XCommon CMake

Release: 1.3.0
Publication Date: 2024/07/31
Document Number: XM-015090-PC

Table of Contents

1 Introduction

XCommon CMake is a build system for xcore applications and libraries. It uses CMake to configure and generate a build environment, using `git` to fetch any missing dependencies, which can then be built with `xmake`.

The aim of XCommon CMake is to use a standard tool (CMake) to accelerate the development of xcore applications without requiring knowledge of the CMake language.

The minimum versions of required tools are listed in the *Software Requirements*.

1.1 Overview

An application executable is built from separate components: application sources, module sources and static libraries.

Each component sets some CMake variables to define its own properties and its dependency relationships with other components.

Then XCommon CMake utility functions provide a CMake implementation to create a build environment with the properties as defined in the components.

To support this functionality, a sandbox structure is assumed. Within this structure, XCommon CMake is able to fetch missing dependencies.

2 Quick Start Guide

2.1 Software Requirements

- CMake (minimum version 3.21)
- Git (minimum version 2.25)
- XTC Tools (minimum version 15.2.1)

2.2 Setup

Note: This step is only required with XMOS XTC Tools prior to 15.3, as XCommon CMake was not distributed with XTC Tools.

Before using XCommon CMake, the environment variable `XMOS_CMAKE_PATH` must be set to the location of the `xcommon_cmake` directory. For example:

```
# MacOS and Linux
export XMOS_CMAKE_PATH=/home/user/xcommon_cmake
```

```
# Windows
set XMOS_CMAKE_PATH=C:\Users\user\xcommon_cmake
```

2.3 Hello World Example

This example is a simple "Hello world" application to demonstrate a minimal project using XCommon CMake.

Create the following file structure, with the contents shown below:

```
app_hello_world/
                |-- CMakeLists.txt
                |-- src/
                       |-- main.c
```

app_hello_world/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(hello_world)

set(APP_HW_TARGET XCORE-AI-EXPLORER)

XMOS_REGISTER_APP()
```

app_hello_world/src/main.c

```
#include <stdio.h>

int main() {
```

(continues on next page)

(continued from previous page)

```
    printf("Hello world!\n");
    return 0;
}
```

Build the executable and run it using the simulator:

```
cmake -G "Unix Makefiles" -B build
cd build
xmake
cd ..
xsim bin/hello_world.xe
```

The message “Hello world!” is displayed.

Note: `xmake` is a build of GNU Make (https://www.gnu.org/software/make/) provided with the XMOS XTC tools for convenience.

Note: `xsim` provides a near cycle-accurate model of systems built from one of more xcore devices. It is supplied with the XMOS XTC tools.

3 Sandbox Structure

XCommon CMake assumes a sandbox structure. A sandbox is a fully contained collection of source code modules which can be used to build one or more applications.

3.1 Definitions

There are three types of top-level directories that XCommon CMake supports in a sandbox structure.

Application

Contains a set of source files that is specific to the application that builds into one or more executables (.xe file). This may represent a simple example or a verified Reference Design. These executables are made by compiling the application's source files and the source files of any modules it uses.

An application directory is prefixed with `sw_`, and each set of application-specific code within is in a directory prefixed with `app_`.

For example, an application may implement a USB Audio Device.

Module

Contains module source code, which can depend on other modules. Module code is compiled into objects which are directly linked into the application executable. XMOS module directories are typically prefixed with `lib_`.

For example, a source module may implement an IO function such as I2C.

Static Library

Contains a pre-built static library archive for each supported architecture. Also, it can optionally contain the source code for the static library to allow the archive to be rebuilt. Finally, it can optionally contain other additional source files which are compiled when the application is built. When an application that uses the static library is built, the static library archive is linked, and any additional source files are compiled and linked into the application. XMOS modules containing a static library are typically prefixed with `lib_`.

For example, a static library module may implement a large stack such as Tensorflow or protected third party IP.

3.2 Repository Layout

All the items defined in the *Definitions* section are placed in the root of the sandbox, each typically representing a separate git repository with no nesting of applications and modules. This allows for the possibility of merging applications and the use of shared dependencies. It also removes issues relating to dependency loops etc.

Each application is expected to include a `CMakeLists.txt` file. Each module is expected to contain a `lib_build_info.cmake` file. These files configure the use of XCommon CMake.

