

---

# **Dawn Documentation**

***Release 0.0.1***

**Fabian Thuring, Carlos Osuna and Tobias Wicky**

**ETH Zurich and MeteoSwiss**

October 13, 2017



## THE BASICS

<b>1</b>	<b>About this project</b>	<b>1</b>
<b>2</b>	<b>First steps</b>	<b>3</b>
<b>3</b>	<b>Stencil Intermediate Representation</b>	<b>7</b>
<b>4</b>	<b>Dawn toolchain</b>	<b>9</b>
<b>5</b>	<b>ISIR</b>	<b>11</b>
<b>6</b>	<b>GridTools</b>	<b>13</b>
<b>7</b>	<b>Dawn Coding standard</b>	<b>15</b>
<b>8</b>	<b>Doxygen</b>	<b>17</b>
<b>9</b>	<b>CMake Modules</b>	<b>19</b>



## ABOUT THIS PROJECT

**Dawn** is a compiler toolchain for developing high-level DSLs for geophysical fluid dynamics models. Development productivity of large scientific codes, like geophysical fluid dynamics (GFD) models, decreased drastically in recent times due to the fact these community models often have to run efficiently on multiple computing architectures that impose the use of different programming models. Embedded domain specific languages (EDSLs) in C++ have been used successfully in the past to enable a clear separation of concerns between the domain algorithms and the implementation strategy, thus allowing a single source code which is performance portable across heterogeneous architectures. Yet, using EDSLs in a correct and efficient manner often requires expert knowledge in high-performance computing. In addition, current DSLs are usually designed and developed for a specific scientific model with little to no reusability among DSLs.

We introduce a new compiler framework, Dawn, that decouples optimization and code generation from high level DSLs. By exposing a Stencil Intermediate Representation (SIR), we allow users to share the toolchain to optimize and generate code among several DSLs. This allows the design of expressive, concise DSLs that can focus on applicability and don't need to bother with the aspect of high-performance computing. Using Dawn as a common compiler infrastructure can drastically reduce development and maintenance effort, while increasing the performance of the generated code, for new and existing DSLs in the GFD model domain.

## Core Features

- Dawn allows the user to generate fast performing code for several back-ends from a relatively simple Stencil Intermediate Representation (SIR).
- Dawn exposes several APIs in different languages (C++, Java, Python) to parse and process the SIR.
- Dawn is able to generate code to be run on Distributed Memory Machines based on MPI, Machines with access to GPUs based on CUDA (via the gridtools EDSL) as well as naive C++ code with close to no parallelism for debugging.
- Dawn offers a wide range of optimization and static analysis passes to guarantee correctness as well as performance of the generated parallel program.



## FIRST STEPS

This section demonstrates the basic features of Dawn. Before getting started, make sure that development environment is set up to compile the included set of test cases.

### Compiling the library

#### Linux

On Linux, you'll need to install a C++11 toolchain (e.g **gcc** or **clang**) as well as **CMake** ( $\geq 3.3$ ). In addition, you will need **Protobuf** ( $\geq 3.4$ ). For example, on Ubuntu the following will install the necessary dependencies

```
sudo apt-get install g++ cmake
```

Note that most Ubuntu distributions don't ship the latest Protobuf library which means you need to compile it yourself (don't worry it's super easy!). First, grab the latest release of Protobuf from [GitHub](https://github.com/google/protobuf), configure it with CMake (the `CMakeLists.txt` file is located in the **cmake/** subdirectory) and then build it

```
git clone https://github.com/google/protobuf.git
cd protobuf
mkdir build
cd build
export PROTOBUF_INSTALL_DIR="$(pwd)/install"
cmake ../cmake/ -DCMAKE_INSTALL_PREFIX="$PROTOBUF_INSTALL_DIR"
make install -j 4
```

which will install the library locally into **build/install**. Next, we can build Dawn. Note that we need to point CMake to the `protobuf-config.cmake` file of Protobuf (which is located in `${PROTOBUF_INSTALL_DIR}/lib/cmake/protobuf/`) via the CMake variable `Protobuf_DIR`.

```
git clone https://github.com/google/protobuf.git
mkdir build
cd build
cmake .. -DProtobuf_DIR="$PROTOBUF_INSTALL_DIR/lib/cmake/protobuf/"
make install -j 4
```

The last line will both install and compile the library locally in **<dawn-dir>/install/**.

