

The `build2` Build System

Copyright © 2014-2017 Code Synthesis Ltd

Permission is granted to copy, distribute and/or modify this document under the terms of the MIT License.

Revision 0.5, May 2017

This revision of the document describes the `build2` build system 0.5.x series.

Table of Contents

Preface 1

1 Name Patterns 1

2 Grammar 3

3 Test Module 4

4 Version Module 4

Preface

This is the preface.

1 Name Patterns

For convenience, in certain contexts, names can be generated with shell-like wildcard patterns. A name is a *name pattern* if its value contains one or more unquoted wildcard characters or character sequences. For example:

```
./: */                # All (immediate) subdirectories
exe{hello}: {hxx cxx}{**} # All C++ header/source files.
pattern = '*.txt'      # Literal '*.txt'.
```

Pattern-based name generation is not performed in certain contexts. Specifically, it is not performed in target names where it is interpreted as a pattern for target type/pattern-specific variable assignments. For example.

```
s = *.txt             # Variable assignment (performed).
./: cxx{*}            # Prerequisite names (performed).
cxx{*}: dist = false  # Target pattern (not performed).
```

In contexts where it is performed, it can be inhibited with quoting, for example:

```
pat = 'foo*bar'
./: cxx{'foo*bar'}
```

The following characters are wildcards:

```
* - match any number of characters (including zero)
? - match any single character
```

If a pattern ends with a directory separator, then it only matches directories. Otherwise, it only matches files. Matches that start with a dot (.) are automatically ignored unless the pattern itself also starts with this character.

In addition to the above wildcard characters, `**` and `***` are recognized as wildcard character sequences. If a pattern contains `**`, then it is matched just like `*` but in all the subdirectories, recursively. The `***` wildcard behaves like `**` but also matches the start directory itself. For example:

```
exe{hello}: cxx{***} # All C++ source files recursively.
```

A group-enclosed (`{ }`) pattern value may be followed by inclusion/exclusion patterns/matches. A subsequent value is treated as an inclusion if it starts with a plus sign (+) and as an exclusion if it starts with a minus (-). A subsequent value that does not start with either of these signs is illegal. For example:

```

exe{hello}: cxx{f* -foo}           # Exclude foo if present.
exe{hello}: cxx{f* +foo}           # Include foo if not present.
exe{hello}: cxx{f* -fo?}           # Exclude foo and fox if present.
exe{hello}: cxx{f* +b* -foo -bar}  # Exclude foo and bar if present.

```

Inclusion and exclusion are applied in the order specified and only to the result produced up to that point. The order of names in the result is unspecified, however, it is guaranteed not to contain duplicates. The pattern and the following inclusions/exclusions must be consistent with regards to the type of filesystem entry they match. That is, they should all match either files or directories. For example:

```

exe{hello}: cxx{f* -foo +*oo}      # Exclusion has no effect.
exe{hello}: cxx{f* +*oo}           # Ok, no duplicates.
./: {*/ -build}                   # Error: exclusion must match a directory.

```

If many inclusions or exclusions need to be specified, then an inclusion/exclusion group can be used. For example:

```

exe{hello}: cxx{f* -{foo bar}}    # Exclude foo and bar if present.

```

This is particularly useful if you would like to list the names to exclude in a variable. For example, this is how we can exclude certain files from compilation but still include them as ordinary file prerequisites (so that they are still included into the distribution):

```

exc = foo.cxx bar.cxx
exe{hello}: cxx{f* -{$exc}} file{$exc}

```

One common situation that calls for exclusions is auto-generated source code. Let's say we have auto-generated command line parser in `options.hxx` and `options.cxx`. Because of the in-tree builds, our name pattern may or may not find these files. Note, however, that we cannot just include them as non-pattern prerequisites. We also have to exclude them from the pattern match since otherwise we may end up with duplicate prerequisites. As a result, this is how we have to handle this case provided we want to continue using patterns to find other, non-generated source files:

```

exe{hello}: {hxx cxx}{* -options} {hxx cxx}{options}

```

If the name pattern includes an absolute directory, then the pattern match is performed in that directory and the generated names include absolute directories as well. Otherwise, the pattern match is performed in the *pattern base* directory. In buildfiles this is `src_base` while on the command line – the current working directory. In this case the generated names are relative to the base directory. For example, assuming we have the `foo.cxx` and `b/bar.cxx` source files:

```

exe{hello}: $src_base/cxx{**}      # $src_base/cxx{foo} $src_base/b/cxx{bar}
exe{hello}:          cxx{**}      #          cxx{foo}          b/cxx{bar}

```

Pattern matching as well as inclusion/exclusion logic is target type-specific. If the name pattern does not contain a type, then the `dir{}` type is assumed if the pattern ends with a directory separator and `file{}` otherwise.

