The Waf Book

Thomas Nagy

Table of Contents

Introduction

On build systems

As software is getting increasingly complex and feature-rich, so is the process of creating it. The complexity has arisen naturally from the fundamental needs of software projects:

1. Usable software (applications) is machine-readable but usually not human-readable

2. As a result, human-readable input data called source code is converted into redistributable software by compilers or archivers
3. The source code is usually decomposed into interdependent units such as files, modules, classes and functions managed independently
4. These units are versioned and their evolution is controlled over time through version control software
5. Additional processes generate additional data from source code such as tests and static analysis results
6. Source code processing is time-consuming and error-prone so it is automated with the help of additional software

While compilers sometimes attempt to do provide full build automation, they are usually limited to very specific features. For example, the Java compiler (javac) can build a whole source code tree at once, but another compiler is necessary to produce archive files (jar). Text editors and Integrated Development Environments (IDE such as Xcode or Eclipse) can provide build automation features, but their user interfaces are best at editing software, not at running builds. Version control systems such as Git are best-suited for managing end-user files, but are typically unfit for calling compilers and running scripts. And though orchestration solutions (Jenkins, Teamcity) are sometimes understood as build systems, they are usually unable to build the software themselves: they require build scripts and build software that is executed on their build agents (Maven, Make, etc).

We thus argue that building software is a unique activity in software development that requires a unique set of tools. The term "build system" is typically used to denote such software, and we believe two definitions should be distinguished:

A. It is a piece of software that helps automating processes in a software project, and that it aimed in particular at processing source code
B. It is the overall tool set required to process a work on a particular software project: compilers, build scripts, orchestration software, version control system, etc.

Waf matches the first definition which is used in the rest of this document.

The Waf framework

Build systems are usually bound to the specific frameworks that they belong to. For example, Visual Studio projects will often require MSBuild and Angular.js projects typically require Npm. Such solutions are usually focused on very specific features and are usually limited where it comes to processing other languages or different projects. For example, Ant is better suited than Make for managing Java projects, but is less convenient than Make for building simple c projects. Since programming languages and solutions evolve constantly, creating the ideal build system for everything is not really possible, so there are trade-offs between framework specialization and genericity.

There is yet a common subset of problems that build systems aim to address:

- Run compilers and scripts as distinct processes
- Run processes only when necessary by recording "what has changed"
- Run processes in parallel for efficiency reasons
- Facilitate the execution of software tests such as configuration tests
- Provide support for typical compilers and tools configurations

Waf fulfills the features above out-of-the-box, and provides a framework to extend its functionality when necessary. The main differences compared to other frameworks lie it its design:

1. Waf only requires Python, there is no dependency on additional software or libraries
2. Waf does not define a new language, it is written entirely in re-usable Python modules
3. Waf does not rely on a code generator (Makefiles) to enable efficient and extensible builds
4. Waf targets are defined as objects which separate the concerns of defining targets from running commands

Objectives of this book

This book is aimed mostly at new and advanced users of the Waf build system; its objective is to show the use of the Waf build system though practical examples, to describe the Waf extension system, and to provide an overview of the Waf internals.

We also know that build systems get re-invented by the day, so we hope that build system writers will be inspired by this documentation to re-use existing patterns and techniques.

The chapters are ordered by difficulty, starting from the basic use of Waf and Python, and diving gradually into the most complex topics. It is therefore recommended to read the chapters in order. It is also possible to start by looking at the examples from the Waf distribution before starting the reading.

2. Download and installation

2.1. Getting the Waf file

2.1.1. How to download Waf

The release files may be downloaded from the main site or from a mirror while the source may be obtained from Github. The downloads and most project commits are signed with the project public key:

- The Waf executable contains an embedded signature which can be verified by means of a script:

```
$ curl -o waf https://waf.io/waf-2.0.8
$ python verify-sig.py waf
```

- The source distribution provides a signature file for its archive:

```
$ curl -o waf-2.0.8.tar.bz2 https://waf.io/waf-2.0.8.tar.bz2
$ curl -o waf-2.0.8.tar.bz2.asc https://waf.io/waf-2.0.8.tar.bz2.asc
$ gpg --verify waf-2.0.8.tar.bz2.asc
```

- Most project commits are signed with the same public key:

```
$ git clone https://github.com/waf-project/waf.git
$ cd waf/
$ git show --show-signature
commit b73ccba03cd5f34b40a36e1d60b6a082a04cd563
gpg: Signature made sam. 16 jul. 2017 17:31:19 CEST
gpg:                using RSA key 0x49B4C67C05277AAA
...
```

2.1.2. How to run Waf

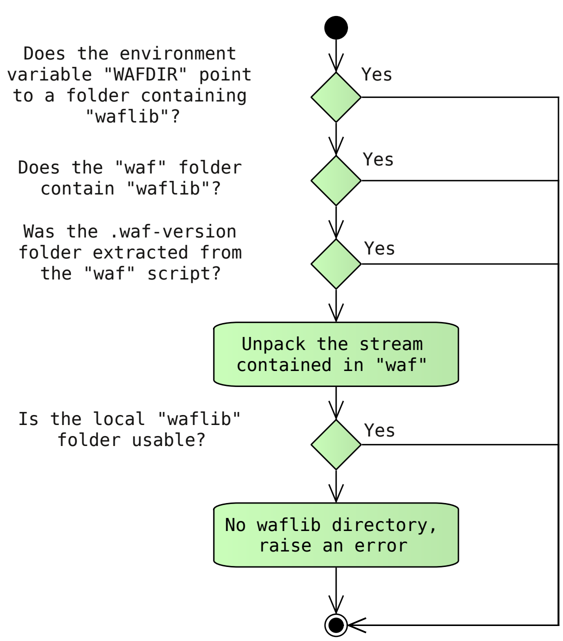
The executable may be run directly through a Python interpreter such as cPython 2.5 to 3.5, Pypy or Jython >= 2.5. It provides its own library compressed as a binary stream within the file. Upon execution, this library is uncompressed into a hidden folder in the directory of the file (it is re-created when removed). This scheme enables different Waf versions to be executed from the same folders and under various Python interpreter versions:

```
$ python waf --help
$ ls -ld .waf*
.waf-2.0.8-2c924e3f453eb715218b9cc852291170
```

No installation is necessary, but the Python interpreter must feature a bzip2 decompressor; if this module is missing, then it may be necessary to build Waf from source (consult the next sections). The folder containing the waf file must also be writable so that the waf file can decompress its library.

If the above conditions are not met, an alternative is to point the environment variable *WAFDIR* to the folder containing the directory named waflib. Though this waflib folder may be provided as part of project source files, the files cannot run in both Python 2 and Python 3 at the same time, so this is generally discouraged.

The following diagram depicts how the `waflib` directory is discovered:



2.1.3. Permissions and aliases

Since the waf file is a Python script, it is usually executed by calling `python` on it:

```
$ python waf
```

On Unix-like systems, it is usually much more convenient to set the executable permissions and avoid calling `python` each time:

```
$ chmod 755 waf
$ ./waf --version
waf 2.0.8 (54dc13ba5f51bfe2ae277451ec5ac1d0a91c7aaf)
```

If the command-line interpreter supports aliases, it is recommended to set one instead of retyping commands:

```
$ alias waf=$PWD/waf
$ waf --version
waf 2.0.8 (54dc13ba5f51bfe2ae277451ec5ac1d0a91c7aaf)
```

Else, the execution path may be modified to point at the location of the waf binary:

```
$ export PATH=$PWD:$PATH
$ waf --version
waf 2.0.8 (54dc13ba5f51bfe2ae277451ec5ac1d0a91c7aaf)
```

For convenience purposes on Windows systems, a waf.bat file is provided to detect the presence of the Python application. It assumes that it is residing in the same folder as the waf file.

2.2. Customization and redistribution

2.2.1. How to build Waf executables

Building Waf requires a Python interpreter having a version number in the range 2.6-3.5. The source code is then processed to support Python 2.5.

```
$ curl -o waf-2.0.8.tar.bz2 https://waf.io/waf-2.0.8.tar.bz2
$ tar xjvf waf-2.0.8.tar.bz2
$ cd waf-2.0.8
$ ./waf-light
Configuring the project
Setting top to                           : /home/user/waf
Setting out to                           : /home/user/waf/build
Checking for program 'python'            : /usr/bin/python
Waf: Entering directory `/waf-2.0.8/build'
[1/1] Creating waf
Waf: Leaving directory `/waf-2.0.8/build'
'build' finished successfully (0.726s)
```

For older Python interpreters, the `waf` file may be created with gzip compression instead of bzip2:

```
$ python waf-light --zip-type=gz
```

Additional extensions can be added to the waf file and redistributed as part of it. For instance, the source distribution contains several extension in testing phase under the folder *waflib/extras*. Passing a relative path in the *--tools* switch will include the corresponding file, while passing an absolute path can refer to any file on the filesystem, and non-python files in particular (they will end up in the local the *waflib/extras/* folder):

```
$ python waf-light --tools=swig,msvs
```

2.2.2. How to provide custom initializers

Extensions that provide an initialization may also be used to execute custom functions before the regular execution. Assuming that a file named `aba.py` is present in the current directory:

```
def foo():
        from waflib.Context import WAFVERSION
        print("This is Waf %s" % WAFVERSION)
```

The following will create a custom waf file that will import and execute the function *foo* before calling the waf library.

```
$ python waf-light --make-waf --tools=msvs,$PWD/aba.py
   --prelude=$'\tfrom waflib.extras import aba\n\taba.foo()'
$ ./waf --help
This is Waf 2.0.8
[...]
```

A return statement may also be added; please consult the contents of waf-light to learn more about this, or consider the examples from the build system kit that illustrate how to create build systems derived from Waf.

2.2.3. License and redistribution

The files included in the waf file (waf-light and all the files under waflib) are published under a BSD license which is reproduced below:

```
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. The name of the author may not be used to endorse or promote products
   derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "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 AUTHOR 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.
```

Though this license is considered very permissive, a copy of the copyright notice must be included in derivative works. In order to lift any doubt, such a copy is added to the waf files by default: just open a waf file with a text editor and read its first lines.

3. Projects and commands

3.1. Waf commands and usage

As the Waf file is meant to be a generic utility for building projects, project-specific details are best kept and versioned in files residing along with the project source code. These files are modules written in the Python programming language and are named **wscript**. Although they can contain any Python code, Waf can use specific functions and classes defined in them. The next sections will explore a particularly useful concept called **function commands**.

3.1.1. Command-line overview

Waf is typically run in a command-line interpreter called terminal or shell; there are three main ways of passing data to the Waf process to tell it to do something:

```
$ CFLAGS=-O3 ❶ waf distclean configure ❷ -j1 --help ❸
```

❶ The **CFLAGS** argument is an environment variable; it is used to provide processes with arbitrary data in an unchecked way way

❷ Waf is instructed to run the two commands called **distclean** and **configure** in this specific order. Commands are passed after the *waf* file and contain no **-** or **=** characters

❸ The **-j1** and **--help** elements are command-line options; they are optional and their position or order in the list of arguments is not meant to be significant.

3.1.2. Waf commands map Python functions

Waf commands assume that a corresponding command function is defined in the project wscript file which usually resides in current folder. They take a single context parameter as input and do not have to return any particular value as in the following example:

```
#! /usr/bin/env python
# encoding: utf-8

def hello(ctx):
    print('hello world')
```

Calling the command instructs `waf` to call function the function **hello**:

```
$ waf hello
hello world
'hello' finished successfully (0.001s)
```

The context object object enables data sharing across scripts, its usage will be described in sections further down.

3.1.3. Waf commands may be chained

As previously mentioned, commands are executed in the order defined on the command-line. A wscript file may thus provide an arbitrary amount of commands in the same *wscript* file:

```
def ping(ctx):
    print(' ping! %d' % id(ctx))

def pong(ctx):
    print(' pong! %d' % id(ctx))
```

And such commands may be called more than once by being repeated on the command-line:

```
$ waf ping pong ping ping
 ping! 140704847272272
'ping' finished successfully (0.001s)
 pong! 140704847271376
'pong' finished successfully (0.001s)
 ping! 140704847272336
'ping' finished successfully (0.001s)
 ping! 140704847272528
'ping' finished successfully (0.001s)
```

When an error occurs, the execution is interrupted and no further commands are called.

> Command functions are passed a new context object when they are called; the class for that object is command-specific: ConfigureContext for configure, BuildContext for build, OptionContext for option, and Context for any other command.

3.1.4. Basic project structure

Although a Waf project must contain a top-level *wscript* file, the contents may be split into several sub-project files. We will now illustrate this concept on a small project:

```
$ tree
|-- src
|   `-- wscript
`-- wscript
```

The commands in the top-level *wscript* will call the same commands from a subproject *wscript* file by calling a context method named *recurse*:

```
def ping(ctx):
        print('→ ping from ' + ctx.path.abspath())
        ctx.recurse('src')
```

And here is the contents of *src/wscript*

```
def ping(ctx):
        print('→ ping from ' + ctx.path.abspath())
```

Upon execution, the results will be:

```
$ cd /tmp/execution_recurse

$ waf ping
→ ping from /tmp/execution_recurse
→ ping from /tmp/execution_recurse/src
'ping' finished successfully (0.002s)

$ cd src

$ waf ping
→ ping from /tmp/execution_recurse/src
'ping' finished successfully (0.001s)
```

The method *recurse*, and the attribute *path* are available on all waf context classes so that all waf commands may use them.

3.2. Fundamental Waf commands

The following sections provide details on major Waf commands which are often used or re-implemented in project files.

3.2.1. Configuring a project (the *configure* command)

Though Waf may be called from any folder containing a *wscript* file, an entry point must be defined in a particular project file. This lifts ambiguities and saves the redefinition of the same imports and function definitions in all sub-wscript files of a project. The following concepts help to structure a Waf project:

1. Project directory: directory containing the source files that will be packaged and redistributed to other developers or to end users
2. Build directory: directory containing the files generated by the project (configuration sets, build files, logs, etc)
3. System files: files and folders which do not belong to the project (operating system files, etc)

The predefined command named *configure* is used to gather and store the information about these folders. We will now extend the example from the previous section with the following top-level wscript file:

```
top = '.' ❶
out = 'build_directory' ❷

def configure(ctx): ❸
        print('→ configuring the project in ' + ctx.path.abspath())

def ping(ctx):
        print('→ ping from ' + ctx.path.abspath())
        ctx.recurse('src')
```

❶ string representing the project directory. In general, top is set to ., except for some proprietary projects where the wscript cannot be added to the top-level, top may be set to *../..* or even some other folder such as */checkout/perforce/project*

❷ string representing the build directory. In general, it is set to *build*, except for some proprietary projects where the build directory may be set to an absolute path such as */tmp/build*. It is important to be able to remove the build directory safely, so it should never be given as . or *..*.

❸ the *configure* function is called by the *configure* command

The script in *src/wscript* is left unchanged:

```
def ping(ctx):
        print('→ ping from ' + ctx.path.abspath())
```

The execution output will be the following:

```
$ cd /tmp/execution_configure ❶
$ tree
|-- src
|   `-- wscript
`-- wscript

$ waf configure ❷
→ configuring the project in /tmp/execution_configure
'configure' finished successfully (0.021s)

$ tree -a
|-- build_directory/ ❸
|   |-- c4che/ ❹
|   |   |-- build.config.py ❺
|   |   `-- _cache.py ❻
|   `-- config.log ❼
|--.lock-wafbuild ❽
|-- src
|   `-- wscript
`-- wscript

$ waf ping
→ ping from /tmp/execution_configure
→ ping from /tmp/execution_configure/src
'ping' finished successfully (0.001s)

$ cd src
$ waf ping ❾
```

```
→ ping from /tmp/execution_configure
→ ping from /tmp/execution_configure/src
'ping' finished successfully (0.001s)
```

1. To configure the project, change to the directory containing the top-level project file
2. The execution is called by calling *waf configure*
3. The build directory was created
4. The configuration data is stored in the folder *c4che/*
5. The command-line options and environment variables in use are stored in *build.config.py*
6. The user configuration set is stored in *_cache.py*
7. Configuration log (duplicate of the output generated during the configuration)
8. Hidden file pointing at the relevant project file and build directory
9. Calling *waf* from a subfolder will execute the commands from the same wscript file used for the configuration

> *waf configure* is always called from the directory containing the wscript file

3.2.2. Removing generated files (the *distclean* command)

A command named *distclean* is provided to remove the build directory and the lock file created during the configuration. On the example from the previous section:

```
$ waf configure
→ configuring the project in /tmp/execution_configure
'configure' finished successfully (0.001s)

$ tree -a
|-- build_directory/
|   |-- c4che/
|   |   |-- build.config.py
|   |   `-- _cache.py
|   `-- config.log
|--.lock-wafbuild
`-- wscript

$ waf distclean ❶
'distclean' finished successfully (0.001s)

$ tree ❷
|-- src
|   `-- wscript
`-- wscript
```

1. The *distclean* command definition is implicit (no declaration in the wscript file)
2. The tree is reverted to its original state: no build directory and no lock file

The behaviour of *distclean* is fairly generic and the corresponding function does not have to be defined in the wscript files. It may be defined to alter its behaviour though, see for example the following:

```
top = '.'
out = 'build_directory'

def configure(ctx):
        print('→ configuring the project')

def distclean(ctx):
        print(' Not cleaning anything!')
```

Upon execution:

```
$ waf distclean
 Not cleaning anything!
'distclean' finished successfully (0.000s)
```

3.2.3. Packaging the project sources (the *dist* command)

The *dist* command is provided to create an archive of the project. By using the script presented previously:

```
top = '.'
out = 'build_directory'

def configure(ctx):
	print('→ configuring the project in ' + ctx.path.abspath())
```

Execute the *dist* command to get:

```
$ cd /tmp/execution_dist

$ waf configure
→ configuring the project in /tmp/execution_dist
'configure' finished successfully (0.005s)

$ waf dist
New archive created: noname-1.0.tar.bz2 (sha='a4543bb438456b56d6c89a6695f17e6cb69061f5')
'dist' finished successfully (0.035s)
```

By default, the project name and version are set to *noname* and *1.0*. To change them, it is necessary to provide two additional variables in the top-level project file:

```
APPNAME = 'webe'
VERSION = '2.0'

top = '.'
out = 'build_directory'

def configure(ctx):
	print('→ configuring the project in ' + ctx.path.abspath())
```

Because the project was configured once, it is not necessary to configure it once again:

```
$ waf dist
New archive created: webe-2.0.tar.bz2 (sha='7ccc338e2ff99b46d97e5301793824e5941dd2be')
'dist' finished successfully (0.006s)
```

More parameters may be given to alter the archive by adding a function *dist* in the script:

```
def dist(ctx):
	ctx.base_name = 'foo_2.0' ❶
	ctx.algo      = 'zip' ❷
	ctx.excl      = ' **/.waf-1* **/*~ **/*.pyc **/*.swp **/.lock-w*' ❸
	ctx.files     = ctx.path.ant_glob('**/wscript') ❹
```

❶ The archive name may be given directly instead of computing from *APPNAME* and *VERSION*

❷ The default compression format is *tar.bz2*. Other valid formats are *zip* and *tar.gz*

❸ Exclude patterns passed to give to *ctx.path.ant_glob()* which is used to find the files

❹ The files to add to the archive may be given as Waf node objects (*excl* is therefore ignored)

3.2.4. Defining command-line options (the *options* command)

The Waf script provides various default command-line options, which may be consulted by executing `waf --help`:

```
$ waf --help
waf [command] [options]

Main commands (example: ./waf build -j4)
  build    : executes the build
  clean    : cleans the project
  configure: configures the project
  dist     : makes a tarball for redistributing the sources
  distcheck: checks if the project compiles (tarball from 'dist')
  distclean: removes the build directory
  install  : installs the targets on the system
  list     : lists the targets to execute
  step     : executes tasks in a step-by-step fashion, for debugging
  uninstall: removes the targets installed

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -j JOBS, --jobs=JOBS  amount of parallel jobs (2)
  -k, --keep            keep running happily even if errors are found
```

```
  -v, --verbose         verbosity level -v -vv or -vvv [default: 0]
  --zones=ZONES         debugging zones (task_gen, deps, tasks, etc)

  configure options:
    -o OUT, --out=OUT   build dir for the project
    -t TOP, --top=TOP   src dir for the project
    --prefix=PREFIX     installation prefix [default: '/usr/local/']
    --download          try to download the tools if missing

  build and install options:
    -p, --progress      -p: progress bar; -pp: ide output
    --targets=TARGETS   task generators, e.g. "target1,target2"

  step options:
    --files=FILES       files to process, by regexp, e.g. "*/main.c,*/test/main.o"

  install/uninstall options:
    --destdir=DESTDIR   installation root [default: '']
    -f, --force         force file installation
```

Accessing a command-line option is possible from any command. Here is how to access the value *prefix*:

```
top = '.'
out = 'build_directory'

def configure(ctx):
        print('→ prefix is ' + ctx.options.prefix)
```

Upon execution, the following will be observed:

```
$ waf configure
→ prefix is /usr/local/
'configure' finished successfully (0.001s)
```

To define project command-line options, a special command named *options* may be defined in user scripts. This command will be called once before any other command executes.

```
top = '.'
out = 'build_directory'

def options(ctx):
        ctx.add_option('--foo', action='store', default=False, help='Silly test')

def configure(ctx):
        print('→ the value of foo is %r' % ctx.options.foo)
```

Upon execution, the following will be observed:

```
$ waf configure --foo=test
→ the value of foo is 'test'
'configure' finished successfully (0.001s)
```

The command context for options is a shortcut to access the optparse functionality. For more information on the optparse module, consult the [Python documentation](#)

3.3. Build-related commands

3.3.1. Building targets (the *build* command)

The *build* command is used for building targets. We will now create a new project in */tmp/execution_build/*, and add a script to create an empty file `foo.txt` and then copy it into another file `bar.txt`:

```
top = '.'
out = 'build_directory'

def configure(ctx):
    pass

def build(ctx):
    ctx(rule='touch ${TGT}', target='foo.txt')
    ctx(rule='cp ${SRC} ${TGT}', source='foo.txt', target='bar.txt')
```

Calling *waf build* directly results in an error:

```
$ cd /tmp/execution_build/

$ waf build
The project was not configured: run "waf configure" first!
```

The build requires a configured folder to know where to look for source files and where to output the created files. Let's try again:

```
$ waf configure build
'configure' finished successfully (0.007s)
Waf: Entering directory `/tmp/execution_build/build_directory'
[1/2] foo.txt:  -> build_directory/foo.txt ❶
[2/2] bar.txt: build_directory/foo.txt -> build_directory/bar.txt
Waf: Leaving directory `/tmp/examples/execution_build/build_directory'
'build' finished successfully (0.041s)

$ tree -a
|-- build_directory/
|   |-- bar.txt ❷
|   |-- c4che/
|   |   |-- build.config.py
|   |   `-- _cache.py
|   |-- foo.txt
|   |-- config.log
|   `-- .wafpickle ❸
|--.lock-wafbuild
`-- wscript

$ waf build
Waf: Entering directory `/tmp/execution_build/build_directory'
Waf: Leaving directory `/tmp/execution_build/build_directory'
'build' finished successfully (0.008s) ❹
```

❶ Note that the build *deduced* that `bar.txt` has to be created after `foo.txt`

❷ The targets are created in the build directory

❸ A pickle file is used to store the information about the targets

❹ Since the targets are up-to-date, they do not have to be created once again

Since the command *waf build* is usually executed very often, a shortcut is provided to call it implicitly:

```
$ waf
Waf: Entering directory `/tmp/execution_build/build_directory'
Waf: Leaving directory `/tmp/execution_build/build_directory'
```

3.3.2. Cleaning targets (the *clean* command)

The *clean* command is used to remove the information about the files and targets created during the build. It uses the same function *build* from the wscript files so there is no need to add a function named *clean* in the wscript file.

After cleaning, the targets will be created once again even if they were up-to-date:

```
$ waf clean build -v
'clean' finished successfully (0.003s)
Waf: Entering directory `/tmp/execution_build/build_directory' ❶
[1/2] foo.txt:  -> build_directory/foo.txt ❷
14:58:34 runner 'touch foo.txt' ❸
[2/2] bar.txt: build_directory/foo.txt -> build_directory/bar.txt
14:58:34 runner 'cp foo.txt bar.txt'
Waf: Leaving directory `/tmp/execution_build/build_directory'
'build' finished successfully (0.040s)
```

❶ All commands are executed from the build directory by default

❷ The information about the files `foo.txt` was lost so it is rebuilt

❸ By using the *-v* flag, the command-lines executed are displayed

3.3.3. More build commands

The following commands all use the same function *build* from the wscript file:

1. `build:` process the source code to create the object files
2. `clean:` remove the object files that were created during a build (unlike distclean, do not remove the configuration)
3. `install:` check that all object files have been generated and copy them on the system (programs, libraries, data files, etc)

4. `uninstall:` undo the installation, remove the object files from the system without touching the ones in the build directory
5. `list:` list the task generators in the build section (to use with waf --targets=name)
6. `step:` force the rebuild of particular files for debugging purposes

The attribute *cmd* holds the name of the command being executed:

```
top = '.'
out = 'build_directory'

def configure(ctx):
        print(ctx.cmd)

def build(ctx):
        if ctx.cmd == 'clean':
                print('cleaning!')
        else:
                print(ctx.cmd)
```

The execution will produce the following output:

```
$ waf configure clean build
Setting top to : /tmp/execution_cmd
Setting out to : /tmp/execution_cmd/build_directory
configure
'configure' finished successfully (0.002s)
cleaning!
'clean' finished successfully (0.002s)
Waf: Entering directory `/tmp/execution_cmd/build_directory'
build
Waf: Leaving directory `/tmp/execution_cmd/build_directory'
'build' finished successfully (0.001s)
```

The build command usage will be described in details in the next chapters.

4. Project configuration

The *configuration* command is used to check if the requirements for working on a project are met and to store the information. The parameters are then stored for use by other commands, such as the build command.

4.1. Using persistent data

4.1.1. Sharing data with the build

The configuration context is used to store data which may be re-used during the build. Let’s begin with the following example:

```
top = '.'
out = 'build'

def options(ctx):
        ctx.add_option('--foo', action='store', default=False, help='Silly test')

def configure(ctx):
        ctx.env.FOO = ctx.options.foo ❶
        ctx.find_program('touch', var='TOUCH') ❷

def build(bld):
        print(bld.env.TOUCH)
        print(bld.env.FOO) ❸
        bld(rule='${TOUCH} ${TGT}', target='foo.txt') ❹
```

❶ Store the option *foo* into the variable *env* (dict-like structure)

❷ Configuration routine used to find the program *touch* and to store it into *ctx.env.TOUCH* [1]

❸ Print the value of *ctx.env.FOO* that was set during the configuration

❹ The variable *${TOUCH}* corresponds to the variable *ctx.env.TOUCH*.

Here is the execution output:

```
$ waf distclean configure build --foo=abcd -v
'distclean' finished successfully (0.005s)
Checking for program touch              : /usr/bin/touch ❶
```

```
'configure' finished successfully (0.007s)
Waf: Entering directory `/tmp/configuration_build/build'
/usr/bin/touch ❷
abcd
[1/1] foo.txt:  -> build/foo.txt
10:56:41 runner '/usr/bin/touch foo.txt' ❸
Waf: Leaving directory `/tmp/configuration_build/build'
'build' finished successfully (0.021s)
```

❶ Output of the configuration test *find_program*

❷ The value of *TOUCH*

❸ Command-line used to create the target *foo.txt*

The variable *ctx.env* is called a **Configuration set**, and is an instance of the class *ConfigSet*. The class is a wrapper around Python dicts to handle serialization. For this reason it should be used for simple variables only (no functions or classes). The values are stored in a python-like format in the build directory:

```
$ tree
build/
|-- foo.txt
|-- c4che
|   |-- build.config.py
|   `-- _cache.py
`-- config.log

$ cat build/c4che/_cache.py
FOO = 'abcd'
PREFIX = '/usr/local'
TOUCH = '/usr/bin/touch'
```

> Reading and writing values to *ctx.env* is possible in both configuration and build commands. Yet, the values are stored to a file only during the configuration phase.