#### Mac OS

On Mac OS, you'll need to install Xcode as well as **CMake** ( $\geq 3.3$ ). In addition, you will need **Protobuf** ( $\geq 3.4$ ). First, make sure you have the Xcode Command Line Tools installed

```
xcode-select --install
```

If you are using [Homebrew](https://brew.sh/), use the following to install **Protobuf** and **CMake**

```
brew update
brew install cmake protobuf
```

After installing the prerequisites, run

```
mkdir build
cd build
cmake ..
make install -j 4
```

The last line will both install and compile the library locally in **<dawn-dir>/install/**.

## Linking the library

We suggest using [CMake](#) for a smooth integration with your project. The installation of Dawn provides a find configuration script (which can be used with [find\\_package](#)) that should be installed into your system.

In your projects `CMakeLists.txt`, simply add

```
find_package(Dawn)
```

to import all the necessary information.

---

**Note:** The script is located in **<dawn-install-dir>/cmake**. If CMake has trouble finding the script you may pass this location directly to CMake by adding `-DDawn_DIR=<dawn-install-dir>/cmake` to the command line flags.

---

If Dawn is found, the following CMake variables will be defined

Variable	Explanation
DAWN_FOUND	True if headers and libraries of Dawn were found.
DAWN_ROOT	Installation prefix of Dawn.
DAWN_VERSION	Version of Dawn (format X.Y.Z).
DAWN_ASSERTS	True if Dawn was compiled with asserts.
DAWN_INCLUDE_DIRS	Dawn include directories.
DAWN_LIBRARY_DIRS	Link directories for Dawn libraries.
DAWN_LIBRARY	Library to link against (this is an alias of <code>DAWN_STATIC_LIBRARY</code> ).
DAWN_STATIC_LIBRARY	Static library of Dawn.
DAWN_HAS_SHARED_LIBRARY	True if the shared library of Dawn is available.
DAWN_SHARED_LIBRARY	Shared library of Dawn.

Finally, just link the static library of Dawn to your own library or executable

```
target_link_libraries(${target} ... PUBLIC ${DAWN_STATIC_LIBRARY})
```

or use `DAWN_SHARED_LIBRARY` instead of `DAWN_STATIC_LIBRARY` if you want to linkt against the shared library.

## Example

If we want to link our file (`foo.cpp`) against the static library of Dawn, we may use something like

```
# Find the Dawn library (abort if we cannot find it!)
find_package(Dawn REQUIRED)

# Expose the Dawn include directories
include_directories(SYSTEM ${DAWN_INCLUDE_DIRS})
```



```
# Compile foo and link against Dawn
add_executable(foo foo.cpp)
target_link_libraries(foo PUBLIC ${DAWN_STATIC_LIBRARY})

# Dawn requires atleast C++11 (you can also set this globally!)
set_property(TARGET foo PROPERTY CXX_STANDARD 11)
```

## Using the library

TODO:

```
#include <dawn/Dawn.h>

int main() {

    return 0;
}
```



## STENCIL INTERMEDIATE REPRESENTATION

Developing a single DSL that is able to support all numerical methods and computational patterns present in GFD models present a serious challenge due to the wide domain it needs to cover. In this project, we accept the reality that multiple scientific communities desire to develop their own DSL language, tailored for their model and needs. The Stencil Intermediate Representation (SIR) allows to define multiple high level DSLs in a lightweight manner by reusing most of the complex toolchain i.e the Dawn library. In addition, a standardized SIR has another major advantage: It allows to easily interact with third-party developers and hardware manufacturers. For instance, the SIR of the [COSMO](#) atmospheric model, serialized to a mark-up Language like [JSON](#), can be distributed to hardware vendors, which in turn have their proprietary, in-house compilers based on the SIR and can return plain C/C++ or CUDA code. This frees third-party developers from compiling the models and hopefully improves collaboration.

In this section we define the specification of the SIR.



