Making build systems not suck!

Jussi Pakkanen
jpakkane@gmail.com
@jpakkane
https://github.com/jpakkane/meson
Disclaimer

https://github.com/jpakkane/meson
“Let's talk about build tools:
All the build tools suck!
Let's just be up-front: that's it!”

Robert Ramey
CppCon 2014

https://github.com/jpakke/meson
How do they suck, exactly?
Productivity is all about the Flow

- originally coined by Mihály Csíkszentmihályi
- intense focus arising from lack of distractions
- hard to achieve (>30 minutes), easy to lose
- impossible to achieve with noisy offices, bad tools or irritating coworkers

https://github.com/jpakkane/meson
The programmer's eternal cycle

Every time this takes longer than 5 seconds, you lose the flow.
“A running compiler holds a mutex on your brain.”
Some practical problems

https://github.com/jpakkane/meson
Simple things must be simple, hard things must be possible¹.

¹ Preferably also easy.
Simplest possible case:
build helloworld
The image is a diagram that outlines the process of configuring and building a software project. The diagram starts with `autoscan` and `autoconf`, followed by `configure.scan` and `configure.ac`, and then `configure`.

- `configure.scan` leads to `configure.ac` which is an input file for `autoconf` and `autoheader`.
- `configure.ac` also points to `config.status`, `config.log`, and `config.h`.
- `config.status` and `config.log` are generated by `autoconf`.
- `config.h` is generated by `autoheader`.
- `Makefile.in` is generated by `automake`.
- `Makefile.am` is required for each `Makefile`.
- `Makefile` is generated by `automake`.
- `Makefile` is used as input for `make`.
- `Makefile` is also input for `install-sh`.
- `install-sh` is a dependency on `COPYING`, `INSTALL`, `NEWS`, `README`, `AUTHORS`, and `ChangeLog`.

The diagram illustrates the flow of files and dependencies used in the configuration and building process. The file `stamp-h1` includes a timestamp for `config.h`.
Almost as simple:
compile a program with a dependency

https://github.com/jpakke/meson
A common pattern with CMake

```cmake
project(sampleapp C)
cmake_minimum_required(VERSION 2.8.9)
include(FindPkgConfig)
pkg_search_module(GTK3 gtk+-3.0)
include_directories(${GTK3_INCLUDE_DIRS})
add_executable(sampleapp sampleapp.c)
target_link_libraries(sampleapp ${GTK3_LIBRARIES})
```

BUG!
BUG!
BUG!
A hard case: precompiled headers
Sorry, but doing this right as a first-class feature is very non-trivial. Every platform does PCH differently, so it is hard to define a common interface. It is probably possible, but we've not had the motivation/time/funding to do it.

Currently CMake does provide enough primitives for projects to do it themselves on each platform. For example, the OBJECT_OUTPUTS and OBJECT_DEPENDS properties can be used to do MSVC-style PCH dependencies in Makefile generators. Custom commands can achieve gcc-style PCH. I think the contributed scripts attached to this bug help with some of this.
Design goals to not sucking.

https://github.com/jpakkane/meson
Either build fully up to date or error out.

Silent stale builds are not acceptable under any circumstances!
Do the common thing by default, allow overrides.
Syntax must not look like line noise.

Addendum: no quoting hell ever!

https://github.com/jpakkane/meson
A second spent writing build definitions is a second wasted.

https://github.com/jpakkane/meson
User must only need to provide info that the system can not deduce otherwise.
Minimize global state

https://github.com/jpakkane/meson
Build speed is essential!