For the `dir{}` target type the trailing directory separator is added to the pattern and all the inclusion/exclusion patterns/matches that do not already end with one. Then the filesystem search is performed for matching directories. For example:

```
./: dir{* -build} # Search for */, exclude build/.
```

For the `file{}` and `file{}`-based target types the default extension (if any) is added to the pattern and all the inclusion/exclusion patterns/matches that do not already contain an extension. Then the filesystem search is performed for matching files.

For example, the `cxx{}` target type obtains the default extension from the `extension` variable. Assuming we have the following line in our `root.build`:

```
cxx{*}: extension = cxx
```

And the following in our `buildfile`:

```
exe(hello): {cxx}{* -foo -bar.cxx}
```

The pattern match will first search for all the files matching the `*.cxx` pattern in `src_base` and then exclude `foo.cxx` and `bar.cxx` from the result. Note also that target type-specific decorations are removed from the result. So in the above example if the pattern match produces `baz.cxx`, then the prerequisite name is `cxx{baz}`, not `cxx{baz.cxx}`.

If the name generation cannot be performed because the base directory is unknown, target type is unknown, or the target type is not directory or file-based, then the name pattern is returned as is (that is, as an ordinary name). Project-qualified names are never considered to be patterns.

2 Grammar

```
eval:          '(' (eval-comma | eval-qual)? ')'
eval-comma:    eval-ternary (',' eval-ternary)*
eval-ternary:  eval-or ('?' eval-ternary ':' eval-ternary)?
eval-or:       eval-and ('||' eval-and)*
eval-and:      eval-comp ('&&' eval-comp)*
eval-comp:     eval-value (('==' | '!=' | '<' | '>' | '<=' | '>=') eval-value)*
eval-value:    value-attributes? (<value> | eval | '!' eval-value)
eval-qual:     <name> ':' <name>
```

```
value-attributes: '[' <key-value-pairs> ']'
```

Note that `?:` (ternary operator) and `!` (logical not) are right-associative. Unlike C++, all the comparison operators have the same precedence. A qualified name cannot be combined with any other operator (including ternary) unless enclosed in parentheses. The `eval` option in the `eval-value` production shall contain single value only (no commas).

3 Test Module

The targets to be tested as well as the tests/groups from testscripts to be run can be narrowed down using the `config.test` variable. While this value is normally specified as a command line override (for example, to quickly re-run a previously failed test), it can also be persisted in `config.build` in order to create a configuration that will only run a subset of tests by default. For example:

```
b test config.test=foo/exe{driver} # Only test foo/exe{driver} target.
b test config.test=bar/baz        # Only run bar/baz testscript test.
```

The `config.test` variable contains a list of @-separated pairs with the left hand side being the target and the right hand side being the testscript id path. Either can be omitted (along with @). If the value contains a target type or ends with a directory separator, then it is treated as a target name. Otherwise – an id path. The targets are resolved relative to the root scope where the `config.test` value is set. For example:

```
b test config.test=foo/exe{driver}@bar
```

To specify multiple id paths for the same target we can use the pair generation syntax:

```
b test config.test=foo/exe{driver}@{bar baz}
```

If no targets are specified (only id paths), then all the targets are tested (with the testscript tests to be run limited to the specified id paths). If no id paths are specified (only targets), then all the testscript tests are run (with the targets to be tested limited to the specified targets). An id path without a target applies to all the targets being considered.

A directory target without an explicit target type (for example, `foo/`) is treated specially. It enables all the tests at and under its directory. This special treatment can be inhibited by specifying the target type explicitly (for example, `dir{foo/}`).

4 Version Module

A project can use any version format as long as it meets the package version requirements. The toolchain also provides additional functionality for managing projects that conform to the *build2 standard version* format. If you are starting a new project that uses *build2*, you are strongly encouraged to use this versioning scheme. It is based on much thought and, often painful, experience. If you decide not to follow this advice, you are essentially on your own when version management is concerned.