## DAWN TOOLCHAIN

The toolchain of Dawn contains several modules and is organized in passes, allowing to customize the process by enabling, disabling or change the order of execution of the individual passes. The default order of execution, depicted in Figure 4.1, starts by translating the SIR into an *internal* SIR (*ISIR*) which enriches the SIR with various information, like the computation of the access patterns, required by the following passes. As the initial construction of the ISIR generates a parallel program that is most likely incorrect under the given parallel model, we invoke several passes (*Parallel Model Builder*) to build a valid parallel program. The *Safety Checker* passes will run various static analysis routines and the *Optimizer* will perform several optimizations such as reordering and fusing groups of statements, setting software managed caches or reducing shared-memory pressure. Finally, the code generation will translate the ISIR into the main DSL of GridTools or directly to C++.

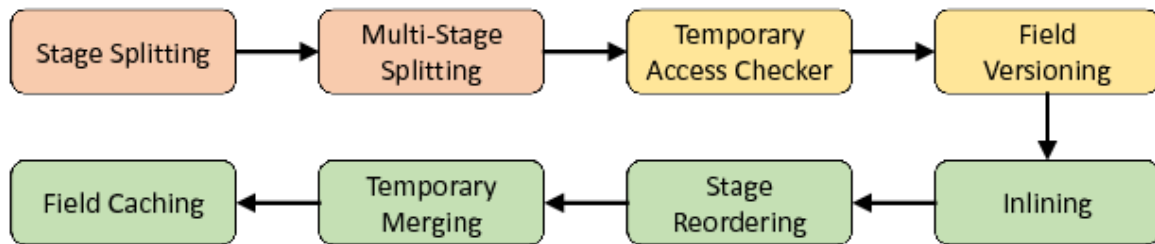


Fig. 4.1: Default order of execution of the passes grouped in *Parallel Model Builder* (orange), *Safety Checker* (yellow) and *Optimizer* (green).

### Parallel Model Builder

The parallel model builder passes transform the initial Stage and Multi-Stage arrangement into a valid parallel program under the *GridTools parallel model*.

### Safety Checker

TODO

### Optimizer

TODO



---

CHAPTER  
**FIVE**

---

**ISIR**

TODO





## GRIDTOOLS

TODO

### Parallel Model

TODO



## DAWN CODING STANDARD

### Introduction

This document attempts to describe a few coding standards that are being used in the Dawn source tree. Although no coding standards should be regarded as absolute requirements to be followed in all instances, you should try to follow the [LLVM](#) coding standard. We deviate from the LLVM standard in certain areas as listed in the following.

### Supported C++11 Language and Library Features

While LLVM restricts the usage of C++11, we do not impose any constraints on the allowed features. While we generally allow the usage of C++ exceptions, you should avoid them at the interface boundary to make exposing the API more convenient - [RTTI](#) should be avoided.

### Style Issues

#### Naming of Types, Functions, Variables, and Enumerators

- **Variable Names** should start with a lower-case letter (e.g `textFileReader`) as opposed to LLVM which starts with an uppercase-letter.
- **Members** should start with a lower-case letter and contain an `_` suffix (e.g `myMember_`)

For types, functions and enumerators you should follow the LLVM style.

### Source Code Formatting

#### Source Code Width

You may use up to 100 columns of text when writing your code (instead of 80 as dictated by LLVM). The rationale behind this is that we do not live in the 90s any more and people usually have wide screens.

#### Spaces Before Parentheses

You should never use spaces before parentheses. For example, this is good

```
if(x) ...
for(i = 0; i != 100; ++i) ...

somefunc(42);
assert(3 != 4 && "laws of math are failing me");
```

```
A = foo(42, 92) + bar(X);
```

and this is bad

```
if (x) ...
for (i = 0; i != 100; ++i) ...

somefunc (42);
assert (3 != 4 && "laws of math are failing me");

A = foo (42, 92) + bar (X);
```

## Pointer and Reference Alignment

You should always align pointers and references on the left i.e directly following the type. For example, this is considered good

```
int* a = ...
int& b = ...

const char** ptr = ...
```

while the following is considered bad

```
int *a = ...
int &b = ...

const char **ptr = ...
```

## Clang Format

To enforce most of these coding standards, CMake can be configured to run `clang-format` on each file with

```
make format
```

The clang-format file is located in the root directory at `<dawn-dir>/ .clangformat`.