Dirty implementation tricks are OK assuming they are reliable and don't leak to the interface.

https://github.com/jpakkanen/meson
Sane, sufficiently rich data types.
Make dependency loops impossible to write.

https://github.com/jpakkane/meson
User experience should be roughly this
Other build systems have good features, let’s steal all of them!
GNU Autotools

Configuration concept

https://github.com/jpakkane/meson
CMake

Platform abstraction
Multiple backends

https://github.com/jpakkane/meson
SCons

Aesthetically pleasing syntax matters

https://github.com/jpakkane/meson
GYP

Definition language not Turing complete

Scalability

https://github.com/jpakkane/meson
QMake/QBS

Native Qt support

https://github.com/jpakkane/meson
By your powers combined, come I:

The Meson Build system
Meson code examples

https://github.com/jpakkane/meson
The helloworld

project('sample project', 'c')
executable('prog', 'sample.c')

https://github.com/jpakkane/meson
What do these two lines get you?

- build on Linux, OSX, Windows, others
- compiler warnings enabled by default
- different build types (debug, optimized etc)
- cross-compilation
- outputs are native binaries, produced by the native toolchain

https://github.com/jpakkane/meson
Using a dependency

project('dep sample', 'c')
gtk3_dep = dependency('gtk+-3.0')
executable('gtkprog', 'gsample.c',
    dependencies : gtk3_dep)
Unit tests

project('sample', 'c')
exe = executable('sample', 'sample.c')
test('simple test', exe)

https://github.com/jpakkane/meson
Precompiled headers

project('sample', 'cpp')
exe = executable('sample', 'sample.cc',
               cpp_pch : 'pch/sample_pch.h')

Compilation time for simple Qt5 dbus tool on Ubuntu phone went from 2 minutes to 55 seconds.
A real world example

- a C++ shared library that uses GLib
- unit test
- install
- create a pkg-config file

https://github.com/jpakka/meson
Top level

project('c++ foolib', 'cpp')

add_global_arguments('-std=c++11', language : 'cpp')
glib_dep = dependency('glib-2.0')

inc = include_directories('include')

subdir('include')
subdir('src')
subdir('test')
include subdir

install_headers('foolib.h')
foolib = shared_library('foo', 'source1.cpp', 'source2.cpp',
    include_directories : inc,
    dependencies : glib_dep,
    install : true)

pkgconfig_gen(libraries : foolib,
    version : '1.0',
    name : 'libfoobarp',
    filebase : 'foobar',
    description : 'A Library to barnicate your foos.')
test_subdirectory

testexe = executable('testexe', 'footest.cpp',
    include_directories : inc,
    link_with : foolib)

test('foolib test', testexe)
That's the build definition in its entirety.

No, really!
Oh, and one more thing ...

```meson
project('qt5 sample', 'cpp')

qt5dep = dependency('qt5', modules : 'Widgets')

q5exe = executable('qt5app',
    sources : ['main.cpp', 'mainWindow.cpp']
    moc_headers : 'mainWindow.h',
    ui_files : 'mainWindow.ui',
    qresources : 'stuff.qrc',
    dependencies : qt5dep)
```
Performance experiment:
Compiling GLib (without GIO)
GLib configuration times

- CFLAGS='-O0 -g' CXXFLAGS='-O0 -g' ./autogen.sh
  - 5 minutes

- default settings for Meson
  - 24 seconds
GLib full build times

- `make -j 2` for Autotools
  - 4m 55s
- `ninja -j 2` for Meson
  - 1m 28s
- **CAVEAT:** Meson builds slightly less code
GLib incremental build times

- rebuild with no changes
  - 3s for Autotools
  - 0.062s for Meson
- rebuild after "touch glib/gprintf.c"
  - 1m 18s for Autotools
  - 1.1s for Meson

https://github.com/jpakkane/meson
Desktop performance

- configuration step usually <5 seconds
- no-op build time <1s even for >10k files
- full CPU saturation due to single Ninja process

https://github.com/jpakkane/meson
Advanced features

https://github.com/jpakka/meson
Source generation

```python
idlcc = executable('idlcompiler', 'idlcompiler.c')

gen = generator(idlcc,
                  output  : ['@BASENAME@.h', '@BASENAME@.c'],
                  arguments : ['@INPUT@', '@OUTPUT0@', '@OUTPUT1@'])

generated = gen.process('class1.idl', 'class2.idl', 'class3.idl')

e2 = executable('prog', 'prog.c', generated)
```

