centos/$releasever/os/$basearch/
```

lines if the `mirrorlist` ones do not suffice to find living mirrors (WARNING: the `/os/` part of the URL would vary for different repository types).

6.2.3 Arch Linux

Update the lower-level OS and package databases:

```
:; pacman -Syu
```

Install tools and prerequisites for NUT:

```
:; pacman -S --needed \
    base-devel \
    autoconf automake libtool libltdl \
    clang gcc \
    ccache \
    git \
    vim python perl \
    pkgconf \
    cppcheck valgrind

# To debug eventual core dump files, or trace programs with an IDE like
# NetBeans (perhaps remotely), you may want the GNU Debugger program:
:; pacman -S --needed \
    gdb

# For spell-checking, highly recommended if you would propose pull requests:
:; pacman -S --needed \
    aspell en-aspell

# For man-page doc types generation:
:; pacman -S --needed \
    asciidoc

# For other doc types (PDF, HTML) generation - massive packages (TEX, X11):
:; pacman -S --needed \
    source-highlight dblatex

# For CGI graph generation - massive packages (X11):
:; pacman -S --needed \
    gd

# Optionally for sd_notify integration:
:; pacman -S --needed \
    systemd
```

```
:; pacman -S --needed \
    cppunit \
    openssl nss \
    augeas \
    libusb \
    neon \
    net-snmp \
    freeipmi \
    avahi

#?# no (lib)gpio(d)

:; pacman -S --needed \
    lua51

:; pacman -S --needed \
    bash dash busybox ksh93
```

Recommended for NUT CI farm integration (matching specific toolkit versions in a build matrix), and note the unusual location of `/usr/lib/ccache/bin/` for symlinks:

```
:; gcc --version
gcc (GCC) 12.2.0
...

# Note: this distro delivers "gcc" et al as file names
# so symlinks like this may erode after upgrades.
# TODO: Rename and then link?..
:; (cd /usr/bin \
    && for V in 12 12.2.0 ; do for T in gcc g++ cpp ; do \
        ln -fsr $T $T-$V ; \
    done; done)
:; (cd /usr/lib/ccache/bin/ \
    && for V in 12 12.2.0 ; do for T in gcc g++ ; do \
        ln -fsr /usr/bin/ccache $T-$V ; \
    done; done)


:; clang --version
clang version 14.0.6
...

:; (cd /usr/bin && ln -fs clang-14 clang++-14 && ln -fs clang-14 clang-cpp-14)
:; (cd /usr/lib/ccache/bin/ \
    && for V in 14 ; do for T in clang clang++ ; do \
        ln -fsr /usr/bin/ccache $T-$V ; \
    done; done)
```

Also for CI build agents, a Java environment (JDK17+ since autumn 2024) is required:

```
# Search for available Java versions:
:; pacman -Ss | egrep 'jre|jdk'

# Pick one:
:; pacman -S --needed \
    jre17-openjdk-headless

# If needed to change default implementation, consult:
:; archlinux-java help
```

6.2.4 Slackware Linux 15

Another long-term presence in the Linux landscape, and sometimes the baseline for appliances, the Slackware project recently hit release 15 in 2022, averaging two years per major release.

It can be installed e.g. in a VM, using ISO images from the project site; see:

- http://www.slackware.com/ ⇒ http://www.slackware.com/getslack/ ⇒ https://mirrors.slackware.com/slackware/slackware-iso/-slackware64-15.0-iso/
- https://slackware.nl/slackware/slackware64-current-iso/

You would have to first log in as `root` and run `cgdisk` to define partitioning for your virtual HDD, such as the common `/boot`, `swap` and `/` Linux layout, and run `setup` to install "everything".

Note that "out of the box" Slackware does not currently on networked package repositories and calculated dependency trees, so one has to know exactly what they want installed.

Third-party projects for package managers are also available, e.g. `slackpkg` ⇒ see also https://docs.slackware.com/slackware:slackpkg and https://slackpkg.org/stable/ :

```
:; wget https://slackpkg.org/stable/slackpkg-15.0.10-noarch-1.txz && \
   installpkg slackpkg-15.0.10-noarch-1.txz
```

Uncomment a mirror from `/etc/slackpkg/mirrors` according to your location and other preferences (or use the top-listed default), and begin with:

```
:; slackpkg update
```

Note that packages may be only installed or re-installed/upgraded as separate explicit operations, so the procedure to bring your system into needed shape is a bit cumbersome (and each command may by default be interactive with a choice menu), e.g.:

```
:; for P in \
    bash mc vim sudo \
   ; do slackpkg info "$P" || slackpkg install $P || break ; done
```

For procedures below, this is automated via `root` profile to become a `slackpkg-install` command:

```
:; grep "slackpkg-install" ~/.profile || {
    echo 'slackpkg-install() { for P in "$@" ; do echo "=== $P:"; slackpkg info "$P" ||  ↩
        slackpkg install "$P" || break ; done; }' >> ~/.profile
   }
:; . ~/.profile
```

If something has a hiccup, it suggests to look for the right name, e.g.:

```
:; slackpkg search python
```

For NUT dependencies and build tools:

```
:; slackpkg update

# Baseline toolkits:
# Note there is no cppcheck, cppunit, valgrind...
# Note clang compiler tools are part of llvm package
:; slackpkg-install \
    ccache time \
    coreutils diffutils \
    git python3 perl curl \
    make autoconf automake libtool binutils \
    pkg-config \
    gcc llvm
```

```
# To debug eventual core dump files, or trace programs with an IDE like
# NetBeans (perhaps remotely), you may want the GNU Debugger program:
:; slackpkg-install gdb

# For spell-checking, highly recommended if you would propose pull requests:
:; slackpkg-install aspell{,-en}

# Note there is no direct "asciidoc" nor "a2x", just the competing project
# "rubygem-asciidoctor" that NUT currently has no recipes for, so you can
# not compile man/html/pdf docs here, per the default repository. See below
# for tools from alternative repositories, which seem to work well. Man page
# compilation would require docbook-xml resources; older versions are in
# this package https://slackbuilds.org/repository/15.0/system/docbook-xml/
# and recent ones are in not-installed part of main repository:
:; slackpkg-install linuxdoc-tools

# More on Python (for NUT-Monitor UI):
:; slackpkg-install \
    python-pip qt5 gettext-tools gettext

# For CGI graph generation - massive packages (X11):
:; slackpkg-install \
    gd

# General dependencies:
:; slackpkg-install \
    openssl openssl-solibs mozilla-nss \
    libusb \
    net-snmp \
    neon