## DOXYGEN

The full [Doxygen](#) documentation of the code can be found here.



## CMAKE MODULES

This section describes the CMake functionality of Dawn and its subprojects.

### Installation

Add the `<dawn>/cmake/modules` directory to the `CMAKE_MODULE_PATH` to use the functions, macros and modules:

```
list(APPEND CMAKE_MODULE_PATH "<dawn>/cmake/modules")
```

Note that all Dawn projects contain a `dawn_cmake_init` macro which tries to find the CMake modules of Dawn.

```
include(DawnCMakeInit)
dawn_cmake_init()
```

### Functions & Macros

Each function and macro uses a `snake-case` identifier and is defined in a separate file using the corresponding `camel-case` filename. For example, to use the function `dawn_add_target_clean_all` include the file `DawnAddTargetCleanAll`.

```
include(DawnAddTargetCleanAll)
dawn_add_target_clean_all()
```

### dawn\_get\_compiler\_info

Get the identification string of the compiler.

```
dawn_get_compiler()
```

The function defines the following variable:

**DAWN\_COMPILER\_STRING** Variable which will contain the identification string of the compiler.

and conditionally one of the following:

**DAWN\_COMPILER\_MSVC** Set to 1 if the compiler in use is MSVC

**DAWN\_COMPILER\_GNU** Set to 1 if the compiler in use is GNU

**DAWN\_COMPILER\_CLANG** Set to 1 if the compiler in use is Clang

## dawn\_add\_executable

Compile the given objects into a runnable executable (.exe).

```
dawn_add_executable(NAME SOURCES DEPENDS [OUTPUT_DIR])
```

**NAME** Name of the executable as well as the CMake target to build it.

**SOURCES** List of source files making up the executable.

**DEPENDS** List of external libraries and/or CMake targets to link against.

**OUTPUT\_DIR [optional]** Directory to place the executable (e.g. \${CMAKE\_BINARY\_DIR}/bin).

**INSTALL\_DESTINATION [optional]** Destination (relative to CMAKE\_INSTALL\_PREFIX) to install the executable.

## dawn\_add\_library

Compile the given sources into an archive or object library. This will provide the CMake targets <NAME>Objects, <NAME>Static and <NAME>Shared respectively.

```
dawn_add_library(NAME SOURCES DEPENDS [OUTPUT_DIR])
```

**NAME** Name of the executable as well as the CMake target to build it.

**SOURCES** List of source files making up the executable.

**DEPENDS** List of external libraries and/or CMake targets to link against.

**OBJECT [optional]** Create a CMake object library.

**ARCHIVE [optional]** Create a static and, if BUILD\_SHARED\_LIBS is ON, shared archive.

## dawn\_add\_target\_clang\_format

Provide a `format` target which runs `clang-format` recursively on all files in the provided directories.

```
dawn_add_target_clang_format(DIRECTORIES PATTERN)
```

**DIRECTORIES** Directories to recursively traverse to find all files with extensions matching `EXTENSION`.

**EXTENSION** Extension to match, separated by `;`. For example: `".h; .cpp"`.

## dawn\_add\_target\_clean\_all

Provide a `clean-all` target which clears the CMake cache and all related CMake files and directories. This effectively removes the following files/directories:

- \${CMAKE\_BINARY\_DIR}/CMakeCache.txt
- \${CMAKE\_BINARY\_DIR}/CTestTestfile.cmake
- \${CMAKE\_BINARY\_DIR}/cmake\_install.cmake
- \${CMAKE\_BINARY\_DIR}/CMakeFiles

```
dawn_add_target_clean_all([ARGN...])
```

**ARGN** Additional files or directories to remove.



## dawn\_add\_unittest

Compile the given objects into a runnable unittest executable (.exe) and register it within CTest. Note that to enable CTest you need to call the builtin command `enable_testing()` in the source root.

```
dawn_add_unittest(NAME SOURCES DEPENDS [OUTPUT_DIR GTEST_ARGS])
```

