ZenMake Documentation

Release 0.11.0

Alexander Magola

2022-09-04

Contents

ZenMake - build system based on the meta build system Waf.

CHAPTER 1

Introduction

1.1 What is it?

ZenMake is a cross-platform build system for C/C++ and some other languages. It uses meta build system Waf as a framework.

Some reasons to create this project can be found *here*.

1.2 Main features

- Build config as python (.py) or as yaml file. Details are *here*.
- Distribution as zip application or as system package (pip). See *Installation*.
- Automatic reconfiguring: no need to run command 'configure'.
- Compiler autodetection.
- Building and running functional/unit tests including an ability to build and run tests only on changes. Details are *here*.
- Build configs in sub directories.
- Building external dependencies.
- Supported platforms: GNU/Linux, MacOS, MS Windows. Some other platforms like OpenBSD/FreeBSD should work as well but it hasn't been tested.
- Supported languages:
  - C: gcc, clang, msvc, icc, xlc, suncc, irixcc
  - C++: g++, clang++, msvc, icpc, xlc++, sunc++
  - D: dmd, ldc2, gdc; MS Windows is not supported yet
  - Fortran: gfortran, ifort (should work but not tested)

  - Assembler: gas (GNU Assembler)
- Supported toolkits/frameworks: - SDL v2 (Linux only) - GTK v3 (Linux only) - Qt v5

1.3 Plans to do

There is no clear roadmap for this project. I add features that I think are needed to include.

1.4 Project links

- Primary git repository: https://github.com/pustotnik/zenmake
- Secondary git repository: https://gitlab.com/pustotnik/zenmake
- Issue tracker: https://github.com/pustotnik/zenmake/issues
- Pypi package: https://pypi.org/project/zenmake
- Documentation: https://zenmake.readthedocs.io

CHAPTER 2

Why?

Short answer: because I could and wanted.

Long answer is below.

https://news.ycombinator.com/item?id=18789162

```
Cool. One more "new" build system...
```

Yes, I know, we already have a lot of them. I decided to create this project because I couldn't find a build tool for Linux which is quick and easy to use, flexible, ready to use, with declarative configuration, without the need to learn one more special language and suitable for my needs. I know about lots of build systems and I have tried some of them.

Well, a little story of the project. In 2010 year I developed a build system in a company where I was working that time. It was a build system based on Waf and it was used successfully for linux projects several years. But that system had a lot of internal problems and I wanted to remake it from scratch. And in 2013 year I tried to begin a new project. But I had no time to develop it at that time. Then, in 2019 year I decided to make some own opensorce project and was selecting a build system for my project. I was considering only opensource cross-platform build systems that can build C/C++ projects on GNU/Linux. Firstly I tried CMake, then Meson and Waf. Also I was looking at some other build systems like Bazel. Eventually, I concluded that I had to try to make my own build tool.

I would do it mostly for myself, but I would be glad if my tool was useful for others.

CHAPTER 3

Quickstart guide

To use ZenMake you need *ZenMake* and *buildconf* file in the root of your project.

Let's consider an example with this structure:

```
testproject
├── buildconf.yml
├── prog
│   └── test.cpp
└── shlib
    ├── util.cpp
    └── util.h
```

For this project `buildconf.yml` can be like that:

```yaml
tasks:
  util :
    features : cxxshlib
    source   : 'shlib/**/*.cpp'
    includes : '.'
  program :
    features : cxxprogram
    source   : 'prog/**/*.cpp'
    includes : '.'
    use      : util

buildtypes:
  debug :
    toolchain : clang++
    cxxflags  : -O0 -g
  release :
    toolchain : g++
    cxxflags  : -O2
  default : debug
```

Lines	Description
1	Section with build tasks
2,6	Names of build tasks. By default they are used as target names. Resulting target names will be adjusted depending on a platform. For example, on Windows 'program' will result to 'program.exe'.
3	Mark build task as a C++ shared library.
4	Specify all *.cpp files in the directory 'shlib' recursively.
5,9	Specify the path for C/C++ headers relative to the project root directory. In this example, this parameter is optional as ZenMake adds the project root directory itself. But it's an example.
7	Mark build task as a C++ executable.
8	Specify all *.cpp files in the directory 'prog' recursively.
10	Specify task 'util' as dependency to task 'program'.
12	Section with build types.
13,16	Names of build types. They can be almost any.
14	Specify Clang C++ compiler for debug.
15	Specify C++ compiler flags for debug.
17	Specify g++ compiler (from GCC) for release.
18	Specify C++ compiler flags for release.
19	Special case: specify default build type that is used when no build type was specified for ZenMake command.

In case of using python config the file `buildconf.py` with the same values as above would look like this:

```
tasks = {
    'util' : {
        'features' : 'cxxshlib',
        'source'   : 'shlib/**/*.cpp',
        'includes' : '.',
    },
    'program' : {
        'features' : 'cxxprogram',
        'source'   : 'prog/**/*.cpp',
        'includes' : '.',
        'use'      : 'util',
    },
}

buildtypes = {
    'debug' : {
        'toolchain' : 'clang++',
        'cxxflags'  : '-O0 -g',
    },
    'release' : {
        'toolchain' : 'g++',
        'cxxflags'  : '-O2',
    },
    'default' : 'debug',
}
```

Once you have the config, run `zenmake` in the root of the project and ZenMake does the build:

```
$ zenmake
* Project name: 'testproject'
* Build type: 'debug'
Setting top to                           : /tmp/testproject
Setting out to                           : /tmp/testproject/build
Checking for 'clang++'                   : /usr/lib/llvm/11/bin/clang++
```

(continues on next page)

(continued from previous page)

```
[1/4] Compiling shlib/util.cpp
[2/4] Compiling prog/test.cpp
[3/4] Linking build/debug/libutil.so
[4/4] Linking build/debug/program
'build' finished successfully (0.531s)
```

Running ZenMake without any parameters in a directory with `buildconf.py` or `buildconf.yml` is the same as running `zenmake build`. Otherwise it's the same as `zenmake help`.

Get the list of all commands with a short description using `zenmake help` or `zenmake --help`. To get help on selected command you can use `zenmake help <selected command>` or `zenmake <selected command> --help`

For example to build `release` of the project above such a command can be used:

```
$ zenmake build -b release
* Project name: 'testproject'
* Build type: 'release'
Setting top to                           : /tmp/testproject
Setting out to                           : /tmp/testproject/build
Checking for 'g++'                       : /usr/bin/g++
[1/4] Compiling shlib/util.cpp
[2/4] Compiling prog/test.cpp
[3/4] Linking build/release/libutil.so
[4/4] Linking build/release/program
'build' finished successfully (0.498s)
```

Here is some possible variant of extended version of the config from above:

```
tasks:
  util :
    features : cxxshlib
    source   : 'shlib/**/*.cpp'
    includes : '.'
    libs     : boost_timer # <-- Add the boost timer library as dependency
  program :
    features : cxxprogram
    source   : 'prog/**/*.cpp'
    includes : '.'
    use      : util

buildtypes:
  debug :
    toolchain : clang++
    cxxflags  : -O0 -g
  release :
    toolchain : g++
    cxxflags  : -O2
  default : debug

configure:
  - do: check-headers
    names : cstdio iostream # <-- Check C++ 'cstdio' and 'iostream' headers
  - do: check-libs          # <-- Check all libraries from the 'libs' parameter
```

One of the effective and simple ways to learn something is to use real examples. So it is recommended to look at examples in `demos` directory which can be found in the repository here.

CHAPTER 4

Installation

Dependencies

- Python >=3.5. Python must have threading support. Python has threading in most cases while nobody uses `--without-threads` for Python building. Python >= 3.7 always has threading.
- PyYAML It's optional and needed only if you use yaml *buildconf*. ZenMake can be used with yaml buildconf file even with PyYAML not installed in an operating system because ZenMake has an internal copy of PyYAML python library. This copy is used only if there is no PyYAML installed in an operating system.

There are different ways to install/use ZenMake:

4.1 Via python package (pip)

ZenMake has its own python package. You can install it by:

```
pip install zenmake
```

In this way pip will install PyYAML if it's not installed already.

Note: `POSIX`: It requires root and will install it system-wide. Alternatively, you can use:

```
pip install --user zenmake
```

which will install it for your user and does not require any special privileges. This will install the package in ~/.local/, so you will have to add ~/.local/bin to your PATH.

`Windows`: It doesn't always require administrator rights.

Note: You need to have `pip` installed. Most of the modern Linux distributions have pip in their packages. On Windows you can use, for example, chocolatey to install pip. Common instructions to install pip can be found here.

Note: You can install ZenMake with pip and virtualenv. In this case you don't touch system packages and it doesn't require root privileges.

After installing you can run ZenMake just by typing:

```
zenmake
```

4.2 Via git

You can use ZenMake from Git repository. But branch `master` can be broken. Also, you can just to switch to the required version using git tag. Each version of ZenMake has a git tag. The body of ZenMake application is located in `src/zenmake` path in the repository. You don't need other directories and files in repository and you can remove them if you want. Then you can make symlink to `src/zenmake/zmrun.py`, shell alias or make executable .sh script (for Linux/MacOS/..) or .bat (for Windows) to run ZenMake. Example for Linux (`zmrepo` is custom directory):

```
$ mkdir zmrepo
$ cd zmrepo
$ git clone https://github.com/pustotnik/zenmake.git .
```

Next step is optional. Switch to existing version, for example to 0.7.0:

```
$ git checkout v0.7.0
```

Here you can make symlink/alias/script to run zenmake.

Other options to run ZenMake:

```
$ <path-to-zenmake-repo>/src/zenmake/zmrun.py
```

or:

```
$ python <path-to-zenmake-repo>/src/zenmake
```

4.3 As a zip application

Zenmake can be run as an executable python zip application. And ZenMake can make such zipapp with the command `zipapp`. Using steps from *Via Git* you can run:

```
$ python src/zenmake zipapp
$ ls *.pyz
zenmake.pyz
$ ./zenmake.pyz
...
```

Resulting file `zenmake.pyz` can be run standalone without the repository and pip. You can copy `zenmake.pyz` to the root of your project and distribute this file with your project. It can be used on any supported platform and doesn't require any additional access and changes in your system.

Note: Since ZenMake 0.10.0 you can download ready to use `zenmake.pyz` from GitHub releases.

CHAPTER 5

Build config

ZenMake uses build configuration file with name `buildconf.py` or `buildconf.yaml`/`buildconf.yml`. First variant is a regular python file and second one is an YAML file. ZenMake doesn't use both files in one directory at the same time. If both files exist in one directory then only `buildconf.py` will be used. Common name `buildconf` is used in this manual.

The format for both config files is the same. YAML variant is a little more readable but in python variant you can add a custom python code if you wish.

Simplified scheme of buildconf is:

```
startdir = path
buildroot = path
realbuildroot = path
project = { ... }
general = { ... }
cliopts = { ... }
conditions = { ... }
tasks = { name: task parameters }
buildtypes = { name: task parameters }
toolchains = { name: parameters }
byfilter = [ { for: {...}, set: task parameters }, ... ]
subdirs = []
edeps = { ... }
```

Also see *syntactic sugar*.

Where symbols '{}' mean an associative array/dictionary and symbols '[]' mean a list. In python notation '{}' is known as dictionary. In some other languages it's called an associative array including YAML (Of course YAML is not programming language but it's markup language). For shortness it's called a `dict` here.

Not all variables are required in the scheme above but buildconf cannot be empty. All variables have reserved names and they all are described here. Other names in buildconf are just ignored by ZenMake (excluding *substitution variables*) if present and it means they can be used for some custom purposes.

Note: About paths in general.

You can use native paths but it's recommended to use wherever possible POSIX paths (Symbol / is used as a separator in a path). With POSIX paths you will ensure the same paths on different platforms/operating systems. POSIX paths will be converted into native paths automatically, but not vice versa. For example, path 'my/path' will be converted into 'my\path' on Windows. Also it's recommended to use relative paths wherever possible.

Warning: `Windows only`: do NOT use short filename notation (8.3 filename) for paths in buildconf files. It can cause unexpected errors.

Below is the detailed description of each buildconf variable.

5.1 startdir

A start path for all paths in a buildconf. It is `.` by default. The path can be absolute or relative to directory where current buildconf file is located. It means that by default all other relative paths in the current buildconf file are considered as the paths relative to directory with the current buildconf file. But you can change this by setting a different value to this variable.

5.2 buildroot

A path to the root of a project build directory. By default it is directory 'build' in the directory with the top-level buildconf file of the project. Path can be absolute or relative to the *startdir*. It is important to be able to remove the build directory safely, so it should never be given as `.` or `..`.

Note: If you change value of `buildroot` with already using/existing build directory then ZenMake will not touch previous build directory. You can remove previous build directory manually or run command `distclean` before changing of `buildroot`. ZenMake cannot do it because it stores all meta information in current build directory and if you change this directory it will lose all such an information.

This can be changed in the future by storing extra information in some other place like user home directory but now it is.

5.3 realbuildroot

A path to the real root of a project build directory and by default it is equal to value of `buildroot`. It is optional parameter and if `realbuildroot` has different value from `buildroot` then `buildroot` will be symlink to `realbuildroot`. Using `realbuildroot` makes sense mostly on linux where '/tmp' is usually on tmpfs filesystem nowadays and it can used to make building in memory. Such a way can improve speed of building. Note that on Windows OS the process of ZenMake needs to be started with enabled "Create symbolic links" privilege and usual user doesn't have a such privilege. Path can be absolute or relative to the *startdir*. It is important to be able to remove the build directory safely, so it should never be given as `.` or `..`.

5.4 project

A *dict* with some parameters for the project. Supported values:

name The name of the project. It's name of the top-level *startdir* directory by default.

version The version of the project. It's empty by default. It's used as default value for *ver-num* field if not empty.

5.5 general

A *dict* array with some general features. Supported values:

autoconfig Execute the command `configure` automatically in the command `build` if it's necessary. It's `True` by default. Usually you don't need to change this value.

monitor-files Set extra file paths to check changes in them. You can use additional files with your buildconf file(s). For example it can be extra python module with some tools. But in this case ZenMake doesn't know about such files when it checks buildconf file(s) for changes to detect if it must call command `configure` for feature `autoconfig`. You can add such files to this variable and ZenMake will check them for changes as it does so for regular buildconf file(s).

If paths contain spaces and all these paths are listed in one string then each such a path must be in quotes.

hash-algo Set hash algorithm to use in ZenMake. It can be `sha1` or `md5`. By default ZenMake uses sha1 algorithm to control changes of config/built files and for some other things. Sha1 has much less collisions than md5 and that's why it's used by default. Modern CPUs often has support for this algorithm and sha1 show better or almost the same performance than md5 in this cases. But in some cases md5 can be faster and you can set here this variant. However, don't expect big difference in performance of ZenMake. Also, if a rare "FIPS compliant" build of Python is used it's always sha1 anyway.

db-format Set format for internal ZenMake db/cache files. Use one of possible values: `py`, `pickle`, `msgpack`.