What does it take to cross compile this example?
Option 2

```python
idlc = executable('idlcompiler', 'idlcompiler.c', native = True)
gen = generator(idlc,
    output : ['@BASENAME@.h', '@BASENAME@.c'],
    arguments : ['@INPUT@', '@OUTPUT0@', '@OUTPUT1@'])
generated = gen.process('class1.idl', 'class2.idl', 'class3.idl')
e2 = executable('prog', 'prog.c', generated)
```

Project options

• **strongly typed user-definable options**

```python
option('testoption', type : 'string', value : 'optval',
       description : 'An option to do something')
option('combo_opt', type : 'combo', choices : ['one', 'two', 'combo'],
       value : 'combo')
```

• **query and set from the command line**

```
mesonconf -Dcombo_opt=one
```
Supported languages

- **Tier 1**: C, C++
- **Tier 2**: ObjC, ObC++, Fortran
- **Tier 3**: Java, C#, Vala, Rust

https://github.com/jpakkane/meson
Code quality

- over 100 unit tests
- each one is also documentation
- all new features must come with a test
The most controversial feature

https://github.com/jpakkane/meson
No in-source builds

- Can only build out-of-source
- Arbitrarily many parallel builds for one source tree
- Turns out you can only reliably do in-source or out-of-source but not both
- Join the dark side, we have cookies

https://github.com/jpakkan/meson
Benefit of OSB: static analyzer

- steps to analyze are the always the same

  mkdir scantmp && cd scantmp
  scan-build meson ..
  scan-build ninja
  cd .. && rm -rf scantmp
Run it with "ninja staticanalyze"

```
run_target('staticanalyze', 'scripts/staticanalyze.sh')
```

```
#!/bin/sh
cd "${MESON_SOURCE_ROOT}"
rm -rf scantmp
mkdir scantmp && cd scantmp
scan-build meson ..
scan-build ninja
cd .. & rm -rf scantmp
```

Impossible to achieve if build system allows in-source builds.

https://github.com/jpakkane/meson
The compatibility matrix hell

<table>
<thead>
<tr>
<th>Autotools</th>
<th>CMake</th>
<th>Meson</th>
<th>Scons</th>
<th>Ant</th>
<th>Premake</th>
<th>Make</th>
<th>Eclipse</th>
<th>Qt Creator</th>
<th>Visual Studio</th>
<th>Code::Blocks</th>
<th>XCode</th>
<th>KDevelop</th>
<th>Gnome Builder</th>
</tr>
</thead>
</table>
The obvious solution

- Autotools
- CMake
- Meson
- Scons
- Ant
- Premake
- Make
- Eclipse
- Qt Creator
- Visual Studio
- Code::Blocks
- XCode
- KDevelop
- Gnome Builder
Format details

- simple JSON schema for deep build system / IDE integration
- introspectable *everything*
  - projects, source files, targets, build flags, project options, unit tests including command line and environment variables
- right click on failed unit test, select “run in debugger”
What can you build with it?

- GLib
- Python 3
- Qt Creator
- SDL2

- Mesa 3D
- Mame
- Mozilla NSPR

https://github.com/jpakkane/meson
Distro packages vs embedded source

https://github.com/jpakkane/meson
Meson subprojects

- any Meson project can be used as a subproject
- becomes a sandboxed part of the parent's build
- projects can query if they are being used as subprojects
- “The Go Github thing” but with C/C++
Sample subproject usage snippet

foolib = dependency('foo', required : false)
if foolib.found()
    # set up project with external lib
else
    subproject('foo')
    # set up project with embedded lib
endif
Here's what it looks like in practice
Further info

- Apache License 2.0
- Reference implementation in Python 3
- Packaged in Ubuntu (14/10) and Debian (Jessie)
- Github has wiki, manual, reference docs ...
- Contributions welcome

https://github.com/jpakkane/meson