```
sandbox/
        |-- lib_mod0/
        |           |-- lib_mod0/
        |                       |-- api/
        |                       |-- src/
        |                       |-- lib_build_info.cmake
        |
```

(continues on next page)

(continued from previous page)

```
|-- lib_mod1/
|           |-- lib_mod1/
|                       |-- api/
|                       |-- src/
|                       |-- lib_build_info.cmake
|
|-- lib_mod2/
|           |-- lib_mod2/
|                       |-- api/
|                       |-- src/
|                       |-- lib_build_info.cmake
|
|-- sw_app0/
|          |-- app_app0_xcore200/
|          |                    |-- src/
|          |                    |-- CMakeLists.txt
|          |-- app_app0_xcoreai/
|                              |-- src/
|                              |-- CMakeLists.txt
|
|-- sw_app1/
|          |-- app_app1/
|                      |-- src/
|                      |-- CMakeLists.txt
|
```

In this example sandbox, `sw_app0` and `sw_app1` could be unrelated applications which share some common module dependencies. This layout would allow an engineer to develop and test multiple applications with a set of shared modules including some local modifications, without having to replicate those changes across multiple sandboxes.

4 Configuration Files

There are four types of CMake configuration file which can be used in an XCommon CMake project.

- Application `CMakeLists.txt`
- Module `lib_build_info.cmake`
- Static library `lib_build_info.cmake`
- Static library `CMakeLists.txt`

4.1 Application CMakeLists.txt

The application `CMakeLists.txt` file is located in the application directory as described in the *Sandbox Structure* section. This file typically has three sections.

The first section is a "header" which must be present to provide mandatory CMake function calls and to load the XCommon CMake function definitions. The three lines in *CMake Header* are required at the beginning of the file.

The second section of this file is usually the largest. It contains the variable definitions that will be used by the XCommon CMake functions to configure the application. There is a set of named variables, documented in *Variables*, which define the dependency relationships and the options for the build configuration.

There are two required variables: `APP_HW_TARGET` is necessary to define the target device, either by a named target defined in the XTC Tools or a local XN file; `XMOS_SANDBOX_DIR` must be set to the path of the root of the sandbox (if the application has no dependencies, this variable isn't strictly required). It is best practice to set this to a path relative to the CMake variable `${CMAKE_CURRENT_LIST_DIR}`, which is the directory containing this application `CMakeLists.txt` file.

The list of dependent modules provided in the `APP_DEPENDENT_MODULES` variable should only be the direct dependencies used in the application source code. Any sub-dependencies that are required will be defined within the modules that require them.

The final part is a call to `XMOS_REGISTER_APP()`. This function performs the necessary actions to populate the sandbox with dependencies and then generate the CMake build environment. All desired XCommon CMake application variables must be set before this function is called.

Example: *sandbox/sw_example/app_example/CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(app_example)

set(XMOS_SANDBOX_DIR ${CMAKE_CURRENT_LIST_DIR}/../..)
set(APP_HW_TARGET XCORE-AI-EXPLORER)
set(APP_COMPILER_FLAGS -O3 -Wall)
set(APP_DEPENDENT_MODULES "lib_foo")

XMOS_REGISTER_APP()
```

4.2 Module lib_build_info.cmake

The module code itself does not contain a `CMakeLists.txt` file because it is never the entry-point for XCommon CMake. Instead, the module directory contains the file `lib_build_info.cmake` which allows it to be included in other XCommon CMake projects.

The `lib_build_info.cmake` file contains variable definitions, followed by a call to `XMOS_REGISTER_MODULE()`. Some variable definitions are required; see *Required module variables*. All desired XCommon CMake module variables must be set before the call to `XMOS_REGISTER_MODULE()`.

In a similar way to the application variables, the `LIB_DEPENDENT_MODULES` variable should only contain the direct dependencies of the module. Any sub-dependencies that are required will be defined within the modules that require them.