**NAME** Name of the unittest executable as well as the CMake target to build it.

**SOURCES** List of source files making up the executable.

**DEPENDS** List of external libraries and/or CMake targets to link against.

**OUTPUT\_DIR [optional]** Directory to place the executable (e.g. `${CMAKE_BINARY_DIR}/bin`).

**GTEST\_ARGS [optional]** Arguments passed to the created GTest executable (e.g. `--gtest_color=yes`)

## dawn\_check\_and\_set\_cxx\_flag

Test if the C++ compiler flag is supported and if so, add it to the `CMAKE_CXX_FLAGS`.

```
dawn_check_and_set_cxx_flag(FLAG NAME)
```

**FLAG** Compiler flag to check (e.g. `-O3`).

**NAME** Name of the check (e.g. `HAVE_GCC_O3`).

## dawn\_check\_cxx\_flag

Test if the C++ compiler flag is supported.

```
dawn_check_cxx_flag(FLAG NAME)
```

**FLAG** Compiler flag to check (e.g. `-O3`).

**NAME** Name of the check (e.g. `HAVE_GCC_O3`).

## dawn\_check\_in\_source\_build

Abort with a meaningful error message if `CMAKE_BINARY_DIR` matches `CMAKE_SOURCE_DIR`.

```
dawn_check_in_source_build()
```

## dawn\_combine\_libraries

Combine multiple object libraries to a single static and, if `BUILD_SHARED_LIBS` is ON, shared library. The CMake target of the library is `<NAME>Static` and `<NAME>Shared` respectively. This will also provide an install target for the libraries as well as an export via `<NAME>Targets`.

```
dawn_combine_libraries(NAME OBJECTS DEPENDS)
```

**NAME** Name of the library.

**OBJECTS** Object libraries to combine (see *dawn\_add\_library*).

**INSTALL\_DESTINATION** Destination (relative to `CMAKE_INSTALL_PREFIX`) to install the libraries.

**DEPENDS [optional]** List of external libraries and/or CMake targets treated as dependencies of the library.

## dawn\_configure\_file

Configure a file and store the output in the same directory hierarchy as the input file while substituting `CMAKE_SOURCE_DIR` for `CMAKE_BINARY_DIR`. If the input file has a `.cmake` extension, it will be removed in the output. This will also handle the installation of the file correctly by replacing `src/` with `include/`.

```
dawn_configure_file(FILE)
```

**FILE** File to configure.

### Example

```
dawn_configure_file(${CMAKE_SOURCE_DIR}/src/dawn/Config.h.cmake)
```

This stores the configured file in `${CMAKE_BINARY_DIR}/src/dawn/Config.h` and, in addition, it will install it in `${CMAKE_INSTALL_PREFIX}/include/dawn/Config.h` (hence it will strip "src/").

## dawn\_create\_package\_string

Create a package string including the location of the library/include directory and version string. Note that the package has to be exported via *dawn\_export\_package*.

```
dawn_create_package_string(NAME PACKAGE_STRING [NOT_FOUND_STRING])
```

**NAME** Name of the package.

**PACKAGE\_STRING** Returned package string.

**NOT\_FOUND\_STRING** String to print if the package was not found, defaults to "Not FOUND" (optional).

### Example

```
dawn_export_package(Foo FOUND TRUE LIBRARIES "/usr/lib/libfoo.so" VERSION "1.2.3")
dawn_create_package_string(Foo FooPackageStr)
message(${FooPackageStr})
```

## dawn\_export\_options

Export a list of options in the variable `<NAME>_OPTIONS` which can be accessed by meta projects.

```
dawn_export_options(NAME ARGN)
```

**NAME** Prefix of the options list.

**ARGN** List of option names to export.

## dawn\_export\_package

Export a package by defining variable for its libraries, include directories, definition and version.

```
dawn_export_package(NAME FOUND [LIBRARIES] [INCLUDE_DIRS]
                    [DEFINITIONS] [VERSION] [EXECUTABLE])
```