The standard *build2* project version conforms to Semantic Versioning and has the following form:

```
<major>.<minor>.<patch>[<-<prerelease>]
```


For example:

```
1.2.3
1.2.3-a.1
1.2.3-b.2
```

The `build2` package version that uses the standard project version will then have the following form (*epoch* is the versioning scheme version and *revision* is the package revision):

```
[<epoch>~]<major>.<minor>.<patch>[-<prerel>][+<revision>]
```

For example:

```
1.2.3
1.2.3+1
1~1.2.3-a.1+2
```

The *major*, *minor*, and *patch* should be numeric values between 0 and 999 and all three cannot be zero at the same time. For initial development it is recommended to use 0 for *major*, start with version 0.1.0, and change to 1.0.0 once things stabilize.

In the context of C and C++ (or other compiled languages), you should increment *patch* when making binary-compatible changes, *minor* when making source-compatible changes, and *major* when making breaking changes. While the binary compatibility must be set in stone, the source compatibility rules can sometimes be bent. For example, you may decide to make a breaking change in a rarely used interface as part of a minor release. Note also that in the context of C++ deciding whether a change is binary-compatible is a non-trivial task. There are resources that list the rules but no automated tooling yet. If unsure, increment *minor*.

If present, the *prerel* component signifies a pre-release. Two types of pre-releases are supported by the standard versioning scheme: *final* and *snapshot* (non-pre-release versions are naturally always final). For final pre-releases the *prerel* component has the following form:

```
(a|b).<num>
```

For example:

```
1.2.3-a.1
1.2.3-b.2
```

The letter 'a' signifies an alpha release and 'b' – beta. The alpha/beta numbers (*num*) should be between 1 and 499.

Note that there is no support for release candidates. Instead, it is recommended that you use later-stage beta releases for this purpose (and, if you wish, call them "release candidates" in announcements, etc).

What version should be used during development? The common approach is to increment to the next version and use that until the release. This has one major drawback: if we publish intermediate snapshots (for example, for testing) they will all be indistinguishable both between each other and, even worse, from the final release. One way to remedy this is to

increment the pre-release number before each publication. However, unless automated, this will be burdensome and error-prone. Also, there is a real possibility of running out of version numbers if, for example, we do continuous integration by testing (and therefore publishing) each commit.

To address this, the standard versioning scheme supports *snapshot pre-releases* with the *prerelease* component having the following form:

```
(a|b) .<num>.<snapsn>[.<snapid>]
```

For example:

```
1.2.3-a.1.1422564055.340c0a26a5efed1f
```

In essence, a snapshot pre-release is after the previous final release but before the next (a.1 and, perhaps, a.2 in the above example) and is uniquely identified by the snapshot sequence number (*snapsn*) and snapshot id (*snapid*).

The *num* component has the same semantics as in the final pre-releases except that it can be 0. The *snapsn* component should be either the special value 'z' or a numeric, non-zero value that increases for each subsequent snapshot. It must fit into an unsigned 64-bit integer. The *snapid* component, if present, should be an alpha-numeric value that uniquely identifies the snapshot. It is not required for version comparison (*snapsn* should be sufficient) and is included for reference. It must not be longer than 16 characters.

Where do the snapshot sn and id come from? Normally from the version control system. For example, for `git`, *snapsn* is the commit date (as UNIX timestamp in the UTC timezone) and *snapid* is a 16-character abbreviated commit id. As discussed below, the `build2` version module extracts and manages all this information automatically (the use of `git` commit dates is not without limitations; see below for details).

The special 'z' *snapsn* value identifies the *latest* or *uncommitted* snapshot. It is chosen to be greater than any other possible *snapsn* value and its use is discussed further below.

As an illustration of this approach, let's examine how versions change during the lifetime of a project:

```
0.1.0-a.0.z      # development after a.0
0.1.0-a.1        # pre-release
0.1.0-a.1.z      # development after a.1
0.1.0-a.2        # pre-release
0.1.0-a.2.z      # development after a.2
0.1.0-b.1        # pre-release
0.1.0-b.1.z      # development after b.1
0.1.0            # release
0.1.1-b.0.z      # development after b.0 (bugfix)
0.2.0-a.0.z      # development after a.0
0.1.1            # release (bugfix)
1.0.0            # release (jumped straight to 1.0.0)
...
```

As shown in the above example, there is nothing wrong with "jumping" to a further version (for example, from alpha to beta, or from beta to release, or even from alpha to release). We cannot, however, jump backwards (for example, from beta back to alpha). As a result, a sensible strategy is to start with `a.0` since it can always be upgraded (but not downgrade) at a later stage.

When it comes to the version control systems, the recommended workflow is as follows: The change to the final version should be the last commit in the (pre-)release. It is also a good idea to tag this commit with the project version. A commit immediately after that should change the version to a snapshot essentially "opening" the repository for development.