The value `py` means text file with python syntax. It is not fastest format but it is human readable one.

The value `pickle` means python pickle binary format. It has good performance and python always supports this format.

The value `msgpack` means msgpack binary format by using python module `msgpack`. Using of this format can decrease ZenMake overhead in building of some big projects because it has the best performance among all supported formats. If the package `msgpack` doesn't exist in the current system then the `pickle` value will be used. Note: ZenMake doesn't try to install package `msgpack`. This package must be installed in some other way.

The default value is `pickle`.

provide-edep-targets Provide target files of *external dependencies* in the *buildroot* directory. It is useful to run built files from the build directory without the need to use such a thing as LD_LIBRARY_PATH for each dependency. Only existing and used target files are provided. Static libraries are also ignored because they are not needed to run built files. On Windows ZenMake copies these files while on other OS (Linux, MacOS, etc) it makes symlinks.

It's `False` by default.

build-work-dir-name Set a name of work directory which is used mostly for object files during compilation. This directory seperates resulting target files from other files in a buildtype directory to avoid file/directory conflicts. Usually you don't need to set this parameter until some target name has conflict with default value of this parameter.

The default value is `@bld`.

5.6 cliopts

A *dict* array with default values for command line options. It can be any existing command line option that ZenMake has. If you want to set an option for selected commands then you can set it in the format of a *dict* where key is a name of specific command or special value 'any' which means any command. If some command doesn't have selected option then it will be ignored.

Example in YAML format:

```
cliopts:
  verbose: 1
  jobs : { build : 4 }
  progress :
    any: false
    build: true
```

Note: Selected command here is a command that is used on command line. It means if you set an option for the `build` command and zenmake calls the `configure` command before this command by itself then this option will be applied for both `configure` and `build` commands. In other words it is like you are running this command with this option on command line.

5.7 conditions

A *dict* with conditions for *selectable parameters*.

5.8 tasks

A *dict* with build tasks. Each task has own unique name and *parameters*. Name of task can be used as dependency id for other build tasks. Also this name is used as a base for resulting target file name if parameter `target` is not set in *task parameters*. In this variable you can set up build parameters particularly for each build task. Example in YAML format:

```
tasks:
  mylib :
    # some task parameters
  myexe :
    # some task parameters
    use : mylib
```

Note: Parameters in this variable can be overridden by parameters in *buildtypes* and/or *byfilter*.

Note: Name of a task cannot contain symbol `:`. You can use parameter `target` if you want to have this symbol in resulting target file names.

5.9 buildtypes

A *dict* with build types like `debug`, `release`, `debug-gcc` and so on. Here is also a special value with name `default` that is used to set default build type if nothing is specified. Names of these build types are just names, they can be any name but not `default`. Also you should remember that these names are used as directory names. So don't use incorrect symbols if you don't want a problem with it.

This variable can be empty or absent. In this case current buildtype is always just an empty string.

Possible parameters for each build type are described in *task parameters*.

Special value `default` must be a string with the name of one of the build types or a *dict* where keys are supported name of the host operating system and values are strings with the names of one of the build types. Special key '_' or 'no-match' can be used in the `default` to define a value that will be used if the name of the current host operating system is not found among the keys in the `default`.

Valid host operating system names: `linux`, `windows`, `darwin`, `freebsd`, `openbsd`, `sunos`, `cygwin`, `msys`, `riscos`, `atheos`, `os2`, `os2emx`, `hp-ux`, `hpux`, `aix`, `irix`.

Note: Only `linux`, `windows` and `darwin` are tested.

Examples in YAML format:

```yaml
buildtypes:
  debug        : { toolchain: auto-c++ }
  debug-gcc    : { toolchain: g++, cxxflags: -fPIC -O0 -g }
  release-gcc  : { toolchain: g++, cxxflags: -fPIC -O2 }
  debug-clang  : { toolchain: clang++, cxxflags: -fPIC -O0 -g }
  release-clang: { toolchain: clang++, cxxflags: -fPIC -O2 }
  debug-msvc   : { toolchain: msvc, cxxflags: /Od /EHsc }
  release-msvc : { toolchain: msvc, cxxflags: /O2 /EHsc }
  default: debug

buildtypes:
  debug:
    toolchain.select:
      default: g++
      darwin: clang++
      windows: msvc
    cxxflags.select:
      default : -O0 -g
      msvc    : /Od /EHsc
  release:
    toolchain.select:
      default: g++
      darwin: clang++
```

(continues on next page)

(continued from previous page)

```
      windows: msvc
    cxxflags.select:
      default : -O2
      msvc    : /O2 /EHsc
  default: debug

buildtypes:
  debug-gcc    : { cxxflags: -O0 -g }
  release-gcc  : { cxxflags: -O2 }
  debug-clang  : { cxxflags: -O0 -g }
  release-clang: { cxxflags: -O2 }
  debug-msvc   : { cxxflags: /Od /EHsc }
  release-msvc : { cxxflags: /O2 /EHsc }
  default:
    _: debug-gcc
    linux: debug-gcc
    darwin: debug-clang
    windows: debug-msvc
```

Note: Parameters in this variable override corresponding parameters in *tasks* and can be overridden by parameters in *byfilter*.

5.10 toolchains

A *dict* with custom toolchain setups. It's useful for simple cross builds for example, or for custom settings for existing toolchains. Each value has unique name and parameters. Parameters are also dict with names of environment variables and special name `kind` that is used to specify the type of toolchain/compiler. Environment variables are usually such variables as `CC`, `CXX`, `AR`, etc that are used to specify name or path to existing toolchain/compiler. Path can be absolute or relative to the *startdir*. Also such variables as `CFLAGS`, `CXXFLAGS`, etc can be set here.

Names of toolchains from this parameter can be used as a value for the `toolchain` in *task parameters*.

Example in YAML format:

```
toolchains:
  custom-g++:
    kind : auto-c++
    CXX  : custom-toolchain/gccemu/g++
    AR   : custom-toolchain/gccemu/ar
  custom-clang++:
    kind : clang++
    CXX  : custom-toolchain/clangemu/clang++
    AR   : custom-toolchain/clangemu/llvm-ar
  g++:
    LINKFLAGS : -Wl,--as-needed
```

5.11 byfilter

This variable describes extra/alternative way to set up build tasks. It's a list of *dicts* with attributes `set` and `for`, `not-for` and/or `if`. Attributes `for`/`not-for`/`if` describe conditions for parameters in attribute

`set`, that is, a filter to set some build task parameters. The attribute `for` is like a `if a` and the attribute `not-for` is like a `if not b` where `a` and `b` are some conditions. And they are like a `if a and if not b` when both of them exist in the same item. The attribute `not-for` has higher priority in the case of the same condition in the both of them.

Since v0.11 ZenMake supports `if` attribute where you can set a string with python like expression.

The `for`/`not-for` are dicts and `if` is an expression. In `for`/`not-for`/`if` you can use such variables as dict keys in `for`/`not-for` and as keywords within an expression:

task Build task name or list of build task names. It can be existing task(s) from *tasks* or new (only in `for`).

buildtype Build type or list of build types. It can be existing build type(s) from *buildtypes* or new (only in `for`).

platform Name of a host platform/operating system or list of them. Valid values are the same as for `default` in *buildtypes*.

The `if` is a real python expression with some builtin functions. You can use standard python operators as '(', ')', 'and', 'or', 'not', '==', '!=' and 'in'. ZenMake supports a little set of extensions as well:

Name	Description
true	The same as python 'True'.
false	The same as python 'False'.
startswith(str, prefix)	Returns true if 'str' starts with the specified 'prefix'.
endswith(str, prefix)	Returns true if 'str' ends with the specified 'suffix'.

The attribute `set` has value of the *task parameters*.

Other features:

- If some key parameter is not specified in `for`/`not-for`/`if` it means that this is for all possible values of this kind of condition. For example if it has no `task` it means 'for all existing tasks'. Special word `all` (without any other parameters) can be used to indicate that current item must be applied to all build tasks. Empty dict (i.e. `{}`) in `for`/`not-for` can be used for the same reason as well.
- Variable 'byfilter' overrides all matched values defined in *tasks* and *buildtypes*.
- Items in `set` with the same names and the same conditions in `for`/`not-for`/`if` override items defined before.
- If `for`/`not-for` and `if` are used for the same `set` then result will be the intersection of resulting sets from `for`/`not-for` and `if`.
- When `set` is empty or not defined it does nothing.

Note: ZenMake applies every item in the `byfilter` in the order as they were written.

It's possible to use `byfilter` without *tasks* and *buildtypes*.

Example in YAML format:

```
GCC_BASE_CXXFLAGS: -std=c++11 -fPIC

buildtypes:
debug-gcc    : { cxxflags: $GCC_BASE_CXXFLAGS -O0 -g }
release-gcc  : { cxxflags: $GCC_BASE_CXXFLAGS -O2 }
```

(continues on next page)

(continued from previous page)

```
debug-clang  : { cxxflags: $GCC_BASE_CXXFLAGS -O0 -g }
release-clang: { cxxflags: $GCC_BASE_CXXFLAGS -O2 }
debug-msvc   : { cxxflags: /Od /EHsc }
release-msvc : { cxxflags: /O2 /EHsc }
default:
  _: debug-gcc
  linux: debug-gcc
  darwin: debug-clang
  windows: debug-msvc

byfilter:
  - for: all
    set: { includes: '.', rpath : '.', }

  - for: { task: shlib shlibmain }
    set: { features: cxxshlib }

  - for: { buildtype: debug-gcc release-gcc, platform: linux }
    set:
      toolchain: g++
      linkflags: -Wl,--as-needed

  - for: { buildtype: release-gcc }
    not-for : { platform : windows }
    set: { cxxflags: -fPIC -O3 }

  - for: { buildtype: [debug-clang, release-clang], platform: linux darwin }
    set: { toolchain: clang++ }

  - if: endswith(buildtype, '-gcc') and platform == 'linux'
    set:
      toolchain: g++
      linkflags: -Wl,--as-needed

  - if: buildtype == 'release-gcc' and platform == 'linux'
    set:
      cxxflags: $GCC_BASE_CXXFLAGS -O3

  - if: endswith(buildtype, '-clang') and platform in ('linux', 'darwin')
    set:
      toolchain: clang++

  - if: endswith(buildtype, '-msvc') and platform == 'windows'
    set:
      toolchain: msvc
```

Note: Parameters in this variable override corresponding parameters in *tasks* and in *buildtypes*.

5.12 subdirs

This variable controls including buildconf files from other sub directories of the project.

- If it is list of paths then ZenMake will try to use this list as paths to sub directories with the buildconf

files and will use all found ones. Paths can be absolute or relative to the *startdir*.

- If it is an empty list or just absent at all then ZenMake will not try to use any sub directories of the project to find buildconf files.

Example in Python format:

```
subdirs = [
    'libs/core',
    'libs/engine',
    'main',
]
```

Example in YAML format:

```
subdirs:
    - libs/core
    - libs/engine
    - main
```

See some details *here*.

5.13 edeps

A *dict* with configurations of external non-system dependencies. Each such a dependency has own unique name which can be used in task parameter *use*.

See full description of parameters *here*. Description of external dependencies is *here*.

Note: More examples of buildconf files can be found in repository here.

CHAPTER 6

Build config: task parameters

It's a *dict* as a collection of build task parameters for a build task. This collection is used in *tasks*, *buildtypes* and *byfilter*. And it's core buildconf element.

Also see *substitutions* for string/text values.

6.1 features

It describes type of a build task. Can be one value or list of values. Supported values:

c Means that the task has a C code. Optional in most cases. Also it's 'lang' feature for C language.

cxx Means that the task has a C++ code. Optional in most cases. Also it's 'lang' feature for C++ language.

d Means that the task has a D code. Optional in most cases. Also it's 'lang' feature for D language.

fc Means that the task has a Fortran code. Optional in most cases. Also it's 'lang' feature for Fortran language.

asm Means that the task has an Assembler code. Optional in most cases. Also it's 'lang' feature for Assembler language.

<lang>stlib Means that result of the task is a static library for the <lang> code. For example: `cstlib`, `cxxstlib`, etc.

<lang>shlib Means that result of the task is a shared library for the <lang> code. For example: `cshlib`, `cxxshlib`, etc.

<lang>program Means that result of the task is an executable file for the <lang> code. For example: `cprogram`, `cxxprogram`, etc.

runcmd Means that the task has the `run` parameter and should run some command. It's optional because ZenMake detects this feature automatically by presence of the `run` in

task parameters. You need to set it explicitly only if you want to try to run <lang>program task without parameter `run`.

test Means that the task is a test. More details about tests are *here*. It is not needed to add `runcmd` to this feature because ZenMake adds `runcmd` itself if necessary.

qt5 Means that the task has Qt5 code. More details are *here*.

Some features can be mixed. For example `cxxprogram` can be mixed with `cxx` for C++ build tasks but it's not necessary because ZenMake adds `cxx` for `cxxprogram` itself. The `cxxshlib` feature cannot be mixed for example with the `cxxprogram` in one build task because they are different types of build task targets. Using of such features as `c` or `cxx` doesn't make sense without *stlib/*shlib/*program features in most cases. The `runcmd` and `test` features can be mixed with any feature.

Examples in YAML format:

```
features : cprogram
features : cxxshlib
features : cxxprogram runcmd
features : cxxprogram test
```

Examples in Python format:

```
'features' : 'cprogram'
'features' : 'cxxshlib'
'features' : 'cxxprogram runcmd'
'features' : 'cxxprogram test'
```

6.2 target

Name of resulting file. The target will have different extension and name depending on the platform but you don't need to declare this difference explicitly. It will be generated automatically. For example the `sample` for *shlib task will be converted into `sample.dll` on Windows and into `libsample.so` on Linux. By default it's equal to the name of the build task. So in most cases it is not needed to be set explicitly.

It's possible to use *selectable parameters* to set this parameter.

6.3 source

One or more source files for compiler/toolchain/toolkit. It can be:

- a string with one or more paths separated by space
- a *dict*, description see below
- a list of items where each item is a string with one or more paths or a dict

The `dict` type is used for `ant_glob` Waf function. Format of patterns for `ant_glob` can be found here https://waf.io/book/. Most significant details from there:

- Patterns may contain wildcards such as * or ?, but they are Ant patterns, not regular expressions.
- The symbol `**` enable recursion. Complex folder hierarchies may take a lot of time, so use with care.
- The '..' sequence does not represent the parent directory.

So such a `dict` can contain fields:

incl Ant pattern or list of patterns to include, required field.

excl Ant pattern or list of patterns to exclude, optional field.

ignorecase Ignore case while matching (False by default), optional field.

startdir Start directory for patterns, optional field. It must be relative to the *startdir* or an absolute path. By default it's '.', that is, it's equal to *startdir*.