# Shells:
:; slackpkg-install \
    bash dash ksh93
```

Some more packages are available on the side, including Java (useful e.g. to make this environment into a fully fledged Jenkins worker). Other common NUT dependencies absent from primary Slackware repositories can be found and downloaded (seek `*.txz` package files, although a few are named `*.tgz`) from here, and passed to `installpkg`:

- http://www.slackware.com/~alien/slackbuilds/openjdk17/
- http://www.slackware.com/~alien/slackbuilds/asciidoc/
- http://www.slackware.com/~alien/slackbuilds/cppunit/
- http://www.slackware.com/alien/slackbuilds/lua/ (5.1 in "stable") and http://www.slackware.com/alien/slackbuilds/lua53/ (5.4 in "current" as of Slackware 15.1 candidate in the works) — note that LUA is not needed for the current NUT code base, but may become needed after import of features from forks

. . . and even the environment for Windows cross-builds, ancient compilers and modern toolkits to cover all bases:

- http://www.slackware.com/~alien/slackbuilds/MinGW-w64/
- http://www.slackware.com/~alien/slackbuilds/docker/
- http://www.slackware.com/~alien/slackbuilds/gcc34/
- http://www.slackware.com/~alien/slackbuilds/gcc5/

FWIW, another "more official" but older Java package seems to be at:

- http://www.slackware.com/~alien/slackbuilds/openjdk/
- https://slackbuilds.org/repository/15.0/development/jdk/

An example routine to install the latest instance of a package could be like this:

```
:; wget -m -l1 http://www.slackware.com/~alien/slackbuilds/asciidoc/pkg/ && \
    find . -name '*.t?z'

:; installpkg ./www.slackware.com/~alien/slackbuilds/asciidoc/pkg/asciidoc-8.1.0-noarch-2. ↩
    tgz
```

Note that some packages are further separated by Slackware version, e.g. with sub-directories for `15.0` and `current`:

```
:; wget -r -l1 -nd -R gif,css,jpg,html,htm --remove-listing \
    http://www.slackware.com/~alien/slackbuilds/openjdk17/pkg64/15.0/

:; installpkg openjdk17-17.0.12_7-x86_64-1alien.txz
```

Upon community members' recommendations, Sotirov's SlackPack is also considered a reputable repository: https://sotirov-bg.net/slackpack/ and should cover most if not all of the dependencies required for NUT building (including PowerMan, IPMI etc.)

Note
If setting up a CI farm agent with builds in RAM disk, keep in mind that default mount options for `/dev/shm` preclude script execution. Either set up the agent in a non-standard fashion (to use another work area), or if this is a dedicated machine — relax the mount options in `/etc/fstab`.
Here is an example with `noexec` which we **must avoid** for such use-case:

```
:; mount | grep /dev/shm
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,inode64)
```

6.2.5 FreeBSD 12.2

Note
As of 2024, this version is way beyond EOL — packages have been removed from mirrors. A discussion at https://forums.freebsd.org/threads/easy-upgrading-from-12-2-rel-in-2024.92695/ touches on upgrades in such situation. An alternate mirror (that worked to bump the system to `openjdk17`, as of this writing in Nov 2024) can be found at https://mirror.sg.gs/freebsd-pkg/FreeBSD:12:amd64/quarterly/ and written into `/etc/pkg/FreeBSD.conf` similarly to existing `url` entry.

Note that `PATH` for builds on BSD should include `/usr/local/...`:

```
:; PATH=/usr/local/libexec/ccache:/usr/local/bin:/usr/bin:$PATH
:; export PATH
```

Note
You may want to reference `ccache` even before all that, as detailed below.

```
:; pkg install \
    git perl5 curl \
    gmake autoconf automake autotools libltdl libtool \
    valgrind \
```

```
    cppcheck \
    pkgconf \
    gcc clang

# To debug eventual core dump files, or trace programs with an IDE like
# NetBeans (perhaps remotely), you may want the GNU Debugger program:
:; pkg install \
    gdb

# See comments below, python version and package naming depends on distro
:; pkg install \
    python

# NOTE: For python, you may eventually have to specify a variant like this
# (numbers depending on default or additional packages of your distro):
#   :; pkg install python2 python27
# and/or:
#   :; pkg install python3 python37
# You can find a list of what is (pre-)installed with:
#   :; pkg info | grep -Ei 'perl|python'

# For spell-checking, highly recommended if you would propose pull requests:
:; pkg install \
    aspell en-aspell

# For other doc types (man-page, PDF, HTML) generation - massive packages (TEX, X11):
:; pkg install \
    asciidoc source-highlight textproc/py-pygments dblatex

# For CGI graph generation - massive packages (X11):
:; pkg install \
    libgd

:; pkg install \
    cppunit \
    nss \
    augeas \
    libmodbus \
    neon \
    net-snmp \
    powerman \
    freeipmi \
    avahi

# NOTE: At least on FreeBSD 12, system-provided crypto exists and is used
# by libnetsnmp, libneon, etc. - but is not marked as a package. Conversely,
# the openssl-1.1.1k (as of this writing) can be installed as a package into
# /usr/local/lib and then causes linking conflicts. The core system-provided
# build of openssl does include headers and is useful for NUT build "as is".
# ONLY INSTALL THIS PACKAGE IF REQUIRED (may get problems to rectify later):
:; test -e /lib/libcrypto.so -a -e /usr/lib/libssl.so || \
   pkg install openssl

:; pkg install \
    lua51

:; pkg install \
    bash dash busybox ksh93
```

Recommended:

```
:; pkg install ccache
```

```
:; ccache-update-links
```

For compatibility with common setups on other operating systems, can symlink `/usr/local/libexec/ccache` as `/usr/lib/cc` and possibly add dash-number suffixed symlinks to compiler tools (e.g. `gcc-10` beside `gcc10` installed by package).

Note
For Jenkins agents, also need to `pkg install openjdk11` (17 or 21 required since autumn 2024) — and do note its further OS configuration suggestions for special filesystem mounts.

Due to BSD specific paths **when not using** an implementation of `pkg-config` or `pkgconf` (so guessing of flags is left to administrator — TBD in NUT `m4` scripts), better use this routine to test the config/build:

```
:; ./configure --with-doc=all --with-all --with-cgi \
    --without-avahi --without-powerman --without-modbus \
    ### CPPFLAGS="-I/usr/local/include -I/usr/include" \
    ### LDFLAGS="-L/usr/local/lib -L/usr/lib"
```

Note the lack of `pkg-config` also precludes `libcppunit` tests, although they also tend to mis-compile/mis-link with GCC (while CLANG seems okay).

6.2.6 OpenBSD 6.5

Note that `PATH` for builds on BSD should include `/usr/local/...`:

```
:; PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin:$PATH
:; export PATH
```

Note
You may want to reference `ccache` even before all that, as detailed below.

OpenBSD delivers many versions of numerous packages, you should specify your pick interactively or as part of package name (e.g. `autoconf-2.69p2`).

Note
For the purposes of builds with Jenkins CI agents, since summer 2022 it requires JDK11 which was first delivered with OpenBSD 6.5. Earlier iterations used OpenBSD 6.4 and version nuances in this document may still reflect that.

FIXME: Since autumn 2024, JDK17+ is required. Maybe time to EOL OpenBSD 6.x workers, or to SSH into them from a nearby machine's Java agent?.. Alternately, consider https://github.com/adoptium/jdk17/blob/master/doc/building.md

During builds, you may have to tell system dispatcher scripts which version to use (which feels inconvenient, but on the up-side for CI — this system allows to test many versions of auto-tools in the same agent), e.g.:

```
:; export AUTOCONF_VERSION=2.69 AUTOMAKE_VERSION=1.13
```

To use the `ci_build.sh` don't forget `bash` which is not part of OpenBSD base installation. It is not required for "legacy" builds arranged by just `autogen.sh` and `configure` scripts.

Note
The OpenBSD 6.5 `install65.iso` installation includes a set of packages that seems to exceed whatever is available on network mirrors; for example, the CD image included `clang` program while it is not available to `pkg_add`, at least not via http://ftp.netbsd.hu/mirrors/openbsd/6.5/packages/amd64/ mirror. The `gcc` version on CD image differed notably from that in the networked repository (4.2.x vs. 4.9.x). You may have to echo a working base URL (part before "6.5/..." into the `/etc/installurl` file, since the old distribution is no longer served by default site.

```
# Optionally, make the environment comfortable, e.g.:
:; pkg_add sudo bash mc wget rsync

:; pkg_add \
    git curl \
    gmake autoconf automake libltdl libtool \
    valgrind \
    cppcheck \
    pkgconf \
    gcc clang

# To debug eventual core dump files, or trace programs with an IDE like
# NetBeans (perhaps remotely), you may want the GNU Debugger program:
:; pkg_add install \
    gdb

# See comments below, python version and package naming depends on distro
:; pkg_add \
    python

# NOTE: For python, you may eventually have to specify a variant like this
# (numbers depending on default or additional packages of your distro):
#   :; pkg_add python-2.7.15p0 py-pip
# and/or:
#   :; pkg_add python-3.6.6p1 py3-pip
# although you might succeed specifying shorter names and the packager
# will offer a list of matching variants (as it does for "python" above).
# NOTE: "perl" is not currently a package, but seemingly part of base OS.
# You can find a list of what is (pre-)installed with:
#   :; pkg_info | grep -Ei 'perl|python'

# For spell-checking, highly recommended if you would propose pull requests:
:; pkg_add \
    aspell

# For other doc types (man-page, PDF, HTML) generation - massive packages (TEX, X11):
:; pkg_add \
    asciidoc source-highlight py-pygments dblatex \
    docbook2x docbook-to-man

# For CGI graph generation - massive packages (X11):
:; pkg_add \
    gd

:; pkg_add \
    cppunit \
    openssl nss \
    augeas \
    libusb1 \
    net-snmp \
    avahi

# For netxml-ups driver the library should suffice; however for nut-scanner
# you may currently require to add a `libneon.so` symlink (the package seems
# to only deliver a numbered SO library name), e.g.:
:; pkg_add neon && \
   if ! test -s /usr/local/lib/libneon.so ; then
    ln -s "`cd /usr/local/lib && ls -1 libneon.so.* | sort -n | tail -1`" /usr/local/lib/ ↩
        libneon.so
   fi
```

```
# Select a LUA-5.1 (or possibly 5.2?) version
:; pkg_add \
    lua

:; pkg_add \
    bash dash ksh93

:; pkg_add \
    argp-standalone \
    freeipmi
```

Recommended:

```
:; pkg_add ccache
:; ( mkdir -p /usr/lib/ccache && cd /usr/lib/ccache && \
        for TOOL in cpp gcc g++ clang clang++ clang-cpp ; do \
            ln -s ../../local/bin/ccache "$TOOL" ; \
        done ; \
   )

:; ( cd /usr/bin && for T in gcc g++ cpp ; do ln -s "$T" "$T-4.2.1" ; done )
:; ( cd /usr/lib/ccache && for T in gcc g++ cpp ; do ln -s "$T" "$T-4.2.1" ; done )

:; ( cd /usr/bin && for T in clang clang++ clang-cpp ; do ln -s "$T" "$T-7.0.1" ; done )
:; ( cd /usr/lib/ccache && for T in clang clang++ clang-cpp ; do ln -s "$T" "$T-7.0.1" ;  ↩
    done )
```

For compatibility with common setups on other operating systems, can add dash-number suffixed symlinks to compiler tools (e.g. `gcc-4.2.1` beside `gcc` installed by package) into `/usr/lib/ccache`.

Note

For Jenkins agents, also need to `pkg_add jdk` (if asked, pick version 11 or 17); can request `pkg_add jdk%11`. You would likely have to update the trusted CA store to connect to NUT CI, see e.g. (raw!) download from https://gist.github.com/-galan/ec8b5f92dd325a97e2f66e524d28aaf8 but ensure that you run it with `bash` and it does `wget` the certificates (maybe with `--no-check-certificate` option if the OS does not trust current internet infrastructure either), and revise the suggested certificate files vs. https://letsencrypt.org/certificates/ and/or comments to that gist.

Due to BSD specific paths **when not using** an implementation of `pkg-config` or `pkgconf` (so guessing of flags is left to administrator — TBD in NUT `m4` scripts), better use this routine to test the config/build:

```
:; ./configure --with-doc=all --with-all --with-cgi \
    --without-avahi --without-powerman --without-modbus \
    ### CPPFLAGS="-I/usr/local/include -I/usr/include"
    ### LDFLAGS="-L/usr/local/lib -L/usr/lib"
```

Note the lack of `pkg-config` also precludes `libcppunit` tests, although they also tend to mis-compile/mis-link with GCC (while CLANG seems okay).

6.2.7 NetBSD 9.2

Instructions below assume that `pkgin` tool (pkg-src component to "install binary packages") is present on the system. Text below was prepared with a VM where "everything" was installed from the ISO image, including compilers and X11. It is possible that some packages provided this way differ from those served by `pkgin`, or on the contrary, that the list of suggested tool installation below would not include something a bare-minimum system would require to build NUT.

Note that `PATH` for builds on NetBSD should include `local` and `pkg`; the default after installation of the test system was:

```
:; PATH="/sbin:/usr/sbin:/bin:/usr/bin:/usr/pkg/sbin:/usr/pkg/bin:/usr/X11R7/bin:/usr/local ↩
    /sbin:/usr/local/bin"
:; export PATH
```

Note
You may want to reference `ccache` even before all that, e.g. in the system-wide `/etc/profile` or build-user's `~/.profile`, as detailed below:

```
:; PATH="/usr/lib/ccache:$PATH"
:; export PATH
```

To use the `ci_build.sh` don't forget `bash` which may be not part of NetBSD base installation. It is not required for "legacy" builds arranged by just `autogen.sh` and `configure` scripts.

Also note that the `install-sh` helper added by autotools from OS-provided resources when generating the `configure` script may be old and its directory creation mode is not safe for parallel-`make` installations. If this happens, you can work around by `make MKDIRPROG="mkdir -p" install -j 8` for example.

Note
Instructions below rely on `pkgin`, an `apt`/`yum`-like front-end tool for managing pkgsrc binary packages. It is available out of the box on the system this document describes, but if an e.g. older release lacks it, it can be added using the older tooling:

```
:; PKG_PATH="http://cdn.NetBSD.org/pub/pkgsrc/packages/NetBSD/$(uname -p)/$(uname -r|cut  ↩
    -f '1 2' -d.)/All/"
:; export PKG_PATH
:; pkg_add pkgin
```

For more details, see https://www.pkgsrc.org/#index1h1 and https://www.librebyte.net/en/cli-en/pkgin-a-netbsd-package-manager/ (the latter also provides a nice cheat-sheet about `pkgin` operations).

Note
On NetBSD 9.2 the `openpimi` and `net-snmp` packages complain that they require either OS ABI 9.0, or that `CHECK_OSABI=no` is set in a `pkg_install.conf`. Such file was not found in the test system, but can be created in `/etc/` and is honoured by `pkgin`:

```
:; grep CHECK_OSABI /etc/pkg_install.conf || \
    ( echo CHECK_OSABI=no >> /etc/pkg_install.conf )
```

```
:; pkgin install \
    git perl curl \
    mozilla-rootcerts mozilla-rootcerts-openssl \
    bmake gmake autoconf automake libltdl libtool \
    cppcheck \
    pkgconf \
    gcc7 gcc14 clang

# See comments below, python version and package naming depends on distro
:; pkgin install \
    python27 python39 python312 python313

:; ( cd /usr/pkg/bin && ( ln -fs python2.7 python2 ; ln -fs python3.13 python3 ) )
# You can find a list of what is (pre-)installed with:
#   :; pkgin list | grep -Ei 'perl|python'
```

```
# For localization maintenance (currently in Python NUT-Monitor app),
# provide an `msgfmt` implementation, e.g.:
#   :; pkgin install gettext
#
# To install the Python NUT-Monitor app, you may need some modules:
# For Python2:
#   :; pkgin install py27-gtk2
# For Python3:
#   :; pkgin install py312-qt5 py313-qt5

# For spell-checking, highly recommended if you would propose pull requests:
:; pkgin install \
    aspell aspell-en

# For man-page doc types, footprint on this platform is moderate:
:; pkgin install \
    asciidoc

# FIXME: Missing packages?
# For other doc types (PDF, HTML) generation - massive packages (TEX, X11):
# :; pkgin install \
#    source-highlight py39-pygments dblatex

# For CGI graph generation - massive packages (X11):
:; pkgin install \
    gd openmp

:; pkgin install \
    cppunit \
    openssl nss \
    augeas \
    libusb libusb1 \
    neon \
    net-snmp \
    avahi

# Select a LUA-5.1 (or possibly 5.2?) version
:; pkgin install \
    lua51 # lua52

:; pkgin install \
    bash dash ast-ksh oksh
```

Note

There seems to be no FreeIPMI for NetBSD, but there's OpenIPMI (can help someone implementing actual support for it):

```
:; pkgin install \
    openipmi
```

It still seems to not work for NUT's expectations of FreeIPMI (we have basic detection support for OpenIPMI but no code yet).

Recommended: For compatibility with common setups on other operating systems, can add dash-number suffixed symlinks to compiler tools (e.g. `gcc-7` beside the `gcc` installed by package) near the original binaries and into `/usr/lib/ccache`:

```
:; ( cd /usr/pkg/bin && for TOOL in cpp gcc g++ ; do \
        for VER in "7" "14" ; do \
            ln -fs "../gcc$VER/bin/$TOOL" "${TOOL}-${VER}" ; \
        done ; \
```

```
     done )

# Note that the one delivered binary is `clang-13` (in originally described
# installation; `clang-18` after an update in 2025 which auto-removed the
# older version) and many (unnumbered) symlinks to it. For NUT CI style of
# support for builds with many compilers, complete the known numbers:
:; ( cd /usr/pkg/bin && for TOOL in clang-cpp clang++ ; do \
        for VER in "-18" ; do \
            ln -s clang-18 "$TOOL$VER" ; \
        done ; \
     done )

:; pkgin install ccache
:; ( mkdir -p /usr/lib/ccache && cd /usr/lib/ccache && \
        for TOOL in cpp gcc g++ clang ; do \
            for VER in "" "-7" "-14" ; do \
                ln -s ../../pkg/bin/ccache "$TOOL$VER" ; \
            done ; \
        done ; \
        for TOOL in clang clang++ clang-cpp ; do \
            for VER in "" "-18" ; do \
                ln -s ../../pkg/bin/ccache "$TOOL$VER" ; \
            done ; \
        done ; \
   )
```

Note
For Jenkins agents, also need to `pkgin install openjdk21` (will be in `JAVA_HOME=/usr/pkg/java/openjdk21`). Note that the location may be not exposed in `PATH` by default, so you may want to edit your system-wide `/etc/profile` or build user's `~/.profile` with:

```
JAVA_HOME="/usr/pkg/java/openjdk21"
export JAVA_HOME
PATH="${JAVA_HOME}/bin:${PATH}"
export PATH
```

6.2.8 OpenIndiana as of releases 2021.10 to 2024.04

Note that due to IPS and `pkg(5)`, a version of python is part of baseline illumos-based OS; this may not be the case on some other illumos distributions which do not use IPS however. Currently they use python 3.7 or newer.

To build older NUT releases (2.7.4 and before), you may need to explicitly `pkg install python-27`.

Typical tooling would include:

```
:; pkg install \
    git curl wget \
    gnu-make autoconf automake libltdl libtool \
    valgrind \
    pkg-config \
    gnu-binutils developer/linker