The project version without the snapshot part can be represented as a 64-bit decimal value comparable as integers (for example, in preprocessor directives). The integer representation has the following form:

```
AAABBBCCCCDDDE
```

```
AAA - major
BBB - minor
CCC - patch
DDD - alpha / beta (DDD + 500)
E   - final (0) / snapshot (1)
```

If the *DDDE* value is not zero, then it signifies a pre-release. In this case one is subtracted from the *AAABBBCCC* value. An alpha number is stored in *DDD* as is while beta – incremented by 500. If *E* is 1, then this is a snapshot after *DDD*.

For example:

```
AAABBBCCCCDDDE
0.1.0          00000100000000
0.1.2          00000100100000
1.2.3          00100200300000
2.2.0-a.1      00200199900100
3.0.0-b.2      00299999950200
2.2.0-a.1.z    00200199900110
```

A project that uses standard versioning can rely on the `build2 version` module to simplify and automate version managements. The `version` module has two primary functions: eliminate the need to change the version anywhere except in the project's manifest file and automatically extract and propagate the snapshot information (serial number and id).

The `version` module must be loaded in the project's `bootstrap.build`. While being loaded, it reads the project's manifest and extracts its version (which must be in the standard form). The version is then parsed and presented as the following build system variables (which can be used in the buildfiles):

```
[string] version          # 2~1.2.3-b.4.1234567.deadbeef+3
[string] version.project  # 1.2.3-b.4.1234567.deadbeef
[uint64] version.project_number # 0010020025041
[string] version.project_id # 1.2.3-b.4.deadbeef
```

```

[bool]    version.stub                # false (true for 0[+<revision>])

[uint64]  version.epoch               # 2

[uint64]  version.major               # 1
[uint64]  version.minor               # 2
[uint64]  version.patch               # 3

[bool]    version.alpha               # false
[bool]    version.beta                # true
[bool]    version.pre_release         # true
[string]  version.pre_release_string  # b.4
[uint64]  version.pre_release_number  # 4

[bool]    version.snapshot            # true
[uint64]  version.snapshot_sn         # 1234567
[string]  version.snapshot_id         # deadbeef
[string]  version.snapshot_string     # 1234567.deadbeef

[uint64]  version.revision            # 3

```

If the version is the latest snapshot (that is, it's in the `.z` form), then the `version` module extracts the snapshot information from the version control system used by the project. Currently only `git` is supported with the following semantics.

If the project's source directory (`src_root`) is clean (that is, it does not have any changed or untracked files), then the `HEAD` commit date and id are used as the snapshot `sn` and `id`, respectively. Otherwise, the snapshot is left in the `.z` form (which signals the latest/uncommitted snapshot). While we can work with such a `.z` snapshot locally, preparing a distribution of such an uncommitted snapshot is an error.

The use of `git` commit dates for snapshot ordering has its limitations: they have one second resolution which means it is possible to create two commits with the same date (but not the same commit id and thus snapshot id). We also need all the committers to have a reasonably accurate clock. Note, however, that in case of a commit date clash/ordering issue, we still end up with distinct versions (because of the commit id) – they are just not ordered correctly. As a result, we feel that the risks are justified when the only alternative is manual version management (which is always an option, nevertheless).

When we prepare a distribution of a snapshot, the `version` module automatically adjusts the package name to include the snapshot information as well as patches the manifest file in the distribution with the snapshot `sn` and `id` (that is, replacing `.z` in the version value with the actual snapshot information). The result is a package that is specific to this commit.

Besides extracting the version information and making it available as individual components, the `version` module also provide rules for automatically generating the `version` (or `Version/VERSION`) file that is customarily found in the root of a project as well as the version headers (or other similar version-based files).

The `version` file rule matches a `doc` target that contains the `version` substring in its name (comparison is case-insensitive) and that depends on the project's `manifest` file. To utilize this rule you would normally have something along these lines to your project's root

buildfile:

```
./: ... doc{version}

doc{version}: file{manifest} # Generated by the version module.
doc{version}: dist = true    # Include into the distribution.
```

The version header rule pre-processes a template file (which means it can be used to generate any kinds of files, not just C/C++ headers). It matches a file-based target that has a corresponding in prerequisite and also depends on the project's manifest file. As an example, let's assume we want to auto-generate a header called `version.hxx` for our `libhello` library. To accomplish this we add the `version.hxx.in` template as well as something along these lines to our buildfile:

```
lib{hello}: ... hxx{version}

hxx{version}: in{version} $src_root/file{manifest}
hxx{version}: dist = true
```