4.1.2. Configuration set usage

We will now provide more examples of the configuration set usage. The object **ctx.env** provides convenience methods to access its contents:

```
top = '.'
out = 'build'

def configure(ctx):
        ctx.env['CFLAGS'] = ['-g'] ❶
        ctx.env.CFLAGS = ['-g'] ❷
        ctx.env.append_value('CXXFLAGS', ['-O2', '-g']) ❸
        ctx.env.append_unique('CFLAGS', ['-g', '-O2'])
        ctx.env.prepend_value('CFLAGS', ['-O3']) ❹

        print(type(ctx.env))
        print(ctx.env)
        print(ctx.env.FOO)
```

❶ Key-based access; storing a list

❷ Attribute-based access (the two forms are equivalent)

❸ Append each element to the list *ctx.env.CXXFLAGS*, assuming it is a list

❹ Insert the values at the beginning. Note that there is no such method as *prepend_unique*

The execution will produce the following output:

```
$ waf configure
<class 'waflib.ConfigSet.ConfigSet'> ❶
'CFLAGS' ['-O3', '-g', '-O2'] ❷
'CXXFLAGS' ['-O2', '-g']
'PREFIX' '/usr/local'
[] ❸

$ cat build/c4che/_cache.py ❹
CFLAGS = ['-O3', '-g', '-O2']
CXXFLAGS = ['-O2', '-g']
PREFIX = '/usr/local'
```

❶ The object *conf.env* is an instance of the class ConfigSet defined in *waflib/ConfigSet.py*

❷ The contents of *conf.env* after the modifications

❸ When a key is undefined, it is assumed that it is a list (used by **append_value** above)

❹ The object *conf.env* is stored by default in this file

Copy and serialization apis are also provided:

```
top = '.'
out = 'build'

def configure(ctx):
        ctx.env.FOO = 'TEST'

        env_copy = ctx.env.derive() ❶

        node = ctx.path.make_node('test.txt') ❷
        env_copy.store(node.abspath()) ❸

        from waflib.ConfigSet import ConfigSet
        env2 = ConfigSet() ❹
        env2.load(node.abspath()) ❺

        print(node.read()) ❻
```

❶ Make a copy of *ctx.env* - this is a shallow copy

❷ Use **ctx.path** to create a node object representing the file `test.txt`

❸ Store the contents of **env_copy** into `test.txt`

❹ Create a new empty ConfigSet object

❺ Load the values from `test.txt`

❻ Print the contents of `test.txt`

Upon execution, the output will be the following:

```
$ waf distclean configure
'distclean' finished successfully (0.005s)
FOO = 'TEST'
PREFIX = '/usr/local'
'configure' finished successfully (0.006s)
```

4.2. Configuration utilities

4.2.1. Configuration methods

The method *ctx.find_program* seen previously is an example of a configuration method. Here are more examples:

```
top = '.'
out = 'build'

def configure(ctx):
        ctx.find_program('touch', var='TOUCH')
        ctx.check_waf_version(mini='2.0.8')
        ctx.find_file('fstab', ['/opt', '/etc'])
```

Although these methods are provided by the context class *waflib.Configure.ConfigurationContext*, they will not appear on it in [API documentation](). For modularity reasons, they are defined as simple functions and then bound dynamically:

```
top = '.'
out = 'build'

from waflib.Configure import conf ❶

@conf ❷
def hi(ctx):
        print('→ hello, world!')

# hi = conf(hi) ❸

def configure(ctx):
        ctx.hi() ❹
```

❶ Import the decorator **conf**

❷ Use the decorator to bind the method *hi* to the configuration context and build context classes. In practice, the configuration methods are only used during the configuration phase.

❸ Decorators are simple python function. Python 2.3 does not support the @ syntax so the function has to be called after the function declaration

❹ Use the method previously bound to the configuration context class

The execution will produce the following output:

```
$ waf configure
→ hello, world!
'configure' finished successfully (0.005s)
```

4.2.2. Loading and using Waf tools

For efficiency reasons, only a few configuration methods are present in the Waf core. Most configuration methods are loaded by extensions called **Waf tools**. The main tools are located in the folder `waflib/Tools`, and the tools in testing phase are located under the folder `waflib/extras`. Yet, Waf tools may be used from any location on the filesystem.

We will now demonstrate a very simple Waf tool named `dang.py` which will be used to set *ctx.env.DANG* from a command-line option:

```
#! /usr/bin/env python
# encoding: utf-8

print('→ loading the dang tool')

from waflib.Configure import conf

def options(opt): ❶
        opt.add_option('--dang', action='store', default='', dest='dang')

@conf
def read_dang(ctx): ❷
        ctx.start_msg('Checking for the variable DANG')
        if ctx.options.dang:
                ctx.env.DANG = ctx.options.dang ❸
                ctx.end_msg(ctx.env.DANG)
        else:
                ctx.end_msg('DANG is not set')

def configure(ctx): ❹
        ctx.read_dang()
```

❶ Provide command-line options

❷ Bind the function *read_dang* as a new configuration method to call ctx.read_dang() below

❸ Set an persistent value from the current command-line options

❹ Provide a command named *configure* accepting a build context instance as parameter

For loading a tool, the method *load* must be used during the configuration:

```
top = '.'
out = 'build'

def options(ctx):
    ctx.load('dang', tooldir='.') ❶

def configure(ctx):
    ctx.load('dang', tooldir='.') ❷

def build(ctx):
    print(ctx.env.DANG) ❸
```

❶ Load the options defined in *dang.py*

❷ Load the tool dang.py. By default, load calls the method *configure* defined in the tools.

❸ The tool modifies the value of *ctx.env.DANG* during the configuration

Upon execution, the output will be the following:

```
$ waf configure --dang=hello
→ loading the dang tool
Checking for DANG                        : hello ❶
'configure' finished successfully (0.006s)

$ waf
```

```
→ loading the dang tool ❷
Waf: Entering directory `/tmp/configuration_dang/build'
hello
Waf: Leaving directory `/tmp/configuration_dang/build'
'build' finished successfully (0.004s)
```

❶ First the tool is imported as a python module, and then the method *configure* is called by *load*

❷ The tools loaded during the configuration will be loaded during the build phase

4.2.3. Multiple configurations

The *conf.env* object is an important point of the configuration which is accessed and modified by Waf tools and by user-provided configuration functions. The Waf tools do not enforce a particular structure for the build scripts, so the tools will only modify the contents of the default object. The user scripts may provide several *env* objects in the configuration and pre-set or post-set specific values:

```
def configure(ctx):
        env = ctx.env ❶
        ctx.setenv('debug') ❷
        ctx.env.CC = 'gcc' ❸
        ctx.load('gcc')

        ctx.setenv('release', env) ❹
        ctx.load('msvc')
        ctx.env.CFLAGS = ['/O2']

        print ctx.all_envs['debug'] ❺
```

❶ Save a reference to *conf.env*

❷ Copy and replace *conf.env*

❸ Modify *conf.env*

❹ Copy and replace *conf.env* again, from the initial data

❺ Recall a configuration set by its name

4.3. Exception handling

4.3.1. Launching and catching configuration exceptions

Configuration helpers are methods provided by the conf object to help find parameters, for example the method *conf.find_program*

```
top = '.'
out = 'build'

def configure(ctx):
        ctx.find_program('some_app')
```

When a test cannot complete properly, an exception of the type *waflib.Errors.ConfigurationError* is raised. This often occurs when something is missing in the operating system environment or because a particular condition is not satisfied. For example:

```
$ waf
Checking for program some_app         : not found
 error: The program some_app could not be found
```

These exceptions may be raised manually by using *conf.fatal*:

```
top = '.'
out = 'build'

def configure(ctx):
        ctx.fatal("I'm sorry Dave, I'm afraid I can't do that")
```

Which will display the same kind of error:

```
$ waf configure
 error: I'm sorry Dave, I'm afraid I can't do that
$ echo $?
1
```

Here is how to catch configuration exceptions:

```
top = '.'
out = 'build'

def configure(ctx):
        try:
                ctx.find_program('some_app')
        except ctx.errors.ConfigurationError: ❶
                ctx.to_log('some_app was not found (ignoring)') ❷
```

❶ For convenience, the module *waflib.Errors* is bound to *ctx.errors*

❷ Adding information to the log file

The execution output will be the following:

```
$ waf configure
Checking for program some_app            : not found
'configure' finished successfully (0.029s) ❶

$ cat build/config.log ❷
# project  configured on Tue Jul 13 19:15:04 2010 by
# waf 2.0.8 (abi 98, python 20605f0 on linux2)
# using /home/waf/bin/waf configure
#
Checking for program some_app
not found
find program=['some_app'] paths=['/usr/local/bin', '/usr/bin'] var=None -> ''
from /tmp/configuration_exception: The program ['some_app'] could not be found
some_app was not found (ignoring) ❸
```

❶ The configuration completes without errors

❷ The log file contains useful information about the configuration execution

❸ Our log entry

Catching the errors by hand can be inconvenient. For this reason, all **@conf** methods accept a parameter named *mandatory* to suppress configuration errors. The code snippet is therefore equivalent to:

```
top = '.'
out = 'build'

def configure(ctx):
        ctx.find_program('some_app', mandatory=False)
```

As a general rule, clients should never rely on exit codes or returned values and must catch configuration exceptions. The tools should always raise configuration errors to display the errors and to give a chance to the clients to process the exceptions.

4.3.2. Transactions

Waf tools called during the configuration may use and modify the contents of *conf.env* at will. Those changes may be complex to track and to undo. Fortunately, the configuration exceptions make it possible to simplify the logic and to go back to a previous state easily. The following example illustrates how to use a transaction to to use several tools at once:

```
top = '.'
out = 'build'

def configure(ctx):
        for compiler in ('gcc', 'msvc'):
                try:
                        ctx.env.stash()
                        ctx.load(compiler)
                except ctx.errors.ConfigurationError:
                        ctx.env.revert()
                else:
                        break
        else:
                ctx.fatal('Could not find a compiler')
```

Though several calls to *stash* can be made, the copies made are shallow, which means that any complex object (such as a list) modification will be permanent. For this reason, the following is a configuration anti-pattern:

```
def configure(ctx):
        ctx.env.CFLAGS.append('-O2')
```

The methods should always be used instead:

```
def configure(ctx):
    ctx.env.append_value('CFLAGS', '-O2')
```

5. Builds

We will now provide a detailed description of the build phase, which is used for processing the build targets.

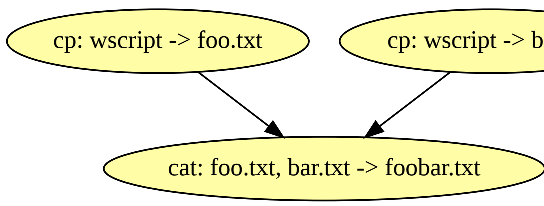
5.1. Essential build concepts

5.1.1. Build order and dependencies

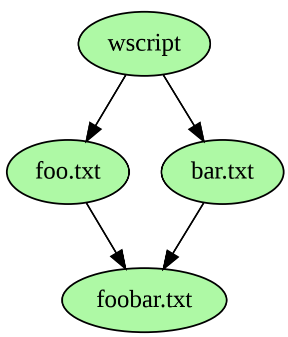
To illustrate the various concepts that are part of the build process, we are now going to use a new example. The files `foo.txt` and `bar.txt` will be created by copying the file `wscript`, and the file `foobar.txt` will be created from the concatenation of the generated files. Here is a summary: [2]

```
cp: wscript -> foo.txt
cp: wscript -> bar.txt
cat: foo.txt, bar.txt -> foobar.txt
```

Each of the three lines represents a command to execute. While the *cp* commands may be executed in any order or even in parallel, the *cat* command may only be executed after all the others are done. The constraints on **the build order** are represented on the following Directed acyclic graph:



When the `wscript` input file changes, the `foo.txt` output file has to be created once again. The file `foo.txt` is said to depend on the `wscript` file. The **file dependencies** can be represented by a Direct acyclic graph too:



Building a project consists in executing the commands according to a schedule which will respect these constraints. Faster build will be obtained when commands are executed in parallel (by using the build order), and when commands can be skipped (by using the dependencies).

In Waf, the commands are represented by **task objects**. The dependencies are used by the task classes, and may be file-based or abstract to enforce particular constraints.

5.1.2. Direct task declaration

We will now represent the build from the previous section by declaring the tasks directly in the build section:

```
def configure(ctx):
        pass

from waflib.Task import Task
class cp(Task): ❶
        def run(self): ❷
                return self.exec_command('cp %s %s' % (
                                self.inputs[0].abspath(), ❸
                                self.outputs[0].abspath()
                        )
                )

class cat(Task):
```

```
	def run(self):
		return self.exec_command('cat %s %s > %s' % (
				self.inputs[0].abspath(),
				self.inputs[1].abspath(),
				self.outputs[0].abspath()
			)
		)

def build(ctx):

	cp_1 = cp(env=ctx.env) ❹
	cp_1.set_inputs(ctx.path.find_resource('wscript')) ❺
	cp_1.set_outputs(ctx.path.find_or_declare('foo.txt'))
	ctx.add_to_group(cp_1) ❻

	cp_2 = cp(env=ctx.env)
	cp_2.set_inputs(ctx.path.find_resource('wscript'))
	cp_2.set_outputs(ctx.path.find_or_declare('bar.txt'))
	ctx.add_to_group(cp_2)

	cat_1 = cat(env=ctx.env)
	cat_1.set_inputs(cp_1.outputs + cp_2.outputs)
	cat_1.set_outputs(ctx.path.find_or_declare('foobar.txt'))
	ctx.add_to_group(cat_1)
```

❶ Task class declaration

❷ Waf tasks have a method named **run** to generate the targets

❸ Instances of *waflib.Task.Task* have input and output objects representing the files to use (Node objects)

❹ Create a new task instance manually

❺ Set input and output files represented as *waflib.Node.Node* objects

❻ Add the task to the build context for execution (but do not execute them immediately)

The execution output will be the following:

```
$ waf clean build ❶
'clean' finished successfully (0.003s)
Waf: Entering directory `/tmp/build_manual_tasks/build'
[1/3] cp: wscript -> build/foo.txt
[2/3] cp: wscript -> build/bar.txt
[3/3] cat: build/foo.txt build/bar.txt -> build/foobar.txt
Waf: Leaving directory `/tmp/build_manual_tasks/build'
'build' finished successfully (0.047s)

$ waf build ❷
Waf: Entering directory `/tmp/build_manual_tasks/build'
Waf: Leaving directory `/tmp/build_manual_tasks/build'
'build' finished successfully (0.007s)

$ echo " " >> wscript ❸

$ waf build
Waf: Entering directory `/tmp/build_manual_tasks/build'
[1/3] cp: wscript -> build/foo.txt ❹
[2/3] cp: wscript -> build/bar.txt
[3/3] cat: build/foo.txt build/bar.txt -> build/foobar.txt
Waf: Leaving directory `/tmp/build_manual_tasks/build'
'build' finished successfully (0.043s)
```

❶ The tasks are not executed in the *clean* command

❷ The build keeps track of the files that were generated to avoid generating them again

❸ Modify one of the source files

❹ Rebuild according to the dependency graph

Please remember:

1. The execution order was **computed automatically**, by using the file inputs and outputs set on the task instances
2. The dependencies were **computed automatically** (the files were rebuilt when necessary), by using the node objects (hashes of the file contents were stored between the builds and then compared)
3. The tasks that have no order constraints are executed in parallel by default

5.1.3. Task encapsulation by task generators

Declaring the tasks directly is tedious and results in lengthy scripts. Feature-wise, the following is equivalent to the previous example:

```
def configure(ctx):
        pass

def build(ctx):
        ctx(rule='cp ${SRC} ${TGT}', source='wscript', target='foo.txt')
        ctx(rule='cp ${SRC} ${TGT}', source='wscript', target='bar.txt')
        ctx(rule='cat ${SRC} > ${TGT}', source='foo.txt bar.txt', target='foobar.txt')
```

The **ctx(…)** call is a shortcut to the class *waflib.TaskGen.task_gen*, instances of this class are called **task generator objects**. The task generators are lazy containers and will only create the tasks and the task classes when they are actually needed:

```
def configure(ctx):
        pass

def build(ctx):
        tg = ctx(rule='touch ${TGT}', target='foo')
        print(type(tg))
        print(tg.tasks)
        tg.post()
        print(tg.tasks)
        print(type(tg.tasks[0]))
```

Here is the output:

```
waf configure build
Setting top to   : /tmp/build_lazy_tg
Setting out to   : /tmp/build_lazy_tg/build
'configure' finished successfully (0.204s)
Waf: Entering directory `/tmp/build_lazy_tg/build'
<class 'waflib.TaskGen.task_gen'> ❶
[] ❷
[{task: foo  -> foo}] ❸
<class 'waflib.Task.foo'> ❹
[1/1] foo:  -> build/foo
Waf: Leaving directory `/tmp/build_lazy_tg/build'
'build' finished successfully (0.023s)
```

❶ Task generator type

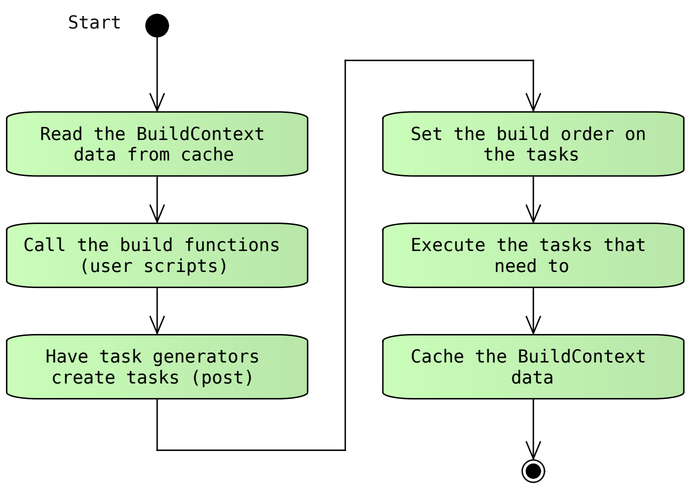
❷ The tasks created are stored in the list *tasks* (0..n tasks may be added)

❸ Tasks are created after calling the method post() - it is usually called automatically internally

❹ A new task class was created dynamically for the target `foo`

5.1.4. Overview of the build phase

A high level overview of the build process is represented on the following diagram:



The tasks are all created before any of them is executed. New tasks may be created after the build has started, but the dependencies have to be set by using low-level apis.

5.2. More build options

Although any operation can be executed as part of a task, a few scenarios are typical and it makes sense to provide convenience functions for them.

5.2.1. Executing specific routines before or after the build

User functions may be bound to be executed at two key moments during the build command (callbacks):

1. immediately before the build starts (bld.add_pre_fun)
2. immediately after the build is completed successfully (bld.add_post_fun)

Here is how to execute a test after the build is finished:

```
top = '.'
out = 'build'

def options(ctx):
        ctx.add_option('--exe', action='store_true', default=False,
                help='execute the program after it is built')

def configure(ctx):
        pass

def pre(ctx): ❶
        print('before the build is started')

def post(ctx):
        print('after the build is complete')
        if ctx.cmd == 'install': ❷
                if ctx.options.exe: ❸
                        ctx.exec_command('/sbin/ldconfig') ❹

def build(ctx):
        ctx.add_pre_fun(pre) ❺
        ctx.add_post_fun(post)
```

❶ The callbacks take the build context as unique parameter *ctx*

❷ Access the command type

❸ Access to the command-line options

❹ A common scenario is to call ldconfig after the files are installed.

❺ Scheduling the functions for later execution. Python functions are objects too.

Upon execution, the following output will be produced:

```
$ waf distclean configure build install --exe
'distclean' finished successfully (0.005s)
'configure' finished successfully (0.011s)
Waf: Entering directory `/tmp/build_pre_post/build'
before the build is started ❶
Waf: Leaving directory `/tmp/build_pre_post/build'
after the build is complete ❷
'build' finished successfully (0.004s)
Waf: Entering directory `/tmp/build_pre_post/build'
before the build is started
Waf: Leaving directory `/tmp/build_pre_post/build'
after the build is complete
/sbin/ldconfig: Can't create temporary cache file /etc/ld.so.cache~: Permission denied ❸
'install' finished successfully (15.730s)
```

❶ output of the function bound by *bld.add_pre_fun*

❷ output of the function bound by *bld.add_post_fun*

❸ execution at installation time

5.2.2. Installing files

Three build context methods are provided for installing files created during or after the build:

1. install_files: install several files in a folder
2. install_as: install a target with a different name
3. symlink_as: create a symbolic link on the platforms that support it

```
def build(bld):
        bld.install_files('${PREFIX}/include', ['a1.h', 'a2.h']) ❶
```

```
        bld.install_as('${PREFIX}/dir/bar.png', 'foo.png') ❷
        bld.symlink_as('${PREFIX}/lib/libfoo.so.1', 'libfoo.so.1.2.3') ❸

        env_foo = bld.env.derive()
        env_foo.PREFIX = '/opt'
        bld.install_as('${PREFIX}/dir/test.png', 'foo.png', env=env_foo) ❹

        start_dir = bld.path.find_dir('src/bar')
        bld.install_files('${PREFIX}/share', ['foo/a1.h'],
                cwd=start_dir, relative_trick=True) ❺

        bld.install_files('${PREFIX}/share', start_dir.ant_glob('**/*.png'), ❻
                cwd=start_dir, relative_trick=True)
```

❶ Install various files in the target destination

❷ Install one file, changing its name

❸ Create a symbolic link

❹ Overriding the configuration set (*env* is optional in the three methods install_files, install_as and symlink_as)

❺ Install src/bar/foo/a1.h as seen from the current script into *${PREFIX}/share/foo/a1.h*

❻ Install the png files recursively, preserving the folder structure read from src/bar/

> the methods *install_files*, *install_as* and *symlink_as* will do something only during *waf install* or *waf uninstall*, they have no effect in other build commands

5.2.3. Listing the task generators and forcing specific task generators

The *list* command is used to display the task generators that are declared:

```
top = '.'
out = 'build'

def configure(ctx):
        pass

def build(ctx):
        ctx(source='wscript', target='foo.txt', rule='cp ${SRC} ${TGT}')
        ctx(target='bar.txt', rule='touch ${TGT}', name='bar')
```

By default, the name of the task generator is computed from the *target* attribute:

```
$ waf configure list
'configure' finished successfully (0.005s)
foo.txt
bar
'list' finished successfully (0.008s)
```

The main usage of the name values is to force a partial build with the *--targets* option. Compare the following:

```
$ waf clean build
'clean' finished successfully (0.003s)
Waf: Entering directory `/tmp/build_list/build'
[1/2] foo.txt: wscript -> build/foo.txt
[2/2] bar:  -> build/bar.txt
Waf: Leaving directory `/tmp/build_list/build'
'build' finished successfully (0.028s)

$ waf clean build --targets=foo.txt
'clean' finished successfully (0.003s)
Waf: Entering directory `/tmp/build_list/build'
[1/1] foo.txt: wscript -> build/foo.txt
Waf: Leaving directory `/tmp/build_list/build'
'build' finished successfully (0.022s)
```

5.2.4. Execution step by step for debugging (the *step* command)

The *step* is used to execute specific tasks and to return the exit status and any error message. It is particularly useful for debugging:

```
waf step --files=test_shlib.c,test_staticlib.c
Waf: Entering directory `/tmp/demos/c/build'
c: shlib/test_shlib.c -> build/shlib/test_shlib.c.1.o
```

```
 -> 0
cshlib: build/shlib/test_shlib.c.1.o -> build/shlib/libmy_shared_lib.so
 -> 0
c: stlib/test_staticlib.c -> build/stlib/test_staticlib.c.1.o
 -> 0
cstlib: build/stlib/test_staticlib.c.1.o -> build/stlib/libmy_static_lib.a
 -> 0
Waf: Leaving directory `/tmp/demos/c/build'
'step' finished successfully (0.201s)
```

In this case the `.so` files were also rebuilt. Since the files attribute is interpreted as a comma-separated list of regular expressions, the following will produce a different output:

```
$ waf step --files=test_shlib.c$
Waf: Entering directory `/tmp/demos/c/build'
c: shlib/test_shlib.c -> build/shlib/test_shlib.c.1.o
 -> 0
Waf: Leaving directory `/tmp/demos/c/build'
'step' finished successfully (0.083s)
```

Finally, the tasks to execute may be prefixed by *in:* or *out:* to specify if it is a source or a target file:

```
$ waf step --files=out:build/shlib/test_shlib.c.1.o
Waf: Entering directory `/tmp/demos/c/build'
cc: shlib/test_shlib.c -> build/shlib/test_shlib.c.1.o
 -> 0
Waf: Leaving directory `/tmp/demos/c/build'
'step' finished successfully (0.091s)
```

note: when using *waf step*, all tasks are executed sequentially, even if some of them return a non-zero exit status

6. Node objects

Node objects represent files or folders and are used to ease the operations dealing with the file system. This chapter provides an overview of their usage.

6.1. Design of the node class

6.1.1. The node tree

The Waf nodes inherit the class *waflib.Node.Node* and provide a tree structure to represent the file system:

1. **parent**: parent node
2. **children**: folder contents - or empty if the node is a file

In practice, the reference to the filesystem tree is bound to the context classes for access from Waf commands. Here is an illustration:

```
top = '.'
out = 'build'

def configure(ctx):
    pass

def dosomething(ctx):
    print(ctx.path.abspath()) ❶
    print(ctx.root.abspath()) ❷
    print("ctx.path contents %r" % ctx.path.children)
    print("ctx.path parent   %r" % ctx.path.parent.abspath())
    print("ctx.root parent   %r" % ctx.root.parent)
```

❶ **ctx.path** represents the path to the `wscript` file being executed

❷ **ctx.root** is the root of the file system or the folder containing the drive letters (win32 systems)

The execution output will be the following:

```
$ waf configure dosomething
Setting top to    : /tmp/node_tree
Setting out to    : /tmp/node_tree/build
'configure' finished successfully (0.007s)
/tmp/node_tree ❶
```

```
/
ctx.path contents {'wscript': /tmp/node_tree/wscript} ❷
ctx.path parent   '/tmp' ❸
ctx.root parent   None ❹
'dosomething' finished successfully (0.001s)
```

❶ Absolute paths are used frequently

❷ The folder contents are stored in the dict *children* which maps names to node objects

❸ Each node keeps reference to his *parent* node

❹ The root node has no *parent*

> There is a strict correspondence between nodes and filesystem elements: a node represents exactly one file or one folder, and only one node can represent a file or a folder.

6.1.2. Node caching

By default, only the necessary nodes are created:

```
def configure(ctx):
    pass

def dosomething(ctx):
    print(ctx.root.children)
```

The filesystem root appears to only contain one node, although the real filesystem root contains more folders than just `/tmp`:

```
$ waf configure dosomething
Setting top to   : /tmp/nodes_cache
Setting out to   : /tmp/nodes_cache/build
'configure' finished successfully (0.086s)
{'tmp': /tmp}
'dosomething' finished successfully (0.001s)

$ ls /
bin boot dev etc home tmp usr var
```

This means in particular that some nodes may have to be read from the file system or created before being used.

6.2. General usage

6.2.1. Searching and creating nodes

Nodes may be created manually or read from the file system. Three methods are provided for this purpose:

```
def configure(ctx):
        pass

def dosomething(ctx):
        print(ctx.path.find_node('wscript')) ❶

        nd1 = ctx.path.make_node('foo.txt') ❷
        print(nd1)

        nd2 = ctx.path.search_node('foo.txt') ❸
        print(nd2)

        nd3 = ctx.path.search_node('bar.txt') ❹
        print(nd3)

        nd2.write('some text') ❺
        print(nd2.read())

        print(ctx.path.listdir())
```

❶ Search for a node by reading the file system

❷ Search for a node or create it if it does not exist

❸ Search for a node but do not try to create it

❹ Search for a file which does not exist

❺ Write to the file pointed by the node, creating or overwriting the file

The output will be the following:

```
$ waf distclean configure dosomething
'distclean' finished successfully (0.005s)
Setting top to    : /tmp/nodes_search
Setting out to    : /tmp/nodes_search/build
'configure' finished successfully (0.006s)
wscript
foo.txt
foo.txt
None
some text
['.lock-wafbuild', 'foo.txt', 'build', 'wscript', '.git']
```

> More methods may be found in the [API documentation]

> The Node methods are not meant to be safe for concurrent access. The code executed in parallel (method run() of task objects for example) must avoid modifying the Node object data structure.