# To debug eventual core dump files, or trace programs with an IDE like
# NetBeans (perhaps remotely), you may want the GNU Debugger program:
:; pkg install \
    gdb

# NOTE: For python, some suitable version should be available since `pkg(5)`
```

```
# tool is written in it. Similarly, many system tools are written in perl
# so some version should be installed. You may specify additional variants
# like this (numbers depending on default or additional packages of your
# distro; recommended to group `pkg` calls with many packages at once to
# save processing time for calculating a build strategy):
#   :; pkg install runtime/python-27
# and/or:
#   :; pkg install runtime/python-37 runtime/python-35 runtime/python-39
# Similarly for perl variants, e.g.:
#   :; pkg install runtime/perl-522 runtime/perl-524 runtime/perl-534
# You can find a list of what is available in remote repositories with:
#   :; pkg info -r | grep -Ei 'perl|python'

# For spell-checking, highly recommended if you would propose pull requests:
:; pkg install \
    aspell text/aspell/en

# For other doc types (man-page, PDF, HTML) generation - massive packages (TEX, X11):
:; pkg install \
    asciidoc libxslt \
    docbook/dtds docbook/dsssl docbook/xsl docbook docbook/sgml-common pygments-39 \
    image/graphviz expect graphviz-tcl

# For CGI graph generation - massive packages (X11):
:; pkg install \
    gd

:; pkg install \
    openssl library/mozilla-nss \
    library/augeas python/augeas \
    libusb-1 libusbugen system/library/usb/libusb system/header/header-usb driver/usb/ugen  ↩
        \
    libmodbus \
    library/neon \
    system/management/snmp/net-snmp \
    system/management/powerman \
    freeipmi \
    avahi

# With 2024.04, some packages were split for development vs. run-time
# and/or based on architecture:
:; pkg install \
    system/library/mozilla-nss/header-nss \
    library/nspr library/nspr/32

:; pkg install \
    lua

:; pkg install \
    dash bash shell/ksh93

### Maybe
:; pkg install \
    gnu-coreutils

### Maybe - after it gets fixed for GCC builds/linkage
:; pkg install \
    cppunit
```

For extra compiler coverage, we can install a large selection of versions, although to meet NUT CI farm expectations we also need to expose "numbered" filenames, as automated below:

```
# NOTE: not all compiler versions may be served, some are obsoleted over time.
# Check current distro repository offers with:
:; pkg info -r '*clang*' '*gcc*'
:; pkg search -r '*bin/g++*'
:; pkg search -r '*bin/clang++*'

:; pkg install \
    gcc-48 gcc-49 gcc-5 gcc-6 gcc-7 gcc-9 gcc-10 gcc-11 \
    clang-80 clang-90 \
    ccache

# As of this writing, clang-13 refused to link (claiming issues with
# --fuse-ld which was never specified) on OI; maybe later it will:
#:; pkg install \
#    developer/clang-13 runtime/clang-13

# With OI 2024.04 there's also clang-18 available in the mix, and as of
# 2025 packaging, clang-19 as well:
:; pkg install \
    developer/clang-19 runtime/clang-19

# Get clang-cpp-X visible in standard PATH (for CI to reference the right one),
# and make sure other frontends are exposed with versions (not all OI distro
# releases have such symlinks packaged right), e.g.:
:; (cd /usr/bin && for X in 8 9 19 ; do for T in "" "++" "-cpp"; do \
    ln -fs "../clang/$X.0/bin/clang$T" "clang${T}-${X}" ; \
   done; done)

# If /usr/lib/ccache/ symlinks to compilers do not appear after package
# installation, or if you had to add links like above, call the service:
:; svcadm restart ccache-update-symlinks
```

We can even include a `gcc-4.4.4-il` version (used to build the illumos OS ecosystems, at least until recently, which is a viable example of an old GCC baseline); but note that so far it conflicts with `libgd` builds at `./configure --with-cgi` stage (its binaries require newer ecosystem):

```
:; pkg install \
    illumos-gcc@4.4.4

# Make it visible in standard PATH
:; (cd /usr/bin && for T in gcc g++ cpp ; do \
    ln -s ../../opt/gcc/4.4.4/bin/$T $T-4.4.4 ; \
   done)

# If /usr/lib/ccache/ symlinks to these do not appear, call the service:
:; svcadm restart ccache-update-symlinks
```

OI currently also does not build `cppunit`-based tests well, at least not with GCC (they segfault at run-time with `ostream` issues); a CLANG build works for that however.

It also lacks out-of-the-box Tex suite and `dblatex` in particular, which `asciidoc` needs to build PDF documents. It may be possible to add these from third-party repositories (e.g. SFE) and/or build from sources.

No pre-packaged `cppcheck` was found, either.

Note

For Jenkins agents, also need to `pkg install runtime/java/openjdk17` for JRE/JDK 17. Java 17 or 21 is required to run Jenkins agents after autumn 2024. If updating from older releases, you may need to update default implementation, e.g.:

```
:; pkg set-mediator -V 17 java
```

6.2.9 OmniOS CE (as of release 151036)

Being a minimal-footprint system, OmniOS CE provides very few packages out of the box. There are additional repositories supported by the project, as well as third-party repositories such as SFE. For some dependencies, it may happen that you would need to roll and install your own builds in accordance with that project's design goals.

Note you may need not just the "Core" IPS package publisher, but also the "Extra" one. As of release 151052, it can be attached right in the installer; otherwise the `pkg (set-)publisher` commands can be used. See OmniOS CE web site for setup details.

```
# Optionally, make the environment comfortable, e.g.:
:; pkg install sudo bash application/mc wget rsync pigz pbzip2 p7zip top

# NOTE: not all compiler versions may be served, some are obsoleted over time.
# While OmniOS LTS 151046 offers a range of clang 13 to 17, current stable
# includes also 18 and 19. OmniOS does not provide implicit (not numbered)
# packages for clang or gcc. Check your distro repository offers with:
:; pkg info -r '*clang*' '*gcc*'
:; pkg search -r '*bin/g++*'
:; pkg search -r '*bin/clang++*'

:; pkg install \
    developer/build/autoconf developer/build/automake \
    developer/build/libtool library/libtool/libltdl \
    build-essential ccache git developer/pkg-config \
    runtime/perl \
    asciidoc \
    libgd \
    developer/clang-17

# To debug eventual core dump files, or trace programs with an IDE like
# NetBeans (perhaps remotely), you may want the GNU Debugger program:
:; pkg install \
    gdb