Example: *sandbox/lib_foo/lib_foo/lib_build_info.cmake*

```
set(LIB_NAME lib_foo)
set(LIB_VERSION 3.2.1)
set(LIB_INCLUDES api)
set(LIB_COMPILER_FLAGS -Os)
set(LIB_DEPENDENT_MODULES "lib_bar")

XMOS_REGISTER_MODULE()
```

4.3 Static library lib_build_info.cmake

For a static library, a `lib_build_info.cmake` file is created to hold the XCommon CMake variable definitions to allow it to be linked into an application. See *Static Libraries* for the variable definitions. All desired XCommon CMake static library variables must be set before the call to `XMOS_REGISTER_STATIC_LIB()`.

Example: *sandbox/lib_bar/lib_bar/lib_build_info.cmake*

```
set(LIB_NAME lib_bar)
set(LIB_VERSION 1.0.0)
set(LIB_ARCHS xs2a xs3a)
set(LIB_ARCHIVE_INCLUDES api)
set(LIB_ARCHIVE_C_SRCS libsrc/bar0.c libsrc/bar1.c)
set(LIB_ARCHIVE_COMPILER_FLAGS -O3)
set(LIB_ARCHIVE_DEPENDENT_MODULES "")

XMOS_REGISTER_STATIC_LIB()
```

4.4 Static library CMakeLists.txt

If the static library repository also contains the source to build it, then a `CMakeLists.txt` file can be created to configure this build. It contains the same initial three lines as the application `CMakeLists.txt` file, with the library name set in the `project()` call, and then it sets the `XMOS_SANDBOX_DIR` variable and includes the `lib_build_info.cmake` described in the previous section. This allows the XCommon CMake variables for the library to be shared between the two workflows: building the static library archive and linking an existing archive into an application.

Example: *sandbox/lib_bar/lib_bar/CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(lib_bar)
```

(continues on next page)

(continued from previous page)

```
set(XMOS_SANDBOX_DIR ${CMAKE_CURRENT_LIST_DIR}/../..)
include(lib_build_info.cmake)
```

5 Dependency Management

XCommon CMake provides a dependency management solution which can fetch modules to be used in an application. These modules are cloned from git repositories and placed in the root of the sandbox.

Starting from the application's CMakeLists.txt, the `APP_DEPENDENT_MODULES` variable defines the immediate dependencies of the application. When each `lib_build_info.cmake` file is included for each dependency, their `LIB_DEPENDENT_MODULES` variables define the sub-dependency relationships. This builds up a tree which is traversed depth-first to populate the sandbox.

As an example, suppose that an application's `CMakeLists.txt` contains `set(APP_DEPENDENT_MODULES lib_mod0 lib_mod1)` and then the modules have the following in their `lib_build_info.cmake` files:

lib_mod0	`set(LIB_DEPENDENT_MODULES "lib_mod2")`
lib_mod1	`set(LIB_DEPENDENT_MODULES "lib_mod2" "lib_mod3")`
lib_mod2	`set(LIB_DEPENDENT_MODULES "lib_mod3")`
lib_mod3	`set(LIB_DEPENDENT_MODULES "")`

Then the dependent modules will be retrieved in the following order: `lib_mod0`, `lib_mod2`, `lib_mod3`, `lib_mod1`.

If a dependency is not present in the sandbox, it will be retrieved the first time it is traversed in this tree. If it then appears again as a dependency of another module, nothing will happen because it is already present in the sandbox.

The `APP_DEPENDENT_MODULES` and `LIB_DEPENDENT_MODULES` variables define lists of dependencies, where each element in the list is a string that specifies where to fetch the source code from and which version to fetch. See the *Dependency Format* specification for full details of the accepted format.

Retrieval happens by cloning from a git repo, which can be accessed via HTTPS (for public repositories) or via an SSH key if one is configured for passwordless access.

All the dependencies in the tree will be retrieved into the sandbox, if they are not already present. The location of the root of the sandbox must be specified by the application using the `XMOS_SANDBOX_DIR` variable. If an application or static library has no dependencies, this variable doesn't need to be set.

During the process of dependency resolution, if a module is already present in the sandbox it will not be modified. If a different version of a module is subsequently required, the procedure is to use standard `git` commands to change to the desired version and then run `cmake build` in the application directory.

5.1 Sandbox Manifest

It is often useful to record the actual version that was used for each module, especially when tracking a branch.

Whenever CMake generates the build environment for an application or static library, a file called `manifest.txt` is created in the `build` directory.

The columns in the manifest file are:

- Name: the name of the application or library
- Location: the git remote from which the repository was cloned
- Branch/tag: the currently checked out branch or tag. A tag takes precedence, so if the head of a branch is explicitly checked out, but that changeset has also been tagged, then the tag will be reported here.
- Changeset: the git commit hash identifying the current changeset checked out in the repository

- Depends_on: (hidden) a list of the libraries the dependency is dependent on. This column is only displayed if CMake is run with the option `-D FULL_MANIFEST=TRUE`.

If any columns are not applicable to a particular dependency, they will contain a hyphen.

6 Examples

This section contains examples that demonstrate some of the features of XCommon CMake. Some examples focus on the contents of the `CMakeLists.txt` files to show how the API functions and variables can be used, without presenting the full application/module source code. Others are complete "mini-applications" with example source code.

6.1 Application Configs

Application `app_cfgs` has two build configs, which in this trivial case change the value of a printed message. When using multiple configs in your application, the same source files are compiled for each config, but different compiler flags can be supplied to each config.

6.1.1 Directory structure