The header rule is a line-based pre-processor that substitutes fragments enclosed between (and including) a pair of dollar signs (\$) with \$\$ being the escape sequence. As an example, let's assume our `version.hxx.in` contains the following lines:

```
#ifndef LIBHELLO_VERSION

#define LIBHELLO_VERSION      $libhello.version.project_number$ULL
#define LIBHELLO_VERSION_STR "$libhello.version.project$"

#endif
```

If our `libhello` is at version 1.2.3, then the generated `version.hxx` will look like this:

```
#ifndef LIBHELLO_VERSION

#define LIBHELLO_VERSION      10020030000ULL
#define LIBHELLO_VERSION_STR "1.2.3"

#endif
```

The first component after the opening \$ should be either the name of the project itself (like `libhello` above) or a name of one of its dependencies as listed in the manifest. If it is the project itself, then the rest can refer to one of the `version.*` variables that we discussed earlier (in reality it can be any variable visible from the project's root scope).

If the name refers to one of the dependencies (that is, projects listed with `depends:` in the manifest), then the following special substitutions are recognized:

```
$<name>.version$           - textual version constraint
$<name>.condition(<VERSION>[,<SNAPSHOT>])$ - numeric satisfaction condition
$<name>.check(<VERSION>[,<SNAPSHOT>])$     - numeric satisfaction check
```

Here *VERSION* is the version number macro and the optional *SNAPSHOT* is the snapshot number macro. The snapshot is only required if you plan to include snapshot information in your dependency constraints.

As an example, let's assume our `libhello` depends on `libprint` which is reflected with the following line in our manifest:

```
depends: libprint >= 2.3.4
```

We also assume that `libprint` provides its version information in the `libprint/version.hxx` header and uses analogous-named macros. Here is how we can add a version check to our `version.hxx.in`:

```
#ifndef LIBHELLO_VERSION

#define LIBHELLO_VERSION      $libhello.version.project_number$ULL
#define LIBHELLO_VERSION_STR "$libhello.version.project$"

#include <libprint/version.hxx>

$libprint.check(LIBPRINT_VERSION) $

#endif
```

After the substitution our `version.hxx` header will look like this:

```
#ifndef LIBHELLO_VERSION

#define LIBHELLO_VERSION      10020030000ULL
#define LIBHELLO_VERSION_STR "1.2.3"

#include <libprint/version.hxx>

#ifdef LIBPRINT_VERSION
# if !(LIBPRINT_VERSION >= 20030040000ULL)
#   error incompatible libprint version, libprint >= 2.3.4 is required
# endif
#endif

#endif
```

The version and condition substitutions are the building blocks of the check substitution. For example, here is how we can implement a check with a customized error message:

```
#if !($libprint.condition(LIBPRINT_VERSION) $)
#   error bad libprint, need libprint $libprint.version$
#endif
```

The version module also treats one dependency in a special way: if you specify the required version of the build system in your manifest, then the module will automatically check it for you. For example, if we have the following line in our manifest:

```
depends: * build2 >= 0.5.0
```

And someone tries to build our project with `build2 0.4.0`, then they will see an error like this:

```
build/bootstrap.build:3:1: error: incompatible build2 version
  info: running 0.4.0
  info: required 0.5.0
```

What version constraints should be use when depending on other project. We start with a simple case where we depend on a release. Let's say `libprint 2.3.0` added a feature that we need in our `libhello`. If `libprint` follows the source/binary compatibility guidelines discussed above, then any `2.X.Y` version should work provided $X \geq 3$. And this how we can specify it in the manifest:

```
depends: libprint [2.3.0 3.0.0-)
```

Let's say we are now working on `libhello 2.0.0` and would like to start using features from `libprint 3.0.0`. However, currently, only pre-releases of `3.0.0` are available. If you would like to add a dependency on a pre-release (most likely from your own pre-release), then the recommendation is to only allow a specific version, essentially "expiring" the combination as soon as newer versions become available. For example:

```
version: 2.0.0-b.1
depends: libprint == 3.0.0-b.2
```

Finally, let's assume we are feeling adventurous and would like to test development snapshots of `libprint` (most likely from our own snapshots). In this case the recommendation is to only allow a snapshot range for a specific pre-release with the understanding and a warning that no compatibility between snapshot versions is guaranteed. For example:

```
version: 2.0.0-b.1.z
depends: libprint [3.0.0-b.2.1 3.0.0-b.3)
```