> The Node methods read/write must be used to prevent file handle inheritance issues on win32 systems instead of plain open/read/write. Such problems arise when spawning processes during parallel builds.

6.2.2. Listing files and folders

The method **ant_glob** is used to list files and folders recursively:

```
top = '.'
out = 'build'

def configure(ctx):
        pass

def dosomething(ctx):
        print(ctx.path.ant_glob('wsc*')) ❶
        print(ctx.path.ant_glob('w?cr?p?')) ❷
        print(ctx.root.ant_glob('usr/include/**/zlib*', ❸ dir=False, src=True)) ❹
        print(ctx.path.ant_glob(['**/*py', '**/*p'], excl=['**/default*'])) ❺
```

❶ The method ant_glob is called on a node object, and not on the build context, it returns only files by default

❷ Patterns may contain wildcards such as *** or *?*, but they are [Ant patterns], not regular expressions

❸ The symbol **** enable recursion. Complex folder hierarchies may take a lot of time, so use with care.

❹ Even though recursion is enabled, only files are returned by default. To turn directory listing on, use *dir=True*

❺ Patterns are either lists of strings or space-delimited values. Patterns to exclude are defined in *waflib.Node.exclude_regs*.

The execution output will be the following:

```
$ waf configure dosomething
Setting top to    : /tmp/nodes_ant_glob
Setting out to    : /tmp/nodes_ant_glob/build
'configure' finished successfully (0.006s)
[/tmp/nodes_ant_glob/wscript]
[/tmp/nodes_ant_glob/wscript]
[/usr/include/zlib.h]
[/tmp/nodes_ant_glob/build/c4che/build.config.py]
```

The sequence *..* represents exactly two dot characters, and not the parent directory. This is used to guarantee that the search will terminate, and that the same files will not be listed multiple times. Consider the following:

```
ctx.path.ant_glob('../wscript') ❶
ctx.path.parent.ant_glob('wscript') ❷
```

❶ Invalid, this pattern will never return anything

❷ Call *ant_glob* from the parent directory

6.2.3. Path manipulation: abspath, path_from

The following example illustrates a few ways of obtaining absolute and relative paths:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(ctx):
        dir = ctx.path ❶
        src = ctx.path.find_resource('wscript')
        bld = ctx.path.find_or_declare('out.out')

        print(src.abspath()) ❷
        print(bld.abspath())
        print(dir.abspath())
        print(src.path_from(dir.parent)) ❸
```

❶ Directory node, source node and build node

❷ Print the absolute path

❸ Compute the path relative to another node

Here is the execution trace on a unix-like system:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.005s)
Waf: Entering directory `/tmp/nested/build'
/tmp/nested/wscript
/tmp/nested/build/out.out
/tmp/nested
nested/wscript
Waf: Leaving directory `/tmp/nested/build'
'build' finished successfully (0.003s)
```

6.3. BuildContext-specific methods

6.3.1. Source and build nodes

Although the *sources* and *targets* in the `wscript` files are declared as if they were in the current directory, the target files are output into the build directory. To enable this behaviour, the directory structure below the *top* directory must be replicated in the *out* directory. For example, the folder **program** from `demos/c` has its equivalent in the build directory:

```
$ cd demos/c
$ tree
.
|-- build
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- _cache.py
|   |-- config.h
|   |-- config.log
|   `-- program
|       |-- main.c.0.o
|       `-- myprogram
|-- program
|   |-- a.h
|   |-- main.c
|   `-- wscript_build
`-- wscript
```

To support this, the build context provides two additional nodes:

1. srcnode: node representing the top-level directory
2. bldnode: node representing the build directory

To obtain a build node from a src node and vice-versa, the following methods may be used:

1. Node.get_src()
2. Node.get_bld()

6.3.2. Using Nodes during the build phase

Although using *find_node*/*search_node*/*make_node* directly is possible, the following two wrappers provide significant simplifications. They accept a string representing the target as input and return a single node:

1. **find_resource**: This method first iterates in the source directory to obtain a file corresponding to the input path given. If not such file is found, it then tries to obtain a declared node from the corresponding path under the build directory. If no source file or build node is found then None is returned.
2. **find_or_declare**: Returns a Node object under the source or build folders. If nothing is found, it creates the corresponding node in the build directory and the folder structure required for writing the file during the build phase. If the path is not under the build directory, then a fake filesystem structure is created for the output files (`__root__`).

Waf uses these two methods internally; in order to avoid ambiguities, it is recommended simplify paths and in particular to resolve .. elements that could cause the object lookup to get out of the source or build folders. The following example is assumed to be defined in the source directory:

```
top = '.'
out = 'build'

def build(bld):
    p = bld.path.find_or_declare('../c/foo.txt') ❶
    p = bld.path.parent.find _node('c').find_or_declare('foo.txt') ❷

    bld(rule='cp ${SRC} ${TGT}', source='wscript', target=p)
```

❶ This ambiguous construct declares `c/foo.txt` in the source directory

❷ With paths properly resolved, this declares the proper file under the build directory `build/c/foo.txt`

6.3.3. Nodes, tasks, and task generators

As seen in the previous chapter, Task objects can process files represented as lists of input and output nodes. The task generators will usually process the input files given as strings to obtain such nodes and bind them to the tasks.

Because the build directory can be enabled or disabled, the following file copy would be ambiguous: [3]

```
def build(bld):
    bld(rule='cp ${SRC} ${TGT}', source='foo.txt', target='foo.txt')
```

To actually copy a file into the corresponding build directory with the same name, the ambiguity must be removed:

```
def build(bld):
    bld(
        rule   = 'cp ${SRC} ${TGT}',
        source = bld.path.make_node('foo.txt'),
        target = bld.path.get_bld().make_node('foo.txt')
    )
```

In practice, it is easier to use a wrapper that conceals these details (more examples can be found in `demos/subst`):

```
def build(bld):
    bld(features='subst', source='wscript', target='wscript', is_copy=True)
```

7. Advanced build definitions

7.1. Custom commands

7.1.1. Context inheritance

An instance of the class *waflib.Context.Context* is used by default for the custom commands. To provide a custom context object it is necessary to create a context subclass:

```
def configure(ctx):
        print(type(ctx))

def foo(ctx): ❶
        print(type(ctx))

def bar(ctx):
        print(type(ctx))

from waflib.Context import Context

class one(Context):
        cmd = 'foo' ❷

class two(Context):
        cmd = 'tak' ❸
        fun = 'bar'
```

1. A custom command using the default context
2. Bind a context class to the command *foo*
3. Declare a new command named *tak*, but set it to call the script function *bar*

The execution output will be:

```
$ waf configure foo bar tak
Setting top to    : /tmp/advbuild_subclass
Setting out to    : /tmp/advbuild_subclass/build
<class 'waflib.Configure.ConfigurationContext'>
'configure' finished successfully (0.008s)
<class 'wscript.one'>
'foo' finished successfully (0.001s)
<class 'waflib.Context.Context'>
'bar' finished successfully (0.001s)
<class 'wscript.two'>
'tak' finished successfully (0.001s)
```

A typical application of custom context is subclassing the build context to use the configuration data loaded in **ctx.env**:

```
def configure(ctx):
        ctx.env.FOO = 'some data'

def build(ctx):
        print('build command')

def foo(ctx):
        print(ctx.env.FOO)

from waflib.Build import BuildContext
class one(BuildContext):
        cmd = 'foo'
        fun = 'foo'
```

The output will be the following:

```
$ waf configure foo
Setting top to    : /tmp/advbuild_confdata
Setting out to    : /tmp/advbuild_confdata/build
'configure' finished successfully (0.006s)
Waf: Entering directory `/disk/comp/waf/docs/book/examples/advbuild_confdata/build'
some data
Waf: Leaving directory `/disk/comp/waf/docs/book/examples/advbuild_confdata/build'
'foo' finished successfully (0.004s)
```

> **Note:** The build commands are using this system: *waf install* → *waflib.Build.InstallContext*, *waf step* → *waflib.Build.StepContext*, etc

7.1.2. Command composition

To re-use commands that have context object of different base classes, insert them in the *command stack*:

```
def configure(ctx):
        pass

def build(ctx):
        pass

def cleanbuild(ctx):
        from waflib import Options
        Options.commands = ['clean', 'build'] + Options.commands
```

This technique is useful for writing testcases. By executing *waf test*, the following script will configure a project, create source files in the source directory, build a program, modify the sources, and rebuild the program. In this case, the program must be rebuilt because a header (implicit dependency) has changed.

```
def options(ctx):
        ctx.load('compiler_c')

def configure(ctx):
        ctx.load('compiler_c')
```

```
def setup(ctx):
        n = ctx.path.make_node('main.c')
        n.write('#include "foo.h"\nint main() {return 0;}\n')

        global v
        m = ctx.path.make_node('foo.h')
        m.write('int k = %d;\n' % v)
        v += 1

def build(ctx):
        ctx.program(source='main.c', target='app')

def test(ctx):
        global v ❶
        v = 12

        import Options ❷
        lst = ['configure', 'setup', 'build', 'setup', 'build']
        Options.commands = lst + Options.commands
```

❶ A global variable may be used to share data between commands deriving from different classes

❷ The test command is used to add more commands

The following output will be observed:

```
$ waf test
'test' finished successfully (0.000s)
Setting top to                           : /tmp/advbuild_testcase
Setting out to                           : /tmp/advbuild_testcase/build
Checking for 'gcc' (c compiler)          : ok
'configure' finished successfully (0.092s)
'setup' finished successfully (0.001s)
Waf: Entering directory `/tmp/advbuild_testcase/build'
[1/2] c: main.c -> build/main.c.0.o
[2/2] cprogram: build/main.c.0.o -> build/app
Waf: Leaving directory `/tmp/advbuild_testcase/build'
'build' finished successfully (0.137s)
'setup' finished successfully (0.002s)
Waf: Entering directory `/tmp/advbuild_testcase/build'
[1/2] c: main.c -> build/main.c.0.o
[2/2] cprogram: build/main.c.0.o -> build/app
Waf: Leaving directory `/tmp/advbuild_testcase/build'
'build' finished successfully (0.125s)
```

7.1.3. Binding a command from a Waf tool

When the top-level wscript is read, it is converted into a python module and kept in memory. Commands may be added dynamically by injecting the desired function into that module. We will now show how to bind a simple command from a Waf tool:

```
top = '.'
out = 'build'

def options(opt):
        opt.load('some_tool', tooldir='.')

def configure(conf):
        pass
```

Waf tools are loaded once for the configuration and for the build. To ensure that the tool is always enabled, it is mandatory to load its options, even if the tool does not actually provide options. Our tool *some_tool.py*, located next to the *wscript* file, will contain the following code:

```
from waflib import Context

def cnt(ctx): ❶
        """do something"""
        print('just a test')

Context.g_module.__dict__['cnt'] = cnt ❷
```

❶ The function to bind must accept a `Context` object as first argument

❷ The main wscript file of the project is loaded as a python module and stored as `Context.g_module`

The execution output will be the following.

```
$ waf configure cnt
Setting top to   : /tmp/examples/advbuild_cmdtool
Setting out to   : /tmp/advbuild_cmdtool/build
'configure' finished successfully (0.006s)
just a test
'cnt' finished successfully (0.001s)
```

7.2. Custom build outputs

7.2.1. Multiple configuration folders

The *WAFLOCK* environment variable is used to control the configuration lock and to point at the default build directory. Observe the results on the following project:

```
def configure(ctx):
        pass

def build(ctx):
        ctx(rule='touch ${TGT}', target='foo.txt')
```

We will change the *WAFLOCK* variable in the execution:

```
$ export WAFLOCK=.lock-wafdebug ❶

$ waf
Waf: Entering directory `/tmp/advbuild_waflock/debug'
[1/1] foo.txt:  -> debug//foo.txt ❷
Waf: Leaving directory `/tmp/advbuild_waflock/debug'
'build' finished successfully (0.012s)

$ export WAFLOCK=.lock-wafrelease

$ waf distclean configure
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.176s)

$ waf
Waf: Entering directory `/tmp/advbuild_waflock/release' ❸
[1/1] foo.txt:  -> release/foo.txt
Waf: Leaving directory `/tmp/advbuild_waflock/release'
'build' finished successfully (0.034s)

$ tree -a
.
|-- .lock-debug ❹
|-- .lock-release
|-- debug
|   |-- .wafpickle-7
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- _cache.py
|   |-- config.log
|   `-- foo.txt
|-- release
|   |-- .wafpickle-7
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- _cache.py
|   |-- config.log
|   `-- foo.txt
`-- wscript
```

❶ The lock file points at the configuration of the project in use and at the build directory to use

❷ The files are output in the build directory debug

❸ The configuration *release* is used with a different lock file and a different build directory.

❹ The contents of the project directory contain the two lock files and the two build folders.

The lock file name may also be changed from the code by changing the appropriate variable in the waf file:

```
from waflib import Options
Options.lockfile = '.lock-wafname'
```

> The output directory pointed at by the waf lock file only has effect when not given in the waf file

7.2.2. Changing the output directory

Variant builds

In the previous section, two different configurations were used for similar builds. We will now show how to inherit the same configuration by two different builds, and how to output the targets in different folders. Let's start with the project file:

```
def configure(ctx):
        pass

def build(ctx):
        ctx(rule='touch ${TGT}', target=ctx.cmd + '.txt') ❶

from waflib.Build import BuildContext
class debug(BuildContext): ❷
        cmd = 'debug'
        variant = 'debug' ❸
```

❶ The command being called is *self.cmd*

❷ Create the *debug* command inheriting the build context

❸ Declare a folder for targets of the *debug* command

This project declares two different builds *build* and *debug*. Let's examine the execution output:

```
waf configure build debug
Setting top to   : /tmp/advbuild_variant
Setting out to   : /tmp/advbuild_variant/build
'configure' finished successfully (0.007s)
Waf: Entering directory `/tmp/advbuild_variant/build'
[1/1] build.txt:  -> build/build.txt
Waf: Leaving directory `/tmp/advbuild_variant/build'
'build' finished successfully (0.020s)
Waf: Entering directory `/tmp/build_variant/build/debug'
[1/1] debug.txt:  -> build/debug/debug.txt ❶
Waf: Leaving directory `/tmp/advbuild_variant/build/debug'
'debug' finished successfully (0.021s)

$ tree
.
|-- build
|   |-- build.txt ❷
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- _cache.py
|   |-- config.log
|   `-- debug
|       `-- debug.txt ❸
`-- wscript
```

❶ Commands are executed from *build/variant*

❷ The default *build* command does not have any variant

❸ The target *debug* is under the variant directory in the build directory

Configuration sets for variants

The variants may require different configuration sets created during the configuration. Here is an example:

```
def options(opt):
        opt.load('compiler_c')

def configure(conf):
        conf.setenv('debug') ❶
        conf.load('compiler_c')
        conf.env.CFLAGS = ['-g'] ❷

        conf.setenv('release')
        conf.load('compiler_c')
        conf.env.CFLAGS = ['-O2']

def build(bld):
        if not bld.variant: ❸
```

```
            bld.fatal('call "waf build_debug" or "waf build_release", and try "waf --help"')
        bld.program(source='main.c', target='app', includes='.') ❹

from waflib.Build import BuildContext, CleanContext, \
        InstallContext, UninstallContext

for x in 'debug release'.split():
        for y in (BuildContext, CleanContext, InstallContext, UninstallContext):
                name = y.__name__.replace('Context','').lower()
                class tmp(y): ❺
                        cmd = name + '_' + x
                        variant = x
```

❶ Create a new configuration set to be returned by *conf.env*, and stored in *c4che/debug_cache.py*

❷ Modify some data in the configuration set

❸ Make sure a variant is set, this will disable the normal commands *build*, *clean* and *install*

❹ *bld.env* will load the configuration set of the appropriate variant (*debug_cache.py* when in *debug*)

❺ Create new commands such as *clean_debug* or *install_debug* (the class name does not matter)

The execution output will be similar to the following:

```
$ waf clean_debug build_debug clean_release build_release
'clean_debug' finished successfully (0.005s)
Waf: Entering directory `/tmp/examples/advbuild_variant_env/build/debug'
[1/2] c: main.c -> build/debug/main.c.0.o
[2/2] cprogram: build/debug/main.c.0.o -> build/debug/app
Waf: Leaving directory `/tmp/examples/advbuild_variant_env/build/debug'
'build_debug' finished successfully (0.051s)
'clean_release' finished successfully (0.003s)
Waf: Entering directory `/tmp/examples/advbuild_variant_env/build/release'
[1/2] c: main.c -> build/release/main.c.0.o
[2/2] cprogram: build/release/main.c.0.o -> build/release/app
Waf: Leaving directory `/tmp/examples/advbuild_variant_env/build/release'
'build_release' finished successfully (0.052s)
```

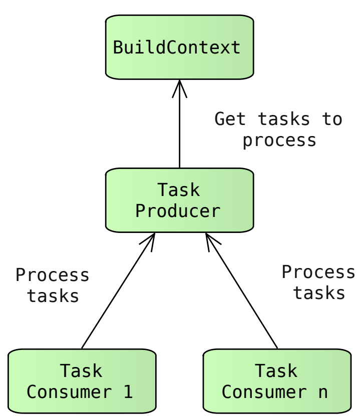
8. Task processing

This chapter provides a description of the task classes which are used during the build phase.

8.1. Task execution

8.1.1. Main actors

The build context is only used to create the tasks and to return lists of tasks that may be executed in parallel. The scheduling is delegated to a task producer which lets task consumers to execute the tasks. The task producer keeps a record of the build state such as the amount of tasks processed or the errors.



The amount of consumers is determined from the number of processors, or may be set manually by using the *-j* option:

```
$ waf -j3
```

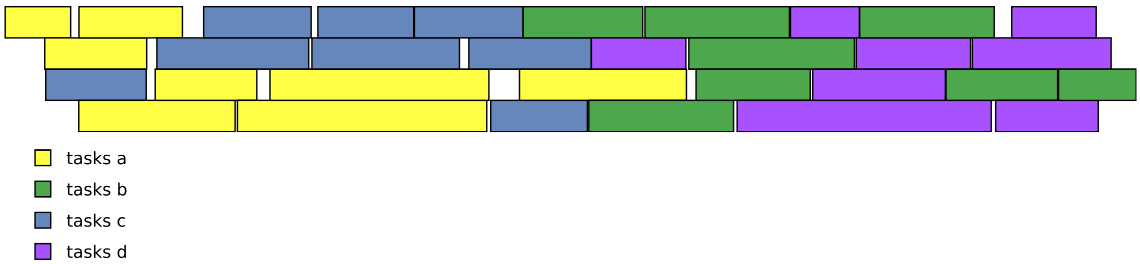
8.1.2. Build groups

The task producer iterates over lists of tasks returned by the build context. Although the tasks from a list may be executed in parallel by the consumer threads, all the tasks from one list must be consumed before processing another list of tasks. The build ends when there are no more tasks to process.

These lists of tasks are called *build groups* and may be accessed from the build scripts. Let's demonstrate this behaviour on an example:

```
def build(ctx):
    for i in range(8):
        ctx(rule='cp ${SRC} ${TGT}', source='wscript', target='wscript_a_%d' % i,
            color='YELLOW', name='tasks a')
        ctx(rule='cp ${SRC} ${TGT}', source='wscript_a_%d' % i, target='wscript_b_%d' % i,
            color='GREEN', name='tasks b')
    for i in range(8)
        ctx(rule='cp ${SRC} ${TGT}', source='wscript', target='wscript_c_%d' % i,
            color='BLUE', name='tasks c')
        ctx(rule='cp ${SRC} ${TGT}', source='wscript_c_%d' % i, target='wscript_d_%d' % i,
            color='PINK', name='tasks d')
```

Each green task must be executed after one yellow task and each pink task must be executed after one blue task. Because there is only one group by default, the parallel execution will be similar to the following:

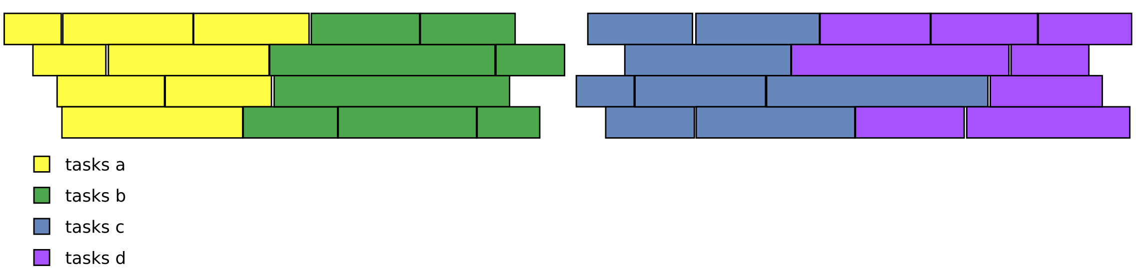


Parallel build representation for "waf -j4" (one group by default)

We will now modify the example to add one more build group.

```
def build(ctx):
    for i in range(8):
        ctx(rule='cp ${SRC} ${TGT}', source='wscript', target='wscript_a_%d' % i,
            color='YELLOW', name='tasks a')
        ctx(rule='cp ${SRC} ${TGT}', source='wscript_a_%d' % i, target='wscript_b_%d' % i,
            color='GREEN', name='tasks b')
    ctx.add_group()
    for i in range(8):
        ctx(rule='cp ${SRC} ${TGT}', source='wscript', target='wscript_c_%d' % i,
            color='BLUE', name='tasks c')
        ctx(rule='cp ${SRC} ${TGT}', source='wscript_c_%d' % i, target='wscript_d_%d' % i,
            color='PINK', name='tasks d')
```

Now a separator will appear between the group of yellow and green tasks and the group of blue and violet tasks:



Parallel build representation for "waf -j4" (two groups)

The tasks and tasks generator are added implicitly to the current group. By giving a name to the groups, it is easy to control what goes where:

```
def build(ctx):

    ctx.add_group('group1')
    ctx.add_group('group2')

    for i in range(8):
        ctx.set_group('group1')
        ctx(rule='cp ${SRC} ${TGT}', source='wscript', target='wscript_a_%d' % i,
            color='YELLOW', name='tasks a')
        ctx(rule='cp ${SRC} ${TGT}', source='wscript_a_%d' % i, target='wscript_b_%d' % i,
```

```
            color='GREEN', name='tasks b')

        ctx.set_group('group2')
        ctx(rule='cp ${SRC} ${TGT}', source='wscript', target='wscript_c_%d' % i,
            color='BLUE', name='tasks c')
        ctx(rule='cp ${SRC} ${TGT}', source='wscript_c_%d' % i, target='wscript_d_%d' % i,
            color='PINK', name='tasks d')
```

In the previous examples, all task generators from all build groups are processed before the build actually starts. This default is provided to ensure that the task count is as accurate as possible. Here is how to tune the build groups:

```
def build(ctx):
        from waflib.Build import POST_LAZY, POST_AT_ONCE
        ctx.post_mode = POST_AT_ONCE ❶
        #ctx.post_mode = POST_LAZY ❷
```

❶ All task generators create their tasks before the build starts (default behaviour)

❷ Groups are processed sequentially: all tasks from previous groups are executed before the task generators from the next group are processed

Build groups can be used for building a compiler to generate more source files to process.

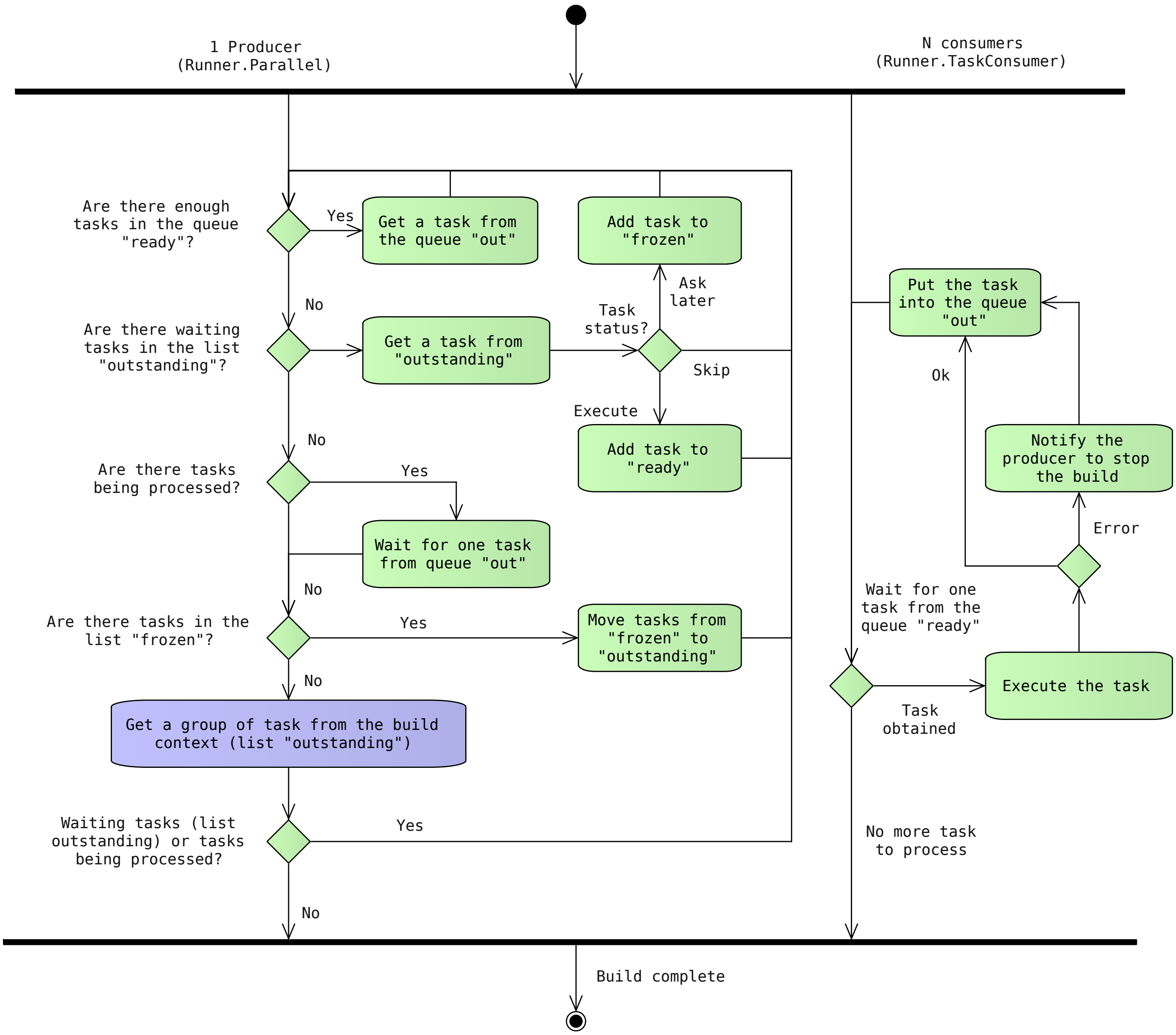
8.1.3. The Producer-consumer system

In most python interpreters, a global interpreter lock prevents parallelization by more than one CPU core at a time. Therefore, it makes sense to restrict the task scheduling on a single task producer, and to let the threads access only the task execution.

The communication between producer and consumers is based on two queues *ready* and *out*. The producer adds the tasks to *ready* and reads them back from *out*. The consumers obtain the tasks from *ready* and give them back to the producer into *out* after executing *task.run*.

The producer uses the an internal list named *outstanding* to iterate over the tasks and to decide which ones to put in the queue *ready*. The tasks that cannot be processed are temporarily output in the list *frozen* to avoid looping endlessly over the tasks waiting for others.