```
sandbox/
       |-- sw_cfgs/
                  |-- app_cfgs/
                              |-- CMakeLists.txt
                              |-- src/
                                     |-- main.c
```

6.1.2 CMake and source file contents

sandbox/sw_cfgs/app_cfgs/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(cfgs)

set(APP_HW_TARGET XCORE-AI-EXPLORER)
set(APP_COMPILER_FLAGS_config0 -DMSG_NUM=0)
set(APP_COMPILER_FLAGS_config1 -DMSG_NUM=1)

XMOS_REGISTER_APP()
```

sandbox/sw_cfgs/app_cfgs/src/main.c

```
#include <stdio.h>

int main() {
    printf("config%d\n", MSG_NUM);
    return 0;
}
```

6.1.3 Build instructions

Commands to build and run app, from working directory `sandbox/sw_cfgs/app_cfgs`:

```
cmake -G "Unix Makefiles" -B build
cd build
xmake
```

The build products are:

- `bin/config0/cfgs_config0.xe`
- `bin/config1/cfgs_config1.xe`

These binaries can be run with xsim to see the difference in their printed output.

```
$> xsim bin/config0/cfgs_config0.xe
config0

$> xsim bin/config1/cfgs_config1.xe
config1
```

An individual executable target can be built, so to build only `cfgs_config1.xe` and not `cfgs_config0.xe`:

```
cd build
xmake config1
```

6.2 Module Dependencies

Application `app_moddeps` requires modules `lib_mod0` and `lib_mod1`, and `lib_mod1` requires `lib_mod2`.

6.2.1 Directory structure

```
sandbox/
        |-- lib_mod0/
        |           |-- lib_mod0/
        |                       |-- api/
        |                       |-- lib_build_info.cmake
        |                       |-- src/
        |-- lib_mod1/
        |           |-- lib_mod1/
        |                       |-- api/
        |                       |-- lib_build_info.cmake
        |                       |-- src/
        |-- lib_mod2/
        |           |-- lib_mod2/
        |                       |-- api/
        |                       |-- lib_build_info.cmake
        |                       |-- src/
        |-- sw_moddeps/
                      |-- app_moddeps/
                                     |-- CMakeLists.txt
                                     |-- src/
```

6.2.2 CMake file contents

sandbox/sw_moddeps/app_moddeps/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(moddeps)

set(APP_HW_TARGET XCORE-AI-EXPLORER)
set(APP_DEPENDENT_MODULES "lib_mod0(3.2.0)"
                          "lib_mod1(1.0.0)")
set(XMOS_SANDBOX_DIR ${CMAKE_CURRENT_LIST_DIR}/../..)