# May be or not be provided, depending on distro age:
:; pkg install \
    net-snmp

# For crypto-enabled builds:
:; pkg install \
    openssl-3

# By r151046, some packages were split for development vs. run-time;
# to provide support for Mozilla NSS, you need libraries and headers:
:; pkg install \
    system/library/mozilla-nss \
    system/library/mozilla-nss/header-nss \
    library/nspr library/nspr/header-nspr

# NOTE: For python, some suitable version should be available since `pkg(5)`
# tool is written in it. You may specify an additional variant like this
# (numbers depending on default or additional packages of your distro):
#   :; pkg install runtime/python-37
# You can find a list of what is available in remote repositories with:
#   :; pkg info -r | grep -Ei 'perl|python'
```

Your OmniOS version may lack a pre-packaged libusb, however the binary build from contemporary OpenIndiana can be used (copy the header files and the library+symlinks for all architectures you would need).

Note
As of July 2022, a `libusb-1` package recipe was proposed for the `omnios-extra` repository (NUT itself and further dependencies may also appear there, per issue #1498), and is available as of release 151046 and later on:

```
:; pkg install \
    libusb-1
```

You may need to set up `ccache` with the same `/usr/lib/ccache` dir used in other OS recipes. Assuming your Build Essentials pulled GCC 11 version, and ccache is under `/opt/ooce` namespace, that would be like:

```
:; GCCVER=11
:; mkdir -p /usr/lib/ccache
:; cd /usr/lib/ccache
:; ln -fs ../../../opt/ooce/bin/ccache gcc
:; ln -fs ../../../opt/ooce/bin/ccache g++
:; ln -fs ../../../opt/ooce/bin/ccache gcpp
:; ln -fs ../../../opt/ooce/bin/ccache gcc-${GCCVER}
:; ln -fs ../../../opt/ooce/bin/ccache g++-${GCCVER}
:; ln -fs ../../../opt/ooce/bin/ccache gcpp-${GCCVER}
```

Given that many of the dependencies can get installed into that namespace, you may have to specify where `pkg-config` will look for them (note that library and binary paths can be architecture bitness-dependent; the `ci_build.sh` script should take care of this for many scenarios):

```
:; ./configure PKG_CONFIG_PATH="/opt/ooce/lib/amd64/pkgconfig" --with-cgi
```

You may also have to fiddle with either `LD_LIBRARY_PATH`, `crle (-c)` or `./configure --with-something-libs` options to compensate for lack of `-R` linker options in the `pkg-config` information provided by that repository: by default the NUT programs can build with the information they get, but fail to find the needed shared objects at run-time. More about this situation is tracked at https://github.com/networkupstools/nut/issues/2782

Note also that the minimal footprint nature of OmniOS CE precludes building any large scope easily, so avoid docs and "all drivers" unless you provide whatever they need to happen.

Note
For Jenkins agents, also need to `pkg install runtime/java/openjdk21` for JRE/JDK 17. Java 17 or 21 is required to run Jenkins agents after autumn 2024. If updating from older releases, you may need to update default implementation, e.g.:

```
:; pkg set-mediator -V 21 java
```

6.2.10 Solaris 8

Builds for a platform as old as this are not currently covered by regular NUT CI farm runs, however since the very possibility of doing this was recently verified, some notes follow.

For context: Following a discussion in the mailing list starting at https://alioth-lists.debian.net/pipermail/nut-upsuser/2022-December/013051.html and followed up by GitHub issues and PR:

- https://github.com/networkupstools/nut/issues/1736
- https://github.com/networkupstools/nut/issues/1737 (about a possible but not yet confirmed platform problem)
- https://github.com/networkupstools/nut/pull/1738

...recent NUT codebase was successfully built and self-tested in a Solaris 8 x86 VM (a circa 2002 release), confirming the project's adherence to the goal that if NUT ran on a platform earlier, so roughly anything POSIX-ish released this millennium and still running, it should still be possible — at least as far as our part of equation is concerned.

That said, platform shows its age vs. later standards (script interpreters and other tools involved), and base "complete install" lacked compilers, so part of the tested build platform setup involved third-party provided package repositories.

One helpful project was extensive notes about preparation of the Solaris 8 VM (and our further comments there), which pointed to the still active "tgcware" repository and contains scripts to help prepare the freshly installed system:

- https://github.com/mac-65/Solaris_8_x86_VM
- https://github.com/mac-65/Solaris_8_x86_VM/issues/1
- http://jupiterrise.com/tgcware/sunos5.8_x86/stable/

Note that scripts attached to the notes refer to older versions of the packages than what is currently published, so I ended up downloading everything from the repository into the VM and using shell wildcards to pick the packages to install (mind the package families with similar names when preparing such patterns).

After the OS, tools and feasible third-party dependencies were installed, certain environment customization was needed to prepare for NUT build in particular (originally detailed in GitHub issues linked above):

- For `CONFIG_SHELL`, system `dtksh` seems to support the syntax (unlike default `/bin/sh`), but for some reason segfaults during `configure` tests. Alternatively `/usr/tgcware/bin/bash` (4.4-ish) can be used successfully. System-provided bash 2.x is too old for these scripts.
- To run `ci_build.sh` CI/dev-testing helper script, either the shebang should be locally fixed to explicitly call `/usr/tgcware/bin` or the build environment's `PATH` should point to this `bash` implementation as a first hit. If we want to primarily use OS-provided tools, this latter option may need a bit of creative setup; I made a symlink into the `/usr/lib/ccache` directory which has to be first anyway (before compilers).
- The system-provided default `grep` lacks the `-E` option which was preferred over generally obsoleted `egrep` since PR #1660 — however pre-pending `/usr/xpg4/bin` early in the `PATH` fixes the problem.
- The builds of `gcc` in TGCWARE repository were picky about shared objects linking as needed to run them, so `LD_LIBRARY_PATH` had to refer to its library directories (generally this is frowned upon and should be a last resort).
- Due to lack of Python in that OS release, NUT augeas support had to be disabled when preparing the build from Git sources (generated files may be available as part of distribution tarballs however): `WITHOUT_NUT_AUGEAS=true; export WITHOUT_NUT_AUGEAS; ./autogen.sh`

Overall, the successful test build using the NUT standard CI helper script `ci_build.sh` had the following shell session settings:

```
### Common pre-sets from .profile or .bashrc:
### bash-2.03$ echo $PATH
### /usr/bin:/usr/dt/bin:/usr/openwin/bin:/bin:/usr/ucb:/usr/tgcware/bin:/usr/tgcware/gnu:/ ↩
    usr/tgcware/gcc42/bin:/usr/tgcware/i386-pc-solaris2.8/bin

### bash-2.03$ echo $LD_LIBRARY_PATH
### /usr/lib:/usr/tgcware/lib:/usr/tgcware/gcc42/lib:/usr/tgcware/i386-pc-solaris2.8/lib