The following illustrates the relationship between the task producers and consumers as performed during the build:



8.1.4. Task states and status

A state is assigned to each task (*task.hasrun = state*) to keep track of the execution. The possible values are the following:

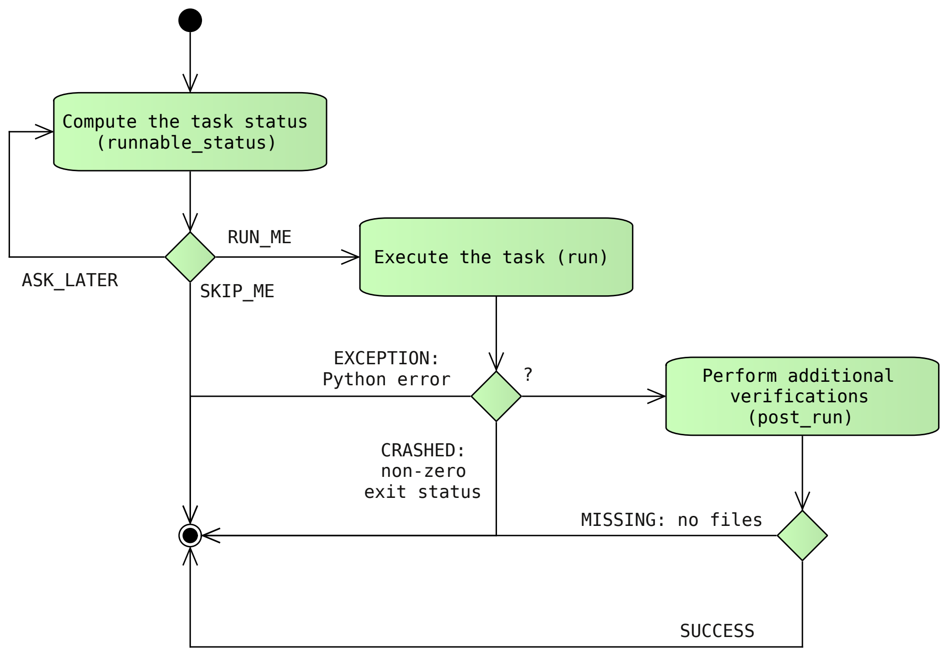
State	Numeric value	Description
NOT_RUN	0	The task has not been processed yet
MISSING	1	The task outputs are missing
CRASHED	2	The task method *run* returned a non-0 value
EXCEPTION	3	An exception occurred in the Task method *run*
SKIPPED	8	The task was skipped (it was up-to-date)
SUCCESS	9	The execution was successful

To decide to execute a task or not, the producer uses the value returned by the task method *runnable_status*. The possible return values are the following:

Code	Description
ASK_LATER	The task may depend on other tasks which have not finished to run (not ready)
SKIP_ME	The task does not have to be executed, it is up-to-date

Code	Description
RUN_ME	The task is ready to be executed

The following diagram represents the interaction between the main task methods and the states and status:



8.2. Build order constraints

8.2.1. The method set_run_after

The method *set_run_after* is used to declare ordering constraints between tasks:

```
task1.set_run_after(task2)
```

The tasks to wait for are stored in the attribute *run_after*. They are used by the method *runnable_status* to yield the status *ASK_LATER* when a task has not run yet. This is merely for the build order and not for forcing a rebuild if one of the previous tasks is executed.

8.2.2. Computed constraints

Attribute after/before

The attributes *before* and *after* are used to declare ordering constraints between tasks:

```
from waflib.Task import Task
class task_test_a(Task):
    before = ['task_test_b']
class task_test_b(Task):
    after  = ['task_test_a']
```

ext_in/ext_out

Another way to force the order is by declaring lists of abstract symbols on the class attributes. This way the classes are not named explicitly, for example:

```
from waflib.Task import Task
class task_test_a(Task):
    ext_in  = ['.h']
class task_test_b(Task):
    ext_out = ['.h']
```

The *extensions* ext_in and ext_out do not mean that the tasks have to produce files with such extensions, but are mere symbols for use as precedence constraints.

Order extraction

Before feeding the tasks to the producer-consumer system, a constraint extraction is performed on the tasks having input and output files. The attributes *run_after* are initialized with the tasks to wait for.

The two functions called on lists of tasks are:

1. *waflib.Task.set_precedence_constraints*: extract the build order from the task classes attributes ext_in/ext_out/before/after
2. *waflib.Task.set_file_constraints*: extract the constraints from the tasks having input and output files

8.2.3. Weak order constraints

A new priority system introduced in Waf 2.0 applies priorities to tasks processed in parallel. The first level of priority ensures that tasks part of long chains or large trees are processed as early as possible to prevent bottlenecks. By default, the priority of a task increases proportionally with the amount of tasks depending on it.

The priority value can yet be enhanced to account for the running time of the task. The following diagram represents a case of a slow red task that increases the total duration of a parallel build (swimlanes represent threads processing tasks):

Same priority for green and red tasks (waf -j2)

Altering the `weight` value on the slow task (on the class or on instances) has the effect of raising the priority of instances:

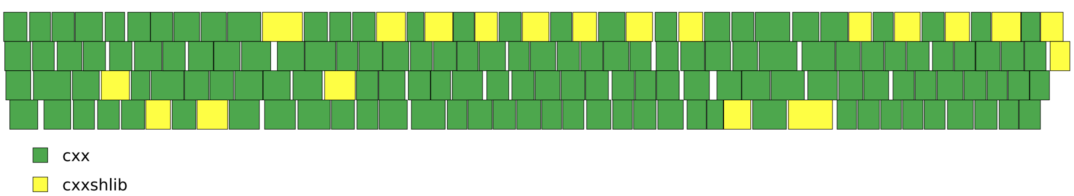
```
class long_red_task(Task.Task):
    run_str = 'sleep 60'
    weight = 3  # default is 0
    color = 'RED'
```

As a result, the slow task is then scheduled to run first and the overall build time is minimized:

Increased red task weight (waf -j2)

For task graphs having mostly the same tree weight, a second priority level sorts tasks by order of task generator declaration. Thanks to this system, user scripts can be optimized to interrupt the build early in case of typical failures (syntax errors, unit tests, etc).

In addition, this system also tends to spread out the execution of similar task types. For example, clusters of slow link tasks are less likely to occur at the very end of a full build, thus limiting the possibilities of memory over-consumption:



Parallel build of 20 shared libraries (waf -j4)

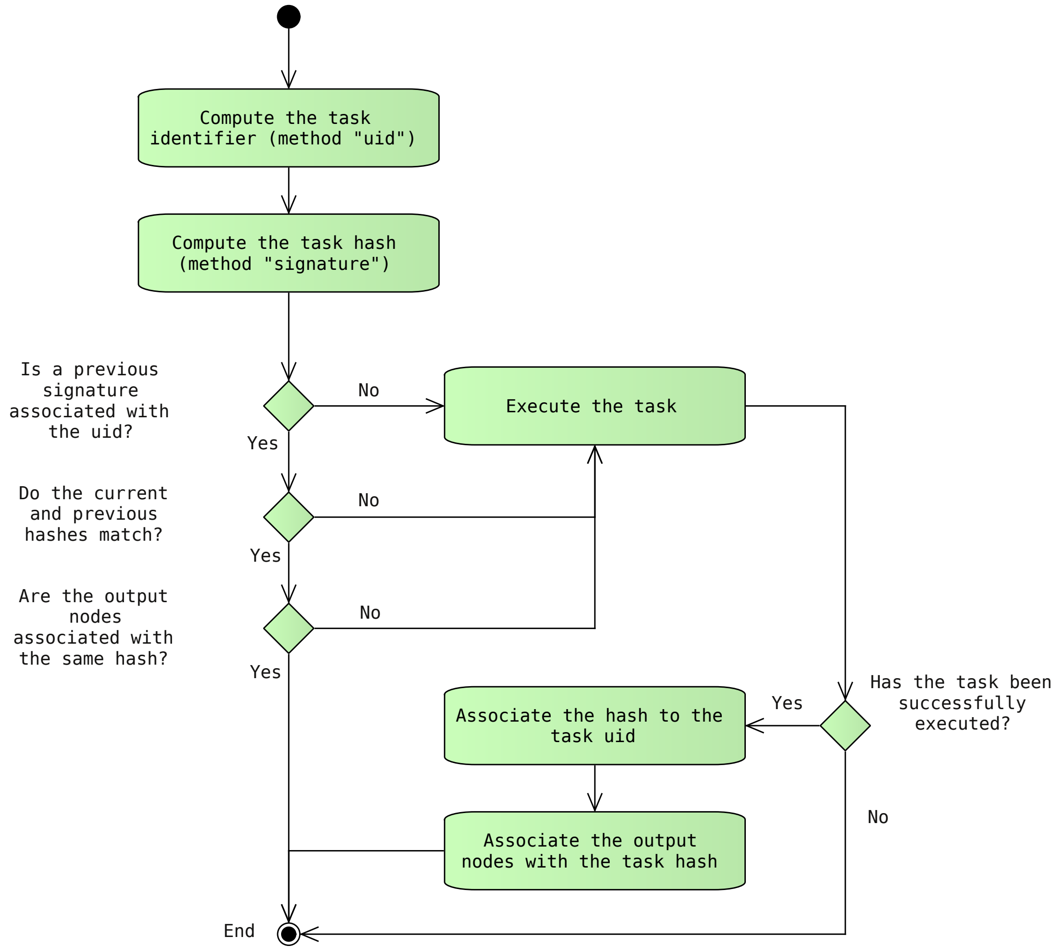
8.3. Dependencies

8.3.1. Task signatures

The direct instances of *waflib.Task.TaskBase* are very limited and cannot be used to track file changes. The subclass *waflib.Task.Task* provides the necessary features for the most common builds in which source files are used to produce target files.

The dependency tracking is based on the use of hashes of the dependencies called **task signatures**. The signature is computed from various dependencies source, such as input files and configuration set values.

The following diagram describes how *waflib.Task.Task* instances are processed:



The following data is used in the signature computation:

1. Explicit dependencies: *input nodes* and dependencies set explicitly by using *bld.depends_on*
2. Implicit dependencies: dependencies searched by a scanner method (the method *scan*)
3. Values: configuration set values such as compilation flags

8.3.2. Explicit dependencies

Input and output nodes

The task objects do not directly depend on other tasks. Other tasks may exist or not, and be executed or nodes. Rather, the input and output nodes hold themselves signatures values, which come from different sources:

1. Nodes for build files usually inherit the signature of the task that generated the file
2. Nodes from elsewhere have a signature computed automatically from the file contents (hash)

Global dependencies on other nodes

The tasks may be informed that some files may depend on other files transitively without listing them in the inputs. This is achieved by the method *add_manual_dependency* from the build context:

```
def configure(ctx):
    pass

def build(ctx):
    ctx(rule='cp ${SRC} ${TGT}', source='wscript', target='somecopy')
    ctx.add_manual_dependency(
        ctx.path.find_node('wscript'),
        ctx.path.find_node('testfile'))
```

The file *somecopy* will be rebuilt whenever *wscript* or *testfile* change, even by one character:

```
$ waf build
Waf: Entering directory `/tmp/tasks_manual_deps/build'
[1/1] somecopy: wscript -> build/somecopy
Waf: Leaving directory `/tmp/tasks_manual_deps/build'
'build' finished successfully (0.034s)
```

```
$ waf
Waf: Entering directory `/tmp/tasks_manual_deps/build'
Waf: Leaving directory `/tmp/tasks_manual_deps/build'
'build' finished successfully (0.006s)

$ echo " " >> testfile

$ waf
Waf: Entering directory `/tmp/tasks_manual_deps/build'
[1/1] somecopy: wscript -> build/somecopy
Waf: Leaving directory `/tmp/tasks_manual_deps/build'
'build' finished successfully (0.022s)
```

8.3.3. Implicit dependencies (scanner methods)

Some tasks can be created dynamically after the build has started, so the dependencies cannot be known in advance. Task subclasses can provide a method named *scan* to obtain additional nodes implicitly. In the following example, the *copy* task provides a scanner method to depend on the wscript file found next to the input file.

```
import time
from waflib.Task import Task
class copy(Task):

    def run(self):
        return self.exec_command('cp %s %s' % (self.inputs[0].abspath(), self.outputs[0].abspath()))

    def scan(self): ❶
        print('→ calling the scanner method')
        node = self.inputs[0].parent.find_resource('wscript')
        return ([node], time.time()) ❷

    def runnable_status(self):
        ret = super(copy, self).runnable_status() ❸
        bld = self.generator.bld ❹
        print('nodes:       %r' % bld.node_deps[self.uid()]) ❺
        print('custom data: %r' % bld.raw_deps[self.uid()]) ❻
        return ret

def configure(ctx):
    pass

def build(ctx):
    tsk = copy(env=ctx.env) ❼
    tsk.set_inputs(ctx.path.find_resource('a.in'))
    tsk.set_outputs(ctx.path.find_or_declare('b.out'))
    ctx.add_to_group(tsk)
```

❶ A scanner method

❷ The return value is a tuple containing a list of nodes to depend on and serializable data for custom uses

❸ Override the method runnable_status to add some logging

❹ Obtain a reference to the build context associated to this task

❺ The nodes returned by the scanner method are stored in the map **bld.node_deps**

❻ The custom data returned by the scanner method is stored in the map **bld.raw_deps**

❼ Create a task manually (encapsulation by task generators will be described in the next chapters)

```
$ waf
→ calling the scanner method ❶
nodes:  [/tmp/tasks_scan/wscript]
custom data: 55.51
[1/1] copy: a.in -> build/b.out
'build' finished successfully (0.021s)

$ waf ❷
nodes:  [/tmp/tasks_scan/wscript]
custom data: 1280561555.512006
'build' finished successfully (0.005s)

$ echo " " >> wscript ❸

$ waf
→ calling the scanner method
nodes:  [/tmp/tasks_scan/wscript]
custom data: 64.31
```

```
[1/1] copy: a.in -> build/b.out
'build' finished successfully (0.022s)
```

❶ The scanner method is always called on a clean build

❷ The scanner method is not called when nothing has changed, although the data returned is retrieved

❸ When a dependency changes, the scanner method is executed once again (the custom data has changed)

If the build order is incorrect, the method *scan* may fail to find dependent nodes (missing nodes) or the signature calculation may throw an exception (missing signature for dependent nodes).

8.3.4. Values

The habitual use of command-line parameters such as compilation flags lead to the creation of *dependencies on values*, and more specifically the configuration set values. The Task class attribute *vars* is used to control what values can enter in the signature calculation. In the following example, the task created has no inputs and no outputs nodes, and only depends on the values.

```
from waflib.Task import Task
class foo(Task): ❶
        vars = ['FLAGS'] ❷
        def run(self):
                print('the flags are %r' % self.env.FLAGS) ❸

def options(ctx):
        ctx.add_option('--flags', default='-f', dest='flags', type='string')

def build(ctx):
        ctx.env.FLAGS = ctx.options.flags ❹
        tsk = foo(env=ctx.env)
        ctx.add_to_group(tsk)

def configure(ctx):
        pass
```

❶ Create a task class named *foo*

❷ The task instances will be executed whenever *self.env.FLAGS* changes

❸ Print the value for debugging purposes

❹ Read the value from the command-line

The execution will produce the following output:

```
$ waf --flags abcdef
[1/1] foo:
the flags are 'abcdef' ❶
'build' finished successfully (0.006s)

$ waf --flags abcdef ❷
'build' finished successfully (0.004s)

$ waf --flags abc
[1/1] foo: ❸
the flags are 'abc'
'build' finished successfully (0.006s)
```

❶ The task is executed on the first run

❷ The dependencies have not changed, so the task is not executed

❸ The flags have changed so the task is executed

8.4. Task tuning

8.4.1. Class access

When a task provides an attribute named *run_str* as in the following example:

```
def configure(ctx):
        ctx.env.COPY      = '/bin/cp'
        ctx.env.COPYFLAGS = ['-f']

def build(ctx):
        from waflib.Task import Task
```

```
        class copy(Task):
                run_str = '${COPY} ${COPYFLAGS} ${SRC} ${TGT}'
        print(copy.vars)

        tsk = copy(env=ctx.env)
        tsk.set_inputs(ctx.path.find_resource('wscript'))
        tsk.set_outputs(ctx.path.find_or_declare('b.out'))
        ctx.add_to_group(tsk)
```

It is assumed that *run_str* represents a command-line, and that the variables in *${}* such as *COPYFLAGS* represent variables to add to the dependencies. A metaclass processes *run_str* to obtain the method *run* (called to execute the task) and the variables in the attribute *vars* (merged with existing variables). The function created is displayed in the following output:

```
$ waf --zones=action
13:36:49 action   def f(tsk):
        env = tsk.env
        gen = tsk.generator
        bld = gen.bld
        wd = getattr(tsk, 'cwd', None)
        def to_list(xx):
                if isinstance(xx, str): return [xx]
                return xx
        lst = []
        lst.extend(to_list(env['COPY']))
        lst.extend(to_list(env['COPYFLAGS']))
        lst.extend([a.path_from(bld.bldnode) for a in tsk.inputs])
        lst.extend([a.path_from(bld.bldnode) for a in tsk.outputs])
        lst = [x for x in lst if x]
        return tsk.exec_command(lst, cwd=wd, env=env.env or None)
[1/1] copy: wscript -> build/b.out
['COPY', 'COPYFLAGS']
'build' finished successfully (0.007s)
```

All subclasses of *waflib.Task.TaskBase* are stored on the module attribute *waflib.Task.classes*. Therefore, the *copy* task can be accessed by using:

```
from waflib import Task
cls = Task.classes['copy']
```

8.4.2. Scriptlet expressions

Although the *run_str* is aimed at configuration set variables, a few special cases are provided for convenience:

1. If the value starts by **env**, **gen**, **bld** or **tsk**, a method call will be made
2. If the value starts by SRC[n] or TGT[n], a method call to the input/output node *n* will be made
3. SRC represents the list of task inputs seen from the root of the build directory
4. TGT represents the list of task outputs seen from the root of the build directory

Here are a few examples:

```
${SRC[0].parent.abspath()} ❶
${bld.root.abspath()} ❷
${tsk.uid()} ❸
${CPPPATH_ST:INCPATHS} ❹
```

❶ Absolute path of the parent folder of the task first source file

❷ File system root

❸ Print the task unique identifier

❹ Perform a map replacement equivalent to *[env.CPPPATH_ST % x for x in env.INCPATHS]*

8.4.3. Command chains

In order to improve script portability *run_str* can also be a list of command strings or functions. All sub-commands must complete for the task to succeed.

The following example shows how to chain three commands to copy `wscript` to `wscript.out`. The output file will be removed first if it already exists. The copy is then performed. When successful, the output file permissions are changed to turn it into an executable.

```
def build(ctx):
        import os
        from waflib import Utils, Task

        def chmod_fun(task): ❶
```

```
                for x in task.outputs:
                        os.chmod(x.abspath(), Utils.O755)
                return 0

        def remove_fun(task):
                for x in task.outputs:
                        try:
                                os.remove(x.abspath())
                        except OSError:
                                if os.path.exists(x.abspath()):
                                        raise ValueError('Cannot remove %r' % x) ❷
                return 0

        class complex_copy(Task.Task):
                run_str = (remove_fun, "${CP} ${SRC} ${TGT}", chmod_fun) ❸

        ctx.env.CP = '/bin/cp' ❹
        tsk = complex_copy(env=ctx.env.derive())
        tsk.set_inputs(ctx.path.find_resource('wscript'))
        tsk.set_outputs(ctx.path.find_or_declare('wscript.out'))
        ctx.add_to_group(tsk)
```

❶ The function arguments is the same as that of *run* methods. Function commands return 0 or None in case of success.

❷ Function commands can raise exceptions such as OSError, ValueError or WafError

❸ Several functions or strings can be chained together; the build will run the commands in the order specified

❹ Rebuilds will occur when variables used in command strings are modified, and when command strings or function definitions change

Note: Chaining is also possible through subclassing.

8.4.4. Direct class modifications

Always execute

The attribute *waflib.Task.always_run* forces a task to be executed whenever a build is performed. The Task method *runnable_status* is then made to simply return *RUN_ME*.

```
def configure(ctx):
    pass

def build(ctx):
    from waflib import Task
    class copy(Task.Task):
        always_run = True
        run_str = 'cp ${SRC} ${TGT}'

    tsk = copy(env=ctx.env)
    tsk.set_inputs(ctx.path.find_resource('wscript'))
    tsk.set_outputs(ctx.path.find_or_declare('b.out'))
    ctx.add_to_group(tsk)
```

For convenience, rule-based task generators can provide an `always` parameter to achieve the same results:

```
def build(ctx):
    ctx(
        rule   = 'echo hello',
        always = True
    )
```

Deep inputs

On incremental builds, only tasks that see their dependencies change are run. The contents of task inputs (Node objects) constitutes one such dependency. After a task is executed, task output changes (Node objects again) may then trigger other tasks in a cascade.

This scheme minimizes rebuilds, and is usually considered a desirable feature. Dependencies onto other files are typically meant to be declared explicitly:

1. As outputs from a scanner method
2. As input Node objects, which can even be unused in a task command
3. As implicit Node dependencies in the list attribute `task.node_deps`

These schemes require the list of dependencies must be specified explicitly so the can sometimes become a hassle. For example, in the case of C/C++ unit tests, test programs are meant to be re-run on upstream shared library changes, and tests can depend on a lot of input libraries. While this particular behaviour can obviously implemented by overriding the method `runnable_status` on a given task, a shortcut can simplify a lot of code.

The decorator called `deep_inputs` enables tasks to re-run whenever the tasks associated to their inputs (Node objects in the lists `task.inputs` or `task.node_deps`) have changed, and even when the input files are identical:

```
from waflib import Task
@Task.deep_inputs
class utest(Task.Task):
        color = 'PINK'
        def run(self):
                ...
```

9. Task generators

9.1. Rule-based task generators (Make-like)

This chapter illustrates the use of rule-based task generators for building simple targets.

9.1.1. Declaration and usage

Rule-based task generators are a particular category of task generators producing exactly one task.

The following example shows a task generator producing the file *foobar.txt* from the project file *wscript* by executing the command *cp* to perform a copy:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld( ❶
                rule   = 'cp ${SRC} ${TGT}', ❷
                source = 'wscript', ❸
                target = 'foobar.txt', ❹
        )
```

❶ To instantiate a new task generator, remember that all arguments have the form *key=value*

❷ The attribute *rule* represents the command to execute in a readable manner (more on this in the next chapters).

❸ Source files, either in a space-delimited string, or in a list of python strings

❹ Target files, either in a space-delimited string, or in a list of python strings

Upon execution, the following output will be observed:

```
$ waf distclean configure build -v
'distclean' finished successfully (0.000s)
'configure' finished successfully (0.021s)
Waf: Entering directory `/tmp/rules_simple/build'
[1/1] foobar.txt: wscript -> build/foobar.txt ❶
10:57:33 runner 'cp ../wscript foobar.txt' ❷
Waf: Leaving directory `/tmp/rules_simple/build'
'build' finished successfully (0.016s)

$ tree
.
|-- build
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- _cache.py
|   |-- config.log
|   `-- foobar.txt
`-- wscript

$ waf ❸
Waf: Entering directory `/tmp/rules_simple/build'
Waf: Leaving directory `/tmp/rules_simple/build'
'build' finished successfully (0.006s)

$ echo " " >> wscript ❹

$ waf
```

```
Waf: Entering directory `/tmp/rules_simple/build'
[1/1] foobar.txt: wscript → build/foobar.txt ❺
Waf: Leaving directory `/tmp/rules_simple/build'
'build' finished successfully (0.013s)
```

❶ In the first execution, the target is correctly created

❷ Command-lines are only displayed in *verbose mode* by using the option *-v*

❸ The target is up-to-date, so the task is not executed

❹ Modify the source file in place by appending a space character

❺ Since the source has changed, the target is created once again

The string for the rule also enters in the dependency calculation. If the rule changes, then the task will be recompiled.

9.1.2. Rule functions

Rules may be given as expression strings or as python function. The function is assigned to the task class created:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        def run(task): ❶
                src = task.inputs[0].abspath() ❷
                tgt = task.outputs[0].abspath() ❸
                cmd = 'cp %s %s' % (src, tgt)
                print(cmd)
                return task.exec_command(cmd) ❹

        bld(
                rule   = run, ❺
                source = 'wscript',
                target = 'same.txt',
        )
```

❶ Rule functions take the task instance as parameter.

❷ Sources and targets are represented internally as Node objects bound to the task instance.

❸ Commands are executed from the root of the build directory. Node methods such as *bldpath* ease the command line creation.

❹ The task class holds a wrapper around subprocess.Popen(…) to execute commands.

❺ Use a function instead of a string expression

The execution trace will be similar to the following:

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/rule_function/out'
[1/1] same.txt: wscript -> out/same.txt
cp /tmp/rule_function/wscript /tmp/rule_function/build/same.txt
Waf: Leaving directory `/tmp/rule_function/out'
'build' finished successfully (0.010s)
```

The rule function must return a null value (0, None or False) to indicate success, and must generate the files corresponding to the outputs. The rule function is executed by threads internally so it is important to write thread-safe code (cannot search or create node objects).

Unlike string expressions, functions may execute several commands at once.

9.1.3. Shell usage

The attribute *shell* is used to enable the system shell for command execution. A few points are worth keeping in mind when declaring rule-based task generators:

1. The Waf tools do not use the shell for executing commands
2. The shell is used by default for user commands and custom task generators
3. String expressions containing the following symbols '>', '<' or '&' cannot be transformed into functions to execute commands without a shell, even if told to
4. In general, it is better to avoid the shell whenever possible to avoid quoting problems (paths having blank characters in the name for example)
5. The shell is creating a performance penalty which is more visible on win32 systems.

Here is an example:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(rule='cp ${SRC} ${TGT}', source='wscript', target='f1.txt', shell=False)
        bld(rule='cp ${SRC} ${TGT}', source='wscript', target='f2.txt', shell=True)
```

Upon execution, the results will be similar to the following:

```
$ waf distclean configure build --zones=runner,action
'distclean' finished successfully (0.004s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/rule/out'
23:11:23 action ❶
def f(task):
        env = task.env
        wd = getattr(task, 'cwd', None)
        def to_list(xx):
                if isinstance(xx, str): return [xx]
                return xx
        lst = []
        lst.extend(['cp'])
        lst.extend([a.srcpath(env) for a in task.inputs])
        lst.extend([a.bldpath(env) for a in task.outputs])
        lst = [x for x in lst if x]
        return task.exec_command(lst, cwd=wd)

23:11:23 action
def f(task):
        env = task.env
        wd = getattr(task, 'cwd', None)
        p = env.get_flat
        cmd = ''' cp %s %s ''' % (" ".join([a.srcpath(env) for a in task.inputs]), ❷
                " ".join([a.bldpath(env) for a in task.outputs]))
        return task.exec_command(cmd, cwd=wd)

[1/2] f1.txt: wscript -> out/f1.txt
23:11:23 runner system command -> ['cp', '../wscript', 'f1.txt'] ❸
[2/2] f2.txt: wscript -> out/f2.txt
23:11:23 runner system command ->  cp ../wscript f2.txt
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.017s)
```

❶ String expressions are converted to functions (here, without the shell).

❷ Command execution by the shell. Notice the heavy use of string concatenation.

❸ Commands to execute are displayed by calling *waf --zones=runner*. When called without the shell, the arguments are displayed as a list.