XMOS_REGISTER_APP()
```

sandbox/lib_mod0/lib_mod0/lib_build_info.cmake

```
set(LIB_NAME lib_mod0)
set(LIB_VERSION 3.2.0)
set(LIB_INCLUDES api)
set(LIB_DEPENDENT_MODULES "")

XMOS_REGISTER_MODULE()
```

sandbox/lib_mod1/lib_mod1/lib_build_info.cmake

```
set(LIB_NAME lib_mod1)
set(LIB_VERSION 1.0.0)
set(LIB_INCLUDES api)
set(LIB_DEPENDENT_MODULES "lib_mod2(2.5.1)")

XMOS_REGISTER_MODULE()
```

sandbox/lib_mod2/lib_mod2/lib_build_info.cmake

```
set(LIB_NAME lib_mod2)
set(LIB_VERSION 2.5.1)
set(LIB_INCLUDES api)
set(LIB_DEPENDENT_MODULES "")

XMOS_REGISTER_MODULE()
```

6.2.3 Build instructions

Commands to build and run app, from working directory `sandbox/sw_moddeps/app_moddeps`:

```
cmake -G "Unix Makefiles" -B build
cd build
xmake
```

The build product is `bin/moddeps.xe`.

6.3 Multi-Application Repository

Repository `sw_multiapp` contains two applications, which have a shared dependency and a unique dependency each. Application `app_multiapp0` requires modules `lib_mod0` and `lib_mod1`; application `app_multiapp1` requires modules `lib_mod0` and `lib_mod2`.

6.3.1 Directory structure

```
sandbox/
       |-- lib_mod0/
       |           |-- lib_mod0/
       |                       |-- api/
       |                       |-- lib_build_info.cmake
       |                       |-- src/
       |-- lib_mod1/
       |           |-- lib_mod1/
       |                       |-- api/
       |                       |-- lib_build_info.cmake
       |                       |-- src/
       |-- lib_mod2/
       |           |-- lib_mod2/
       |                       |-- api/
       |                       |-- lib_build_info.cmake
       |                       |-- src/
       |-- sw_multiapp/
                      |-- app_multiapp0/
                      |                |-- CMakeLists.txt
                      |                |-- src/
                      |-- app_multiapp1/
                      |                |-- CMakeLists.txt
                      |                |-- src/
                      |-- CMakeLists.txt
                      |-- deps.cmake
```

6.3.2 CMake file contents

sandbox/sw_multiapp/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(sw_multiapp)

add_subdirectory(app_multiapp0)
add_subdirectory(app_multiapp1)
```

sandbox/sw_multiapp/deps.cmake

```
set(APP_DEPENDENT_MODULES "lib_mod0"
                          "lib_mod1"
                          "lib_mod2")
```

sandbox/sw_multiapp/app_multiapp0/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(multiapp0)
```

(continues on next page)

(continued from previous page)

```
set(APP_HW_TARGET XCORE-AI-EXPLORER)
include(${CMAKE_CURRENT_LIST_DIR}/../deps.cmake)
set(XMOS_SANDBOX_DIR ${CMAKE_CURRENT_LIST_DIR}/../..)

XMOS_REGISTER_APP()
```

sandbox/sw_multiapp/app_multiapp0/src/main.c

```
#include "mod0.h"
#include "mod1.h"

int main() {
    mod0();
    mod1();
    return 0;
}
```

sandbox/sw_multiapp/app_multiapp1/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(multiapp1)

set(APP_HW_TARGET XCORE-AI-EXPLORER)
include(${CMAKE_CURRENT_LIST_DIR}/../deps.cmake)
set(XMOS_SANDBOX_DIR ${CMAKE_CURRENT_LIST_DIR}/../..)

XMOS_REGISTER_APP()
```

sandbox/sw_multiapp/app_multiapp1/src/main.c

```
#include "mod0.h"
#include "mod2.h"

int main() {
    mod0();
    mod2();
    return 0;
}
```

sandbox/lib_mod0/lib_mod0/lib_build_info.cmake

```
set(LIB_NAME lib_mod0)
set(LIB_VERSION 1.0.0)
set(LIB_INCLUDES api)
set(LIB_DEPENDENT_MODULES "")

XMOS_REGISTER_MODULE()
```

sandbox/lib_mod1/lib_mod1/lib_build_info.cmake

```
set(LIB_NAME lib_mod1)
set(LIB_VERSION 1.0.0)
set(LIB_INCLUDES api)
set(LIB_DEPENDENT_MODULES "")

XMOS_REGISTER_MODULE()
```

sandbox/lib_mod2/lib_mod2/lib_build_info.cmake

```
set(LIB_NAME lib_mod2)
set(LIB_VERSION 1.0.0)
set(LIB_INCLUDES api)
set(LIB_DEPENDENT_MODULES "")

XMOS_REGISTER_MODULE()
```

6.3.3 Build instructions

Commands to build and run both applications, from working directory `sandbox/sw_multiapp`:

```
cmake -G "Unix Makefiles" -B build
cd build
xmake
```

The build products are `app_multiapp0/bin/multiapp0.xe` and `app_multiapp1/bin/multiapp1.xe`.