ZenMake always adds several patterns to exclude files for any ant pattern. These patterns include *Default Excludes* from Ant patterns and some additional patterns like `**/*.swp`.

There is simplified form of ant patterns using: if string value contains '*' or '?' it will be converted into `dict` form to use patterns. See examples below.

Any path or pattern should be relative to the *startdir*. But for pattern (in dict) can be used custom `startdir` parameter.

Note: If paths contain spaces and all these paths are listed in one string then each such a path must be in quotes.

YAML: You can write a string without quotes (as a plain scalar) in many cases but there are some special symbols which cannot be used at the beginning without quotes, for example `*` and `?<space>`. So a value like `**/*.cpp` must be always in qoutes (`'` or `"`).

See details here: https://www.yaml.info/learn/quote.html.

Examples in YAML format:

```
# just one file
source : test.cpp

# list of two files
source : main.c about.c
# or
source : [main.c, about.c]

# get all *.cpp files in the 'startdir' recursively
source : { incl: '**/*.cpp' }
# or
source :
    incl: '**/*.cpp'
# or (shortest record with the same result)
source : '**/*.cpp'

# get all *.c and *.cpp files in the 'startdir' recursively
source :  { incl: '**/*.c **/*.cpp' }
# or (shorter record with the same result)
source : '**/*.c **/*.cpp'

# get all *.cpp files in the 'startdir'/mylib recursively
source :  mylib/**/*.cpp

# get all *.cpp files in the 'startdir'/src recursively
# but don't include files according pattern 'src/extra*'
source :
    incl: src/**/*.cpp
```

(continues on next page)

(continued from previous page)

```
    excl: src/extra*

# get all *.c files in the 'src' and in '../others' recursively
source :
    - src/**/*.c
    - incl: '**/*.c'
      startdir: ../others

# pattern with space, it's necessary to use both types of quotes here:
source : '"my prog/**/*.c"'

# two file paths with spaces
source : '"my shlib/my util.c" "my shlib/my util2.c"'
```

Examples in python format:

```
# just one file
'source' : 'test.cpp'

# list of two files
'source' : 'main.c about.c'
'source' : ['main.c', 'about.c'] # the same result

# get all *.cpp files in the 'startdir' recursively
'source' :  dict( incl = '**/*.cpp' )
# or
'source' :  { 'incl': '**/*.cpp' }
# or (shortest record with the same result)
'source' :  '**/*.cpp'

# get all *.c and *.cpp files in the 'startdir' recursively
'source' :  { 'incl': ['**/*.c', '**/*.cpp'] }
# or (shorter record with the same result)
'source' :  ['**/*.c', '**/*.cpp']

# get all *.cpp files in the 'startdir'/mylib recursively
'source' :  'mylib/**/*.cpp'

# get all *.cpp files in the 'startdir'/src recursively
# but don't include files according pattern 'src/extra*'
'source' :  dict( incl = 'src/**/*.cpp', excl = 'src/extra*' )

# get all *.c files in the 'src' and in '../others' recursively
'source'   : [
    'src/**/*.c',
    { 'incl': '**/*.c', 'startdir' : '../others' },
]

# pattern with space:
'source' : '"my prog/**/*.c"'

# two file paths with spaces
'source' : '"my shlib/my util.c" "my shlib/my util2.c"'
```

It's possible to use *selectable parameters* to set this parameter.

6.4 includes

Include paths are used by the C/C++/D/Fortran compilers for finding headers/files. Paths should be relative to *startdir* or absolute. But last variant is not recommended.

If paths contain spaces and all these paths are listed in one string then each such a path must be in quotes.

Examples in YAML format:

```
includes : myinclude
includes : include myinclude
includes : [ include, myinclude ]
```

It's possible to use *selectable parameters* to set this parameter.

This parameter can be *exported*.

6.5 toolchain

Name of toolchain/compiler to use in the task. It can be any system compiler that is supported by ZenMake or a toolchain from custom *toolchains*. There are also the special names for autodetecting in format `auto-*` where `*` is a 'lang' feature for programming language, for example `auto-c`, `auto-c++`, etc.

Known names for C: `auto-c`, `gcc`, `clang`, `msvc`, `icc`, `xlc`, `suncc`, `irixcc`.
Known names for C++: `auto-c++`, `g++`, `clang++`, `msvc`, `icpc`, `xlc++`, `sunc++`.
Known names for D: `auto-d`, `ldc2`, `gdc`, `dmd`.
Known names for Fortran: `auto-fc`, `gfortran`, `ifort`.
Known names for Assembler: `auto-asm`, `gas`, `nasm`.

Note: If you don't set `toolchain` then ZenMake will try to set `auto-*` itself according values in *features*.

In some rare cases this parameter can contain more than one value as a string with values separated by space or as list. For example, for case when C and Assembler files are used in one task, it can be `"gcc gas"`.

If toolchain from custom *toolchains* or some system toolchain contain spaces in their names and all these toolchains are listed in one string then each such a toolchain must be in quotes.

It's possible to use *selectable parameters* to set this parameter.

6.6 cflags

One or more compiler flags for C.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.7 cxxflags

One or more compiler flags for C++.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.8 dflags

One or more compiler flags for D.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.9 fcflags

One or more compiler flags for Fortran.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.10 asflags

One or more compiler flags for Assembler.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.11 cppflags

One or more compiler flags added at the end of compilation commands for C/C++.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.12 linkflags

One or more linker flags for C/C++/D/Fortran.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.13 ldflags

One or more linker flags for C/C++/D/Fortran at the end of the link command.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.14 aslinkflags

One or more linker flags for Assembler.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.15 arflags

Flags to give the archive-maintaining program.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.16 defines

One or more defines for C/C++/Assembler/Fortran.

Examples in YAML format:

```
defines : MYDEFINE

defines : [ ABC=1, DOIT ]

defines :
    - ABC=1
    - DOIT

defines : 'ABC=1 DOIT AAA="some long string"'
```

Examples in Python format:

```
'defines' : 'MYDEFINE'

'defines' : ['ABC=1', 'DOIT']

'defines' : 'ABC=1 DOIT AAA="some long string"'
```

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.17 use

This attribute enables the link against libraries (static or shared). It can be used for local libraries from other tasks or to declare dependencies between build tasks. Also it can be used to declare using of *external dependencies*. For external dependencies the format of any dependency in `use` must be: `dependency-name:target-reference-name`.

It can contain one or more the other task names.

If a task name contain spaces and all these names are listed in one string then each such a name must be in quotes.

Examples in YAML format:

```
use : util
use : util mylib
use : [util, mylib]
use : 'util "my lib"'
use : ['util', 'my lib']
use : util mylib someproject:somelib
```

Examples in Python format:

```
'use' : 'util'
'use' : 'util mylib'
'use' : ['util', 'mylib']
'use' : 'util "my lib"'
'use' : ['util', 'my lib']
'use' : 'util mylib someproject:somelib'
```

It can be used to specify libraries of qt5 as well. More details are *here*.

It's possible to use *selectable parameters* to set this parameter.

6.18 libs

One or more names of existing shared libraries as dependencies, without prefix or extension. Usually it's used to set system libraries.

If you use this parameter to specify non-system shared libraries for some task you may need to specify the same libraries for all other tasks which depend on the current task. For example, you set library 'mylib' to the task A but the task B has parameter `use` with 'A', then it's recommended to add 'mylib' to the parameter `libs` for the task B. Otherwise you can get link error `... undefined reference to ...` or something like that. Some other ways to solve this problem include using environment variable `LD_LIBRARY_PATH` or changing of /etc/ld.so.conf file. But usually last method is not recommended.

Example in YAML format:

```
libs : m rt
```

Example in Python format:

```
'libs' : 'm rt'
```

It's possible to use *selectable parameters* to set this parameter.

6.19 libpath

One or more additional paths to find libraries. Usually you don't need to set it.

If paths contain spaces and all these paths are listed in one string then each such a path must be in quotes.

Paths should be absolute or relative to *startdir*.

Examples in YAML format:

```
libpath : /local/lib
libpath : '/local/lib "my path"' # in case of spaces in a path
```

Examples in Python format:

```
'libpath' : '/local/lib'
'libpath' : '/local/lib "my path"' # in case of spaces in a path
```

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.20 monitlibs

One or more names from `libs` to monitor changes.

For example, a project has used some system library 'superlib' and once this library was upgraded by a system package manager. After that the building of the project will not make a relink with the new version of 'superlib' if no changes in the project which can trigger such a relink. Usually it is not a problem because a project is changed much more frequently than upgrading of system libraries during development.

Any names not from `libs` are ignored.

It can be True or False as well. If it is True then value of `libs` is used. If it is False then it means an empty list.

By default it's False.

Using of this parameter can slow down a building of some projects with a lot of values in this parameter. ZenMake uses sha1/md5 hashes to check changes of every library file.

It's possible to use *selectable parameters* to set this parameter.

6.21 stlibs

The same as `libs` but for static libraries.

It's possible to use *selectable parameters* to set this parameter.

6.22 stlibpath

The same as `libpath` but for static libraries.

It's possible to use *selectable parameters* to set this parameter.

Also this parameter can be *exported*.

6.23 monitstlibs

The same as `monitlibs` but for static libraries. It means it's affected by parameter `stlibs`.

It's possible to use *selectable parameters* variables to set this parameter.

6.24 moc

One or more header files (.h) with C++ class declarations with Q_OBJECT. These files are handled with Qt Meta-Object Compiler, moc. Format for this parameter is the same as for the *source* parameter.

You can specify header files without Q_OBJECT here because ZenMake filters such files by itself. So you can specify just all .h files of your directory with header files if you wish.

It can be used only for tasks with `qt5` in *features*.

6.25 rclangprefix

Value of `qresource prefix` in generated .qrc file for a qt5 task. When .ts files are specified in the *source* parameter ZenMake compiles these files into .qm files. If you set the `rclangprefix` parameter ZenMake will insert all compiled .qm files in .qrc file to embed .qm files as internal binary resourses inside compiled task target file. And the value of this parameter can be used in the QTranslator::load method in the 'directory' argument in your Qt5 code.

The *bld-langprefix*, *unique-qmpaths* and *install-langdir* parameters are ignored if the `rclangprefix` is set.

It can be used only for tasks with `qt5` in *features*.

6.26 langdir-defname

Name of a define to set for your Qt5 code to detect current directory with compiled .qm files to use in the QTranslator::load method. When .ts files are specified in the *source* parameter ZenMake compiles these files into .qm files. But when you use the `install` command ZenMake copies these files from build directory into install directory. So during regular building and for installed application the directory with .qm files are different. Value of the define with the name from the `langdir-defname` is the install directory of .qm files for the `install` command and the build directory of .qm files in other cases.

This parameter is ignored if *rclangprefix* is set.

It can be used only for tasks with `qt5` in *features*.

6.27 bld-langprefix

Set build directory path prefix for compiled .qm files. It is relative to *buildtypedir* and defaults to `@translations`. Usually you don't need to use this parameter.

This parameter is ignored if *rclangprefix* is set.

It can be used only for tasks with `qt5` in *features*.

6.28 unique-qmpaths

Make unique file paths for compiled .qm files by adding name of current buld task by the pattern: `$(buildtypedir)/<bld-langprefix>/<task name>_<original .qm file name>` where *buildtypedir* is the built-in variable. Usually you don't need to use this parameter.

This parameter is ignored if *rclangprefix* is set.

It can be used only for tasks with `qt5` in *features*.

6.29 rpath

One or more paths to hard-code into the binary during linking time. It's ignored on platforms that do not support it.

If paths contain spaces and all these paths are listed in one string then each such a path must be in quotes.

It's possible to use *selectable parameters* to set this parameter.

6.30 ver-num

Enforce version numbering on shared libraries. It can be used with *shlib `features` for example. It can be ignored on platforms that do not support it.

It's possible to use *selectable parameters* to set this parameter.

6.31 run

A *dict* with parameters to run something in the task. It' used with task features `runcmd` and `test`. It can be also just a string or a python function (for buildconf.py only). In this case it's the same as using dict with one parameter `cmd`.

cmd Command line to run. It can be any suitable command line. For convenience special *built-in substitution* variables `src` and `tgt` can be used here. The `tgt` variable contains string with the absolute path to resulting target file of the current task. And the `src` contains string with all source files of the task.

Environment variables also can be used here but see *bash-like substitutions*.

For python variant of buildconf it can be python function as well. In this case such a function gets one argument as a python dict with parameters:

taskname Name of current build task

startdir Current *startdir*

buildroot Root directory for building

buildtype Current buildtype

target Absolute path to resulting target. It may not be existing.

waftask Object of Waf class Task. It's for advanced use.

cwd Working directory where to run `cmd`. By default it's build directory for current buildtype. Path can be absolute or relative to the *startdir*.

env Environment variables for `cmd`. It's a `dict` where each key is a name of variable and value is a value of env variable.

timeout Timeout for `cmd` in seconds. It works only when ZenMake is run with python 3. By default there is no timeout.

shell If shell is True, the specified command will be executed through the shell. By default to avoid some common problems it is True. But in many cases it's safe to set False. In this case it avoids some overhead of using shell. In some cases it can be set to True by ZenMake/Waf even though you set it to False.

repeat Just amount of running of `cmd`. It's mostly for tests. By default it's 1.

If current task has parameter `run` with empty `features` or with only `runcmd` in the `features` then it is standalone runcmd task.

If current task is not standalone runcmd task then command from parameter `run` will be run after compilation and linking. If you want to have a command that will be called before compilation and linking you can make another standalone runcmd task and specify this new task in the parameter `use` of the current task.

By default ZenMake expects that any build task produces a target file and if it doesn't find this file when the task is finished it will throw an error. And it is true for standalone runcmd tasks also. If you want to create standalone runcmd task which doesn't produce target file you can set task parameter *target* to an empty string.