For performance and maintainability, try avoiding the shell whenever possible

9.1.4. Inputs and outputs

Source and target arguments are optional for make-like task generators, and may point at one or several files at once. Here are a few examples:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld( ❶
                rule   = 'cp ${SRC} ${TGT[0].abspath()} && cp ${SRC} ${TGT[1].abspath()}',
                source = 'wscript',
                target = 'f1.txt f2.txt',
                shell  = True
        )
```

```
	bld( ❷
		source = 'wscript',
		rule   = 'echo ${SRC}'
	)

	bld( ❸
		target = 'test.k3',
		rule   = 'echo "test" > ${TGT}',
	)

	bld( ❹
		rule   = 'echo 1337'
	)

	bld( ❺
		rule   = "echo 'task always run'",
		always = True
	)
```

❶ Generate *two files* whenever the input or the rule change. Likewise, a rule-based task generator may have multiple input files.

❷ The command is executed whenever the input or the rule change. There are no declared outputs.

❸ No input, the command is executed whenever it changes

❹ No input and no output, the command is executed only when the string expression changes

❺ No input and no output, the command is executed each time the build is called

For the record, here is the output of the build:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.093s)
Waf: Entering directory `/tmp/rule/out'
[1/5] echo 1337:
1337
[2/5] echo 'task always run':
[3/5] echo ${SRC}: wscript
../wscript
[4/5] f1.txt f2.txt: wscript -> out/f1.txt out/f2.txt
task always run
[5/5] test.k3:  -> out/test.k3
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.049s)

$ waf
Waf: Entering directory `/tmp/rule/out'
[2/5] echo 'task always run':
task always run
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.014s)
```

9.2. Name and extension-based file processing

Transformations may be performed automatically based on the file name or on the extension.

9.2.1. Refactoring repeated rule-based task generators into implicit rules

The explicit rules described in the previous chapter become a limitation for processing several files of the same extension. The following code may lead to unmaintainable scripts and to slow builds (large amount of objects):

```
def build(bld):
	for x in 'a.lua b.lua c.lua'.split():
		y = x.replace('.lua', '.luac')
		bld(source=x, target=y, rule='${LUAC} -s -o ${TGT} ${SRC}')
		bld.install_files('${LUADIR}', x)
```

It is desirable to extract the rule from the user scripts in the following manner:

```
def build(bld):
	bld(source='a.lua b.lua c.lua')
```

The following piece of code will enable this functionality. It may be inserted in a waf tool or in the same `wscript` file:

```
from waflib import TaskGen
TaskGen.declare_chain(
        name         = 'luac', ❶
        rule         = '${LUAC} -s -o ${TGT} ${SRC}', ❷
        shell        = False,
        ext_in       = '.lua', ❸
        ext_out      = '.luac', ❹
        reentrant    = False, ❺
        install_path = '${LUADIR}', ❻
)
```

❶ The name for the corresponding task class to use

❷ The rule is the same as for any rule-based task generator. It is passed to the `run_str` attribute of a task class.

❸ Input file, processed by extension

❹ Output files extensions separated by spaces. In this case there is only one output file

❺ The reentrant attribute is used to add the output files as source again, for processing by another implicit rule

❻ String representing the installation path for the output files, similar to the destination path from *bld.install_files*. To disable installation, set it to `False` or `None`.

9.2.2. Chaining more than one command

Now consider the long chain *uh.in* → *uh.a* → *uh.b* → *uh.c*. The following implicit rules demonstrate how to generate the files while maintaining a minimal user script:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(source='uh.in')

from waflib import TaskGen
TaskGen.declare_chain(name='a', rule='cp ${SRC} ${TGT}', ext_in='.in', ext_out='.a',)
TaskGen.declare_chain(name='b', rule='cp ${SRC} ${TGT}', ext_in='.a',  ext_out='.b',)
TaskGen.declare_chain(name='c', rule='cp ${SRC} ${TGT}', ext_in='.b',  ext_out='.c', reentrant = False)
```

During the build phase, the correct compilation order is computed based on the extensions given:

```
$ waf distclean configure build
'distclean' finished successfully (0.000s)
'configure' finished successfully (0.090s)
Waf: Entering directory `/comp/waf/demos/simple_scenarios/chaining/build'
[1/3] a: uh.in -> build/uh.a
[2/3] b: build/uh.a -> build/uh.b
[3/3] c: build/uh.b -> build/uh.c
Waf: Leaving directory `/comp/waf/demos/simple_scenarios/chaining/build'
'build' finished successfully (0.034s)
```

9.2.3. Scanner methods

Because transformation chains rely on implicit transformations, it may be desirable to hide some files from the list of sources. Or, some dependencies may be produced conditionally and may not be known in advance. A *scanner method* is a kind of callback used to find additional dependencies just before the target is generated. For illustration purposes, let us start with an empty project containing three files: the *wscript*, *ch.in* and *ch.dep*

```
$ cd /tmp/smallproject

$ tree
.
|-- ch.dep
|-- ch.in
`-- wscript
```

The build will create a copy of *ch.in* called *ch.out*. Also, *ch.out* must be rebuild whenever *ch.dep* changes. This corresponds more or less to the following Makefile:

```
ch.out: ch.in ch.dep
        cp ch.in ch.out
```

The user script should only contain the following code:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(source = 'ch.in')
```

The code below is independent from the user scripts and may be located in a Waf tool.

```
def scan_meth(task): ❶
        node = task.inputs[0]
        dep = node.parent.find_resource(node.name.replace('.in', '.dep')) ❷
        if not dep:
                raise ValueError("Could not find the .dep file for %r" % node)
        return ([dep], []) ❸

from waflib import TaskGen
TaskGen.declare_chain(
        name      = 'copy',
        rule      = 'cp ${SRC} ${TGT}',
        ext_in    = '.in',
        ext_out   = '.out',
        reentrant = False,
        scan      = scan_meth, ❹
)
```

❶ The scanner method accepts a task object as input (not a task generator)

❷ Use node methods to locate the dependency (and raise an error if it cannot be found)

❸ Scanner methods return a tuple containing two lists. The first list contains the list of node objects to depend on. The second list contains private data such as debugging information. The results are cached between build calls so the contents must be serializable.

❹ Add the scanner method to chain declaration

The execution trace will be the following:

```
$ echo 1 > ch.in
$ echo 1 > ch.dep ❶

$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/smallproject/build'
[1/1] copy: ch.in -> build/ch.out ❷
Waf: Leaving directory `/tmp/smallproject/build'
'build' finished successfully (0.010s)

$ waf
Waf: Entering directory `/tmp/smallproject/build'
Waf: Leaving directory `/tmp/smallproject/build'
'build' finished successfully (0.005s) ❸

$ echo 2 > ch.dep ❹

$ waf
Waf: Entering directory `/tmp/smallproject/build'
[1/1] copy: ch.in -> build/ch.out ❺
Waf: Leaving directory `/tmp/smallproject/build'
'build' finished successfully (0.012s)
```

❶ Initialize the file contents of *ch.in* and *ch.dep*

❷ Execute a first clean build. The file *ch.out* is produced

❸ The target *ch.out* is up-to-date because nothing has changed

❹ Change the contents of *ch.dep*

❺ The dependency has changed, so the target is rebuilt

Here are a few important points about scanner methods:

1. they are executed only when the target is not up-to-date.
2. they may not modify the *task* object or the contents of the configuration set *task.env*
3. they are executed in a single main thread to avoid concurrency issues

4. the results of the scanner (tuple of two lists) are re-used between build executions (and it is possible to access programmatically those results)
5. the make-like rules also accept a *scan* argument (scanner methods are bound to the task rather than the task generators)
6. they are used by Waf internally for c/c++ support, to add dependencies dynamically on the header files (*.c* → *.h*)

9.2.4. Extension callbacks

In the chain declaration from the previous sections, the attribute *reentrant* was described to control if the generated files are to be processed or not. There are cases however where one of the two generated files must be declared (because it will be used as a dependency) but where it cannot be considered as a source file in itself (like a header in c/c\++). Now consider the following two chains (*uh.in* → *uh.a1* + *uh.a2*) and (*uh.a1* → *uh.b*) in the following example:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        obj = bld(source='uh.in')

from waflib import TaskGen
TaskGen.declare_chain(
        name      = 'a',
        rule      = 'cp ${SRC} ${TGT}',
        ext_in    = '.in',
        ext_out   = ['.a1', '.a2'],
        reentrant = True,
)

TaskGen.declare_chain(
        name      = 'b',
        rule      = 'cp ${SRC} ${TGT}',
        ext_in    = '.a1',
        ext_out   = '.b',
        reentrant = False,
)
```

The following error message will be produced:

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/smallproject'
Waf: Leaving directory `/tmp/smallproject'
Cannot guess how to process bld:///tmp/smallproject/uh.a2 (got mappings ['.a1', '.in'] in
   class TaskGen.task_gen) -> try conf.load(..)?
```

The error message indicates that there is no way to process *uh.a2*. Only files of extension *.a1* or *.in* can be processed. Internally, extension names are bound to callback methods. The error is raised because no such method could be found, and here is how to register an extension callback globally:

```
@TaskGen.extension('.a2')
def foo(*k, **kw):
        pass
```

To register an extension callback locally, a reference to the task generator object must be kept:

```
def build(bld):
        obj = bld(source='uh.in')
        def callback(*k, **kw):
                pass
        obj.mappings['.a2'] = callback
```

The exact method signature and typical usage for the extension callbacks is the following:

```
from waflib import TaskGen
@TaskGen.extension(".a", ".b") ❶
def my_callback(task_gen_object❷, node❸):
        task_gen_object.create_task(
                task_name, ❹
                node,  ❺
                output_nodes) ❻
```

❶ Comma-separated list of extensions (strings)

❷ Task generator instance holding the data

❸ Instance of Node, representing a file (either source or build)

❹ The first argument to create a task is the name of the task class

❺ The second argument is the input node (or a list of nodes for several inputs)

❻ The last parameter is the output node (or a list of nodes for several outputs)

The creation of new task classes will be described in the next section.

9.2.5. Task class declaration

Waf tasks are instances of the class Task.TaskBase. Yet, the base class contains the real minimum, and the immediate subclass *Task.Task* is usually chosen in user scripts. We will now start over with a simple project containing only one project *wscript* file and and example file named *ah.in*. A task class will be added.

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(source='uh.in')

from waflib import Task, TaskGen

@TaskGen.extension('.in')
def process(self, node):
        tsk = self.create_task('abcd') ❶
        print(tsk.__class__)

class abcd(Task.Task): ❷
        def run(self): ❸
                print('executing...')
                return 0 ❹
```

❶ Create a new instance of *abcd*. The method *create_task* is a shortcut to make certain the task will keep a reference on its task generator.

❷ Inherit the class Task located in the module Task.py

❸ The method run is called when the task is executed

❹ The task return status must be an integer, which is zero to indicate success. The tasks that have failed will be executed on subsequent builds

The output of the build execution will be the following:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/simpleproject/build'
<class 'wscript_main.abcd'>
[1/1] abcd:
executing...
Waf: Leaving directory `/tmp/simpleproject/build'
'build' finished successfully (0.005s)
```

Although it is possible to write down task classes in plain python, two functions (factories) are provided to simplify the work, for example:

```
Task.simple_task_type( ❶
        'xsubpp', ❷
        rule    = '${PERL} ${XSUBPP} ${SRC} > ${TGT}', ❸
        color   = 'BLUE', ❹
        before  = 'cc') ❺

def build_it(task):
        return 0

Task.task_type_from_func(❻
        'sometask', ❼
        func    = build_it, ❽
        vars    = ['SRT'],
        color   = 'RED',
        ext_in  = '.in',
        ext_out = '.out') ❾
```

❶ Create a new task class executing a rule string

❷ Task class name

❸ Rule to execute during the build

❹ Color for the output during the execution

❺ Execute the task instance before any instance of task classes named *cc*. The opposite of *before* is *after*

❻ Create a new task class from a custom python function. The *vars* attribute represents additional configuration set values to use as dependencies

❼ Task class name

❽ Function to use

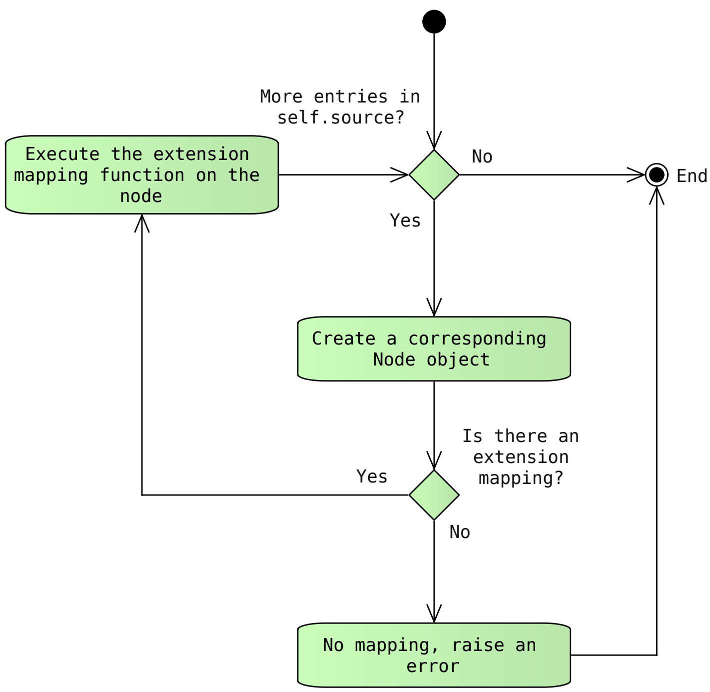
❾ In this context, the extension names are meant to be used for computing the execution order with other tasks, without naming the other task classes explicitly

Note that most attributes are common between the two function factories. More usage examples may be found in most Waf tools.

9.2.6. Source attribute processing

The first step in processing the source file attribute is to convert all file names into Nodes. Special methods may be mapped to intercept names by the exact file name entry (no extension). The Node objects are then added to the task generator attribute *source*.

The list of nodes is then consumed by regular extension mappings. Extension methods may re-inject the output nodes for further processing by appending them to the the attribute *source* (hence the name re-entrant provided in declare_chain).



9.3. General purpose task generators

So far, various task generators uses have been demonstrated. This chapter provides a detailed description of task generator structure and usage.

9.3.1. Task generator definition

The chapter on make-like rules illustrated how the attribute *rule* is processed. Then the chapter on name and extension-based file processing illustrated how the attribute *source* is processed (in the absence of the rule attribute). To process *any attribute*, the following properties should hold:

1. Attributes should be processed only when the task generator is set to generate the tasks (lazy processing)
2. There is no list of authorized attributes (task generators may be extended by user scripts)
3. Attribute processing should be controllable on a task generator instance basis (special rules for particular task generators)
4. The extensions should be split into independent files (low coupling between the Waf tools)

Implementing such a system is a difficult problem which lead to the creation of very different designs:

1. *A hierarchy of task generator subclasses* It was abandoned due to the high coupling between the Waf tools: the C tools required knowledge from the D tool for building hybrid applications

2. *Method decoration (creating linked lists of method calls)* Replacing or disabling a method safely was no longer possible (addition-only), so this system disappeared quickly
3. *Flat method and execution constraint declaration* The concept is close to aspect-oriented programming and might scare programmers.

So far, the third design proved to be the most flexible and was kept. Here is how to define a task generator method:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        v = bld(myattr='Hello, world!')
        v.myattr = 'Hello, world!' ❶
        v.myMethod() ❷

from waflib import TaskGen

@TaskGen.taskgen_method ❸
def myMethod(tgen): ❹
        print(getattr(self, 'myattr', None)) ❺
```

❶ Attributes may be set by arguments or by accessing the object. It is set two times in this example.

❷ Call the task generator method explicitly

❸ Use a python decorator

❹ Task generator methods have a unique argument representing the current instance

❺ Process the attribute *myattr* when present (the case in the example)

The output from the build will be the following:

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/simpleproject/build'
hello world
Waf: Leaving directory `/tmp/simpleproject/build'
'build' finished successfully (0.003s)
```

> The method could be bound by using *setattr* directly, like for binding any new method on a python class.

9.3.2. Executing the method during the build

So far, the task generator methods defined are only executed through explicit calls. Another decorator is necessary to have a task generator executed during the build phase automatically. Here is the updated example:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(myattr='Hello, world!')

from waflib import TaskGen

@TaskGen.taskgen_method ❶
@TaskGen.feature('*') ❷
def methodName(self):
        print(getattr(self, 'myattr', None))
```

❶ Bind a method to the task generator class (redundant when other methods such as *TaskGen.feature* are used)

❷ Bind the method to the symbol *myfeature*

The execution results will be the following:

```
$ waf distclean configure build --zones=task_gen ❶
'distclean' finished successfully (0.004s)
'configure' finished successfully (0.001s)
```

```
Waf: Entering directory `/tmp/simpleproject/build'
23:03:44 task_gen posting objects (normal)
23:03:44 task_gen posting >task_gen '' of type task_gen defined in dir:///tmp/simpleproject> 139657958706768
❷
23:03:44 task_gen -> process_rule (139657958706768) ❸
23:03:44 task_gen -> process_source (139657958706768) ❹
23:03:44 task_gen -> methodName (139657958706768) ❺
Hello, world!
23:03:44 task_gen posted ❻
Waf: Leaving directory `/tmp/simpleproject/build'
23:03:44 task_gen posting objects (normal)
'build' finished successfully (0.004s)
```

1. The debugging zone *task_gen* is used to display the task generator methods being executed
2. Display which task generator is being executed
3. The method *process_rule* is used to process the *rule*. It is always executed.
4. The method *process_source* is used to process the *source* attribute. It is always executed except if the method *process_rule* processes a *rule* attribute
5. Our task generator method is executed, and prints *Hello, world!*
6. The task generator methods have been executed, the task generator is marked as done (posted)

9.3.3. Task generator features

So far, the task generator methods we added were declared to be executed by all task generator instances. Limiting the execution to specific task generators requires the use of the *feature* decorator:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(features='ping')
        bld(features='ping pong')

from waflib import TaskGen

@TaskGen.feature('ping')
def ping(self):
        print('ping')

@TaskGen.feature('pong')
def pong(self):
        print('pong')
```

The execution output will be the following:

```
$ waf distclean configure build --zones=task_gen
'distclean' finished successfully (0.003s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/simpleproject/build'
16:22:07 task_gen posting objects (normal)
16:22:07 task_gen posting <task_gen '' of type task_gen defined in dir:///tmp/simpleproject> 140631018237584
16:22:07 task_gen -> process_rule (140631018237584)
16:22:07 task_gen -> process_source (140631018237584)
16:22:07 task_gen -> ping (140631018237584)
ping
16:22:07 task_gen posted
16:22:07 task_gen posting <task_gen '' of type task_gen defined in dir:///tmp/simpleproject> 140631018237776
16:22:07 task_gen -> process_rule (140631018237776)
16:22:07 task_gen -> process_source (140631018237776)
16:22:07 task_gen -> pong (140631018237776)
pong
16:22:07 task_gen -> ping (140631018237776)
ping
16:22:07 task_gen posted
Waf: Leaving directory `/tmp/simpleproject/build'
16:22:07 task_gen posting objects (normal)
'build' finished successfully (0.005s)
```

Although the task generator instances are processed in order, the task generator method execution requires a specific declaration for the order of execution. Here, the method *pong* is executed before the method *ping*

9.3.4. Task generator method execution order

To control the execution order, two new decorators need to be added. We will now show a new example with two custom task generator methods *method1* and *method2*, executed in that order:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(myattr='Hello, world!')

from waflib import TaskGen

@TaskGen.feature('*')
@TaskGen.before('process_source', 'process_rule')
def method1(self):
        print('method 1 %r' % getattr(self, 'myattr', None))

@TaskGen.feature('*')
@TaskGen.before('process_source')
@TaskGen.after('method1')
def method2(self):
        print('method 2 %r' % getattr(self, 'myattr', None))
```

The execution output will be the following:

```
$ waf distclean configure build --zones=task_gen
'distclean' finished successfully (0.003s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/simpleproject/build'
15:54:02 task_gen posting objects (normal)
15:54:02 task_gen posting <task_gen of type task_gen defined in dir:///tmp/simpleproject> 139808568487632
15:54:02 task_gen -> method1 (139808568487632)
method 1 'Hello, world!'
15:54:02 task_gen -> process_rule (139808568487632)
15:54:02 task_gen -> method2 (139808568487632)
method 2 'Hello, world!'
15:54:02 task_gen -> process_source (139808568487632)
15:54:02 task_gen posted
Waf: Leaving directory `/tmp/simpleproject/build'
15:54:02 task_gen posting objects (normal)
'build' finished successfully (0.005s)
```

9.3.5. Adding or removing a method for execution

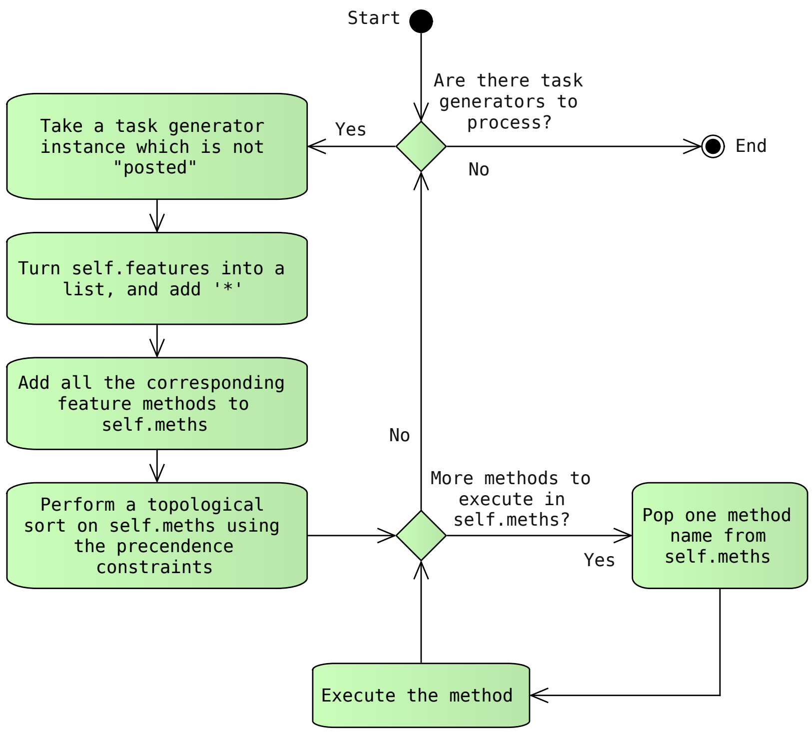
The order constraints on the methods (after/before), are used to sort the list of methods in the attribute *meths*. The sorting is performed once, and the list is consumed as methods are executed. Though no new feature may be added once the first method is executed, new methods may be added dynamically in self.meths. Here is how to create an infinite loop by adding the same method at the end:

```
from waflib.TaskGen import feature

@feature('*')
def infinite_loop(self):
        self.meths.append('infinite_loop')
```

Likewise, methods may be removed from the list of methods to execute:

```
from waflib.TaskGen import feature

@feature('*')
@before_method('process_source')
def remove_process_source(self):
        self.meths.remove('process_source')
```

The task generator method workflow is represented in the following illustration:



9.3.6. Expressing abstract dependencies between task generators

We will now illustrate how task generator methods can be used to express abstract dependencies between task generator objects. Here is a new project file located under */tmp/targets/*:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(rule='echo A', always=True, name='A')
        bld(rule='echo B', always=True, name='B')
```

By executing *waf --targets=B*, only the task generator *B* will create its tasks, and the output will be the following:

```
$ waf distclean configure build --targets=B
'distclean' finished successfully (0.000s)
'configure' finished successfully (0.042s)
Waf: Entering directory `/tmp/targets/build'
[1/1] B:
B
Waf: Leaving directory `/tmp/targets/build'
'build' finished successfully (0.032s)
```

Here is a way to ensure that the task generator *A* has created its tasks when *B* does:

```
top = '.'
out = 'build'

def configure(conf):
    pass

def build(bld):
    bld(rule='echo A', always=True, name='A')
    bld(rule='echo B', always=True, name='B', depends_on='A')

from waflib.TaskGen import feature, before_method
@feature('*') ❶
@before_method('process_rule')
def post_the_other(self):
    deps = getattr(self, 'depends_on', []) ❷
    for name in self.to_list(deps):
        other = self.bld.get_tgen_by_name(name) ❸
```

```
        print('other task generator tasks (before) %s' % other.tasks)
        other.post() ❹
        print('other task generator tasks (after) %s' % other.tasks)
```

❶ This method will be executed for all task generators, before the attribute `rule` is processed

❷ Try to process the attribute `depends_on`, if present

❸ Obtain the task generator by name, and for the same variant

❹ Force the other task generator to create its tasks

The output will be:

```
$ waf distclean configure build --targets=B
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/targets/build'
other task generator tasks (before) [] ❶
other task generator tasks (after) [ ❷
        {task: A  -> }]
[1/2] B:
B
[2/2] A: ❸
A
Waf: Leaving directory `/tmp/targets/build'
'build' finished successfully (0.014s)
```

❶ The other task generator has not created any task yet

❷ A task generator creates all its tasks by calling its method `post()`

❸ Although `--targets=B` was requested, the task from target *A* was created and executed too

In practice, the dependencies will often re-use the task objects created by the other task generator: node, configuration set, etc. This is used by the uselib system (see the next chapter on c/c++ builds).

10. C and C++ projects

Although Waf is language neutral, it is used very often for C and C++ projects. This chapter describes the Waf tools and functions used for these languages.

10.1. Common declaration for C, C++, Fortran and D applications

10.1.1. Predefined task generators

The C/C++ builds consist in transforming (compiling) source files into object files, and to assemble (link) the object files at the end. In theory a single programming language should be sufficient for writing any application, but the situation is usually more complicated:

1. Source files may be created by other compilers in other languages (IDL, ASN1, etc)
2. Additional files may enter in the link step (libraries, object files) and applications may be divided in dynamic or static libraries
3. Different platforms may require different processing rules (manifest files on MS-Windows, etc)

To conceal the implementation details and the portability concerns, each target (program, library) can be wrapped as single task generator object as in the following example:

```
def options(opt):
        opt.load('compiler_c')

def configure(conf):
        conf.load('compiler_c') ❶

def build(bld):
        bld.program(source='main.c', target='app', use='myshlib mystlib') ❷
        bld.stlib(source='a.c', target='mystlib', use='myobjects') ❸
        bld.shlib(source='b.c', target='myshlib') ❹
        bld.objects(source='c.c', target='myobjects')
```

❶ The extensions compiler_c, compiler_cxx, compiler_fc and compiler_d provide support for the C, C++, Fortran and D languages respectively

❷ This declares a program built from *main.c* and using two other libraries

❸ This declares a static library, using the objects from *myobjects*

❹ This declares a shared library

The targets will have different extensions and names depending on the platform. For example on Linux, the contents of the build directory will be:

```
$ tree build
build/
|-- c4che
|   |-- build.config.py
|   `-- _cache.py
|-- config.log
|-- app ❶
|-- a.c.1.o ❷
|-- b.c.2.o
|-- c.c.3.o ❸
|-- libmyshlib.so ❹
|-- libmystlib.a
`-- main.c.0.o
```

❶ Programs have no extension on Linux but will have *.exe* on Windows

❷ Object files suffixes contain a number that is meant to provide some protection against name collisions

❸ The index value is incremented for each task generator declared in a particular wscript file [4]

❹ The *.so* extension for shared libraries on Linux will be *.dll* on Windows or *.dylib* on Mac. These extensions are determined at configuration time.