Alternatively, a single application can be configured and built. From working directory `sandbox/sw_multiapp/app_multiapp1`:

```
cmake -G "Unix Makefiles" -B build
cd build
xmake
```

The build product is `bin/multiapp1.xe`. Application `app_multiapp0` has not been built.

6.3.4 Dependency best practice

For a repository which contains multiple applications, each with different dependencies, if each has its own definition of the `APP_DEPENDENT_MODULES` variable, trying to keep the common dependencies synchronised is error-prone.

In a multi-application repository, CMake can configure and generate the build environment at different levels: either for a single application from within that application's subdirectory, or for all applications from the `CMakeLists.txt` file in the root of the repository. For simplicity, it is preferable for the manifest to show a common view of the whole sandbox, rather than only reporting the dependencies in the sandbox which are used by a single application.

Therefore, it is strongly recommended to set the `APP_DEPENDENT_MODULES` variable with the full list of dependencies for all applications in the repository in the common `deps.cmake` file, as in the example above. Individual applications should not modify the `APP_DEPENDENT_MODULES` variable in their own `CMakeLists.txt` files, otherwise the generated manifest file may be incorrect.

6.4 Compiler Flags

Options to the compiler can be set for all sources in an application or module, and also independent sets of compiler options can be specified for build configs and individual source files.

This example demonstrates the hierarchy of how these options interact. The `MSG_NUM` macro is defined for a config, so it applies to all sources. Then the `FLAG0` and `FLAG1` macros are defined for specific files, so they are undefined in the other sources (and successful compilation of this example proves this as the `#error` directives are not reached.

6.4.1 Directory structure

```
sandbox/
       |-- sw_cflags/
                    |-- app_cflags/
                                  |-- CMakeLists.txt
                                  |-- src/
                                         |-- flag0.c
                                         |-- flag1.c
                                         |-- main.c
```

6.4.2 CMake and source file contents

sandbox/sw_cflags/app_cflags/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(cflags)

set(APP_HW_TARGET XCORE-AI-EXPLORER)

set(APP_COMPILER_FLAGS_config0 -DMSG_NUM=0)
set(APP_COMPILER_FLAGS_config1 -DMSG_NUM=1)

set(APP_COMPILER_FLAGS_flag0.c -DFLAG0=0)
set(APP_COMPILER_FLAGS_flag1.c -DFLAG1=1)

XMOS_REGISTER_APP()
```

sandbox/sw_cflags/app_cflags/src/flag0.c

```
#include <stdio.h>

#ifndef FLAG0
#error
#endif

#ifdef FLAG1
#error
#endif

void flag0() {
    printf("%d:%d\n", MSG_NUM, FLAG0);
}
```

sandbox/sw_cflags/app_cflags/src/flag1.c

```
#include <stdio.h>

#ifdef FLAG0
#error
#endif

#ifndef FLAG1
#error
#endif

void flag1() {
    printf("%d:%d\n", MSG_NUM, FLAG1);
}
```

sandbox/sw_cflags/app_cflags/src/main.c

```
#include <stdio.h>

#ifdef FLAG0
#error
#endif

#ifdef FLAG1
#error
#endif

void flag0();
void flag1();

int main() {
    printf("config%d\n", MSG_NUM);
    flag0();
    flag1();
    return 0;
}
```

6.4.3 Build instructions

Commands to build and run app, from working directory `sandbox/sw_cflags/app_cflags`:

```
cmake -G "Unix Makefiles" -B build
cd build
xmake
```

The build products are:

- `bin/config0/cflags_config0.xe`
- `bin/config1/cflags_config1.xe`

These binaries can be run with xsim to see the difference in their printed output.