Examples in YAML format:

```yaml
echo:
    run: "echo 'say hello'"
    target: ''

test.py:
    run:
        cmd   : python tests/test.py
        cwd   : .
        env   : { JUST_ENV_VAR: qwerty }
        shell : false
    target: ''
    configure :
        - do: find-program
          names: python

shlib-test:
    features : cxxprogram test
    # ...
    run:
        cmd     : '$(tgt) a b c'
        env     : { ENV_VAR1: '111', ENV_VAR2: 'false' }
        repeat  : 2
        timeout : 10 # in seconds
        shell   : false

foo.luac:
    source : foo.lua
```

(continues on next page)

(continued from previous page)

```
    configure : [ { do: find-program, names: luac } ]
    run: '${LUAC} -s -o $(tgt) $(src)'
```

Examples in Python format:

```
'echo' : {
    'run' : "echo 'say hello'",
    'target': '',
},

'test.py' : {
    'run'       : {
        'cmd'   : 'python tests/test.py',
        'cwd'   : '.',
        'env'   : { 'JUST_ENV_VAR' : 'qwerty', },
        'shell' : False,
    },
    'target': '',
    'configure'  : [ dict(do = 'find-program', names = 'python'), ]
},

'shlib-test' : {
    'features' : 'cxxprogram test',
    # ...
    'run'       : {
        'cmd'     : '$(tgt) a b c',
        'env'     : { 'ENV_VAR1' : '111', 'ENV_VAR2' : 'false'},
        'repeat'  : 2,
        'timeout' : 10, # in seconds, Python 3 only
        'shell'   : False,
    },
},

'foo.luac' : {
    'source' : 'foo.lua',
    'configure' : [ dict(do = 'find-program', names = 'luac'), ],
    'run': '${LUAC} -s -o $(tgt) $(src)',
},
```

It's possible to use *selectable parameters* to set this parameter.

6.32 configure

A list of configuration actions (configuration checks and others). Details are *here*. These actions are called on **configure** step (in command **configure**).

It's possible to use *selectable parameters* to set this parameter.

Results of these configuration actions can be *exported* with the name *config-results*.

6.33 export-<param> / export

Some task parameters can be exported to all dependent build tasks.

There two forms: `export-<param>` and `export`.

In first form `<param>` is the name of the exported task parameter. The boolean True/False value or specific valid value to the `<param>` can be used to export. If value is True then ZenMake exports the value of the parameter from current task to all dependent build tasks. If value is False then ZenMake exports nothing.

Supported names: `includes`, `defines`, `config-results`, `libpath`, `stlibpath`, `moc` and all `*flags`.

But the parameter `export-config-results` accepts boolean True/False only value.

In second form it must be string or list with the names of parameters to export. Second form is simplified form of the first form when all values are True. And this form cannot be used to set specific value to export.

Note: By default ZenMake exports nothing (all values are False).

Exporting values are inserted in the beginning of the current parameter values in dependent tasks. It was made to have ability to overwrite parent values. For example, task A has `defines` with value `AAA=q` and task B depends on task A and has `defines` with value `BBB=v`. So if task A has `export-defines` with True, then actual value of `defines` in task B will be `AAA=q BBB=v`.

Examples in YAML format:

```
# export all includes from current task
export-includes: true
# the same result:
export: includes

# export all includes and defines from current task
export-includes: true
export-defines: true
# the same result:
export: includes defines

# export specific includes, value of parameter 'includes' from current
# task is not used
export-includes: incl1 incl2

# export specific defines, value of parameter 'defines' from current
# task is not used
export-defines  : 'ABC=1 DOIT AAA="some long string"'

# export results of all configuration actions from current task
export-config-results: true

# export all includes, defines and results of configuration actions
export: includes defines config-results
```

Specific remarks:

includes If specified paths contain spaces and all these paths are listed in one string then each such a path must be in quotes.

defines Defines from *configuration actions* are not exported. Use `export-config-results` or `export` with `config-results` for that.

It's possible to use *selectable parameters* (in strings) to set this parameter.

6.34 install-path

String representing the installation directory for the task *target* file. It is used in the `install` and `uninstall` commands. This path must be absolute. To disable installation, set it to False or to empty string. If it's absent then built-in `prefix`, `bindir` and `libdir` variables will be used to detect path. You can use any *built-in substitution* variable including `prefix`, `bindir` and `libdir` here like this:

Example in YAML format:

```
install-path : '$(prefix)/bin'
```

This parameter is false for standalone runcmd tasks by default.

It's possible to use *selectable parameters* to set this parameter.

6.35 install-files

A list of additional files to install. Each item in this list must be a *dict* with following parameters:

do It is what to do and it can be `copy`, `copy-as` or `symlink`. The `copy` value means copying specified files to a directory from the `dst`. The `copy-as` value means copying one specified file to a path from the `dst` so you can use a difference file name. The `symlink` value means creation of symlink. It's for POSIX platforms only and does nothing on MS Windows.

You may not set this parameter in some cases. If this parameter is absent:

- It's `symlink` if parameter `symlink` exists in current dict.
- It's `copy` in other cases.

src If `do` is `copy` then rules for this parameter are the same as for *source* but with one addition: you can specify one or more paths to directory if you don't use any ant pattern. In this case all files from specified directory will be copied recursively with directories hierarchy.

If `do` is `copy-as`, it must be one path to a file. And it must be relative to the *startdir* or an absolute path.

If `do` is `symlink`, it must be one path to a file. Created symbolic link will point to this path. Also it must be relative to the *startdir* or an absolute path.

dst If `do` is `copy` then it must be a path to a directory. If `do` is `copy-as`, it must be one path to a file. If `do` is `symlink`, this parameter cannot be used. See parameter `symlink`.

It must be relative to the *startdir* or an absolute path.

Any path here will have value of `destdir` at the beginning if this `destdir` is set to non-empty value. This `destdir` can be set from command line argument `--destdir` or from environment variable `DESTDIR` and it is not set by default.

symlink It is like `dst` for `copy-as` but for creating a symlink. This parameter can be used only if `do` is `symlink`.

It must be relative to the *startdir* or an absolute path.

chmod Change file mode bits. It's for POSIX platforms only and does nothing on MS Windows. And it cannot be used for `do = symlink`.

It must be integer or string. If it is an integer it must be correct value for python function os.chmod. For example: 0o755.

If it is a string then value will be converted to integer as octal representation of an integer. For example, '755' will be converted to 493 (it's 755 in octal representation).

By default it is 0o644.

user Change file owner. It's for POSIX platforms only and does nothing on MS Windows. It must be a name of existing user. It is not set by default and the value from original file will be used.

group Change file user's group. It's for POSIX platforms only and does nothing on MS Windows. It must be a name of existing user's group. It is not set by default and the value from original file will be used.

follow-symlinks Follow symlinks from `src` if `do` is `copy` or `copy-as`. If it is false, symbolic links in the paths from `src` are represented as symbolic links in the `dst`, but the metadata of the original links is NOT copied; if true or omitted, the contents and metadata of the linked files are copied to the new destination.

It's true by default.

relative This parameter can be used only if `do` is `symlink`. If it is true, relative symlink will created.

It's false by default.

Some examples can be found in the directory 'mixed/01-cshlib-cxxprogram' in the repository here.

It's possible to use *selectable parameters* to set this parameter.

6.36 install-langdir

Installation directory for .qm files. It defaults to `$(appdatadir)/translations` where *appdatadir* is the built-in variable.

This parameter is ignored if *rclangprefix* is set.

It can be used only for tasks with `qt5` in *features*.

6.37 normalize-target-name

Convert `target` name to ensure the name is suitable for file name and has not any potential problems. It replaces all space symbols for example. Experimental. By default it is False.

It's possible to use *selectable parameters* to set this parameter.

6.38 enabled

If it's False then current task will not be used at all. By default it is True.

It makes sense mostly to use with *selectable parameters* or with *byfilter*. With this parameter you can make a build task which can be used, for example, on Linux only or for specific toolchain or with another condition.

6.39 group-dependent-tasks

Although runtime jobs for the tasks may be executed in parallel, some preparation is made before this in one thread. It includes, for example, analyzing of the task dependencies and file paths in *source*. Such list of tasks is called *build group* and, by default, it's only one build group for each project which uses ZenMake. If this parameter is true, ZenMake creates a new build group for all other dependent tasks and preparation for these dependent tasks will be run only when all jobs for current task, including all dependencies, are done.

For example, if some task produces source files (*.c, *.cpp, etc) that don't exist at the time of such a preparation, you can get a problem because ZenMake cannot find not existing files. It is not a problem if such a file is declared in *target* and then this file is specified without use of ant pattern in `source` of dependent tasks. In other cases you can solve the problem by setting this parameter to True for a task which produces these source files.

By default it is False. Don't set it to True without reasons because it can slow building down.

6.40 objfile-index

Counter for the object file extension. By default it's calculated automatically as unique index number for each build task.

If you set this for one task but not for others in the same project and your index number is matched with one of automatic generated indexes then it can cause compilation errors if different tasks use the same files in parameter `source`.

Also you can set the same value for the all build tasks and often it's not a problem while different tasks use the different files in parameter `source`.

Set this parameter only if you know what you do.

It's possible to use *selectable parameters* to set this parameter.

Note: More examples of buildconf files can be found in repository here.

CHAPTER 7

Build config: selectable parameters

ZenMake provides ability to select values for parameters in *task params* depending on some conditions. This feature of ZenMake is similar to *Configurable attributes* from Bazel build system and main idea was borrowed from that system. But implementation is different.

It can be used for selecting different source files, includes, compiler flags and others on different platforms, different toolchains, etc.

Example in YAML format:

```
tasks:
    # ...

conditions:
    windows-msvc:
        platform: windows
        toolchain: msvc

buildtypes:
    debug: {}
    release:
        cxxflags.select:
            windows-msvc: /O2
            default: -O2
```

Example in Python format:

```
tasks = {
    # ...
}

conditions = {
    'windows-msvc' : {
        'platform' : 'windows',
        'toolchain' : 'msvc',
    },
```

(continues on next page)

(continued from previous page)

```
}

buildtypes = {
    'debug' : {
    },
    'release' : {
        'cxxflags.select' : {
            'windows-msvc': '/O2',
            'default': '-O2',
        },
    },
}
```

In this example for build type 'release' we set value '/O2' to 'cxxflags' if toolchain 'msvc' is used on MS Windows and set '-02' for all other cases.

This method can be used for any parameter in *task params* excluding *features* in the form:

YAML format:

```
<parameter name>.select:
    <condition name1>: <value>
    <condition name2>: <value>
    ...
    default: <value>
```

Python format:

```
'<parameter name>.select' : {
    '<condition name1>' : <value>,
    '<condition name2>' : <value>,
    ...
    'default' : <value>,
}
```

A <parameter name> here is a parameter from *task params*. Examples: 'toolchain.select', 'source.select', 'use.select', etc.

Each condition name must refer to a key in *conditions* or to one of built-in conditions (see below). There is also special optional key `default` wich means default value if none of the conditions has been selected. If the key `default` doen't exist then ZenMake tries to use the value of <parameter name> if it exists. If none of the conditions has been selected and no default value for the parameter then this parameter will not be used.

Keys in *conditions* are just strings which consist of latin characters, digits and symbols '+', '-', '_' . A value for each condition is a dict with one or more such parameters:

platform Selected platform like 'linux', 'windows', 'darwin', etc. Valid values are the same as for `default` in the *buildtypes*.

It can be one value or list of values or string with more than one value separated by spaces like this: 'linux windows'.

host-os Selected basic name of a host operating system. It is almost the same as the `platform` parameter but for the MSYS2 and cygwin platforms it is always 'windows' and for the darwin platform it is 'macos'.

distro Name of a Linux distribution like 'debian', 'fedora', etc. This name is empty string for other operating systems.

cpu-arch Selected current CPU architecture. Actual it's a result of the python function platform.machine() See https://docs.python.org/library/platform.html. Some possible values are: arm, i386, i686, x86_64, AMD64. Real value depends on a platform. For example, on Windows you can get AMD64 while on Linux you gets x86_64 on the same host.

Current value can be obtained also with the command `zenmake sysinfo`.

It can be one value or list of values or string with more than one value separated by spaces like this: 'i686 x86_64'.

toolchain Selected/detected toolchain.

It can be one value or list of values or string with more than one value separated by spaces like this: 'gcc clang'.

task Selected build task name.

It can be one value or list of values or string with more than one value separated by spaces like this: 'mylib myprogram'.

buildtype Selected buildtype.

It can be one value or list of values or string with more than one value separated by spaces like this: 'debug release'.

env Check system environment variables. It's a dict of pairs <variable> : <value>.

Example in YAML format:

```yaml
conditions:
    my-env:
        env:
            TEST: 'true' # use 'true' as a string
            CXX: gcc
```

Example in Python format:

```python
conditions = {
     'my-env' : {
         'env' : {
             'TEST' : 'true',
             'CXX' : 'gcc',
         }
     },
 }
```

If a parameter in a condition contains more than one value then any of these values will fulfill selected condition. It means if some condition, for example, has `platform` which contains `'linux windows'` without other parameters then this condition will be selected on any of these platforms (on GNU/Linux and on MS Windows). But with parameter `env` the situation is different. This parameter can contain more than one environment variable and a condition will be selected only when all of these variables are equal to existing variables from the system environment. If you want to have condition to select by any of such variables you can do it by making different conditions in *conditions*.

Note: There is a constraint for `toolchain.select` - it's not possible to use a condition with the 'toolchain' parameter inside `toolchain.select`.

Only one record from `*.select` for each parameter can be selected for each task during configuring but a condition name in `*.select` can be string with more than one name from `conditions`. Such names can be used with 'and',

'or', 'not' and '()' to form different conditions in `*.select`.

Example in YAML format:

```
conditions:
    linux:
        platform: linux
    g++:
        toolchain: g++

buildtypes:
    debug: {}
    release:
        cxxflags.select:
            # will be selected only on linux with selected/detected toolchain g++
            linux and g++: -Ofast
            # will be selected in all other cases
            default: -O2
```

Example in Python format:

```
conditions = {
    'linux' : {
        'platform' : 'linux',
    },
    'g++' : {
        'toolchain' : 'g++',
    },
}

buildtypes = {
    'debug' : {
    },
    'release' : {
        'cxxflags.select' : {
            # will be selected only on linux with selected/detected toolchain g++
            'linux and g++': '-Ofast',
            # will be selected in all other cases
            'default': '-O2',
        },
    },
}
```

For convenience there are ready to use built-in conditions for known platforms and supported toolchains. So in example above the `conditions` variable is not needed at all because conditions with names `linux` and `g++` already exist:

in YAML format:

```
# no declaration of conditions

buildtypes:
    debug: {}
    release:
        cxxflags.select:
            # will be selected only on linux with selected/detected toolchain g++
            linux and g++: -Ofast
            # will be selected in all other cases
            default: -O2
```

in Python format:

```
# no declaration of conditions

buildtypes = {
    'debug' : {
    },
    'release' : {
        'cxxflags.select' : {
            # will be selected only on linux with selected/detected toolchain g++
            'linux and g++': '-Ofast',
            # will be selected in all other cases
            'default': '-O2',
        },
    },
}
```

Also you can use built-in conditions for supported buildtypes. But if any name of supported buildtype is the same as one of known platforms or supported toolchains then such a buildtype cannot be used as a built-in condition. For example, you may want to make/use the buildtype 'linux' and it will be possible but you have to declare a different name to use it in conditions in this case because the 'linux' value is one of known platforms.