The build context methods *program*, *shlib*, *stlib* and *objects* are aliases that return a single task generator with the appropriate features detected from the source list. In case more control is required over the linking mode - for example when no source files are given - it may be necessary to fallback to the feature-based declaration. The example above is equivalent to the following:

```
def build(bld):
        bld(features='c cprogram', source='main.c', target='app', use='myshlib mystlib')
        bld(features='c cstlib', source='a.c', target='mystlib', use='myobjects')
        bld(features='c cshlib', source='b.c', target='myshlib')
        bld(features='c', source='c.c', target='myobjects')
```

The feature names follow the same pattern where the prefix is the language (c/cxx/fc/d for C, C++, Fortran and D), and the suffix is the target type (program/shlib/stlib for programs, shared libraries and static libraries). Targets mixing several language types typically split source files by language type, but they may be combined in a single declaration:

```
def options(opt):
        opt.load('compiler_c compiler_cxx')

def configure(conf):
        conf.load('compiler_c compiler_cxx')

def build(bld):
        bld(features='c cxx cxxprogram', source='a.c b.c++', target='app')
```

10.1.2. Additional attributes

The methods described previously can process many more attributes than just *use* or features. Here is a longer example:

```
def options(opt):
        opt.load('compiler_c')

def configure(conf):
        conf.load('compiler_c')

def build(bld):
        bld.program(
                source       = 'main.c', ❶
                target       = 'appname', ❷

                includes     = ['.'], ❸
                defines      = ['LINUX=1', 'BIDULE'],

                lib          = ['m'], ❹
                libpath      = ['/usr/lib'],

                rpath        = ['/opt/kde/lib'] ❺
                vnum         = '1.2.3',

                idx          = 123, ❻
                install_path = '${SOME_PATH}/bin', ❼
                cflags       = ['-O2', '-Wall'], ❽
        )
```

1. Source files are provided as either space-separated file names, or as a list of string/Node objects
2. The target is converted automatically to `target.exe` or `libtarget.so`, depending on the platform and type
3. Includes should be provided relative to the current script as the build directory is added as-needed
4. For portability reasons, library names should be provided without file extensions
5. Certain attributes such as rpath and vnum are ignored on platforms that do not support them
6. Setting the counter for the object file extension is sometimes necessary to avoid object file name collisions
7. Programs and shared libraries are installed by default. To disable the installation, set None.
8. Miscellaneous flags, applied to the source files that support them (if present)

One drawback of of passing flags such as **cflags** directly is that the wscript files need to be modified when such flags are changed. The section on the **use** system describes how to leverage configuration variables to propagate such flags.

10.2. Include processing

10.2.1. Execution path and flags

Include paths are used by the C/C++ compilers for finding headers. When one header changes, the files are recompiled automatically. For example on a project having the following structure:

```
$ tree
.
|-- foo.h
|-- src
|   |-- main.c
|   `-- wscript
`-- wscript
```

The file *src/wscript* will contain the following code:

```
def build(bld):
    bld.program(
        source   = 'main.c',
        target   = 'myapp',
        includes = '.. .')
```

The command-line (output by `waf -v`) will have a form similar to the following:

```
cc -I. -I.. -Isrc -I../src ../src/main.c -c -o src/main_1.o
```

This command contains the include paths as seen from the build directory:

```
.. -> -I..      and  -I.
.  -> -I../src  and  -Isrc
```

The most important points to remember are:

1. Includes paths are given relative to the directory containing the wscript file. Providing absolute paths are best avoided as they are a source of portability problems.
2. The includes add both the source directory and the corresponding build directory for the task generator variant. Hard-coding the build directory is best avoided.
3. System include paths are best defined during the configuration and added to INCLUDES variables (uselib)

10.2.2. Headers and dependencies

Waf preprocessor

Waf uses a preprocessor written in Python for adding the dependencies on the headers. A simple parser looking at #include statements would miss constructs such as:

```
#define mymacro "foo.h"
#include mymacro
```

System headers are not tracked by default for performance reasons, so the preprocessor can miss dependencies obtained through external macros (boost):

```
#if SOMEMACRO
        /* an include in the project */
        #include "foo.h"
#endif
```

To write portable code and to ease debugging, it is recommended to put all the conditions used within a project into a *config.h* file.

```
def configure(conf):
        conf.check(
                fragment    = 'int main() { return 0; }\n',
                define_name = 'FOO',
                mandatory   = True)
        conf.write_config_header('config.h')
```

Generated headers

Dependencies are obtained on generated headers, but this requires declaring the headers as output files:

```
def create_header(tsk):
        tsk.outputs[0].write('int abc = 12;\n')
bld(
        rule     = create_header,
        target   = 'b.h', ❶
        ext_out  = ['.h']) ❷

tg = bld.program(
        features = 'app',
        source   = 'main.c',
        includes = '.', ❸
        target   = 'myprogram')
```

❶ All headers created by the build must be declared. [5]

❷ While optional, build tasks that create headers should indicate that header files are generated in order to improve build performance

❸ Include paths must be provided in such variables and not as **cflags** for instance

Performance concerns

If performance is not a concern, then dependencies can be computed on system headers by setting the following:

```
from waflib import c_preproc
c_preproc.go_absolute = True
```

Additional tools such as gccdeps, msvcdeps or dumbpreproc provide alternate dependency scanners that can be faster in certain cases (boost). Such extensions still require proper include paths and header declaration in order to avoid race conditions in the build system.

Dependency troubleshooting

To display the dependencies obtained or missed, use the following:

```
$ waf --zones=deps

23:53:21 deps deps for src:///comp/waf/demos/qt4/src/window.cpp: ❶
  [src:///comp/waf/demos/qt4/src/window.h, bld:///comp/waf/demos/qt4/src/window.moc]; ❷
  unresolved ['QtGui', 'QGLWidget', 'QWidget'] ❸
```

❶ File preprocessed

❷ List of headers found

❸ System headers discarded or missing dependencies

The Waf preprocessor also provides a specific debug zone which may be useful for troubleshooting complex macros:

```
$ waf --zones=preproc
```

> note: The scanner is only called when C files or dependencies are modified. In the rare case of adding headers after a successful compilation, then it may be necessary to run *waf clean build* to force a full dependency re-scan.

10.3. Library interaction (use)

10.3.1. Local libraries

The attribute *use* enables the link against libraries (static or shared), or the inclusion of object files when the task generator referenced is not a library.

```
def build(bld):
        bld.stlib(
                source   = 'test_staticlib.c',
                target   = 'mylib',
                name     = 'stlib1') ❶

        bld.program(
                source   = 'main.c',
                target   = 'app',
                includes = '.',
                use      = ['stlib1']) ❷
```

❶ The name attribute must point at exactly one task generator

❷ The attribute *use* contains the task generator names to use

In this example, the file *app* will be re-created whenever *mylib* changes (order and dependency). By using task generator names, the programs and libraries declarations may appear in any order and across scripts. For convenience, the name does not have to be defined, and will be pre-set from the target name:

```
def build(bld):
        bld.stlib(
                source   = 'test_staticlib.c',
                target   = 'mylib')

        bld.program(
                source   = 'main.c',
                target   = 'app',
                includes = '.',
                use      = ['mylib'])
```

The *use* processing also exhibits a recursive behaviour. Let's illustrate it by the following example:

```
def build(bld):
        bld.shlib(
                source = 'a.c', ❶
                target = 'lib1')

        bld.stlib(
                source = 'b.c',
                use    = 'cshlib', ❷
                target = 'lib2')

        bld.shlib(
                source = 'c.c',
                target = 'lib3',
                use    = 'lib1 lib2') ❸

        bld.program( ❹
                source = 'main.c',
                target = 'app',
                use    = 'lib3')
```

❶ A simple shared library

❷ The *cshlib* flags will be propagated to both the library and the program.

❸ *lib3* uses both a shared library and a static library

❹ A program using *lib3*

Because of the shared library dependency *lib1* → *lib2*, the program *app* should link against both *lib1* and *lib3*, but not against *lib2*:

```
$ waf -v
'clean' finished successfully (0.004s)
Waf: Entering directory `/tmp/cprog_propagation/build'
[1/8] c: a.c -> build/a.c.0.o
12:36:17 runner ['/usr/bin/gcc', '-fPIC', '../a.c', '-c', '-o', 'a.c.0.o']
[2/8] c: b.c -> build/b.c.1.o
12:36:17 runner ['/usr/bin/gcc', '../b.c', '-c', '-o', 'b.c.1.o']
[3/8] c: c.c -> build/c.c.2.o
12:36:17 runner ['/usr/bin/gcc', '-fPIC', '../c.c', '-c', '-o', 'c.c.2.o']
[4/8] c: main.c -> build/main.c.3.o
12:36:17 runner ['/usr/bin/gcc', '../main.c', '-c', '-o', 'main.c.3.o']
[5/8] cstlib: build/b.c.1.o -> build/liblib2.a
12:36:17 runner ['/usr/bin/ar', 'rcs', 'liblib2.a', 'b.c.1.o']
[6/8] cshlib: build/a.c.0.o -> build/liblib1.so
12:36:17 runner ['/usr/bin/gcc', 'a.c.0.o', '-o', 'liblib1.so', '-shared']
```

```
[7/8] cshlib: build/c.c.2.o -> build/liblib3.so
12:36:17 runner ['/usr/bin/gcc', 'c.c.2.o', '-o', 'liblib3.so', '-Wl,-Bstatic', '-L.', '-llib2', '-Wl,-
Bdynamic', '-L.', '-llib1', '-shared']
[8/8] cprogram: build/main.c.3.o -> build/app
12:36:17 runner ['/usr/bin/gcc', 'main.c.3.o', '-o', 'app', '-Wl,-Bdynamic', '-L.', '-llib1', '-llib3']
Waf: Leaving directory `/tmp/cprog_propagation/build'
'build' finished successfully (0.144s)
```

To sum up the two most important aspects of the *use* attribute:

1. The task generators may be created in any order and in different files, but must provide a unique name for the *use* attribute
2. The *use* processing will iterate recursively over all the task generators involved, but the flags added depend on the target kind (shared/static libraries)

10.3.2. Special local libraries

Includes folders

The use keyword may point at special libraries that do not actually declare a target. For example, header-only libraries are commonly used to add specific include paths to several targets:

```
def build(bld):
        bld(
                includes        = '. src',
                export_includes = 'src', ❶
                name            = 'com_includes')

        bld.stlib(
                source          = 'a.c',
                target          = 'shlib1',
                use             = 'com_includes') ❷

        bld.program(
                source          = 'main.c',
                target          = 'app',
                use             = 'shlib1', ❸
                )
```

❶ The *includes* attribute is private, but *export_includes* will be used by other task generators

❷ The paths added are relative to the other task generator

❸ The *export_includes* will be propagated to other task generators

Object files

Here is how to enable specific compilation flags for particular files:

```
def build(bld):
        bld.objects( ❶
                source  = 'test.c',
                cflags  = '-O3',
                target  = 'my_objs')

        bld.shlib(
                source  = 'a.c',
                cflags  = '-O2', ❷
                target  = 'lib1',
                use     = 'my_objs') ❸

        bld.program(
                source  = 'main.c',
                target  = 'test_c_program',
                use     = 'lib1') ❹
```

❶ Files will be compiled in c mode, but no program or library will be produced

❷ Different compilation flags may be used

❸ The objects will be added automatically in the link stage

❹ There is no object propagation to other programs or libraries to avoid duplicate symbol errors

> Like static libraries, object files are often abused to copy-paste binary code which can lead to huge binaries and obscure crashes. Try to minimize executable size by using shared libraries whenever possible.

Fake libraries

Local libraries will trigger a recompilation whenever they change. The methods *read_shlib* and *read_stlib* can be used to add this behaviour to external libraries or to binary files present in the project.

```
def build(bld):
        bld.read_shlib('m', paths=['.', '/usr/lib64'])
        bld.program(source='main.c', target='app', use='m')
```

The methods will try to find files such as *libm.so* or *libm.dll* in the specified paths to compute the required paths and dependencies. In this example, the target *app* will be re-created whenever */usr/lib64/libm.so* changes. These libraries are propagated between task generators just like shared or static libraries declared locally.

10.3.3. Foreign libraries and flags

When an element in the attribute *use* does not match a local library, it is assumed that it represents a system library, and the the required flags are present in the configuration set *env*. This system enables the addition of several compilation and link flags at once, as in the following example:

```
import sys

def options(opt):
        opt.load('compiler_c')

def configure(conf):
        conf.load('compiler_c')
        conf.env.INCLUDES_TEST      = ['/usr/include'] ❶

        if sys.platform != 'win32': ❷
                conf.env.DEFINES_TEST   = ['TEST']
                conf.env.CFLAGS_TEST    = ['-O0'] ❸
                conf.env.LIB_TEST       = ['m']
                conf.env.LIBPATH_TEST   = ['/usr/lib']
                conf.env.LINKFLAGS_TEST = ['-g']
                conf.env.INCLUDES_TEST  = ['/opt/gnome/include']

def build(bld):
        mylib = bld.stlib(
                source   = 'test_staticlib.c',
                target   = 'teststaticlib',
                use      = 'TEST') ❹

        if mylib.env.CC_NAME == 'gcc':
                mylib.cxxflags = ['-O2'] ❺
```

❶ For portability reasons, it is recommended to use INCLUDES instead of giving flags of the form -I/include. Note that the INCLUDES use used by both c and c++

❷ Variables may be left undefined in platform-specific settings, yet the build scripts will remain identical.

❸ Declare a few variables during the configuration, the variables follow the convention VAR_NAME

❹ Add all the VAR_NAME corresponding to the *use variable* NAME, which is *TEST* in this example

❺ *Model to avoid*: setting the flags and checking for the configuration should be performed in the configuration section

The variables used for C/C++ are the following:

Table 1. Use variables and task generator attributes for C/C++

Uselib variable	Attribute	Usage
LIB	lib	list of shared library names to use, without prefix or extension
LIBPATH	libpath	list of search path for shared libraries
STLIB	stlib	list of static library names to use, without prefix or extension
STLIBPATH	stlibpath	list of search path for static libraries
LINKFLAGS	linkflags	list of link flags (prefer other variables whenever possible)
LDFLAGS	ldflags	list of link flags at the end of the link command (prefer other variables)
RPATH	rpath	list of paths to hard-code into the binary during linking time
CFLAGS	cflags	list of compilation flags for c files
CXXFLAGS	cxxflags	list of compilation flags for c++ files

Uselib variable	Attribute	Usage
CPPFLAGS	cppflags	list of flags added at the end of compilation commands (C/C++/Fortran modes)
FCFLAGS	fcflags	list of Fortran compilation flags
DFLAGS	dflags	list of compilation flags for d files
INCLUDES	includes	include paths (absolute paths or Node objects). [6]
CXXDEPS		a variable/list to trigger c++ file recompilations when it changes
CCDEPS		same as above, for c
LINKDEPS		same as above, for the link tasks
DEFINES	defines	list of defines in the form ['key=value', …]
FRAMEWORK	framework	list of frameworks to use
FRAMEWORKPATH	frameworkpath	list of framework paths to use
ARCH	arch	list of architectures in the form [*ppc*, *x86*]

The variables may be left empty for later use, and will not cause errors. During the development, the configuration cache files (for example, _cache.py) may be modified from a text editor to try different configurations without forcing a whole project reconfiguration. The files affected will be rebuilt however.

10.4. Configuration helpers

10.4.1. Configuration tests

The method *check* is used to detect parameters using a small build project. The main parameters are the following

1. msg: title of the test to execute
2. okmsg: message to display when the test succeeds
3. errmsg: message to display when the test fails
4. env: environment to use for the build (conf.env is used by default)
5. compile_mode: *cc* or *cxx*
6. define_name: sets a define of the form **define_name=x** when the test succeeds (a default value for define_name is usually calculated automatically)
7. global_define: sets **define_name=X** to **env.DEFINES** when True and to **env.DEFINES_X** when False (X represents value of the uselib_store variable)
8. uselib_store: stores arguments of the form lib,cflag,cxxflags to LIB_X,CFLAG_X,CXXFLAGS_X respectively (X represents the value of the uselib_store variable)

The errors raised are instances of *waflib.Errors.ConfigurationError*. There are no return codes.

Besides the main parameters, the attributes from c/c++ task generators may be used. Here are a few examples:

```
def configure(conf):

        conf.check(header_name='time.h', features='c cprogram') ❶
        conf.check_cc(fragment='int main() {2+2==4;}\n', define_name="boobah") ❷
        conf.check_cc(lib='m', cflags='-Wall', defines=['var=foo', 'x=y'],
                uselib_store='M') ❸
        conf.check_cxx(lib='linux', use='M', cxxflags='-O2') ❹

        conf.check_cc(fragment='''
                        #include <stdio.h>
                        int main() { printf("4"); return 0; } ''',
                define_name = "booeah",
                execute     = True,
                define_ret  = True,
                msg         = "Checking for something") ❺

        conf.check(features='c', fragment='int main(){return 0;}') ❻

        conf.write_config_header('config.h') ❼
```

❶ Try to compile a program using the configuration header time.h, if present on the system, if the test is successful, the define HAVE_TIME_H will be added

❷ Try to compile a piece of code, and if the test is successful, define the name boobah

❸ Modifications made to the task generator environment are not stored. When the test is successful and when the attribute uselib_store is provided, the names lib, cflags and defines will be converted into *use variables* LIB_M, CFLAGS_M and DEFINES_M and the flag values are added to the configuration environment.

4. Try to compile a simple c program against a library called *linux*, and reuse the previous parameters for libm by *use*
5. Execute a simple program, collect the output, and put it in a define when successful
6. The tests create a build with a single task generator. By passing the *features* attribute directly it is possible to disable the compilation or to create more complicated configuration tests.
7. After all the tests are executed, write a configuration header in the build directory (optional). The configuration header is used to limit the size of the command-line.

Here is an example of a `config.h` produced with the previous test code:

```
/* Configuration header created by Waf - do not edit */
#ifndef _CONFIG_H_WAF
#define _CONFIG_H_WAF

#define HAVE_PRINTF 1
#define HAVE_TIME_H 1
#define boobah 1
#define booeah "4"

#endif /* _CONFIG_H_WAF */
```

The file `_cache.py` will contain the following variables:

```
DEFINES_M = ['var=foo', 'x=y']
CXXFLAGS_M = ['-Wall']
CFLAGS_M = ['-Wall']
LIB_M = ['m']
boobah = 1
booeah = '4'
defines = {'booeah': '"4"', 'boobah': 1, 'HAVE_TIME_H': 1, 'HAVE_PRINTF': 1}
dep_files = ['config.h']
waf_config_files = ['/compilation/waf/demos/adv/build/config.h']
```

10.4.2. Advanced tests

The methods *conf.check* create a build context and a task generator internally. This means that the attributes *includes*, *defines*, *cxxflags* may be used (not all shown here). Yet some tests can be require several targets at once. In order to facilitate this, a custom build function can be passed directly, for example:

```
def build(bld):
        lib_node = bld.srcnode.make_node('libdir/iotest.c')
        lib_node.parent.mkdir()
        lib_node.write('#include <stdio.h>\nint lib_func(void) { FILE *f = fopen("foo", "r");}\n', 'w')
        bld(features='c cshlib', source=[lib_node], linkflags=['-g3'], target='mylib')
conf.check(build_fun=build, msg='Custom test')
```

As a result, tests can also be written for other programming languages and compilers. Here is for example a custom test for LaTeX packages:

```
def build_latex_test(bld):
        def write_tex(tsk):
                tsk.outputs[0].write(r'''\documentclass[a4paper,12pt]{article}  \usepackage{ucs}
\begin{document} test \end{document} ''')
        bld(rule=write_tex, target='main.tex')
        bld(features='tex', type='pdflatex', source='main.tex', prompt=0)
conf.test(build_fun=build_latex_test, msg='Checking for UCS', okmsg='ok', errmsg='ucs.sty is missing install
latex-extras')
```

10.4.3. Configuration headers

Adding lots of command-line define values increases the size of the command-line and makes it harder to review the flags when errors occur. Besides that, the defines passed on the command-line may fail unexpectedly with different compilers and command execution contexts. For example, define values containing quotes may be misinterpreted in Visual Studio response files. It is therefore a best practice to use configuration headers whenever possible.

Writing configuration headers can be performed using the following methods:

```
def configure(conf):
        conf.define('NOLIBF', 1)
        conf.undefine('NOLIBF')
        conf.define('LIBF', 1)
        conf.define('LIBF_VERSION', '1.0.2')
        conf.write_config_header('config.h')
```

The code snipped will produce the following *config.h* in the build directory:

```
build/
|-- c4che
|   |-- build.config.py
|   `-- _cache.py
|-- config.log
`-- config.h
```

The contents of the config.h for this example are:

```
/* Configuration header created by Waf - do not edit */
#ifndef _CONFIG_H_WAF
#define _CONFIG_H_WAF

/* #undef NOLIBF */
#define LIBF 1
#define LIBF_VERSION "1.0.2"

#endif /* _CONFIG_H_WAF */
```

> note: By default, the defines are moved from the command-line into the configuration header. This means that the attribute *conf.env.DEFINE* is cleared by this operation. To prevent this behaviour, use *conf.write_config_header(remove=False)*

10.4.4. Pkg-config

Instead of duplicating the configuration detection in all dependent projects, configuration files may be written when libraries are installed. To ease the interaction with build systems based on Make (cannot query databases or apis), small applications have been created for reading the cache files and to interpret the parameters (with names traditionally ending in *-config*): pkg-config, wx-config, sdl-config, etc.

The method *check_cfg* is provided to ease the interaction with these applications. Here are a few examples:

```
def options(opt):
        opt.load('compiler_c')

def configure(conf):
        conf.load('compiler_c')

        conf.check_cfg(atleast_pkgconfig_version='0.0.0') ❶
        pango_version = conf.check_cfg(modversion='pango') ❷

        conf.check_cfg(package='pango') ❸
        conf.check_cfg(package='pango', uselib_store='MYPANGO',
                args=['--cflags', '--libs']) ❹

        conf.check_cfg(package='pango', ❺
                args=['pango >= 0.1.0', 'pango < 9.9.9', '--cflags', '--libs'],
                msg="Checking for 'pango 0.1.0'") ❻

        conf.check_cfg(path='sdl-config', args='--cflags --libs',
                package='', uselib_store='SDL') ❼
        conf.check_cfg(path='mpicc', args='--showme:compile --showme:link',
                package='', uselib_store='OPEN_MPI', mandatory=False) ❽
```

❶ Check for the pkg-config version

❷ Retrieve the module version for a package as a string. If there were no errors, *PANGO_VERSION* is defined. It can be overridden with the attribute *uselib_store='MYPANGO'*.

❸ Check if the pango package is present, and define *HAVE_PANGO* (calculated automatically from the package name)

❹ Beside defining *HAVE_MYPANGO*, extract and store the relevant flags to the *use variable* MYPANGO (*LIB_MYPANGO*, *LIBPATH_MYPANGO*, etc)

❺ Like the previous test, but with pkg-config clauses to enforce a particular version number

❻ Display a custom message on the output. The attributes *okmsg* and *errmsg* represent the messages to display in case of success and error respectively

❼ Obtain the flags for sdl-config. The example is applicable for other configuration programs such as wx-config, pcre-config, etc

❽ Suppress the configuration error which is raised whenever the program to execute is not found or returns a non-zero exit status

Due to the amount of flags, the lack of standards between config applications, and to the compiler-dependent flags (-I for gcc, /I for msvc), the pkg-config output is parsed before setting the corresponding *use variables* in a go. The function *parse_flags(line, uselib, env)* in the Waf module c_config.py performs the flag extraction.

The outputs are written in the build directory into the file *config.log*:

```
# project  configured on Tue Aug 31 17:30:21 2010 by
# waf 2.0.8 (abi 98, python 20605f0 on linux2)
# using /home/waf/bin/waf configure
#
---
Setting top to
/disk/comp/waf/docs/book/examples/cprog_pkgconfig
---
Setting out to
/disk/comp/waf/docs/book/examples/cprog_pkgconfig/build
---
Checking for program pkg-config
/usr/bin/pkg-config
find program=['pkg-config'] paths=['/usr/local/bin', '/usr/bin'] var='PKGCONFIG' -> '/usr/bin/pkg-config'
---
Checking for pkg-config version >= 0.0.0
['/usr/bin/pkg-config', '--atleast-pkgconfig-version=0.0.0']
yes
['/usr/bin/pkg-config', '--modversion', 'pango']
out: 1.28.0

---
Checking for pango
['/usr/bin/pkg-config', 'pango']
yes
---
Checking for pango
['/usr/bin/pkg-config', 'pango']
yes
---
Checking for pango 0.1.0
['/usr/bin/pkg-config', 'pango >= 0.1.0', 'pango < 9.9.9', '--cflags', '--libs', 'pango']
out: -pthread -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib64/glib-2.0/include
     -pthread -lpango-1.0 -lgobject-2.0 -lgmodule-2.0 -lgthread-2.0 -lrt -lglib-2.0

yes
---
Checking for sdl-config
['sdl-config', '--cflags', '--libs']
out: -I/usr/include/SDL -D_GNU_SOURCE=1 -D_REENTRANT
-L/usr/lib64 -lSDL -lpthread

yes
---
Checking for mpicc
['mpicc', '--showme:compile', '--showme:link']
out: -pthread libtool: link: -pthread -L/usr/lib64 -llammpio -llamf77mpi -lmpi -llam -lutil -ldl
```

After such a configuration, the configuration set contents will be similar to the following:

```
'CFLAGS_OPEN_MPI' ['-pthread']
'CFLAGS_PANGO' ['-pthread']
'CXXFLAGS_OPEN_MPI' ['-pthread']
'CXXFLAGS_PANGO' ['-pthread']
'DEFINES' ['HAVE_PANGO=1', 'HAVE_MYPANGO=1', 'HAVE_SDL=1', 'HAVE_OPEN_MPI=1']
'DEFINES_SDL' ['_GNU_SOURCE=1', '_REENTRANT']
'INCLUDES_PANGO' ['/usr/include/pango-1.0', '/usr/include/glib-2.0', '/usr/lib64/glib-2.0/include']
'INCLUDES_SDL' ['/usr/include/SDL']
'LIBPATH_OPEN_MPI' ['/usr/lib64']
'LIBPATH_SDL' ['/usr/lib64']
'LIB_OPEN_MPI' ['lammpio', 'lamf77mpi', 'mpi', 'lam', 'util', 'dl']
'LIB_PANGO' ['pango-1.0', 'gobject-2.0', 'gmodule-2.0', 'gthread-2.0', 'rt', 'glib-2.0']
'LIB_SDL' ['SDL', 'pthread']
'LINKFLAGS_OPEN_MPI' ['-pthread']
'LINKFLAGS_PANGO' ['-pthread']
'PKGCONFIG' '/usr/bin/pkg-config'
'PREFIX' '/usr/local'
'define_key' ['HAVE_PANGO', 'HAVE_MYPANGO', 'HAVE_SDL', 'HAVE_OPEN_MPI']
```

10.5. Extensions

10.5.1. Additional tools

Additional C or C++ tools may be included in the Waf file and loaded to detect compilers ahead of the Waf ones. For this, the Waf tools such as **compiler_fc** load all files that start by the **fc_** prefix automatically.

It is thus necessary to include extensions as part of `waflib/extras` by building a custom waf file:

```
./waf-light --tools=fc_bgxlf
```

The extension then only needs to inserts itself in the list of modules to use for a particular platform. This Fortran module provides for example:

```
from waflib.Tools.compiler_fc import fc_compiler
fc_compiler['linux'].insert(0, 'fc_bgxlf')
```

10.5.2. Cross-compilation concerns

Simple cross builds using environment variables

In general, cross builds do not require building host compilers can be performed by setting environment variables such as CFLAGS, LINKFLAGS, CC and AR:

```
CC=/path/to/gcc waf configure build
```

Staged cross-compilation builds

Such builds typically require some preparation from the project maintainers. While certain extensions exist, it is usually best to set config sets for the host and target builds, and then update the target build environment variables that need changing:

```
def configure(conf):
        conf.setenv('host')
        conf.setenv('target') ❶

        conf.setenv('host')
        conf.load('gcc') ❷

        conf.setenv('target') ❸
        conf.env.CC = ['/opt/arm/bin/gcc'] ❹
        conf.gcc_modifier_win32()

def build(bld):
        bld.env = bld.all_envs['host']
        bld.program(source='main.c', target='app', install_path=None)
        bld.add_group()
        bld.env = bld.all_envs['target'] ❺
        bld.program(source='main.c', target='app')
```

❶ Make environment copies

❷ Configure for the host builds

❸ Configure the target data

❹ Set relevant variables

❺ Declare target with the appropriate configuration data

11. Advanced scenarios

This chapter demonstrates a few examples of the waf library for more complicated and less common scenarios.