**NAME** Name of the package.

**FOUND** True if the package was found.

**LIBRARIES** List of libraries to link against (optional).

**INCLUDE\_DIRS** List of include directories required by this package (optional).

**DEFINITIONS** List of definitions required by the package (optional).

**EXECUTABLE** Executable of the package (optional).

**VERSION** Version of the package (optional).

The following variables are defined:

**DAWN\_<NAME>\_FOUND** True if package was found.

**DAWN\_<NAME>\_LIBRARIES** Libraries of the package to link against.

**DAWN\_<NAME>\_INCLUDE\_DIRS** Include directories required by this package.

**DAWN\_<NAME>\_DEFINITIONS** Definitions required by the package.

**DAWN\_<NAME>\_EXECUTABLE** Executable of the package.

**DAWN\_<NAME>\_VERSION** Version string of the package.

To create a formatted string of the exported packages *dawn\_create\_package\_string*.

## dawn\_find\_python\_module

Try to find a specific python module.

```
dawn_find_python_module(MODULE [REQUIRED])
```

**MODULE** Python module to find.

**REQUIRED** If set, abort with an error if MODULE was not found.

The module makes use of the PYTHON\_EXECUTABLE which needs to be set before calling this function (see [FindPythonInterp](#)).

Variables defined:

**PYTHON\_<module>\_FOUND** Python module is available where <module> is the python module to search for in all uppercase.

**PYTHON\_<module>\_PATH** Path to the module where <module> is the python module to search for in all uppercase.

## dawn\_get\_architecture\_info

Get the identification of the architecture.

```
dawn_get_architecture_info()
```

The function defines the following variable:

**DAWN\_ARCHITECTURE\_STRING** String of the architecture.

## dawn\_get\_compiler\_info

Get the identification string of the compiler.

```
dawn_get_compiler()
```

The function defines the following variable:

**DAWN\_COMPILER\_STRING** String of the currently used compiler.

and conditionally one of the following:

**DAWN\_COMPILER\_MSVC** Set to 1 if the compiler in use is MSVC

**DAWN\_COMPILER\_GNU** Set to 1 if the compiler in use is GNU

**DAWN\_COMPILER\_CLANG** Set to 1 if the compiler in use is Clang

### dawn\_get\_git\_head\_revision

These functions force a re-configure on each git commit so that you can trust the values of the variables in your build system. It returns the refspec and sha1 hash of the current head revision.

```
dawn_get_git_head_revision(HEAD_REF HEAD_HASH)
```

**HEAD\_REF** The Refspec.

**HEAD\_HASH** sha1 hash of the current head revision.

### dawn\_get\_platform\_info

Get the identification string of the platform.

```
dawn_get_platform_info()
```

The functions defines the following variable:

**DAWN\_PLATFORM\_STRING** String of the platform.

and conditionally the following:

**DAWN\_ON\_WIN32** Set to 1 if the platform is Win32-ish

**DAWN\_ON\_UNIX** Set to 1 if the platform is Unix-ish

**DAWN\_ON\_APPLE** Set to 1 if the platform is Darwin

**DAWN\_ON\_LINUX** Set to 1 if the platform is Linux

### dawn\_get\_script\_dir

Get the directory of the scripts located <dawn-root>/cmake/scripts.

```
dawn_get_script_dir (SCRIPT_DIR_VAR)
```

**SCRIPT\_DIR\_VAR** Variable which will contain the script directory on output.

---

**Note:** This function is for internal use only.

---

### dawn\_include\_guard

Prevent frequently-included CMake files from being re-parsed multiple times.

```
dawn_include_guard()
```

### dawn\_make\_string\_pair

Create a formatted string of the given string pair.

```
dawn_make_string_pair(FIRST SECOND NUM_CHARS OUT_STRING)
```

**FIRST** First string.

**SECOND** Second string.

**NUM\_CHARS** Characters before **SECOND** is printed.

**OUT\_STRING** Output string.

This is equivalent to the python snippet:

```
OUT_STRING = "%s%s%s" % (FIRST, " " * (NUM_CHARS - len(FIRST)), SECOND)
```