There is one detail about built-in conditions for toolchains - only toolchains supported for current build tasks can be used. ZenMake detects them from all `features` of all existing build tasks in current project during configuring. For example, if tasks exist for C language only then supported toolchains for all other languages cannot be used as a built-in condition.

If you declare condition in *conditions* with the same name of a built-in condition then your condition will be used instead of that built-in condition.

CHAPTER 8

Build config: edeps

The config parameter `edeps` is a *dict* with configurations of external non-system dependencies. General description of external dependencies is *here*.

Each such a dependency can have own unique name and parameters:

8.1 rootdir

A path to the root of the dependency project. It should be path to directory with the build script of the dependency project. This path can be relative to the *startdir* or absolute.

8.2 targets

A *dict* with descriptions of targets of the dependency project. Each target has a reference name which can be in *use* in format `dependency-name:target-reference-name` and parameters:

dir A path with the current target file. Usually it's some build directory. This path can be relative to the *startdir* or absolute.

type It's type of the target file. This type has effects to the link of the build tasks and some other things. Supported types:

stlib The target file is a static library.

shlib The target file is a shared library.

program The target file is an executable file.

file The target file is any file.

name It is a base name of the target file which is used for detecting of resulting target file name depending on destination operating system, selected toolchain, value of `type`, etc.

If it's not set the target reference name is used.

ver-num It's a version number for the target file if it is a shared library. It can have effect on resulting target file name.

fname It's a real file name of the target. Usually it's detected by ZenMake from other parameters but you can set it manually but it's not recommended until you really need it. If parameter `type` is equal to `file` the value of this parameter is always equal to value of parameter `name` by default.

Example in YAML format for non-ZenMake dependency:

```
targets:
    # 'shared-lib' and 'static-lib' are target reference names
    shared-lib:
        dir : ../foo-lib/_build_/debug
        type: shlib
        name: fooutil

    static-lib:
        dir : ../foo-lib/_build_/debug
        type: stlib
        name: fooutil
```

Example in Python format for non-ZenMake dependency:

```
'targets': {
    # 'shared-lib' and 'static-lib' are target reference names
    'shared-lib' : {
        'dir' : '../foo-lib/_build_/debug',
        'type': 'shlib',
        'name': 'fooutil',
    },
    'static-lib' : {
        'dir' : '../foo-lib/_build_/debug',
        'type': 'stlib',
        'name': 'fooutil',
    },
},
```

8.3 export-includes

A list of paths with 'includes' for C/C++/D/Fortran compilers to export from the dependency project for all build tasks which depend on the current dependency. Paths should be relative to the *startdir* or absolute but last variant is not recommended.

If paths contain spaces and all these paths are listed in one string then each such a path must be in quotes.

8.4 rules

A *dict* with descriptions of rules to produce targets files of dependency. Each rule has own reserved name and parameters to run. The rule names that allowed to use are: `configure`, `build`, `test`, `clean`, `install`, `uninstall`.

The parameters for each rule can be a string with a command line to run or a dict with attributes:

cmd A command line to run. It can be any suitable command line.

cwd A working directory where to run `cmd`. By default it's the *rootdir*. This path can be relative to the *startdir* or absolute.

env Environment variables for `cmd`. It's a `dict` where each key is a name of variable and value is a value of env variable.

timeout A timeout for `cmd` in seconds. By default there is no timeout.

shell If shell is True, the specified command will be executed through the shell. By default it is False. In some cases it can be set to True by ZenMake even though you set it to False.

trigger A dict that describes conditions to run the rule. If any configured trigger returns True then the rule will be run. You can configure one or more triggers for each rule. ZenMake supports the following types of trigger:

always If it's True then the rule will be run always. If it's False and no other triggers then the rule will not be run automatically.

paths-exist This trigger returns True only if configured paths exist on a file system. You can set paths as a string, list of strings or as a dict like for config task parameter *source*.

Examples in YAML format:

```
trigger:
    paths-exist: /etc/fstab

trigger:
    paths-exist: [ /etc/fstab, /tmp/somefile ]

trigger:
    paths-exist:
        startdir: '../foo-lib'
        incl: '**/*.label'
```

Examples in Python format:

```
'trigger': {
    'paths-exist' : '/etc/fstab',
}

'trigger': {
    'paths-exist' : ['/etc/fstab', '/tmp/somefile'],
}

'trigger': {
    'paths-exist' : dict(
        startdir = ../foo-lib,
        incl = '**/*.label',
    ),
}
```

paths-dont-exist This trigger is the same as `paths-exist` but returns True if configured paths don't exist.

env This trigger returns True only if all configured environment variables exist and equal to configured values. Format is simple: it's a `dict` where each key is a name of variable and value is a value of environment variable.

no-targets If it is True this trigger returns True only if any of target files for current dependency doesn't exist. It can be useful to detect the need to run

'build' rule. This trigger cannot be used in ZenMake command 'configure'.

func This trigger is a custom python function that must return True or False. This function gets the following parameters as arguments:

zmcmd It's a name of the current ZenMake command that has been used to run the rule.

targets A list of configured/detected targets. It's can be None if rule has been run from command 'configure'.

It's better to use ***kwargs* in this function because some new parameters can be added in the future.

This trigger cannot be used in YAML buildconf file.

Note: For any non-ZenMake dependency there are following default triggers for rules:

configure: { always: true }

build: { no-targets: true }

Any other rule: { always: false }

Note: You can use command line option `-E`/`--force-edeps` to run rules for external dependencies without checking triggers.

---

zm-commands A list with names of ZenMake commands in which selected rule will be run. By default each rule can be run in the ZenMake command with the same name only. For example, rule 'configure' by default can be run with the command 'configure' and rule 'build' with the command 'build', etc. But here you can set up a different behavior.

8.5 buildtypes-map

This parameter is used only for external dependencies which are other ZenMake projects. By default ZenMake uses value of current `buildtype` for all such dependencies to run rules but in some cases names of buildtype can be not matched. For example, current project can have buildtypes `debug` and `release` but project from dependency can have buildtypes `dbg` and `rls`. In this case you can use this parameter to set up the map of these buildtype names.

Example in YAML format:

```
buildtypes-map:
    debug   : dbg
    release : rls
```

Example in Python format:

```
buildtypes-map: {
    'debug'   : 'dbg',
    'release' : 'rls',
}
```

Some examples can be found in the directory 'external-deps' in the repository here.

CHAPTER 9

Build config: extended syntax

For convenience, ZenMake supports some syntax extensions in buildconf files.

9.1 Syntactic sugar

There are some syntactic sugar constructions that can be used to make a buildconf a little shorter.

9.1.1 configure

It can be used as a replacement for *configure* task param.

For example you have (in YAML format):

```
tasks:
  util:
    features : cshlib
    source   : shlib/**/*.c
    configure:
      - do: check-headers
        names : stdio.h

  test:
    features : cprogram
    source   : prog/**/*.c
    use      : util
    configure:
      - do: check-headers
        names : stdio.h
```

So it can be converting into this:

```
tasks:
  util:
    features : cshlib
    source   : shlib/**/*.c

  test:
    features : cprogram
    source   : prog/**/*.c
    use      : util

configure:
  - do: check-headers
    names : stdio.h
```

The `configure` above is the same as following construction:

```
byfilter:
  - for: all
    set:
      configure:
        - do: check-headers
          names : stdio.h
```

In addition to regular arguments for *configure* task param you can use `for`/`not-for`/`if` in the same way as in the *byfilter*.

Example:

```
tasks:
  # .. skipped

configure:
  - do: check-headers
    names : stdio.h
    not-for: { task: mytask }
```

9.1.2 install

Like as previous `configure` this can be used as a replacement for *install-files* task param.

Example:

```
tasks:
  # .. skipped

install:
  - for: { task: gui }
    src: 'some/src/path/ui.res'
    dst: '$(prefix)/share/$(prjname)'
```

9.2 Substitutions

There are two types of substitutions in ZenMake: bash-like variables with ability to use system environment variables and built-in variables.

9.2.1 Bash-like variables

ZenMake supports substitution variables with syntax similar to syntax of bash variables.

Both $VAR and ${VAR} syntax are supported. These variables can be used in any buildconf parameter value of string/text type.

in YAML format:

```
param: '${VAR}/some-string'
```

in Python format:

```
'param' : '${VAR}/some-string'
```

ZenMake looks such variables in environment variables at first and then in the buildconf file. You can use a $$ (double-dollar sign) to prevent use of environment variables.

Example in YAML format:

```
# set 'fragment' variable
fragment: |
  program
  end program

# set 'GCC_BASE_FLAGS' variable
GCC_BASE_FLAGS: -std=f2018 -Wall

tasks:

# ... skipped values

  test:
    features : fcprogram
    source   : src/calculator.f90 src/main.f90
    includes : src/inc
    use      : staticlib sharedlib
    configure:
      - do: check-code
        text: $$fragment # <-- substitution without env
        label: fragment

buildtypes:
  # GCC_BASE_FLAGS can be overwritten by environment variable with the same name
  debug   : { fcflags: $GCC_BASE_FLAGS -O0 }
  release: { fcflags: $GCC_BASE_FLAGS -O2 }
  default: debug
```

Note: These substitution variables inherit values from parent buildconf in *subdirs*.

Also values for such variables can be set by some *configuration actions*. For example see `var` in configuration action `find-program`. But in this case these values are not visible everywhere.

For YAML format there are some constraints with ${VAR} form due to YAML specification:

```
debug   : { fcflags: $GCC_BASE_FLAGS -O0 }      # works
debug   : { fcflags: "$GCC_BASE_FLAGS -O0" }    # works
```

(continues on next page)

(continued from previous page)

```
debug   : { fcflags: ${GCC_BASE_FLAGS} -O0 }    # doesn't work
debug   : { fcflags: "${GCC_BASE_FLAGS} -O0" } # works
debug   :
  fcflags: ${GCC_BASE_FLAGS} -O0                # works
```

9.2.2 Built-in variables

ZenMake has some built-in variables that can be used as substitutions. To avoid possible conflicts with environment and bash-like variables the syntax of substitutions is a little bit different in this case:

in YAML format:

```
param: '$(var)/some-string'
```

in Python format:

```
'param' : '$(var)/some-string'
```

List of built-in variables:

prjname Name of the current project. It can be changed via `name` from *here*.

topdir Absolute path of *startdir* of the top-level buildconf file. Usually it is root directory of the current project.

buildrootdir Absolute path of *buildroot*.

buildtypedir Absolute path of current buildtype directory. It is current value of *buildroot* plus current buildtype.

prefix The installation prefix. It is a directory that is prepended onto all install directories and it defaults to `/usr/local` on UNIX and `C:/Program Files/$(prjname)` on Windows. It can be changed via environment variable *PREFIX* or via `--prefix` on the command line.

execprefix The installation prefix for machine-specific files. In most cases it is the same as the `$(prefix)` variable. It was introduced mostly for compatibility with GNU standard: https://www.gnu.org/prep/standards/html_node/Directory-Variables.html. It can be changed via environment variable *EXEC_PREFIX* or via `--execprefix` on the command line.

bindir The directory for installing executable programs that users can run. It defaults to `$(exeprefix)/bin` on UNIX and `$(exeprefix)` on Windows. It can be changed via environment variable *BINDIR* or via `--bindir` on the command line.

sbindir The directory for installing executable programs that can be run, but are only generally useful to system administrators. It defaults to `$(exeprefix)/sbin` on UNIX and `$(exeprefix)` on Windows. It can be changed via environment variable *SBINDIR* or via `--sbindir` on the command line.

libexecdir The directory for installing executable programs to be run by other programs rather than by users. It defaults to `$(exeprefix)/libexec` on UNIX and `$(exeprefix)` on Windows. It can be changed via environment variable *LIBEXECDIR* or via `--libexecdir` on the command line.

libdir The installation directory for object files and libraries of object code. It defaults to `$(exeprefix)/lib` or `$(exeprefix)/lib64` on UNIX and `$(exeprefix)` on Windows. On Debian/Ubuntu, it may be `$(exeprefix)/lib/<multiarch-tuple>`. It can be changed via environment variable *LIBDIR* or via `--libdir` on the command line.

sysconfdir The installation directory for read-only single-machine data. It defaults to `$(prefix)/etc` on UNIX and `$(prefix)` on Windows. It can be changed via environment variable *SYSCONFDIR* or via `--sysconfdir` on the command line.

sharedstatedir The installation directory for modifiable architecture-independent data. It defaults to `/var/lib` on UNIX and `$(prefix)` on Windows. It can be changed via environment variable *SHAREDSTATEDIR* or via `--sharedstatedir` on the command line.

localstatedir The installation directory for modifiable single-machine data. It defaults to `$(prefix)/var`. It can be changed via environment variable *LOCALSTATEDIR* or via `--localstatedir` on the command line.

includedir The installation directory for C header files. It defaults to `$(prefix)/include`. It can be changed via environment variable *INCLUDEDIR* or via `--includedir` on the command line.

datarootdir The installation root directory for read-only architecture-independent data. It defaults to `$(prefix)/share` on UNIX and `$(prefix)` on Windows. It can be changed via environment variable *DATAROOTDIR* or via `--datarootdir` on the command line.

datadir The installation directory for read-only architecture-independent data. It defaults to `$(datarootdir)`. It can be changed via environment variable *DATADIR* or via `--datadir` on the command line.

appdatadir The installation directory for read-only architecture-independent application data. It defaults to `$(datarootdir)/$(prjname)` on UNIX and `$(datarootdir)` on Windows. It can be changed via environment variable *APPDATADIR* or via `--appdatadir` on the command line.

docdir The installation directory for documentation. It defaults to `$(datarootdir)/doc/$(prjname)` on UNIX and `$(datarootdir)/doc` on Windows. It can be changed via environment variable *DOCDIR* or via `--docdir` on the command line.

mandir The installation directory for man documentation. It defaults to `$(datarootdir)/man`. It can be changed via environment variable *MANDIR* or via `--mandir` on the command line.

infodir The installation directory for info documentation. It defaults to `$(datarootdir)/info`. It can be changed via environment variable *INFODIR* or via `--infodir` on the command line.

localedir The installation directory for locale-dependent data. It defaults to `$(datarootdir)/locale`. It can be changed via environment variable *LOCALEDIR* or via `--localedir` on the command line.

In some cases some extra variables are provided. For example, variables `src` and `tgt` are provided for the `cmd` in the task parameter *run*.