11.1. Project organization

11.1.1. Building the compiler first

The example below demonstrates how to build a compiler which is used for building the remaining targets. The requirements are the following:

1. Create the compiler and all its intermediate tasks
2. Re-use the compiler in a second build step
3. The compiler will transform *.src* files into *.cpp* files, which will be processed too
4. Call the compiler again if it was rebuilt (add the dependency on the compiler)

The first thing to do is to write the expected user script:

```
top = '.'
out = 'build'

def configure(ctx):
    ctx.load('g++')
    ctx.load('src2cpp', tooldir='.')
```

```
def build(ctx):
    ctx.program( ❶
        source = 'comp.cpp',
        target = 'comp')

    ctx.add_group() ❷

    ctx.program(
        source = 'main.cpp a.src', ❸
        target = 'foo')
```

❶ Build the compiler first, it will result in a binary named *comp*

❷ Add a new build group to make certain the compiler is complete before processing the next tasks

❸ The file *a.src* is to be transformed by *comp* into *a.cpp*

The code for the *src → cpp* conversion will be the following:

```
from waflib.Task import Task
class src2cpp(Task): ❶
    run_str = '${SRC[0].abspath()} ${SRC[1].abspath()} ${TGT}'
    color   = 'PINK'

from waflib.TaskGen import extension

@extension('.src')
def process_src(self, node): ❷
    tg = self.bld.get_tgen_by_name('comp') ❸
    comp = tg.link_task.outputs[0]
    tsk = self.create_task('src2cpp', [comp, node], node.change_ext('.cpp')) ❹
    self.source.extend(tsk.outputs) ❺
```

❶ Declare a new task class for processing the source file by our compiler

❷ Files of extension *.src* are to be processed by this method

❸ Obtain a reference on the task generator producing the compiler

❹ Create the task *src → cpp*, the compiler being as used as the first source file

❺ Add the generated *cpp* file to be processed too

The compilation results will be the following:

```
$ waf distclean configure build -v
'distclean' finished successfully (0.006s)
Setting top to                           : /tmp/scenarios_compiler
Setting out to                           : /tmp/scenarios_compiler/build
Checking for program g++,c++             : /usr/bin/g++
Checking for program ar                  : /usr/bin/ar
'configure' finished successfully (0.118s)
Waf: Entering directory `/tmp/scenarios_compiler/build'
[1/6] cxx: comp.cpp -> build/comp.cpp.1.o
01:06:00 runner ['/usr/bin/g++', '../comp.cpp', '-c', '-o', 'comp.cpp.1.o']
[2/6] cxxprogram: build/comp.cpp.1.o -> build/comp ❶
01:06:00 runner ['/usr/bin/g++', 'comp.cpp.1.o', '-o', 'build/comp', '-Wl,-Bstatic', '-Wl,-Bdynamic']
[3/6] cxx: main.cpp -> build/main.cpp.2.o
01:06:00 runner ['/usr/bin/g++', '../main.cpp', '-c', '-o', 'main.cpp.2.o']
[4/6] src2cpp: build/comp a.src -> build/a.cpp
01:06:00 runner ['build/comp', 'scenarios_compiler/a.src', 'a.cpp'] ❷
[5/6] cxx: build/a.cpp -> build/a.cpp.2.o
01:06:00 runner ['/usr/bin/g++', 'a.cpp', '-c', '-o', 'a.cpp.2.o']
[6/6] cxxprogram: build/main.cpp.2.o build/a.cpp.2.o -> build/foo ❸
01:06:00 runner ['/usr/bin/g++', 'main.cpp.2.o', 'a.cpp.2.o', '-o', 'build/foo', '-Wl,-Bstatic', '-Wl,-
Bdynamic']
Waf: Leaving directory `/tmp/scenarios_compiler/build'
'build' finished successfully (0.171s)
```

❶ Creation of the *comp* program

❷ Use the compiler to generate *a.cpp*

❸ Compile and link *a.cpp* and *main.cpp* into the program *foo*

When 'waf --targets=foo' is called, the task generator 'comp' will create its tasks too (task generators from previous groups are processed).

11.1.2. Providing arbitrary configuration files

A file is copied into the build directory before the build starts. The build may use this file for building other targets.

```
cfg_file = 'somedir/foo.txt'

def configure(conf):

    orig = conf.root.find_node('/etc/fstab')
    txt = orig.read() ❶

    dest = conf.bldnode.make_node(cfg_file)
    dest.parent.mkdir() ❷
    dest.write(txt) ❸

    conf.env.append_value('cfg_files', dest.abspath()) ❹

def build(ctx):
    ctx(rule='cp ${SRC} ${TGT}', source=cfg_file, target='bar.txt')
```

❶ Read the file */etc/fstab*

❷ Create the destination directory in case it does not already exist

❸ Create a new file in the build directory

❹ Mark the output as a configuration file so it can be used during the build

The execution output will be the following:

```
$ waf configure build
Setting top to     : /tmp/scenarios_impfile
Setting out to     : /tmp/scenarios_impfile/build
'configure' finished successfully (0.003s)
Waf: Entering directory `/tmp/scenarios_impfile/build'
[1/1] bar.txt: build/somedir/foo.txt -> build/bar.txt
Waf: Leaving directory `/tmp/scenarios_impfile/build'
'build' finished successfully (0.008s)

$ tree
.
|-- build
|   |-- bar.txt
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- _cache.py
|   |-- config.log
|   `-- somedir
|       `-- foo.txt
`-- wscript
```

11.2. Mixing extensions and C/C++ features

11.2.1. Files processed by a single task generator

Now let's illustrate the @extension decorator on idl file processing. Files with .idl extension are processed to produce .c and .h files (`foo.idl` → `foo.c` + `foo.h`). The .c files must be compiled after being generated.

First, here is the declaration expected in user scripts:

```
top = '.'
out = 'build'

def configure(conf):
    conf.load('g++')

def build(bld):
    bld.program(
        source = 'foo.idl main.cpp',
        target = 'myapp'
        )
```

The file `foo.idl` is listed as a source. It will be processed to `foo.cpp` and compiled and linked with `main.cpp`

Here is the code to support this scenario:

```
from waflib.Task import Task
from waflib.TaskGen import extension
```

```
class idl(Task):
    run_str = 'cp ${SRC} ${TGT[0].abspath()} && touch ${TGT[1].abspath()}' ❶
    color   = 'BLUE'
    ext_out = ['.h'] ❷

@extension('.idl')
def process_idl(self, node):
    cpp_node = node.change_ext('.cpp')
    hpp_node = node.change_ext('.hpp')
    self.create_task('idl', node, [cpp_node, hpp_node]) ❸
    self.source.append(cpp_node) ❹
```

❶ Dummy command for demonstration purposes. In practice the rule to use would be like *omniidl -bcxx ${SRC} -C${TGT}*

❷ Because the idl task produces headers, it must be executed before any other `cpp` file is compiled

❸ Create the task from the *.idl* extension.

❹ Re-inject the file to compile by the C++ compiler

The execution results will be the following:

```
$ waf distclean configure build -v
'distclean' finished successfully (0.002s)
Setting top to   : /tmp/scenarios_idl
Setting out to   : /tmp/scenarios_idl/build
Checking for program g++,c++   : /usr/bin/g++
Checking for program ar        : /usr/bin/ar
'configure' finished successfully (0.072s)
Waf: Entering directory `/tmp/scenarios_idl/build'
[1/4] idl: foo.idl -> build/foo.cpp build/foo.hpp
19:47:11 runner 'cp ../foo.idl foo.cpp && touch foo.hpp'
[2/4] cxx: main.cpp -> build/main.cpp.0.o
19:47:11 runner ['/usr/bin/g++', '-I.', '-I..', '../main.cpp', '-c', '-o', 'main.cpp.0.o']
[3/4] cxx: build/foo.cpp -> build/foo.cpp.0.o
19:47:11 runner ['/usr/bin/g++', '-I.', '-I..', 'foo.cpp', '-c', '-o', 'foo.cpp.0.o']
[4/4] cxxprogram: build/main.cpp.0.o build/foo.cpp.0.o -> build/myapp
19:47:11 runner ['/usr/bin/g++', 'main.cpp.0.o', 'foo.cpp.0.o', '-o', 'myapp']
Waf: Leaving directory `/tmp/scenarios_idl/build'
'build' finished successfully (0.149s)
```

> **Note:** The drawback of this declaration is that the source files produced by the idl transformation can be used by only one task generator.

11.2.2. Resources shared by several task generators

Let's suppose now that the idl outputs will be shared by several task generators. We will first start by writing the expected user script:

```
top = '.'
out = 'out'

def configure(ctx):
    ctx.load('g++')

def build(ctx):
    ctx( ❶
        source   = 'notify.idl',
        name     = 'idl_gen')

    ctx.program( ❷
        source   = ['main.cpp'],
        target   = 'testprog',
        includes = '.',
        add_idl  = 'idl_gen') ❸
```

❶ Process an idl file in a first task generator. Name this task generator *idl_gen*

❷ Somewhere else (maybe in another script), another task generator will use the source generated by the idl processing

❸ Reference the idl processing task generator by the name *idl_gen*.

The code to support this scenario will be the following:

```
from waflib.Task import Task
from waflib.TaskGen import feature, before_method, extension
```

```
class idl(Task):
        run_str = 'cp ${SRC} ${TGT[0].abspath()} && touch ${TGT[1].abspath()}'
        color   = 'BLUE'
        ext_out = ['.h'] ❶

@extension('.idl')
def process_idl(self, node):
        cpp_node = node.change_ext('.cpp')
        hpp_node = node.change_ext('.hpp')
        self.create_task('idl', node, [cpp_node, hpp_node])
        self.more_source = [cpp_node] ❷

@feature('*')
@before_method('process_source') ❸
def process_add_source(self):
        for x in self.to_list(getattr(self, 'add_idl', [])): ❹
                y = self.bld.get_tgen_by_name(x)
                y.post() ❺
                if getattr(y, 'more_source', None):
                        self.source.extend(y.more_source) ❻
```

❶ The idl processing must be performed before any C++ task is executed

❷ Bind the output file to a new attribute

❸ Add the source from another task generator object

❹ Process *add_idl*, finding the other task generator

❺ Ensure that the other task generator has created its tasks

❻ Update the source list

The task execution output will be very similar to the output from the first example:

```
$ waf distclean configure build -v
'distclean' finished successfully (0.007s)
Setting top to  : /tmp/scenarios_idl2
Setting out to  : /tmp/scenarios_idl2/build
Checking for program g++,c++    : /usr/bin/g++
Checking for program ar         : /usr/bin/ar
'configure' finished successfully (0.080s)
Waf: Entering directory `/tmp/scenarios_idl2/build'
[1/4] idl: foo.idl -> build/foo.cpp build/foo.hpp
20:20:24 runner 'cp ../foo.idl foo.cpp && touch foo.hpp'
[2/4] cxx: main.cpp -> build/main.cpp.1.o
20:20:24 runner ['/usr/bin/g++', '-I.', '-I..', '../main.cpp', '-c', '-o', 'main.cpp.1.o']
[3/4] cxx: build/foo.cpp -> build/foo.cpp.1.o
20:20:24 runner ['/usr/bin/g++', '-I.', '-I..', 'foo.cpp', '-c', '-o', 'foo.cpp.1.o']
[4/4] cxxprogram: build/main.cpp.1.o build/foo.cpp.1.o -> build/testprog
20:20:24 runner ['/usr/bin/g++', 'main.cpp.1.o', 'foo.cpp.1.o', '-o', 'testprog']
Waf: Leaving directory `/tmp/scenarios_idl2/build'
'build' finished successfully (0.130s)
```

11.3. Task generator methods

11.3.1. Replacing particular attributes

In general, task generator attributes are not replaced, so the following is not going to be compile `main.c`:

```
bld.env.FOO = '/usr/includes'
bld.env.MAIN = 'main.c'
bld(
    features = 'c cprogram',
    source   = '${MAIN}',
    target   = 'app',
    includes = '. ${FOO}')
```

This design decision is motivated by two main reasons:

1. Processing the attributes has a negative performance impact
2. For consistency all attributes would have to be processed

Nevertheless, it is we will demonstrate how to provide Waf with a method to process some attributes. To add a new task generator method, it is necessary to think about its integration with other methods: is there a particular order? The answer is yes, for example, the source attribute is used to create the compilation tasks. To display what methods are in use, execute Waf with the following logging key:

```
$ waf --zones=task_gen
...
19:20:51 task_gen posting task_gen 'app' declared in 'scenarios_expansion' ❶
19:20:51 task_gen -> process_rule (9232720) ❷
19:20:51 task_gen -> process_source (9232720)
19:20:51 task_gen -> apply_link (9232720)
19:20:51 task_gen -> apply_objdeps (9232720)
19:20:51 task_gen -> process_use (9232720)
19:20:51 task_gen -> propagate_uselib_vars (9232720)
19:20:51 task_gen -> apply_incpaths (9232720)
19:20:51 task_gen posted app
```

❶ Task generator execution

❷ Method name and task generator id in parentheses

From the method list, we find that **process_rule** and **process_source** are processing the *source* attribute. The *includes* attribute is processed by **apply_incpaths**.

```
from waflib import Utils, TaskGen
@TaskGen.feature('*') ❶
@TaskGen.before('process_source', 'process_rule', 'apply_incpaths') ❷
def transform_strings(self):
        for x in 'includes source'.split(): ❸
                val = getattr(self, x, None)
                if val:
                        if isinstance(val, str):
                                setattr(self, x, Utils.subst_vars(val, self.env)) ❹
                        elif isinstance(val, list):
                                for i in range(len(val)):
                                        if isinstance(val[i], str):
                                                val[i] = Utils.subst_vars(val[i], self.env)
```

❶ Execute this method in all task generators

❷ Methods to take into account

❸ Iterate over all interesting attributes

❹ Substitute the attributes

11.3.2. Inserting special include flags

A scenario that appears from times to times in C/C++ projects is the need to insert specific flags before others, regardless of how flags are usually processed. We will now consider the following case: execute all C++ compilations with the flag '-I.' in first position (before any other include).

First, a look at the definition of the C++ compilation rule shows that the variable *INCPATHS* contains the include flags:

```
class cxx(Task.Task):
    color   = 'GREEN'
    run_str = '${CXX} ${CXXFLAGS} ${CPPPATH_ST:INCPATHS} ${CXX_SRC_F}${SRC} ${CXX_TGT_F}${TGT}'
    vars    = ['CXXDEPS']
    ext_in  = ['.h']
    scan    = c_preproc.scan
```

Those include flags are set by the method *apply_incpaths*. The trick is then to modify *INCPATHS* after that method has been executed:

```
top = '.'
out = 'build'

def configure(conf):
    conf.load('g++')

def build(bld):
    bld.program(features='cxx cxxprogram', source='main.cpp', target='test')

from waflib.TaskGen import after, feature

@feature('cxx')
@after_method('apply_incpaths')
def insert_blddir(self):
    self.env.prepend_value('INCPATHS', '.')
```

A related case is how to add the top-level directory containing a configuration header:

```
@feature('cxx')
@after_method('apply_incpaths', 'insert_blddir')
def insert_srcdir(self):
    path = self.bld.srcnode.abspath()
    self.env.prepend_value('INCPATHS', path)
```

11.4. Custom tasks

11.4.1. Force the compilation of a particular task

In some applications, it may be interesting to keep track of the date and time of the last build. In C this may be done by using the macros '*DATE*' and '*TIME*', for example, the following `about.c` file will contain:

```
void ping() {
    printf("Project compiled: %s %s\n", __DATE__, __TIME__);
}
```

The files are only compiled when they change though, so it is necessary to find a way to force the `about.c` recompilation. To sum up, the compilation should be performed whenever:

1. One of the c files of the project is compiled
2. The link flags for any task change
3. The link task including the object for our macro is removed

To illustrate this behaviour, we will now set up a project will use various c files:

```
def options(opt):
    opt.load('compiler_c')

def configure(conf):
    conf.load('compiler_c')

def build(bld):
    bld.program(
        source   = 'main.c about.c',
        target   = 'app',
        includes = '.',
        use      = 'my_static_lib')

    bld.stlib(
        source   = 'test_staticlib.c',
        target   = 'my_static_lib')
```

The main file will just call the function *ping* defined `about.c` to display the date and time:

```
#include "a.h"

int main() {
    ping();
    return 0;
}
```

The task method *runnable_status* must be overridden to take into account the dependencies:

```
import os
from waflib import Task
def runnable_status(self):
    if self.inputs[0].name == 'about.c': ❶
        h = 0 ❷
        for g in self.generator.bld.groups:
            for tg in g:
                if isinstance(tg, TaskBase):
                    continue ❸

                h = hash((self.generator.bld.hash_env_vars(self.generator.env, ['LINKFLAGS']), h))
                for tsk in getattr(tg, 'compiled_tasks', []): # all .c or .cpp compilations
                    if id(tsk) == id(self):
                        continue
                    if not tsk.hasrun:
                        return Task.ASK_LATER
                    h = hash((tsk.signature(), h)) ❹
        self.env.CCDEPS = h

        try:
            os.stat(self.generator.link_task.outputs[0].abspath()) ❺
```

```
        except:
            return Task.RUN_ME

    return Task.Task.runnable_status(self) ❻

from waflib.Tools.c import c ❼
c.runnable_status = runnable_status
```

❶ If the task processes `about.c`

❷ Define a hash value that the task will depend on (CCDEPS)

❸ Iterate over all task generators of the project

❹ Hash the link flags and the signatures of all other compilation tasks

❺ Make sure to execute the task if it was never executed before

❻ Normal behaviour

❼ Modify the *c* task class

The execution will produce the following output:

```
$ waf
Waf: Entering directory `/tmp/scenarios_end/build'
[2/5] c: test_staticlib.c -> build/test_staticlib.c.1.o
[3/5] cstlib: build/test_staticlib.c.1.o -> build/libmy_static_lib.a
[4/5] c: about.c -> build/about.c.0.o
[5/5] cprogram: build/main.c.0.o build/about.c.0.o -> build/app
Waf: Leaving directory `/tmp/scenarios_end/build' ❶
'build' finished successfully (0.088s)

$ ./build/app
Project compiled: Jul 25 2010 14:05:30

$ echo " " >> main.c ❷

$ waf
Waf: Entering directory `/tmp/scenarios_end/build'
[1/5] c: main.c -> build/main.c.0.o
[4/5] c: about.c -> build/about.c.0.o ❸
[5/5] cprogram: build/main.c.0.o build/about.c.0.o -> build/app
Waf: Leaving directory `/tmp/scenarios_end/build'
'build' finished successfully (0.101s)

$ ./build/app
Project compiled: Jul 25 2010 14:05:49
```

❶ All files are compiled on the first build

❷ The file `main.c` is modified

❸ The build generates `about.c` again to update the build time string

11.4.2. A compiler producing source files with names unknown in advance

The requirements for this problem are the following:

1. A compiler **creates source files** (one .src file → several .c files)
2. The source file names to create are **known only when the compiler is executed**
3. The compiler is slow so it should run **only when absolutely necessary**
4. Other tasks will **depend on the generated files** (compile and link the .c files into a program)

To do this, the information on the source files must be shared between the build executions.

```
top = '.'
out = 'build'

def configure(conf):
    conf.load('gcc')
    conf.load('mytool', tooldir='.')

def build(bld):
    bld.env.COMP = bld.path.find_resource('evil_comp.py').abspath() ❶
    bld.stlib(source='x.c foo.src', target='astaticlib') ❷
```

❶ Compiler path

❷ An example, having a *.src* file

The contents of *mytool* will be the following:

```
import os
from waflib import Task, Utils, Context
from waflib.Utils import subprocess
from waflib.TaskGen import extension

@extension('.src')
def process_shpip(self, node): ❶
    self.create_task('src2c', node)

class src2c(Task.Task):
    color = 'PINK'
    quiet = True ❷
    before = ['cstlib'] ❸

    def run(self):
        cmd = '%s %s' % (self.env.COMP, self.inputs[0].abspath())
        n = self.inputs[0].parent.get_bld()
        n.mkdir()
        cwd = n.abspath()
        out = self.generator.bld.cmd_and_log(cmd, cwd=cwd, quiet=Context.STDOUT) ❹

        out = Utils.to_list(out)
        self.outputs = [self.generator.path.find_or_declare(x) for x in out]
        self.generator.bld.raw_deps[self.uid()] = [self.signature()] + self.outputs ❺
        self.add_c_tasks(self.outputs) ❻

    def add_c_tasks(self, lst):
        self.more_tasks = []
        for node in lst:
            if node.name.endswith('.h'):
                continue
            tsk = self.generator.create_compiled_task('c', node)
            self.more_tasks.append(tsk) ❼

            tsk.env.append_value('INCPATHS', [node.parent.abspath()])

            if getattr(self.generator, 'link_task', None): ❽
                self.generator.link_task.set_run_after(tsk)
                self.generator.link_task.inputs.append(tsk.outputs[0])
        self.generator.link_task.inputs.sort(key=lambda x: x.abspath())

    def runnable_status(self):
        ret = super(src2c, self).runnable_status()
        if ret == Task.SKIP_ME:

            lst = self.generator.bld.raw_deps[self.uid()]
            if lst[0] != self.signature():
                return Task.RUN_ME

            nodes = lst[1:]
            for x in nodes:
                try:
                    os.stat(x.abspath())
                except:
                    return Task.RUN_ME

            nodes = lst[1:]
            self.set_outputs(nodes)
            self.add_c_tasks(nodes) ❾

        return ret
```

❶ The processing will be delegated to the task

❷ Disable the warnings raised when a task has no outputs

❸ Make certain the task will be processed before the link task is considered

❹ When the task is executed, collect the process stdout which contains the generated file names

❺ Store the output file nodes in a persistent cache

❻ Create the tasks to compile the outputs

❼ The c tasks will be processed after the current task is done. This does not mean that the c tasks will always be executed.

❽ If the task generator of the *src* file has a link task, set the build order

❾ When this task can be skipped, force the dynamic c task creation

The output will be the following:

```
$ waf distclean configure build build
'distclean' finished successfully (0.006s)
Setting top to  : /tmp/scenarios_unknown
Setting out to  : /tmp/scenarios_unknown/build
Checking for program gcc,cc              : /usr/bin/gcc
Checking for program ar                  : /usr/bin/ar
'configure' finished successfully (0.115s)
Waf: Entering directory `/tmp/scenarios_unknown/build'
[1/3] src2c: foo.src
[2/5] c: build/shpip/a12.c -> build/shpip/a12.c.0.o
[3/5] c: build/shpop/a13.c -> build/shpop/a13.c.0.o
[4/5] c: x.c -> build/x.c.0.o
[5/5] cstlib: build/x.c.0.o build/shpip/a12.c.0.o build/shpop/a13.c.0.o -> build/libastaticlib.a
Waf: Leaving directory `/tmp/scenarios_unknown/build'
'build' finished successfully (0.188s)
Waf: Entering directory `/tmp/scenarios_unknown/build'
Waf: Leaving directory `/tmp/scenarios_unknown/build'
'build' finished successfully (0.013s)
```

12. Using the development version

12.1. Execution traces

12.1.1. Logging

The generic flags to add more information to the stack traces or to the messages is *-v* (verbosity), it is used to display the command-lines executed during a build:

```
$ waf -v
```

To display all the traces (useful for bug reports), use the following flag:

```
$ waf -vvv
```

Debugging information can be filtered easily with the flag *zones*:

```
$ waf --zones=action
```

Tracing zones must be comma-separated, for example:

```
$ waf --zones=action,envhash,task_gen
```

The Waf module *Logs* replaces the Python module logging. In the source code, traces are provided by using the *debug* function, they must obey the format "zone: message" like in the following:

```
Logs.debug('task: executing %r - it was never run before or its class changed', self)
```

The following zones are used in Waf:

Table 2. Debugging zones

Zone	Description
runner	command-lines executed (enabled when -v is provided without debugging zones)
deps	implicit dependencies found (task scanners)
task_gen	task creation (from task generators) and task generator method execution
action	functions to execute for building the targets
env	environment contents
envhash	hashes of the environment objects - helps seeing what changes
build	build context operations such as filesystem access
preproc	preprocessor execution
group	groups and task generators

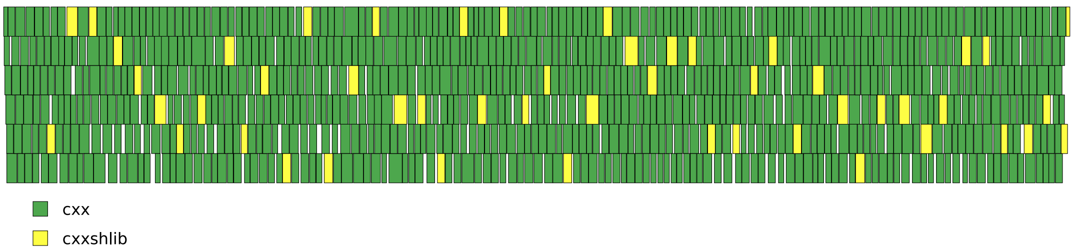
Debugging information can be displayed only after the command-line has been parsed. For example, no debugging information will be displayed when a waf tool is being by for the command-line options *opt.load()* or by the global init method function *init.tool()*

12.1.2. Build visualization

The Waf tool named *parallel_debug* injects code in Waf modules to obtain detailed execution traces. This module is provided in the folder `waflib/extras` and must be imported in one's project before use:

```
def options(ctx):
        ctx.load('parallel_debug', tooldir='.')

def configure(ctx):
        ctx.load('parallel_debug', tooldir='.')

def build(ctx):
        ...
```

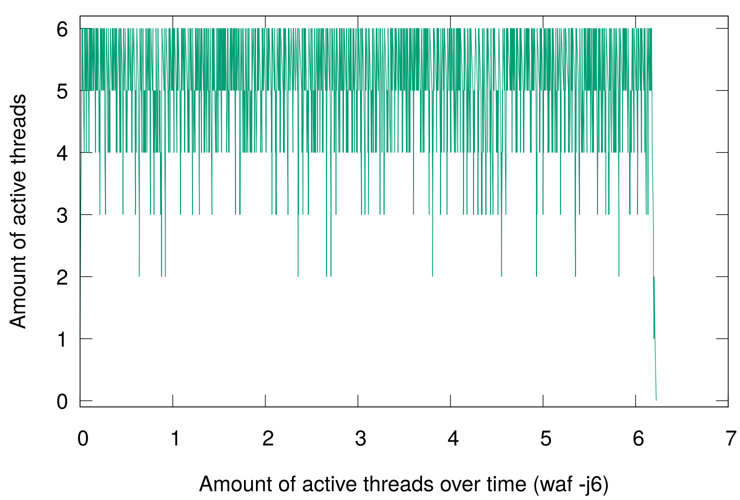
The execution generates a diagram of the tasks executed during the build in the file `pdebug.svg`:



Parallel build representation - 50 shared libraries (waf -j6)

Numeric values are output in the file `pdebug.dat` as space-separated values. That file can be processed by other applications such as Gnuplot to obtain other diagrams:

```
#! /usr/bin/env gnuplot
set terminal png
set output "output.png"
set ylabel "Amount of active threads"
set xlabel "Amount of active threads over time (waf -j6)"
unset label
set yrange [-0.1:6.2]
set ytic 1
plot 'pdebug.dat' using 3:7 with lines title "" lt 2
```



The data file columns are the following:

Table 3. pdebug file format

Column	Type	Description
1	int	Identifier of the thread which has started or finished processing a task

Column	Type	Description
2	int	Identifier of the task processed
3	float	Event time
4	string	Type of the task processed
5	int	Amount of tasks processed
6	int	Amount of tasks waiting to be processed by the task consumers
7	int	Amount of active threads

12.2. Profiling

12.2.1. Benchmark projects

The script `utils/genbench.py` is used as a base to create large c-like project files. The habitual use is the following:

```
$ utils/genbench.py /tmp/build 50 100 15 5
$ cd /tmp/build
$ waf configure
$ waf -p -j2
```

The C++ project created will generate 50 libraries from 100 class files for each, each source file having 15 include headers pointing at the same library and 5 headers pointing at other headers randomly chosen.