## Example

```
dawn_make_string_pair(Foo Bar1 10 out_var1)
dawn_make_string_pair(LongerFoo Bar2 10 out_var2)
message(${out_var1}) # "Foo      Bar1"
message(${out_var2}) # "LongerFoo Bar2"
```

## dawn\_report\_result

Report a list of strings, encompassed by a box.

```
dawn_report_result(HEADER [ARGN...])
```

**HEADER** Header to print

**ARGN** List of strings to report

## dawn\_set\_cxx\_standard

Set the minimum standard of C++.

```
dawn_set_cxx_standard(MIN_CXX_STANDARD)
```

**MIN\_CXX\_STANDARD** Minimum C++ standard which needs to be supported, one of [c++11, c++14, c++1z, c++17, c++2a].

## Modules

Load settings for an external project via `find_package`.

### FindClang

```
find_package(Clang [REQUIRED] [QUIET])
```

Find the `Clang` libraries and includes. Uses the same include and library paths detected by *FindLLVM*.

### Result Variables

Defines the following variables:

**CLANG\_FOUND** True if Clang was found.

**CLANG\_INCLUDE\_DIRS** Where to find Clang includes.

**CLANG\_LIBS** List of all Clang libraries.

**CLANG\_LIBCLANG\_LIB** C Interface to Clang.

**CLANG\_LIBRARY\_DIRS** Where to find Clang libraries.

**CLANG\_RESSOURCE\_INCLUDE\_PATH** Path to the internal Clang headers needed by Frontend tools.

**CLANG\_<COMPONENT>\_LIB** Path to the <COMPONENT> library.

## Hints

Set **LLVM\_ROOT** to a directory that contains a LLVM/Clang installation.

## FindLLVM

```
find_package(LLVM [REQUIRED] [QUIET])
```

Find the LLVM libraries and includes.

## Result Variables

Defines the following variables:

**LLVM\_FOUND** True if llvm found.

**LLVM\_INCLUDE\_DIRS** Where to find llvm include files

**LLVM\_LIBRARY\_DIRS** Where to find llvm libs

**LLVM\_CPPFLAGS** LLVM preprocessor flags

**LLVM\_LDFLAGS** LLVM linker flags.

**LLVM\_CXXFLAGS** LLVM c++ compiler flags

**LLVM\_INSTALL\_PREFIX** LLVM installation prefix

**LLVM\_VERSION** Version string (llvm-config --version)

## Hints

Set **LLVM\_ROOT** to a directory that contains a LLVM installation.

## FindSphinx

```
find_package(Sphinx [REQUIRED] [QUIET])
```

This module locates [Sphinx](#).

## Result Variables

Defines the following variables:

**SPHINX\_FOUND** System has sphinx.

**SPHINX\_EXECUTABLE** Path to **sphinx-build** executable.

**SPHINX\_VERSION** Version of sphinx.

## Hints

You can directly set `SPHINX_EXECUTABLE` if the module has trouble finding sphinx.

## Findbash

```
find_package(bash [REQUIRED] [QUIET])
```

This module locates `bash`. `BASH_FOUND` will report if bash has been found.

## Result Variables

Defines the following variables:

**BASH\_FOUND** System has bash.

**BASH\_EXECUTABLE** Path to the bash executable.

## Hints

You can directly set `BASH_EXECUTABLE` if the module has trouble finding bash.

## Findccache

```
find_package(ccache [REQUIRED] [QUIET])
```

This module locates `ccache`. `CCACHE_FOUND` will report if ccache has been found.

## Result Variables

Defines the following variables:

**CCACHE\_FOUND** System has ccache.

**CCACHE\_EXECUTABLE** Path to the ccache executable.

## Hints

You can directly set `CCACHE_EXECUTABLE` if the module has trouble finding ccache.

## Findclang-format

```
find_package(clang-format [REQUIRED] [QUIET])
```

This module locates `clang-format`. `CLANG-FORMAT_FOUND` will report if clang-format has been found.

## Result Variables

Defines the following variables:

**CLANG-FORMAT\_FOUND** System has clang-format.

**CLANG-FORMAT\_EXECUTABLE** Path to the clang-format executable.

## Hints

You can directly set `CLANG-FORMAT_EXECUTABLE` if the module has trouble finding clang-format.