### Further tuning for the build itself:
:; git clean -fffdddxxx
:; CONFIG_SHELL=/usr/tgcware/bin/bash \
   WITHOUT_NUT_AUGEAS=true \
   PATH="/usr/xpg4/bin:$PATH" \
    /usr/tgcware/bin/bash ./ci_build.sh
```

6.2.11 MacOS with homebrew

Some CI tests happen on MacOS using a mix of their default xcode environment for compilers, and Homebrew community packaging for dependencies (including `bash` since the system one is too old for `ci_build.sh` script syntax).

See `.travis.yml` and `.circleci/config.yml` for practical details of the setup, and https://brew.sh if you want to install it on your MacOS system (note that its default packaged locations known as `HOMEBREW_PREFIX` differ depending on architecture — see https://docs.brew.sh/Installation for more details; find via `brew config | grep HOMEBREW_PREFIX: | awk '{print $2}'`).

Note
The quickest pre-configuration for `ci_build.sh` integration with this non-default build system would be to add this line into your shell profile:

```
eval "$(brew shellenv)"
```

Note
Homebrew is not the only build/packaging system available for MacOS, so NUT scripts do not make any assumptions nor try to find a build system they were not told about (via `HOMEBREW_PREFIX` in this case).

Currently known dependencies for basic build include:

```
# Optional for a quick spin:
:; HOMEBREW_NO_AUTO_UPDATE=1; export HOMEBREW_NO_AUTO_UPDATE

# Required:
:; brew install ccache bash libtool binutils autoconf automake git m4 \
        pkg-config aspell asciidoc docbook-xsl cppunit gd \
        libusb neon net-snmp \
        nss openssl \
        libmodbus freeipmi powerman

# To debug eventual core dump files, or trace programs with an IDE like
# NetBeans (perhaps remotely), you may want the GNU Debugger program:
:; brew install gdb

# Recommended:
:; brew install curl wget midnight-commander
```

Note
for `asciidoc/a2x` to work, you should `export XML_CATALOG_FILES` with the location of packaged resources (`${HOMEBREW_PREFIX}/etc/xml/catalog`). On one test system, man page builds spewed warnings like `<unknown>:1: SyntaxWarning: invalid escape sequence '\S'` due to incompatibility of older asciidoc with new Python syntax requirements, but seemed to produce reasonable results otherwise.

Note that `ccache` is installed in a different location than expected by default in the `ci_build.sh` script, so if your system allows to add the symbolic link to `/opt/homebrew/opt/ccache/libexec` (`/usr/local/opt/ccache/libexec` on x86) as `/usr/lib/ccache` — please do so as the easiest way out.

Alternatively, to prepare building sessions with `ci_build.sh` you can:

```
:; export CI_CCACHE_SYMLINKDIR="/opt/homebrew/opt/ccache/libexec"

### ...or for x86 builders:
#:; export CI_CCACHE_SYMLINKDIR="/usr/local/opt/ccache/libexec"
```

Note
For Jenkins agents, also need to `brew install --cask temurin@21` for JRE/JDK 21. Java 17 or 21 (an LTS) is required to run Jenkins agents after summer 2024.

The compiler is part of Apple's XCode ecosystem. Just try to run `clang` in a GUI terminal, and a pop-up will appear offering to install it. Note that you would have to create symbolic links to version-numbered names of compilers, e.g. `clang-14` and `clang++-14` in both `/usr/local/bin` (pointing to `/bin/clang(++)` and in the `ccache` location prepared above (pointing to `../bin/ccache`), and repeat that in locations prepared by XCode installation such as `/Library/Developer/Comma` and `/usr/local/Homebrew/Library/Homebrew/shims/mac/super/` just as `ln -s clang{,-14} ; ln -s clang++{,-14}`. Apparently `clang` is the only compiler available; various names of `gcc*` are links to the same binaries.

Warning
Take care to **NOT** symlink a `clang-cpp(-14)` which is not a name recognized by XCode dispatcher program, so requests to it freeze.

On a machine dedicated to CI purposes, it may be wise to disable Spotlight search indexing with `sudo mdutil -a -i off`, disable active desktop themes, etc. (normally you would not even log in interactively).

For scratch workers (VMs) also consider going to System Preferences ⇒ Software Updates ⇒ Advanced, and un-checking all the boxes (disable the overheads of pulling in the updates you won't remember in an hour anyway).

If you eventually find that the `kernel_task` consumes a lot of CPU, this is usually the system's way of throttling (scheduling A LOT of high-priority no-op cycles to preempt any other workload). In this case please investigate cooling and hardware compatibility (especially in a VM), or find and follow community documentation about neutering `IOPlatformPluginFamily.kext` module (probably not on the bare-metal MacOS instance — find the cooling problem there instead).

- https://grafxflow.co.uk/blog/mac-os-x/delete-ioplatformpluginfamilykext-macos-big-sur

6.2.12 Windows builds

There have been several attempts to adjust NUT codebase to native builds for Windows, as well as there are many projects helping to produce portable programs.

Further TODO for Windows would include:

- MSVCRT (ubiquitous) vs. UCRT (more standards compliant, but since Windows 10) https://docs.microsoft.com/en-us/cpp/porting/upgrade-your-code-to-the-universal-crt?view=msvc-160
- libusb-0.1 vs. libusb-1.0

Note
Native mingw, MSYS2, etc. builds on Windows are known to suffer from interaction with antivirus software which holds executable files open and so not writable by the linker. This may cause random steps in the `configure` script or later during the build to fail. If that happens to you, disable the antivirus completely or exempt at least the NUT build area from protection.

Windows with mingw

See `scripts/Windows/README.adoc` for original recommendations for the effort, including possibilities of cross-builds with mingw available in Linux.

Unfortunately these did not work for me at the time of testing, yielding some issues downloading mingw both in Windows and Linux environments. So I explored other downloads, as detailed below.

See also:

- https://winlibs.com/
- https://azrael.digipen.edu/~mmead/www/public/mingw/
- https://www.mingw-w64.org/downloads/

Note
Seems the mingw installer has problems with current authentication and redirect on SourceForge. You can download and unpack 7z archives from https://sourceforge.net/projects/mingw-w64/files/mingw-w64/mingw-w64-release/ into e.g. `C:\Progra~1\mingw-w64\x86_64-8.1.0-release-posix-seh-rt_v6-rev0` location on your Windows system. Then for building further NUT dependencies see `scripts/Windows/README.adoc`.

Windows with MSYS2

The MSYS2 ecosystem is available at https://www.msys2.org/ and builds upon earlier work by MinGW-w64 (in turn a fork of MinGW.org (aka mingw-w32)) and Cygwin projects, to name a few related efforts. It also includes `pacman` similar to that in Arch Linux for easier dependency installation, and many packages are available "out of the box" this way.

The project is currently sponsored by Microsoft and seems to be supported by Visual Studio Code IDE for building and debugging projects, for more details see https://code.visualstudio.com/docs/cpp/config-mingw

Notable pages of the project include:

- https://www.msys2.org/ with current download link and first-installation instructions
- https://www.msys2.org/wiki/MSYS2-introduction/ for general overview
- https://packages.msys2.org/search?t=binpkg for search in package repository

After downloading and installing MSYS2 archive for the first time, they suggest to start by updating the base ecosystem (using their terminal):