The compilation time may be discarded easily by disabling the actual compilation, for example:

```
def build(bld):
        from waflib import Task
        def touch_func(task):
                for x in task.outputs:
                        x.write('')
        for x in Task.classes:
                cls = Task.classes[x]
                cls.func = touch_func
                cls.color = 'CYAN'
```

12.2.2. Profile traces

Profiling information is obtained by calling the module cProfile and by injecting specific code. The most interesting methods to profile is *waflib.Build.BuildContext.compile*. The amount of function calls is usually a bottleneck, and reducing it results in noticeable speedups. Here is an example on the method compile:

```
from waflib.Build import BuildContext
def ncomp(self):
        import cProfile, pstats
        cProfile.runctx('self.orig_compile()', {}, {'self': self}, 'profi.txt')
        p = pstats.Stats('profi.txt')
        p.sort_stats('time').print_stats(45)

BuildContext.orig_compile = BuildContext.compile
BuildContext.compile = ncomp
```

Here the output obtained on a benchmark build created as explained in the previous section:

```
Fri Jul 23 15:11:15 2010    profi.txt

         1114979 function calls (1099879 primitive calls) in 5.768 CPU seconds

   Ordered by: internal time
   List reduced from 139 to 45 due to restriction 45

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   109500    0.523    0.000    1.775    0.000 /comp/waf/waflib/Node.py:615(get_bld_sig)
     5000    0.381    0.000    1.631    0.000 /comp/waf/waflib/Task.py:475(compute_sig_implicit_deps)
   154550    0.286    0.000    0.286    0.000 {method 'update' of '_hashlib.HASH' objects}
   265350    0.232    0.000    0.232    0.000 {id}
40201/25101    0.228    0.000    0.228    0.000 /comp/waf/waflib/Node.py:319(abspath)
    10000    0.223    0.000    0.223    0.000 {open}
    20000    0.197    0.000    0.197    0.000 {method 'read' of 'file' objects}
    15000    0.193    0.000    0.349    0.000 /comp/waf/waflib/Task.py:270(uid)
    10000    0.189    0.000    0.850    0.000 /comp/waf/waflib/Utils.py:96(h_file)
```

A few known hot spots are present in the library:

1. The persistence implemented by the cPickle module (the cache file to serialize may take a few megabytes)
2. Accessing configuration data from the Environment instances
3. Computing implicit dependencies in general

12.2.3. Optimizations tips

The Waf source code has already been optimized in various ways. In practice, the projects may use additional assumptions to replace certain methods or parameters from its build scripts. For example, if a project is always executed on Windows, then the *framework* and *rpath* variables may be removed:

```
from waflib.Tools.ccroot import USELIB_VARS
USELIB_VARS['cprogram'] = USELIB_VARS['cxxprogram'] = \
        set(['LIB', 'STLIB', 'LIBPATH', 'STLIBPATH', 'LINKFLAGS', 'LINKDEPS'])
```

12.3. Waf programming

12.3.1. Setting up a Waf directory for development

Waf is hosted on Github, and uses Git for source control. To obtain the development copy, use:

```
$ git clone https://github.com/waf-project/waf.git wafdir
$ cd wafdir
$ ./waf-light --make-waf
```

To avoid regenerating Waf each time, the environment variable **WAFDIR** should be used to point at the directory containing *waflib*:

```
$ export WAFDIR=/path/to/directory/
```

12.3.2. Specific guidelines

Though Waf is written in Python, additional restrictions apply to the source code:

1. Indentation is tab-only, and the maximum line length should be about 200 characters
2. The development code is kept compatible with Python 2.6, and the code is processed so that it runs on Python 2.5
3. The *waflib* modules must be insulated from the *Tools* modules to keep the Waf core small and language independent
4. API compatibility is maintained in the cycle of a minor version (from 2.0.0 to 2.0.n)

note

More code always means more bugs. Whenever possible, unnecessary code must be removed, and the existing code base should be simplified.

13. Waf architecture overview

This chapter provides describes the Waf library and the interaction between the components.

13.1. Modules and classes

13.1.1. Core modules

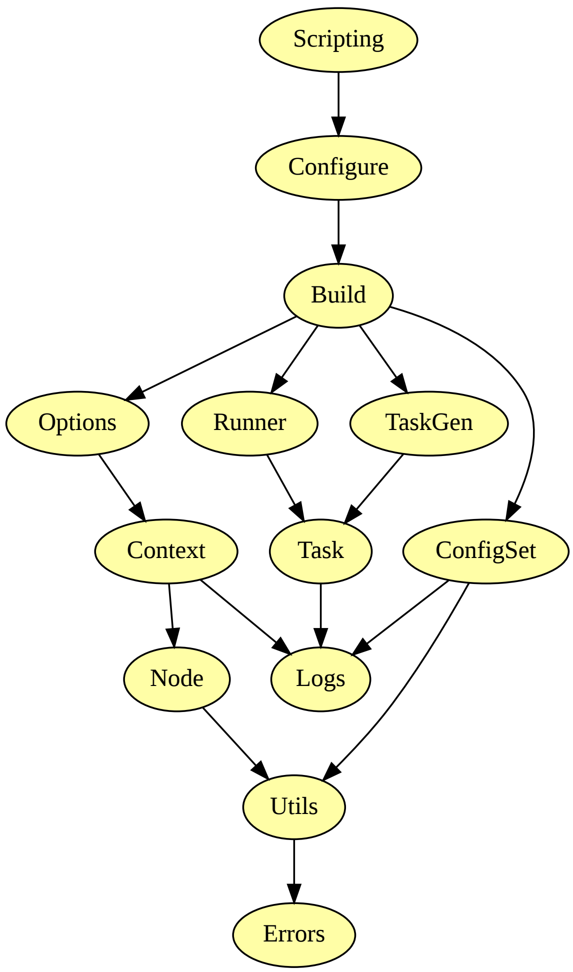
Waf consists of the following modules which constitute the core library. They are located in the directory `waflib/`. The modules located under `waflib/Tools` and `waflib/extras` are extensions which are not part of the Waf core.

Table 4. List of core modules

Module	Role
Build	Defines the build context classes (build, clean, install, uninstall), which holds the data for one build (paths, configuration data)
Configure	Contains the configuration context class, which is used for launching configuration tests and writing the configuration settings for the build
ConfigSet	Contains a dictionary class which supports a lightweight copy scheme and provides persistence services
Context	Contains the base class for all waf commands (context parameters of the Waf commands)
Errors	Exceptions used in the Waf code
Logs	Logging system wrapping the calls to the python logging module

Module	Role
Node	Contains the file system representation class
Options	Provides a custom command-line option processing system based on optparse
Runner	Contains the task execution system (thread-based producer-consumer)
Scripting	Constitutes the entry point of the Waf application, executes the user commands such as build, configuration and installation
TaskGen	Provides the task generator system, and its extension system based on method addition
Task	Contains the task class definitions, and factory functions for creating new task classes
Utils	Contains support functions and classes used by other Waf modules

Not all core modules are required for using Waf as a library. The dependencies between the modules are represented on the following diagram. For example, the module *Node* requires both modules *Utils* and *Errors*. Conversely, if the module *Build* is used alone, then the modules *Scripting* and *Configure* can be removed safely.



Dependencies between the core modules

13.1.2. Context classes

User commands, such as *configure* or *build*, are represented by classes derived from *waflib.Context.Context*. When a command does not have a class associated, the base class *waflib.Context.Context* is used instead.

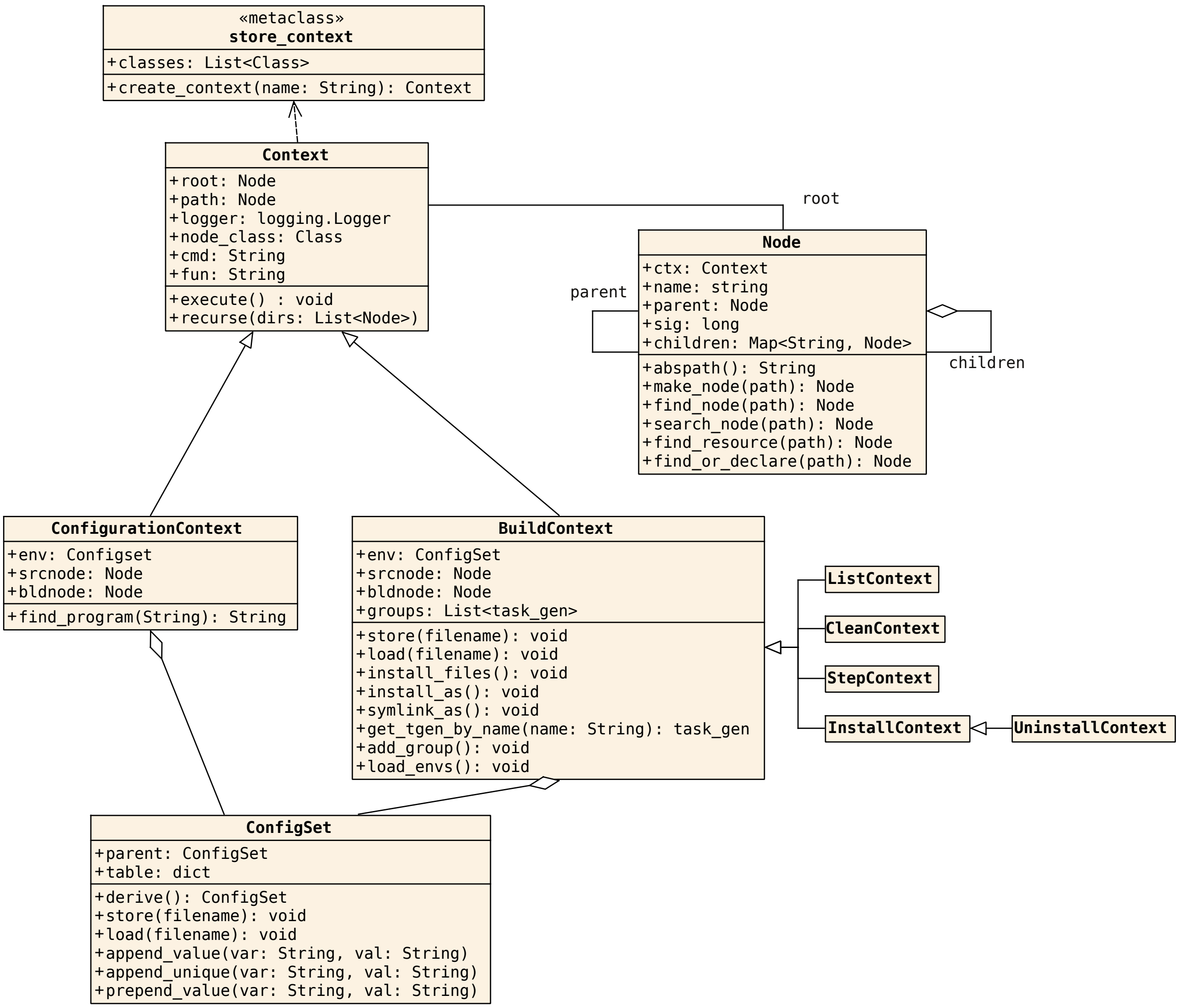
The method *execute* is the start point for a context execution, it often calls the method *recurse* to start reading the user scripts and execute the functions referenced by the *fun* class attribute.

The command is associated to a context class by the class attribute *cmd* set on the class. Context subclasses are added in *waflib.Context.classes* by the metaclass *store_context* and loaded through the function *waflib.Context.create_context*. The classes defined last will replace existing commands.

As an example, the following context class will define or override the *configure* command. When calling *waf configure*, the function *foo* will be called from wscript files:

```
from waflib.Context import Context
class somename(Context):
```

```
    cmd = 'configure'
    fun = 'foo'
```



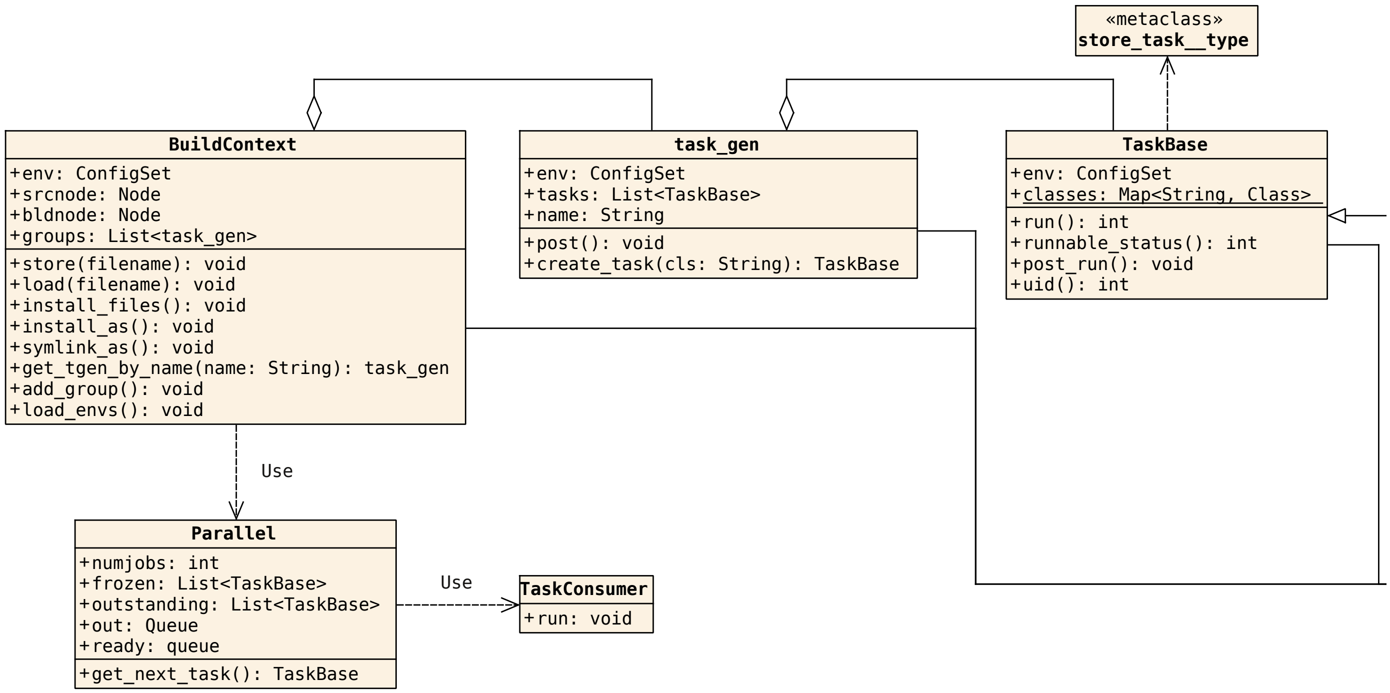
13.1.3. Build classes

The class *waflib.Build.BuildContext* and its subclasses such as *waflib.Build.InstallContext* or *waflib.Build.StepContext* have task generators created when reading the user scripts. The task generators will usually have task instances, depending on the operations performed after all task generators have been processed.

The *ConfigSet* instances are copied from the build context to the tasks (*waflib.ConfigSet.ConfigSet.derive*) to propagate values such as configuration flags. A copy-on-write is performed through most methods of that class (append_value, prepend_value, append_unique).

The *Parallel* object encapsulates the iteration over all tasks of the build context, and delegates the execution to thread objects (producer-consumer).

The overall structure is represented on the following diagram:



13.2. Context objects

13.2.1. Context commands and recursion

The context commands are designed to be as independent as possible, and may be executed concurrently. The main application is the execution of small builds as part of configuration tests. For example, the method *waflib.Configure.run_build* creates a private build context internally to perform the tests. Here is an example of a build that creates and executes simple configuration contexts concurrently:

```
import os
from waflib.Configure import conf, ConfigurationContext
from waflib import Task, Build, Logs

def options(ctx):
        ctx.load('compiler_c')

def configure(ctx):
        ctx.load('compiler_c')

def build(ctx):
        ctx(rule=run_test, always=True, header_name='stdio.h') ❶
        ctx(rule=run_test, always=True, header_name='unistd.h')

def run_test(self):
        top = self.generator.bld.srcnode.abspath()
        out = self.generator.bld.bldnode.abspath()

        ctx = ConfigurationContext(top_dir=top, out_dir=out) ❷
        ctx.init_dirs() ❸

        ctx.in_msg = 1 ❹
        ctx.msg('test') ❺

        header = self.generator.header_name
        logfile = self.generator.path.get_bld().abspath() + os.sep \
                + header + '.log'
        ctx.logger = Logs.make_logger(logfile, header) ❻

        ctx.env = self.env.derive() ❼
        ctx.check(header_name=header) ❽
```

❶ Create task generators which will run the method *run_test* method defined below

❷ Create a new configuration context as part of a *Task.run* call

❸ Initialize ctx.srcnode and ctx.bldnode (build and configuration contexts only)

❹ Set the internal counter for the context methods *msg*, *start_msg* and *end_msg*

❺ The console output is disabled (non-zero counter value to disable nested messages)

❻ Each context may have a logger to redirect the error messages

❼ Initialize the default environment to a copy of the task one

❽ Perform a configuration check

After executing *waf build*, the project folder will contain the new log files:

```
$ tree
.
|-- build
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- _cache.py
|   |-- config.log
|   |-- stdio.h.log
|   `-- unistd.h.log
`-- wscript
```

A few measures are set to ensure that the contexts can be executed concurrently:

1. Context objects may use different loggers derived from the *waflib.Logs* module.
2. Each context object is associated to a private subclass of *waflib.Node.Node* to ensure that the node objects are unique. To pickle Node objects, it is important to prevent concurrent access by using the lock object *waflib.Node.pickle_lock*.

13.2.2. Build context and persistence

The build context holds all the information necessary for a build. To accelerate the start-up, a part of the information is stored and loaded between the runs. The persistent attributes are the following:

Table 5. Persistent attributes

Attribute	Description	Type
root	Node representing the root of the file system	Node
node_deps	Implicit dependencies	dict mapping Node to signatures
raw_deps	Implicit file dependencies which could not be resolved	dict mapping Node ids to any serializable type
task_sigs	Signature of the tasks executed	dict mapping a Task computed uid to a hash

13.3. Support for c-like languages

13.3.1. Compiled tasks and link tasks

The tool *waflib.Tools.ccroot* provides a system for creating object files and linking them into a single final file. The method *waflib.Tools.ccroot.apply_link* is called after the method *waflib.TaskGen.process_source* to create the link task. In pseudocode:

```
call the method process_source:
  for each source file foo.ext:
    process the file by extension
      if the method create_compiled_task is used:
        create a new task
        set the output file name to be foo.ext.o
        add the task to the list self.compiled_tasks

call the method apply_link
  for each name N in self.features:
    find a class named N:
      if the class N derives from 'waflib.Tools.ccroot.link_task':
        create a task of that class, assign it to self.link_task
        set the link_task inputs from self.compiled_tasks
        set the link_task output name to be env.N_PATTERN % self.target
        stop
```

This system is used for *assembly*, *C*, *C++*, *D* and *fortran* by default. Note that the method *apply_link* is supposed to be called after the method *process_source*.

We will now demonstrate how to support the following mini language:

```
cp: .ext -> .o
cat: *.o -> .exe
```

Here is the project file:

```
def configure(ctx):
        pass

def build(ctx):
        ctx(features='mylink', source='foo.ext faa.ext', target='bingo')

from waflib.Task import Task
from waflib.TaskGen import feature, extension, after_method
from waflib.Tools import ccroot ❶

@after_method('process_source')
@feature('mylink')
def call_apply_link(self): ❷
        self.apply_link()

class mylink(ccroot.link_task): ❸
        run_str = 'cat ${SRC} > ${TGT}'

class ext2o(Task):
        run_str = 'cp ${SRC} ${TGT}'

@extension('.ext')
def process_ext(self, node):
        self.create_compiled_task('ext2o', node) ❹
```

❶ This import will bind the methods such as *create_compiled_task* and *apply_link_task*

❷ An alternate definition would be calling *waflib.TaskGen.feats['mylink'] = ['apply_link']*

❸ The link task must be a subclass of another link task class

❹ Calling the method *create_compiled_task*

The execution outputs will be the following:

```
$ waf distclean configure build -v
'distclean' finished successfully (0.005s)
Setting top to   : /tmp/architecture_link
Setting out to   : /tmp/architecture_link/build
'configure' finished successfully (0.008s)
Waf: Entering directory `/tmp/architecture_link/build'
[1/3] ext2o: foo.ext -> build/foo.ext.0.o
12:50:25 runner ['cp', '../foo.ext', 'foo.ext.0.o']
[2/3] ext2o: faa.ext -> build/faa.ext.0.o
12:50:25 runner ['cp', '../faa.ext', 'faa.ext.0.o']
[3/3] mylink: build/foo.ext.0.o build/faa.ext.0.o -> build/bingo
12:50:25 runner 'cat foo.ext.0.o faa.ext.0.o > bingo'
Waf: Leaving directory `/tmp/architecture_link/build'
'build' finished successfully (0.041s)
```

> **Note:** Task generator instances have at most one link task instance

13.4. Writing re-usable Waf tools

13.4.1. Adding a waf tool

Importing the code

The intent of the Waf tools is to promote high cohesion by moving all conceptually related methods and classes into separate files, hidden from the Waf core, and as independent from each other as possible.

Custom Waf tools can be left in the projects, added to a custom waf file through the *waflib/extras* folder, or used through *sys.path* changes.

The tools can import other tools directly through the *import* keyword. The scripts however should always import the tools to the *ctx.load* to limit the coupling. Compare for example:

```
def configure(ctx):
    from waflib.extras.foo import method1
    method1(ctx)
```

and:

```
def configure(ctx):
    ctx.load('foo')
    ctx.method1()
```

The second version should be preferred, as it makes fewer assumptions on whether *method1* comes from the module *foo* or not, and on where the module *foo* is located.

Naming convention for C/C++/Fortran

The tools *compiler_c*, *compiler_cxx* and *compiler_fc* use other waf tools to detect the presense of particular compilers. They provide a particular naming convention to give a chance to new tools to register themselves automatically and save the import in user scripts. The tools having names beginning by *c_*, *cxx_* and *fc_* will be tested.

The registration code will be similar to the following:

```
from waflib.Tools.compiler_X import X_compiler
X_compiler['platform'].append('module_name')
```

where **X** represents the type of compiler (*c*, *cxx* or *fc*), **platform** is the platform on which the detection should take place (linux, win32, etc), and **module_name** is the name of the tool to use.

13.4.2. Command methods

Subclassing is only for commands

As a general rule, subclasses of *waflib.Context.Context* are created only when a new user command is necessary. This is the case for example when a command for a specific variant (output folder) is required, or to provide a new behaviour. When this happens, the class methods *recurse*, *execute* or the class attributes *cmd*, *fun* are usually overridden.

> If there is no new command needed, do not use subclassing.

Domain-specific methods are convenient for the end users

Although the Waf framework promotes the most flexible way of declaring tasks through task generators, it is often more convenient to declare domain-specific wrappers in large projects. For example, the samba project provides a function used as:

```
bld.SAMBA_SUBSYSTEM('NDR_NBT_BUF',
    source    = 'nbtname.c',
    deps      = 'talloc',
    autoproto = 'nbtname.h'
    )
```

How to bind new methods

New methods are commonly bound to the build context or to the configuration context by using the *@conf* decorator:

```
from waflib.Configure import conf

@conf
def enterprise_program(self, *k, **kw):
    kw['features'] = 'c cprogram debug_tasks'
    return self(*k, **kw)

def build(bld):
    # no feature line
    bld.enterprise_program(source='main.c', target='app')
```

The methods should always be bound in this manner or manually, as subclassing may create conflicts between tools written for different purposes.

14. Further reading

Due to the amount of features provided by Waf, this book cannot be both complete and up-to-date. For greater understanding and practice the following links are recommended to the reader:

Table 6. Recommended links

Link	Description
https://waf.io/apidocs/index.html	API documentation
https://waf.io	The Waf project page and downloads
https://github.com/waf-project/waf	Source code repository and issue tracker
https://waf.io/blog/2011/01/python-32-and-build-system-kit.html	Information on the build system kit

15. Appendices

15.1. Environment variables

Environment variable name	Usage
DESTDIR	Default installation base directory when *--destdir* is not provided on the command-line.
JOBS	Default value for the amount of parallel jobs. Has no effect when *-j* is provided on the command-line.
MSYSTEM	When set to a non-empty value on Windows platforms, the msys root directory is used as root path.
NOCLIMB	When set to a non-empty value, the process will not search for a build configuration in upper folders. This can be used to force automatic project configuration in sub-folders (Configure.autoconfig) in the context of nested projects.
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.
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.
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.
PREFIX	Default installation prefix when *--prefix* is not provided on the command-line.
TERM	When set to *dumb* or *emacs*, console colors are disabled. May have no effect when *--color* is provided on the command-line.
WAF_CMD_FORMAT	When set to *string*, commands executed by *Context.exec_command* are also displayed as plain strings to facilitate copy-pasting.
WAFDIR	Path to the directory containing *waflib/*; the version in the waf file and in *waflib/Context.py* must match else an error is raised.
WAF_HOUR_FORMAT	Time format string, the default value is *%H:%M:%S*.
WAFLOCK	Name of the lock file indicating whether a project is configured. Overriding this variable can be necessary when a project is shared over a network file system.
WAF_LOG_FORMAT	Debug message format string, the default value is *%(asctime)s %(c1)s%(zone)s%(c2)s %(message)s*.
WAF_PRINT_FAILURE_LOG	When set to a non-empty value, failing configuration tests are printed to the console.

15.2. Glossary

Build Order
: The build order is the sequence in which tasks must be executed. Because tasks can be executed in parallel, several build orders can be computed depending on the constraints between the tasks. When a build order cannot be computed (usually by contradictory order constraints), the build is said to be in a deadlock.

Dependency
: A dependency represents the conditions by which a task can be considered up-to-date or not (execution status). The dependencies can be explicit (file inputs and outputs) or abstract (dependency on a value for example).

Task generator
: A task generator is an object instance of the class Task.task_gen. The task generators encapsulate the creation of various task instances at a time, and simplify the creation of ordering constraints between them (for example, compilation tasks are executed before link tasks).

Task
: A Waf task is an object instance of the class Task.TaskBase. Waf tasks may be simple (Task.TaskBase) or related to the filesystem (Task.Task). Tasks represent the production of something during the build (files in general), and may be executed in sequence (with ordering constraints) or in parallel.

Tool
: A Waf tool is a python module containing Waf-specific extensions. The Waf tools are located in the folder `waflib/Tools/` and usually contain a global variable *configure* which may reference functions to execute in the configuration.

Feature (Task Generator)
: The feature system enables binding new behaviours to task generators instances by scheduling class methods for execution. The methods are weaved in an order that this specified on the method definition (after/before constraints). One major benefit is that additional behaviours can be weaved onto existing targets without modifying the build scripts. This approach is comparable to the CSS class declarations in the web design context.

Node
: The Node class is a data structure used to represent the filesystem in an efficient manner. The node objects may represent files or folders. File nodes are associated to signatures objects. The signature can be hashes of the file contents (source files) or task signatures (build files).

Command
: Function present in the top-level project file (wscript) and accepting a *waflib.Context.Context* instance as unique input parameter. The function is executed when its name is given on the command-line (for example running *waf configure* will execute the function *configure*)

Variant
: Additional output directory used to enable several (build) commands to create the same targets with different compilation flags.

---

1. *find_program* may use the same variable from the OS environment during the search, for example *CC=gcc waf configure*

2. It is actually considered a best practice to avoid copying files. When this is required, consider installing files or re-using the examples provided under the folder `demos/subst` of the source distribution.

3. When file copies cannot be avoided, the best practice is to change the file names

4. A particular idx value can be forced in order to prevent name collisions

5. If the build generates a lot of headers with arbitrary names, a brute-force approach to obtain rebuilds can be to compute and assign a hash to **env.CCDEPS**

6. For performance reasons, Node objects should be passed in includes/INCLUDES in the build phase. Yet, env objects containing Node objects are not serializable, so this technique cannot be used in the configuration phase

---

Last updated 2018-05-22 23:18:10 CEST