Built-in variables cannot be used in buildconf parameters which are used to determine values of that built-in variables. These parameters are:

- *startdir*, *buildroot*, *realbuildroot*
- **buildtypedir** only: the `default` in the *buildtypes*
- **buildtypedir** only: the `buildtypes`, `platform` and `task` in the *byfilter*

CHAPTER 10

Commands

Here are some descriptions of general commands. You can get the list of the all commands with a short description by `zenmake help` or `zenmake --help`. To get help on selected command you can use `zenmake help <selected command>` or `zenmake <selected command> --help`. Some commands have short aliases. For example you can use `bld` instead of `build` and `dc` instead of `distclean`.

configure Configure the project. In most cases you don't need to call this command directly. The `build` command calls this command by itself if necessary. This command processes most of values from *buildconf* of a project. Any change in *buildconf* leads to call of this command. You can change this behaviour with parameter `autoconfig` in buildconf *general features*.

build Build the project in the current directory. It's the main command. To see all possible parameters use `zenmake help build` or `zenmake build --help`. For example you can use `-v` to see more info about building process or `-p` to use progress bar instead of text logging. By default it calls the `configure` command by itself if necessary.

test Build (if necessery) and run tests in the current directory. If the project has no tests it's almost the same as running the `build` command. The `test` command builds and runs tests by default while the `build` command doesn't.

run Build the project (if necessery) and run one executable target from the build directory. You can specify build task/target to run if the project has more than one executable targets or omit it if the project has only one executable target. To provide command line args directly to your program you can put them after '–' in command line after all args for ZenMake. This command is for fast checking of the built project.

clean Remove build files for selected `buildtype` of the project. It doesn't touch other build files.

cleanall Remove the build directory of the project with everything in it.

install Install the build targets in some destination directory using installation prefix. It builds targets by itself if necessary. You can control paths with *environment variables* or command line parameters (see `zenmake help install`). It looks like classic `make install` in common.

uninstall Remove the build targets installed with the `install` command.

CHAPTER 11

Environment variables

ZenMake supports some environment variables that can be used. Most of examples are for POSIX platforms (Linux/MacOS) with `gcc` and `clang` installed. Also see *bash-like substitutions*.

AR Set archive-maintaining program.

CC Set C compiler. It can be name of installed a system compiler or any path to existing compiler. It overrides values from *build config* if present. Example:

```
CC=clang zenmake build -B
```

CXX Set C++ compiler. It can be name of installed a system compiler or any path to existing compiler. It overrides values from *build config* if present. Example:

```
CXX=clang++ zenmake build -B
```

DC Set D compiler. It can be name of installed a system compiler or any path to existing compiler. It overrides values from *build config* if present. Example:

```
DC=ldc2 zenmake build -B
```

FC Set Fortran compiler. It can be name of installed a system compiler or any path to existing compiler. It overrides values from *build config* if present. Example:

```
FC=gfortran zenmake build -B
```

AS Set Assembler. It can be name of installed a system compiler or any path to existing compiler. It overrides values from *build config* if present. Example:

```
AS=gcc zenmake build -B
```

ARFLAGS Flags to give the archive-maintaining program.

CFLAGS Extra flags to give to the C compiler. Example:

```
CFLAGS='-O3 -fPIC' zenmake build -B
```

CXXFLAGS Extra flags to give to the C++ compiler. Example:

```
CXXFLAGS='-O3 -fPIC' zenmake build -B
```

CPPFLAGS Extra flags added at the end of compilation commands for C/C++.

DFLAGS Extra flags to give to the D compiler. Example:

```
DFLAGS='-O' zenmake build -B
```

FCFLAGS Extra flags to give to the Fortran compiler. Example:

```
FCFLAGS='-O0' zenmake build -B
```

ASFLAGS Extra flags to give to the Assembler. Example:

```
ASFLAGS='-Os' zenmake build -B
```

LINKFLAGS Extra list of linker flags for C/C++/D/Fortran. Example:

```
LINKFLAGS='-Wl,--as-needed' zenmake build -B
```

LDFLAGS Extra list of linker flags at the end of the link command for C/C++/D/Fortran. Example:

```
LDFLAGS='-Wl,--as-needed' zenmake build -B
```

ASLINKFLAGS Extra list of linker flags for Assembler files. Example:

```
ASLINKFLAGS='-s' zenmake build -B
```

JOBS Default value for the amount of parallel jobs. Has no effect when `-j` is provided on the command line. Example:

```
JOBS=2 zenmake build
```

NUMBER_OF_PROCESSORS Default value for the amount of parallel jobs when the JOBS environment variable is not provided; it is usually set on windows systems. Has no effect when `-j` is provided on the command line.

NOCOLOR When set to a non-empty value, colors in console outputs are disabled. Has no effect when `--color` is provided on the command line. Example:

```
NOCOLOR=1 zenmake build
```

NOSYNC When set to a non-empty value, console outputs are displayed in an asynchronous manner; console text outputs may appear faster on some platforms. Example:

```
NOSYNC=1 zenmake build
```

BUILDROOT A path to the root of a project build directory. The path can be absolute or relative to the current directory. See also *buildroot*. Example:

```
BUILDROOT=bld zenmake build
```

DESTDIR Default installation base directory when `--destdir` is not provided on the command line. It's mostly for installing to a temporary directory. For example it can be used to create deb/rpm/etc packages. Example:

```
DESTDIR=dest zenmake install
```

PREFIX Set value of built-in variable *prefix* as the installation prefix. This path is always considered as an absolute path or as a relative path to `DESTDIR`. Example:

```
PREFIX=/usr zenmake install
```

EXEC_PREFIX Set value of built-in variable *execprefix* as the installation prefix for machine-specific files.

BINDIR Set value of built-in variable *bindir* as the directory for installing executable programs that users can run. This path is always considered as an absolute path or as a relative path to `DESTDIR`. Example:

```
BINDIR=/usr/bin zenmake install
```

SBINDIR Set value of built-in variable *sbindir* as the directory for installing executable programs that can be run, but are only generally useful to system administrators. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

LIBEXECDIR Set value of built-in variable *libexecdir* as the directory for installing executable programs to be run by other programs rather than by users. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

LIBDIR Set value of built-in variable *libdir* as the installation directory for object files and libraries of object code. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

SYSCONFDIR Set value of built-in variable *sysconfdir* as the installation directory for read-only single-machine data. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

SHAREDSTATEDIR Set value of built-in variable *sharedstatedir* as the installation directory for modifiable architecture-independent data. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

LOCALSTATEDIR Set value of built-in variable *localstatedir* as the installation directory for modifiable single-machine data. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

INCLUDEDIR Set value of built-in variable *includedir* as the installation directory for C header files. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

DATAROOTDIR Set value of built-in variable *datarootdir* as the installation root directory for read-only architecture-independent data. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

DATADIR Set value of built-in variable *datadir* as the installation directory for read-only architecture-independent data. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

APPDATADIR Set value of built-in variable *appdatadir* as the installation directory for read-only architecture-independent application data. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

DOCDIR Set value of built-in variable *docdir* as the installation directory for documentation. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

MANDIR Set value of built-in variable *mandir* as the installation directory for man documentation. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

INFODIR Set value of built-in variable *infodir* as the installation directory for info documentation. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

LOCALEDIR Set value of built-in variable *localedir* as the installation directory for locale-dependent data. This path is always considered as an absolute path or as a relative path to `DESTDIR`.

QT5_BINDIR Set the bin directory of the installed Qt5 toolkit. This directory must contain such tools like qmake, moc, uic, etc. This path must be absolute native path or path relative to the current working directory but last variant is not recommended. This variable can be especially useful for standalone installation of Qt5, for example on Windows. The `PATH` and `QT5_SEARCH_ROOT` environment variables are ignored if `QT5_BINDIR` is not empty.

QT5_LIBDIR Set the library directory of the installed Qt5 toolkit. This path must be absolute native path or path relative to the current working directory but last variant is not recommended. Usually you don't need to use this variable if you set the `QT5_BINDIR` variable.

QT5_INCLUDES Set the directory with 'includes' of the installed Qt5 toolkit. This path must be absolute native path or path relative to the current working directory but last variant is not recommended. Usually you don't need to use this variable if you set the `QT5_BINDIR` variable. This variable has no effect on systems with pkg-config/pkgconf installed (while you don't turn on the *QT5_NO_PKGCONF*).

QT5_SEARCH_ROOT Set the root directory to search for installed Qt5 toolkit(s). ZenMake will try to find the bin directories of all Qt5 toolkits in this directory recursively. Dot not set this variable to path like `/` or `C:\` because it will slow down the detection very much. Qt5 toolkits found in this directory have priority over values from the `PATH` environment variable. You can set more than one directories using path separator (`;` on Windows and `:` on other OS) like this:

```
QT5_SEARCH_ROOT=/usr/local/qt:/usr/local/opt/qt zenmake
```

It defaults to `C:\Qt` on Windows. Usually you don't need to use this variable on Linux.

QT5_MIN_VER Set minimum version of Qt5. For example it can be `5.1` or `5.1.2`.

QT5_MAX_VER Set maximum version of Qt5. For example it can be `5.12` or `5.12.2`.

QT5_USE_HIGHEST_VER By default ZenMake will use first useful version of Qt5. When this variable set to a 'True', 'true', 'yes' or non-zero number then ZenMake will try to use the highest version of Qt5 among found versions.

QT5_NO_PKGCONF When set to a 'True', 'true', 'yes' or non-zero number, ZenMake will not use pkg-config/pkgconf to configure building with Qt5. Usually you don't need to use this variable.

QT5_{MOC,UIC,RCC,LRELEASE,LUPDATE} These variables can be used to specify full file paths to Qt5 tools `moc`, `uic`, `rcc`, `lrelease` and `lupdate`. Usually you don't need to use these variables.

ZM_CACHE_CFGACTIONS When set to a 'True', 'true', 'yes' or non-zero number, ZenMake tries to use a cache for some *configuration actions*. Has no effect when `--cache-cfg-actions` is provided on the command line. It can speed up next runs of some configuration actions but also it can ignore changes in toolchains, system paths, etc. In general, it is safe to use it if there were no changes in the current system. Example:

```
ZM_CACHE_CFGACTIONS=1 zenmake configure
```

CHAPTER 12

Supported languages

12.1 C/C++

C an C++ are main languages that ZenMake supports. And the most of ZenMake features were made for these languages.

Supported compilers:

- C:
  - GCC C (gcc): regularly tested
  - CLANG C from LLVM (clang): regularly tested
  - Microsoft Visual C/C++ (msvc): regularly tested
  - Intel C/C++ (icc): should work but not tested
  - IBM XL C/C++ (xlc): should work but not tested
  - Oracle/Sun C (suncc): should work but not tested
  - IRIX/MIPSpro C (irixcc): may be works, not tested
- C++:
  - GCC C++ (g++): regularly tested
  - CLANG C++ from LLVM (clang++): regularly tested
  - Microsoft Visual C/C++ (msvc): regularly tested
  - Intel C/C++ (icpc): should work but not tested
  - IBM XL C/C++ (xlc++): should work but not tested
  - Oracle/Sun C++ (sunc++): should work but not tested

Examples of projects can be found in the directory `c` and `cpp` in the repository here.

12.2 Assembler

ZenMake supports gas (GNU Assembler) and has experimental support for nasm/yasm.

Examples of projects can be found in the directory `asm` in the repository here.

12.3 D

ZenMake supports compiling for D language. You can configure and build D code like C/C++ code but there are some limits:

- There is no support for MS Windows yet.
- There is no support for D package manager DUB.

While nobody uses ZenMake for D, there are no plans to resolve these issues.

Supported compilers:

- DMD Compiler - official D compiler (dmd): regularly tested
- GCC D Compiler (gdc): regularly tested
- LLVM D compiler (ldc2): regularly tested

Examples of projects can be found in the directory `d` in the repository here.

12.4 FORTRAN

ZenMake supports compiling for Fortran language.

Supported compilers:

- GCC Fortran Compiler (gfortran): regularly tested
- Intel Fortran Compiler (ifort): should work but not tested

Examples of projects can be found in the directory `fortran` in the repository here.

CHAPTER 13

Supported toolkits

13.1 Qt5

To build C++ project with Qt5 you can put `qt5` in *features*. In such tasks in the *source* parameter not only .cpp files but .qrc, .ui and .ts files can be specified as well.

There are additional task parameters for Qt5 tasks: *moc*, *rclangprefix*, *langdir-defname*, *bld-langprefix*, *unique-qmpaths*, *install-langdir*.

There are also several additional environment variables for Qt5 toolkit such as: *QT5_BINDIR*, *QT5_SEARCH_ROOT*, *QT5_LIBDIR* and some others.

ZenMake tries to find Qt5 with `qmake` and searches for it in `QT5_SEARCH_ROOT` and in the system `PATH` environment variables. You can use `QT5_BINDIR` to set directory path with `qmake` in it. The `PATH` and `QT5_SEARCH_ROOT` environment variables are ignored in this case.

You can specify minimum/maximum version of Qt5 with the *QT5_MIN_VER* and *QT5_MAX_VER* environment variables.

To specify needed Qt5 modules you should use the *use* parameter like this:

```
use : QtWidgets QtDBus # original title case of Qt5 modules must be used
```

ZenMake always adds `QtCore` module to the `use` for tasks with `qt5` in *features* because every other Qt5 module depends on `QtCore` module. So you don't need to specify `QtCore` to the `use` parameter.

Simple Qt5 task can be like that:

```
tasks:
  myqt5app:
    features  : cxxprogram qt5
    source    : prog/**/*.cpp prog/**/*.qrc prog/**/*.ui prog/**/*.ts
    moc       : prog/**/*.h
    use       : QtWidgets
```

Also it is recommended to look at examples in the `qt5` directory in the repository here.

CHAPTER 14

Configuration actions

ZenMake supports some configuration actions. They can be used to check system libraries, headers, etc. To set configuration actions use the `configure` parameter in *task params*. The value of the `configure` parameter must be a list of such actions. An item in the list can be a `dict` where `do` specifies what to do, in other words it is some type of configuration action. It's like a function where `do` describes the name of a function and others parameters are parameters for the function.

There is another possible value for such an item in python format of buildconf file and it is a python function which must return True/False on Success/Failure. If such a function raises some exception then ZenMake interprets it as if the function returns False. This function can be without arguments or with named arguments: `taskname`, `buildtype`. It's better to use ***kwargs* to have universal way to work with any input arguments.

These actions can be run sequentially or in parallel (see `do` = `parallel`). And they all are called on the **configure** step (in command **configure**).

Results of the same configuration actions are cached when it's possible but not between runnings of ZenMake.

These configuration actions in `dict` format:

do = check-headers *Parameters*: `names`, `defname` = '', `defines` = [], `mandatory` = True.

Supported languages: C, C++.

Check existence of C/C++ headers from the `names` list.

The `defname` parameter is a name of a define to set for your code when the action is over. By default the name for each header is generated in the form 'HAVE_<HEADER NAME>=1'. For example, if you set 'cstdio' in the `names` then the define 'HAVE_CSTDIO=1' will be generated. If you set 'stdio.h' in the `names` then the define 'HAVE_STDIO_H=1' will be generated.

The `defines` parameter can be used to set additional C/C++ defines to use in compiling of the action. These defines will not be set for your code, only for the action.