```
:; pacman -Syu
```

Wait for metadata and base package downloads, agree that all MSYS2 programs including your terminal would be closed/restarted, and wait for this stage to complete.

Run it again to refresh more of the ecosystem, now without restarting it:

```
:; pacman -Syu
```

Finally, install tools and prerequisites for building NUT; note that some of the recommended package names are "umbrellas" for several implementations, and the `pacman` would ask you which (or "all") to install in those cases.

Note
Suggestions below use `x86_64` generic variants where possible, and `clang` where available to try both build toolkits on the platform. If you want to build `i686` (32-bit) or alternate backends (e.g. `ucrt` instead of default `msvcrt`), poke the repository search to see what is available.

Note
To build NUT with `ci_build.sh` (and generally — to help `configure` script find the dependencies listed below), start the terminal session with "MSYS2 MinGW x64" shortcut. Other options set up the environment variables for toolkits listed in their shortcut names, and so tend to prefer "wrong" flags and paths to dependencies (if you have several variants installed). The "MSYS2 MinGW UCRT x64" was also reported to work.

To avoid toolkit variant mismatches, you may require to use their specific builds preferentially:

```
PATH="/mingw64/bin:$PATH"
export PATH
```

. . . and also add these lines to the `~/.bashrc` file.

```
# This covers most of the common FOSS development baseline, including
# python, perl, autotools, gcc, clang, git, binutils, make, pkgconf...
:; pacman -S --needed \
    base-devel mingw-w64-x86_64-toolchain \
    autoconf-wrapper automake-wrapper libtool mingw-w64-x86_64-libltdl \
    clang gcc \
    ccache mingw-w64-x86_64-ccache \
    git aspell aspell-en \
    vim python \
    mingw-w64-x86_64-python-pygments

# PThreads come as an extra feature; note there are many variants,
# see https://packages.msys2.org/search?t=binpkg&q=pthread
:; pacman -S --needed \
    mingw-w64-x86_64-winpthreads-git \
    mingw-w64-clang-x86_64-winpthreads-git

# Note that MSYS2 includes libusb-1.0 "natively"
# The NUT codebase adjustments for Windows might at this moment expect older
# ecosystem via https://github.com/mcuee/libusb-win32 -- subject to fix then.
:; pacman -S --needed \
    mingw-w64-x86_64-libusb \
    mingw-w64-clang-x86_64-libusb

# Seems that the older libusb-win32 (libusb-0.1) is also available as:
:; pacman -S --needed \
    mingw-w64-x86_64-libusb-win32 \
    mingw-w64-clang-x86_64-libusb-win32

# Alternately there is libusb-compat (libusb-1.0 codebase exposing the older
# libusb-0.1 API) which SHOULD NOT be installed along with the real libusb-0.1:
# :; pacman -S --needed mingw-w64-x86_64-libusb-compat-git mingw-w64-clang-x86_64-libusb- ↩
    compat-git

# This also pulls *-devel of several other projects:
:; pacman -S --needed \
    mingw-w64-x86_64-neon libneon-devel

# Other dependencies:
:; pacman -S --needed \
    mingw-w64-x86_64-libmodbus-git \
    mingw-w64-clang-x86_64-libmodbus-git \
    mingw-w64-x86_64-libgd \
    mingw-w64-clang-x86_64-libgd

# For C++ tests:
:; pacman -S --needed \
    mingw-w64-x86_64-cppunit \
    mingw-w64-clang-x86_64-cppunit
```

`ccache` wrapper scripts are available as e.g. `/mingw64/lib/ccache/bin/gcc` and lack a set for `clang` tools; easy-peasy fix with:

```
:; cd /mingw64/lib/ccache/bin
:; for T in clang clang++ clang-cpp ; do sed "s/gcc/$T/" < gcc > "$T" ; chmod +x "$T" ;  ↩
    done
```

Note that default `ccache` seems quirky on Windows MSYS2, possibly due to mixing of the path separator characters and/or embedding and choking on the `C:` in path names. Overall it seems unable to create the cache files after it has created the cache directory tree (though you might have to pre-create the `${HOME}/.ccache` anyway, as NUT `ci_build.sh` script does. As found in experimentation, setting the `PATH` consistently for toolkits involved is very important.

- https://github.com/ccache/ccache/discussions/784
- https://sourceforge.net/p/msys2/tickets/253/

Notable packages **not found** in the repo:

- snmp (net-snmp, ucd-snmp) — instructions in `scripts/Windows/README.adoc` document now covers building it from source in MSYS2 MinGW x64 environment, essentially same as for Linux cross builds with proper `ARCH` and `PREFIX`
- libregex (C version, direct NUT `configure` script support was added by the Windows branch); MSYS2 however includes `libpcre` pulled by some of the dependencies above...
- augeas
- avahi
- powerman
- ipmi

Not installed above (yet?):

- https://packages.msys2.org/search?t=binpkg&q=serial — for these need to first check if termios is part of baseline

Note that `ccache` symlinks for MSYS2 are installed into `/usr/lib/ccache/bin` directory (not plain `/usr/lib/ccache` as elsewhere).

Note

After you successfully build NUT (perhaps using `ci_build.sh`), if you install it into a prototype area by `make DESTDIR=... install` then you should add the third-party shared libraries involved, for that file set to be usable. Something along these lines:

```
:; find "$DESTDIR" -name '*.exe' -type f  | while read F ; do ldd "$F" \
   | grep ' /mingw64/' ; done  | awk '{print $3}' | sort | uniq \
   | while read LIB ; do cp -pf "$LIB" "$DESTDIR/mingw64/bin/" ; done
```

Keep in mind that a similar trick (or links to `*.dll` — and symlinks are problematic on that platform) may be needed in other directories, such as `sbin` and `cgi-bin`:

```
:; ( cd "$DESTDIR/mingw64/bin/" && ln *.dll ../sbin && ln *.dll ../cgi-bin )
```