The *toolchain*, *includes* and *libpath* task parameters affect this type of action.

do = check-libs *Parameters*: `names` = [], `fromtask` = True, `defines` = [], `autodefine` = False, `mandatory` = True.

Supported languages: C, C++.

Check existence of the shared libraries from the `libs` task parameter or/and from the `names` list. If `fromtask` is set to False then names of libraries from the `libs` task parameter will not be used for checking. If `autodefine` is set to True it generates C/C++ define name like `HAVE_LIB_LIBNAME=1`.

The `defines` parameter can be used to set additional C/C++ defines to use in compiling of the action. These defines will not be set for your code, only for the action.

The *toolchain*, *includes* and *libpath* task parameters affect this type of action.

do = check-code *Parameters*: `text` = '', `file` = '', `label` = '', `defines` = [], `defname` = '', `execute` = False, `mandatory` = True.

Supported languages: C, C++, D, Fortran.

Provide piece of code for the test. Code can be provided with the `text` parameter as a plane text or with the `file` parameter as a path to the file with a code. This path can be absolute or relative to the *startdir*. At least one of the `text` or `file` parameters must be set.

The `label` parameter can be used to mark message of the test. If the `execute` parameter is True it means that the resulting binary will be executed and the result will have effect on the current configuration action.

The `defname` parameter is a name of C/C++/D/Fortran define to set for your code when the test is over. There is no such a name by default.

The `defines` parameter can be used to set additional C/C++/D/Fortran defines to use in compiling of the test. These defines will not be set for your code, only for the test.

The *toolchain*, *includes* and *libpath* task parameters affect this type of action.

do = find-program *Parameters*: `names`, `paths`, `var` = '', `mandatory` = True.

Supported languages: all languages supported by ZenMake.

Find a program. The `names` parameter must be used to specify one or more possible file names for the program. Do not add an extension for portability. This action does nothing if `names` is empty.

The `paths` parameter can be used to set paths to find the program, but usually you don't need to use it because by default the `PATH` system environment variable is used. Also the Windows Registry is used on MS Windows if the program was not found.

The `var` parameter can be used to set *substitution* variable name. By default it's a first name from the `names` in upper case and without symbols '-' and '.'. If this name is found in environment variables, ZenMake will use it instead of trying to find the program. Also this name can be used in parameter *run* like this:

in YAML format:

```
foo.luac:
  source : foo.lua
  configure : [ { do: find-program, names: luac } ]
  # var 'LUAC' will be set in 'find-program' if 'luac' is found.
  run: '${LUAC} -s -o $(tgt) $(src)'
```

in Python format:

```
'foo.luac' : {
    'source' : 'foo.lua',
    'configure' : [ dict(do = 'find-program', names = 'luac'), ],
    # var 'LUAC' will be set in 'find-program' if 'luac' is found.
```

(continues on next page)

(continued from previous page)

```
    'run': '${LUAC} -s -o $(tgt) $(src)',
},
```

do = find-file *Parameters*: names, paths, var = '', mandatory = True.

Supported languages: all languages supported by ZenMake.

Find a file on file system. The names parameter must be used to specify one or more possible file names. This action does nothing if names is empty.

The paths parameter must be used to set paths to find the file. Each path can be absolute or relative to the *startdir*. By default it's '.' which means *startdir*.

The var parameter can be used to set *substitution* variable name. By default it's a first name from the names in upper case and without symbols '-' and '.'.

do = call-pyfunc *Parameters*: func, mandatory = True.

Supported languages: any but only in python format of buildconf file.

Call a python function. It'a another way to use python function as an action. In this way you can use the mandatory parameter.

do = pkgconfig *Parameters*: toolname = 'pkg-config', toolpaths, packages, cflags = True, libs = True, static = False, defnames = True, def-pkg-vars, tool-atleast-version, pkg-version = False, mandatory = True.

Supported languages: C, C++.

Execute pkg-config or compatible tool (for example pkgconf) and use results. The toolname parameter can be used to set name of the tool and it is 'pkg-config' by default. The toolpaths parameter can be used to set paths to find the tool, but usually you don't need to use it.

The packages parameter is required parameter to set one or more names of packages in database of pkg-config. Each such a package name can be used with '>', '<', '=', '<=' or '>=' to check version of a package.

The parameters named cflags (default: True), libs = (default: True), static (default: False) are used to set corresponding command line parameters --cflags, --libs, --static for 'pkg-config' to get compiler/linker options. If cflags or libs is True then obtained compiler/linker options are used by ZenMake in a build task. Parameter static means forcing of static libraries and it is ignored if cflags and libs are False.

The defnames parameter is used to set C/C++ defines. It can be True/False or dict. If it's True then default names for defines will be used. If it's False then no defines will be used. If it's dict then keys must be names of used packages and values must be dicts with keys have and version and values as names for defines. By default it's True. Each package can have 'HAVE_PKGNAME' and 'PKGNAME_VERSION' define where PKGNAME is a package name in upper case. And it's default patterns. But you can set custom defines. Name of 'PKGNAME_VERSION' is used only if pkg-version is True.

The pkg-version parameter can be used to get 'define' with version of a package. It can be True of False. If it's True then 'define' will be set. If it's False then corresponding 'define' will not be set. It's False by default. This parameter will not set 'define' if defnames is False.

The def-pkg-vars parameter can be used to set custom values of variables for 'pkg-config'. It must be dict where keys and values are names and values of these variables. ZenMake uses the command line option --define-variable for this parameter. It's empty by default.

The `tool-atleast-version` parameter can be used to check minimum version of selected tool (pkg-config).

Examples in YAML format:

```yaml
# Elements like 'tasks' and other task params are skipped

# ZenMake will check package 'gtk+-3.0' and set define 'HAVE_GTK_3_0=1'
configure:
    - do: pkgconfig
      packages: gtk+-3.0

# ZenMake will check packages 'gtk+-3.0' and 'pango' and
# will check 'gtk+-3.0' version > 1 and <= 100.
# Before checking of packages ZenMake will check that 'pkg-config'␣
↪version
# is greater than 0.1.
# Also it will set defines 'WE_HAVE_GTK3=1', 'HAVE_PANGO=1',
# GTK3_VER="gtk3-ver" and LIBPANGO_VER="pango-ver" where 'gtk3-ver'
# and 'pango-ver' are values of current versions of
# 'gtk+-3.0' and 'pango'.
configure:
    - do: pkgconfig
      packages: 'gtk+-3.0 > 1 pango gtk+-3.0 <= 100'
      tool-atleast-version: '0.1'
      pkg-version: true
      defnames:
          gtk+-3.0: { have: WE_HAVE_GTK3, version: GTK3_VER }
          pango: { version: LIBPANGO_VER }
```

Examples in Python format:

```python
# Elements like 'tasks' and other task params are skipped

# ZenMake will check package 'gtk+-3.0' and set define 'HAVE_GTK_3_0=1'
'configure'  : [
    { 'do' : 'pkgconfig', 'packages' : 'gtk+-3.0' },
]

# ZenMake will check packages 'gtk+-3.0' and 'pango' and
# will check 'gtk+-3.0' version > 1 and <= 100.
# Before checking of packages ZenMake will check that 'pkg-config'␣
↪version
# is greater than 0.1.
# Also it will set defines 'WE_HAVE_GTK3=1', 'HAVE_PANGO=1',
# GTK3_VER="gtk3-ver" and LIBPANGO_VER="pango-ver" where 'gtk3-ver'
# and 'pango-ver' are values of current versions of
# 'gtk+-3.0' and 'pango'.
'configure'  : [
    {
        'do' : 'pkgconfig',
        'packages' : 'gtk+-3.0 > 1 pango gtk+-3.0 <= 100 ',
        'tool-atleast-version' : '0.1',
        'pkg-version' : True,
        'defnames' : {
            'gtk+-3.0' : { 'have' : 'WE_HAVE_GTK3', 'version': 'GTK3_VER
↪' },
            'pango' : { 'version': 'LIBPANGO_VER' },
```

(continues on next page)

(continued from previous page)

```
        },
    },
],
```

do = toolconfig *Parameters*: `toolname` = 'pkg-config', `toolpaths`, `args` = '–cflags –libs', `static` = False, `parse-as` = 'flags-libs', `defname`, `var`, `msg`, `mandatory` = True.

Supported languages: any.

Execute any `*-config` tool. It can be pkg-config, sdl-config, sdl2-config, mpicc, etc.

ZenMake doesn't know which tool will be used and therefore this action can be used in any task including standalone runcmd task.

The `toolname` parameter must be used to set name of such a tool. The `toolpaths` parameter can be used to set paths to find the tool, but usually you don't need to use it.

The `args` parameter can be used to set command line arguments. By default it is '–cflags –libs'.

The `static` parameter means forcing of static libraries and it is ignored if `parse-as` is not set to 'flags-libs'.

The `parse-as` parameter describes how to parse output. If it is 'none' then output will not be parsed. If it is 'flags-libs' then ZenMake will try to parse the output for compiler/linker options but ZenMake knows how to parse C/C++ compiler/linker options only, other languages are not supported for this value. And if it is 'entire' then output will not be parsed but value of output will be set to define name from the `defname` and/or `var` if they are defined. By default `parse-as` is set to 'flags-libs'.

The `defname` parameter can be used to set C/C++ define. If `parse-as` is set to 'flags-libs' then ZenMake will try to set define name by using value of the `toolname` discarding '-config' part if it exists. For example if the `toolname` is 'sdl2-config' then 'HAVE_SDL2=1' will be used. For other values of `parse-as` there is no default value for `defname` but you can set some custom define name.

The `var` parameter can be used to set *substitution* variable name. This parameter is ignored if value of `parse-as` is not 'entire'. By default it is not defined.

The `msg` parameter can be used to set custom message for this action.

Examples in YAML format:

```yaml
tasks:
  myapp:
    # other task params are skipped
    configure:
      # ZenMake will get compiler/linker options for SDL2 and
      # set define to 'HAVE_SDL2=1'
      - do: toolconfig
        toolname: sdl2-config
        # ZenMake will get SDL2 version and put it in the define 'SDL2_
↪VERSION'
      - do: toolconfig
        toolname: sdl2-config
        msg: Getting SDL2 version
        args: --version
        parse-as: entire
        defname: SDL2_VERSION
```

Examples in Python format:

```
tasks = {
    'myapp' : {
        # other task params are skipped
        'configure'  : [
            # ZenMake will get compiler/linker options for SDL2 and
            # set define to 'HAVE_SDL2=1'
            { 'do' : 'toolconfig', 'toolname' : 'sdl2-config' },
            # ZenMake will get SDL2 version and put it in the define
↪'SDL2_VERSION'
            {
                'do' : 'toolconfig',
                'toolname' : 'sdl2-config',
                'msg' : 'Getting SDL2 version',
                'args' : '--version',
                'parse-as' : 'entire',
                'defname' : 'SDL2_VERSION',
            },
        ]
    },
}
```

do = write-config-header *Parameters*: `file` = '', `guard` = '', `remove-defines` = True, `mandatory` = True.

Supported languages: C, C++.

Write a configuration header in the build directory after some configuration actions. By default file name is `<task name>_config.h`. The `guard` parameter can be used to change C/C++ header guard. The `remove-defines` parameter means removing the defines after they are added into configuration header file and it is True by default.

In your C/C++ code you can just include this file like that:

```
#include "yourconfig.h"
```

You can override file name by using the `file` parameter.

do = parallel *Parameters*: `actions`, `tryall` = False, `mandatory` = True.

Supported languages: all languages supported by ZenMake.

Run configuration actions from the `actions` parameter in parallel. Not all types of actions are supported. Allowed actions are `check-headers`, `check-libs`, `check-code` and `call-pyfunc`.

If you use `call-pyfunc` in `actions` you should understand that python function must be thread safe. If you don't use any shared data in such a function you don't need to worry about concurrency.

If the `tryall` parameter is True then all configuration actions from the `actions` parameter will be executed despite of errors. By default the `tryall` is False.

You can control order of the actions by using the parameters `before` and `after` with the parameter `id`. For example, one action can have `id = 'base'` and then another action can have `after = 'base'`.

Any configuration action has the `mandatory` parameter which is True by default. It also has effect for any action inside `actions` for parallel actions and for the whole bundle of parallel actions as well.

All results (defines and some other values) of configuration actions (excluding `call-pyfunc`) in one build task can be exported to all dependent build tasks. Use *export* with the name *config-results* for this ability. It allows you to avoid writing the same config actions in tasks and reduce configuration actions time run.

Example in python format:

```
def check(**kwargs):
    buildtype = kwargs['buildtype']
    # some checking
    return True

tasks = {
    'myapp' : {
        'features'   : 'cxxshlib',
        'libs'   : ['m', 'rt'],
        # ...
        'configure'  : [
            # do checking in function 'check'
            check,
            # Check libs from param 'libs'
            # { 'do' : 'check-libs' },
            { 'do' : 'check-headers', 'names' : 'cstdio', 'mandatory' : True },
            { 'do' : 'check-headers', 'names' : 'cstddef stdint.h', 'mandatory' :␣
↪False },
            # Each lib will have define 'HAVE_LIB_<LIBNAME>' if autodefine = True
            { 'do' : 'check-libs', 'names' : 'pthread', 'autodefine' : True,
                        'mandatory' : False },
            { 'do' : 'find-program', 'names' = 'python' },
            { 'do' : 'parallel',
                'actions' : [
                    { 'do' : 'check-libs', 'id' : 'syslibs' },
                    { 'do' : 'check-headers', 'names' : 'stdlib.h iostream' },
                    { 'do' : 'check-headers', 'names' : 'stdlibasd.h', 'mandatory' :␣
↪False },
                    { 'do' : 'check-headers', 'names' : 'string', 'after' : 'syslibs'␣
↪},
                ],
                'mandatory' : False,
                #'tryall' : True,
            },

            #{ 'do' : 'write-config-header', 'file' : 'myapp_config.h' }
            { 'do' : 'write-config-header' },
        ],
    },
}
```

CHAPTER 15

Dependencies

ZenMake supports several types of dependencies for build projects:

15.1 System libraries

System libraries can be specified by using the config parameter *libs*. Usually you don't need to set paths to system libraries but you can set them using the config parameter *libpath*.

15.2 Local libraries

Local libraries are libraries from your project. Use the config parameter *use* to specify such dependencies.

15.3 Sub buildconfs

You can organize building of your project by using more than one *buildconf* file in some sub directories of your project. In this case ZenMake merges parameters from all such buildconf files. But you must specify these sub directories by using the config parameter *subdirs*.

Parameters in the sub buildconf can always overwrite matching parameters from the parent *buildconf*. But some parameters are not changed.

These parameters can be set only in the the top-level buildconf:

`buildroot`, `realbuildroot`, `project`, `general`, `cliopts`

Also default build type can be set only in the top-level buildconf.

These parameters are always used without merging with parent buildconfs:

`startdir`, `subdirs`, `tasks`

ZenMake doesn't merge your own variables in your buildconf files if you use some of them. Other variables are merged including `byfilter`. But build tasks in the `byfilter` which are not from the current buildconf are ignored excepting explicit specified ones.

Some examples can be found in the directory 'subdirs' in the repository here.

15.4 External dependencies

A few basic types of external dependencies can be used:

- *Depending on other ZenMake projects*
- *Depending on non-ZenMake projects*

See full description of buildconf parameters for external dependencies *here*.

15.4.1 ZenMake projects

Configuration for this type of dependency is simple in most cases: you set up the config variable *edeps* with the *rootdir* and the *export-includes* (if it's necessary) and then specify this dependency in *use*, using existing task names from dependency buildconf.

Example in YAML format:

```
edeps:
    zmdep:
        rootdir: ../zmdep
        export-includes: ../zmdep

tasks:
    myutil:
        features : cxxshlib
        source   : 'shlib/**/*.cpp'
        # Names 'calclib' and 'printlib' are existing tasks in 'zmdep' project
        use: zmdep:calclib zmdep:printlib
```

Example in Python format:

```
edeps = {
    'zmdep' : {
        'rootdir': '../zmdep',
        'export-includes' : '../zmdep',
    },
}

tasks = {
    'myutil' : {
        'features' : 'cxxshlib',
        'source'   : 'shlib/**/*.cpp',
        # Names 'calclib' and 'printlib' are existing tasks in 'zmdep' project
        'use' : 'zmdep:calclib zmdep:printlib',
    },
}
```

Additionally, in some cases, the parameter *buildtypes-map* can be useful.

Also it's recommended to use always the same version of ZenMake for all such projects. Otherwise there are some compatible problems can be occured.

Note: Command line options `--force-edeps` and `--buildtype` for current project will affect rules for its external dependencies while all other command line options will be ignored. You can use *environment variables* to have effect on all external dependencies. And, of course, you can set up each buildconf in the dependencies to have desirable behavior.

15.4.2 Non-ZenMake projects

You can use external dependencies from some other build systems but in this case you need to set up more parameters in the config variable *edeps*. Full description of these parameters can be found *here*. Only one parameter `buildtypes-map` is not used for such dependencies.

If it's necessary to set up different targets for different buildtypes you can use *selectable parameters* in build tasks of your ZenMake project.

Example in Python format:

```
foolibdir = '../foo-lib'

edeps = {
    'foo-lib-d' : {
        'rootdir': foolibdir,
        'export-includes' : foolibdir,
        'targets': {
            'shared-lib' : {
                'dir' : foolibdir + '/_build_/debug',
                'type': 'shlib',
                'name': 'fooutil',
            },
        },
        'rules' : {
            'build' : 'make debug',
        },
    },
    'foo-lib-r' : {
```

(continues on next page)

(continued from previous page)

```
        'rootdir': foolibdir,
        'export-includes' : foolibdir,
        'targets': {
            'shared-lib' : {
                'dir' : foolibdir + '/_build_/release',
                'type': 'shlib',
                'name': 'fooutil',
            },
        },
        'rules' : {
            'build' : 'make release',
        },
    },
}

tasks = {
    'util' : {
        'features' : 'cxxshlib',
        'source'   : 'shlib/**/*.cpp',
    },
    'program' : {
        'features' : 'cxxprogram',
        'source'   : 'prog/**/*.cpp',
        'use.select' : {
            'debug'   : 'util foo-lib-d:shared-lib',
            'release' : 'util foo-lib-r:shared-lib',
        },
    },
}
```

15.4.3 Common notes

You can use command line option `-E/--force-edeps` to run rules for external dependencies without checking triggers.

Some examples can be found in the directory 'external-deps' in the repository here.

CHAPTER 16

Tests

ZenMake supports building and running tests. It has no special support for particular testing framework/library but it can be any testing framework/library.

To set up a test you need to specify task feature `test` in `buildconf` file. Then you have a choice:

- If selected task has feature *program then you may not need to do anything more. ZenMake wil try to build/run this task as test as is. But you can specify task parameter *run* to set up additional arguments.
- If selected task has no feature *program and has no *run* but has *stlib/*shlib then this task is considered as a task with test but ZenMake will not try to run this task as a test. It's useful for creation of separated libraries for tests only.
- Specify task parameter *run*.

Tests are always built only on `build` stage and run only on `test` stage. Order of buiding and running test tasks is controlled by their depedencies as for just build tasks. So it's possible to use task parameter `use` to control order of running of tests.

Example of test tasks in YAML format:

```
stlib-test:
    features : cxxprogram test'
    source   : tests/test_stlib.cpp
    # testcmn here is some library with common code for tests
    use      : stlib testcmn

test from script:
    features: test
    run:
        cmd  : python tests/test.py
        cwd  : .
        shell: false
    use: complex
    configure: [ { do: find-program, names: python } ]

# testcmn is a library with common code for tests only
```

(continues on next page)

(continued from previous page)

```
testcmn:
    features: cxxshlib test
    source  : tests/common.cpp
    includes: .

shlib-test:
    features: cxxprogram test
    source  : tests/test_shlib.cpp
    use     : shlib testcmn
    run:
        cmd     : '$(tgt) a b c'
        env     : { AZ : '111', BROKEN_TEST : 'false' }
        repeat  : 2
        timeout : 10 # in seconds
        shell   : false

shlibmain-test:
    features: cxxprogram test
    source  : tests/test_shlibmain.cpp
    use     : shlibmain testcmn
```

Example of test tasks in python format:

```
'stlib-test' : {
    'features' : 'cxxprogram test',
    'source'   : 'tests/test_stlib.cpp',
    # testcmn here is some library with common code for tests
    'use'      : 'stlib testcmn',
},

'test from script' : {
    'features' : 'test',
    'run'      : {
        'cmd'     : 'python tests/test.py',
        'cwd'     : '.',
        'shell'   : False,
    },
    'use'      : 'complex',
    'configure' : [ dict(do = 'find-program', names = 'python'), ]
},
# testcmn is a library with common code for tests only
'testcmn' : {
    'features' : 'cxxshlib test',
    'source'   :  'tests/common.cpp',
    'includes' : '.',
},
'shlib-test' : {
    'features'    : 'cxxprogram test',
    'source'      : 'tests/test_shlib.cpp',
    'use'         : 'shlib testcmn',
    'run'      : {
        'cmd'     : '$(tgt) a b c',
        'env'     : { 'AZ' : '111', 'BROKEN_TEST' : 'false'},
        'repeat'  : 2,
        'timeout' : 10, # in seconds
        'shell'   : False,
    },
```

(continues on next page)

(continued from previous page)

```
},
'shlibmain-test' : {
    'features'    : 'cxxprogram test',
    'source'      : 'tests/test_shlibmain.cpp',
    'use'         : 'shlibmain testcmn',
},
```

Use can build and/or run tests with command `test`. You can do it with command `build` as well but `build` doesn't do it by default, only if some command line arguments are used.

To build and run all tests with command `test`:

```
zenmake test
```

The same action with command `build`:

```
zenmake build -t yes -T all
```

To build but not run tests with command `test`:

```
zenmake test -T none
```

You can run all tests but also you can run tests only on changes. For this you can use `--run-tests` with value `on-changes`:

```
zenmake test -T on-changes
```

To specify additional command line arguments for all compiled testing executables you can use `--`:

```
zenmake test -- -vs
```

Everything after `--` is considered as extra command line arguments for executable target files.

CHAPTER 17

Performance tips

Here are some tips which can help to improve performance of ZenMake in some cases.

17.1 Hash algorithm

By default ZenMake uses sha1 algorithm to control changes of config/built files and for some other things. Modern CPUs often have support for this algorithm and sha1 shows better or almost the same performance as md5 in this cases. But in some other cases md5 can be faster and you can switch to use this hash algorithm. However, don't expect a big difference in performance of ZenMake.

It's recommended to check if it really has positive effect before using of md5. To change hash algorithm you can use parameter `hash-algo` in buildconf *general features*.

CHAPTER 18

FAQ

Why is python used?

It's mostly because Waf is implemented in python.

Can I use buildconf.py as usual python script?

Yes, you can. Such a behavior is supported while you don't try to use reserved config variable names for inappropriate reasons.

I want to install my project via zenmake without 'bin' and 'lib64' in one directory

Example on Linux:

```
DESTDIR=your_install_path PREFIX=/ BINDIR=/ LIBDIR=/ zenmake install
```

or:

```
PREFIX=your_install_path BINDIR=your_install_path LIBDIR=your_install_path zenmake␣
↪install
```

CHAPTER 19

Changelog

19.1 Version 0.11.0 (2022-09-04)

Added

- embed pyyaml
- add value ‘all’ for variable ‘for’ in the ‘byfilter’ parameter
- add buildconf parameter export-* for libpath, stlibpath and all *flags
- add the ‘cleanall’ command as replacement for the ‘distclean’ command
- remake/improve/extend substitutions (buildconf variables inside a text)
- add some syntactic sugar for buildconf
- get rid of ${TARGET} and rewrite substitution of ${SRC} and ${TGT}
- add ability to use ‘and’, ‘or’ and ‘not’ in the ‘*.select’
- add ‘host-os’ and ‘distro’ for the ‘*.select’ conditions
- add ‘if’ for the ‘byfilter’ parameter
- add the ‘run’ command
- support qt5 for c++ (almost done) #31
- enable absolute paths in path patterns
- add runtime lib paths for the ‘run’ command and for the ‘run’ feature
- support python 3.10

Changed

- update waf to 2.0.23
- fix bug with auto detection of interpreter in ‘runcmd’
- rename ‘include’ to ‘incl’ and ‘exclude’ to ‘excl’ for buildconf parameter ‘source’

- rename buildconf parameter ‘matrix’ to ‘byfilter’
- rename ‘export-config-actions’ to ‘export-config-results’
- rename buildconf parameter ‘config-actions’ to ‘configure’
- remake and improve the buildconf parameters ‘export-*’
- prioritize yaml buildconf format
- fix bug of no automatic reconfiguration with changed env/cli args for install/uninstall
- rename buildconf ‘features’ to ‘general’
- fix bug with ‘enabled.select’
- improve buildconf validator
- extend/improve install directory vars
- fix problem when not all values from buildconf.cliopts have effect
- fix order of reading config values from env, cli and config file
- fix terminal width detection in CLI
- improve system libraries detection
- fix bug when zenmake could not find toolchain from sys env vars like CC, CXX, etc
- fix problem with found zero-byte executables (mostly windows problem)
- fix problem with short file names (8.3 filename) on windows
- fix bug when getting rid of CXX in cmd line does not induce reconfigure
- make stop child procces in the ‘run’ command on keyboard interrupt
- many other fixes

Removed

- drop python 2.x, 3.4 and pypy
- remove task features aliases: more problems than profits
- remove redundant ‘default-buildtype’ parameter
- remove the ‘platforms’ parameter

19.2 Version 0.10.0 (2020-09-23)

Added

- support Fortran language
- add basic D language support
- add selectable parameters for buildconf task parameters
- support external dependencies
- add ‘tryall’ and ‘after’/’before’ for parallel configuration actions
- add correct buildconf validation for nested types
- add configuration action ‘call-pyfunc’ (‘check-by-pyfunc’) to parallel actions

- add configuration action 'check-code'
- add configuration actions 'pkgconfig' and 'toolconfig' (support pkg-config and other *-config tools)
- add configuration action 'find-file'
- add 'remove-defines' for configuration action 'write-config-header'
- add option to add extra files to monitor ('monitor-files')
- add buildconf task parameters 'stlibs' and 'stlibpath'
- add buildconf task parameters 'monitlibs' and 'monitstlibs'
- add buildconf task parameter 'export-config-actions'
- add buildconf task parameter 'enabled'
- add buildconf task parameter 'group-dependent-tasks'
- add add buildconf task parameter 'install-files'
- add parameter 'build-work-dir-name' to buildconf 'features'
- add simplified form of patterns using for buildconf task parameter 'source'
- add custom substitution variables
- add detection of msvc, gfortran, ifort and D compilers for command 'sysinfo'
- add number of CPUs for command 'sysinfo'
- add 'not-for' condition for config var 'matrix'
- add ability to set compiler flags in buildconf parameter 'toolchains'
- add ability to use 'run' in buildconf as a string or function
- add cdmline options –verbose-configure (-A) and –verbose-build (-B)
- add cmdline option '–force-edeps'
- add c++ demo project with boost libraries
- add demo project with luac
- add demo project with 'strip' utility on linux
- add demo project with dbus-binding-tool
- add demo projects for gtk3
- add demo project for sdl2
- add codegen demo project

Changed

- improve support of spaces in values (paths, etc)
- improve unicode support
- use sha1 by default for hashes
- correct some english text in documentation
- detach build obj files from target files
- remove locks in parallel configuration actions
- small optimization of configuration actions

- improve validation for parallel configuration actions
- improve error handling for configuration actions with python funcs
- improve buildconf errors handling
- improve use of buildconf parameter 'project.version'
- remake/improve handling of cache/db files (see buildconf parameter 'db-format')
- reduce size of zenmake.pyz by ignoring some unused waf modules
- apply solution from waf issue 2272 to fix max path limit on windows with msvc
- rename '–build-tests' to '–with-tests', enable it for 'configure' and add ability to use -t and -T as flags
- rename 'sys-lib-path' to 'libpath' and fix bug with incorrect value
- rename 'sys-libs' to 'libs'
- rename 'conftests' to 'config-actions'
- rename config action 'check-programs' to 'find-program' and change behaviour
- make ordered configuration actions
- disable ':' in task names
- refactor code to support task features in separated python modules
- don't merge buildconf parameter 'project' in sub buildconfs (see 'subdirs')
- fix bug with toolchain supported more than one language
- fix some bugs with env vars
- fix compiling problem with the same files in different tasks
- fix bug with object file indexes
- fix command 'clean' for case when build dir is symlink
- fix Waf bug of broken 'vnum' for some toolchains
- fix parsing of cmd line in 'runcmd' on windows
- fix processing of destdir, prefix, bindir, libdir

Removed

- remove configuration action (test) 'check'

19.3 Version 0.9.0 (2019-12-10)

Added

- add config parameter 'startdir'
- add config parameter 'subdirs' to support sub configs
- add 'buildroot' as the command-line arg and the environment variable
- print header with some project info
- add parallel configuration tests

Changed

- fix default command-line command
- fix problem of too long paths in configuration tests on Windows
- fix some small bugs in configuration tests
- rid of the wscript file during building
- improve buildconf validator
- improve checking of the task features
- update Waf to version 2.0.19

Removed

- remove config parameters 'project.root' and 'srcroot'

CHAPTER 20

License

BSD 3-Clause License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.