```
$> xsim bin/config0/cflags_config0.xe
config0
0:0
0:1

$> xsim bin/config1/cflags_config1.xe
config1
1:0
1:1
```

6.5 Optional Headers

Application `app_opthdr` requires module `lib_mod0`, and `lib_mod0` supports an optional header file.

Optional headers are a module feature which allows module code to be conditionally compiled based on the presence of a header file in the application or another module. One use for this feature is to allow an application to override the definitions of constants in the module.

6.5.1 Directory structure

```
sandbox/
       |-- lib_mod0/
       |           |-- lib_mod0/
       |                       |-- api/
       |                       |-- lib_build_info.cmake
       |                       |-- src/
       |-- sw_opthdr/
                    |-- app_opthdr/
                                 |-- CMakeLists.txt
                                 |-- src/
                                        |-- mod0_conf.h
                                        |-- main.c
```

6.5.2 CMake file contents

sandbox/sw_opthdr/app_opthdr/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(opthdr)

set(APP_HW_TARGET XCORE-AI-EXPLORER)
set(APP_DEPENDENT_MODULES "lib_mod0")

XMOS_REGISTER_APP()
```

sandbox/lib_mod0/lib_mod0/lib_build_info.cmake

```
set(LIB_NAME lib_mod0)
set(LIB_VERSION 1.0.0)
set(LIB_INCLUDES api)
set(LIB_OPTIONAL_HEADERS mod0_conf.h)
set(LIB_DEPENDENT_MODULES "")

XMOS_REGISTER_MODULE()
```

Files in `lib_mod0` (source or headers) can now conditionally compile code using preprocessor directives like

```
#ifdef __mod0_conf_h_exists__
```

For example, `mod0_conf.h` could be conditionally included into files in `lib_mod0`, so that the application can define or override constants in module `lib_mod0`.

6.5.3 Build instructions

Commands to configure and build the app, from working directory `sandbox/sw_opthdr/app_opthdr`:

```
cmake -G "Unix Makefiles" -B build
cd build
xmake
```

6.6 Static Library

Library `lib_abc` will be compiled as a static library to link into applications, rather than be used as a module. It has one dependency, `lib_mod0`.

6.6.1 Directory structure

```
sandbox/
       |-- lib_mod0/
       |           |-- lib_mod0/
       |                       |-- api/
       |                       |      |-- mod0.h
       |                       |-- lib_build_info.cmake
       |                       |-- src/
       |                              |-- mod0.c
       |
       |-- lib_abc/
                  |-- lib_abc/
                             |-- api/
                             |      |-- abc.h
                             |-- CMakeLists.txt
                             |-- lib_build_info.cmake
                             |-- libsrc/
                                       |-- abc.c
```

6.6.2 CMake file contents

sandbox/lib_abc/lib_abc/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(lib_abc)

set(XMOS_SANDBOX_DIR ${CMAKE_CURRENT_LIST_DIR}/../..)
include(lib_build_info.cmake)
```

sandbox/lib_abc/lib_abc/lib_build_info.cmake

```
set(LIB_NAME lib_abc)
set(LIB_VERSION 1.2.3)

# Define debug and release versions of the archive, with different compiler flags
set(LIB_ARCHIVES archive_dbg archive_rel)
set(LIB_ARCHIVE_ARCHS xs2a xs3a)
set(LIB_ARCHIVE_COMPILER_FLAGS_archive_dbg -g -O0)
set(LIB_ARCHIVE_COMPILER_FLAGS_archive_rel -O3)

set(LIB_ARCHIVE_INCLUDES api)
set(LIB_ARCHIVE_DEPENDENT_MODULES "lib_mod0(1.0.0)")

XMOS_REGISTER_STATIC_LIB()
```

sandbox/lib_mod0/lib_mod0/lib_build_info.cmake

```
set(LIB_NAME lib_mod0)
set(LIB_VERSION 1.0.0)
```

(continues on next page)

(continued from previous page)

```
set(LIB_INCLUDES api)
set(LIB_DEPENDENT_MODULES "")

XMOS_REGISTER_MODULE()
```

6.6.3 Build instructions

Commands to build the static libraries, from working directory `sandbox/lib_abc/lib_abc`:

```
cmake -G "Unix Makefiles" -B build
cd build
xmake
```

A static library archive is created for each architecture:

- `sandbox/lib_abc/lib_abc/lib/xs2a/lib_abc.a`
- `sandbox/lib_abc/lib_abc/lib/xs3a/lib_abc.a`

7 API Reference

7.1 Functions

7.1.1 CMake Header

Some CMake function calls are required in the application or static library `CMakeLists.txt` file.

`cmake_minimum_required`
This is used to set the minimum version of CMake based on the language features used. Your version of CMake must not be lower than the version set in this function call. An appropriate value is the minimum version of CMake supported by XCommon CMake, as reported in the Quick Start Guide.

`include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)`
This is the inclusion of the xcore toolchain and the functions provided by XCommon CMake. The environment variable XMOS_CMAKE_PATH will be set by enabling the XTC Tools environment.

`project`
This function takes an argument which will be used as the base name for the application. If `my_app` is set here, the XE executable for the default config build will be called `my_app.xe`.

These three lines must be present at the beginning of an application or static library `CMakeLists.txt` file.

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(my_app)

# Now ready for the XCommon CMake code for the application or static library
```

7.1.2 XCommon CMake Functions

`XMOS_REGISTER_APP()`
This function is called after setting the *Required application variables* and any *Optional application variables* inside an application, to perform the following:

- define the application build targets
- add application sources to the executable build targets
- set compiler options for each build config
- populate the manifest with an entry for the application
- fetch any missing dependencies
- configure the immediate dependencies
- check presence of optional headers
- create commands for PCA, if enabled

Note: Pre-compilation Analysis (PCA) provides whole program optimisation but is only applicable to XC source files.

XMOS_REGISTER_MODULE()

This function is called after setting the *Required module variables* and any *Optional application variables*, to perform the following:

- check the major version number of the module for compatibility
- fetch any missing dependencies of the module
- set compiler options for module source files
- add module sources to the executable build targets
- populate the manifest with an entry for the module

This function is called recursively when adding module dependencies which use this function.

XMOS_REGISTER_STATIC_LIB()

This function is called after setting the *Static Libraries* and it can be used in two ways.

Firstly, if CMake is being run from the static library directory, this function will:

- define the static library build targets
- add static library sources to the build targets
- set compiler options for the static library sources
- populate the manifest with an entry for the static library
- fetch any missing dependencies
- configure the immediate dependencies

Alternatively, if the static library is a dependency of an application, this function is called as a result of the dependency configuration for the application. In that case, it will link the static library into all of the application build targets.

7.2 Variables

XCommon CMake relies on named variables which can be set for application and library code. These variables must be set before calling the *XCommon CMake Functions*. The order in which the variables are set does not matter.

7.2.1 Applications

7.2.1.1 Required application variables

APP_HW_TARGET

The target name or filename of an XN file to define the target platform. If a filename is provided, the full path is not required; the child directories of the application directory will be searched and the first file matching this name is used. Examples:

```
set(APP_HW_TARGET XCORE-AI-EXPLORER)
set(APP_HW_TARGET xk-316-mc.xn)
```

Advanced: this variable is not required if exclusively performing *Native CPU Builds*.

XMOS_SANDBOX_DIR

The path to the root of the sandbox directory. This is only required if `APP_DEPENDENT_MODULES` is non-empty. See *Sandbox Structure*.

```
set(XMOS_SANDBOX_DIR ${CMAKE_CURRENT_LIST_DIR}/../..)
```

7.2.1.2 Optional application variables

APP_ASM_SRCS
List of assembly source files to compile. File paths are relative to the application directory. If not set, all `*.S` files in the `src` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any assembly sources. Examples:

```
set(APP_ASM_SRCS src/feature0/f0.S src/feature1/f1.S)
set(APP_ASM_SRCS "")
```

APP_C_SRCS
List of C source files to compile. File paths are relative to the application directory. If not set, all `*.c` files in the `src` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any C sources. Examples:

```
set(APP_C_SRCS src/feature0/f0.c src/feature1/f1.c)
set(APP_C_SRCS "")
```

APP_COMPILER_FLAGS
List of options to the compiler for use when compiling all source files, except those which have their own options via the `APP_COMPILER_FLAGS_<filename>` variable. This variable should also be used for compiler definitions via the `-D` option. Default: empty list which provides no compiler options. Example:

```
set(APP_COMPILER_FLAGS -g -O3 -Wall -DMY_DEF=123)
```

APP_COMPILER_FLAGS_<config>
List of options to the compiler for use when compiling all source files for the specified config, except those which have their own options via the `APP_COMPILER_FLAGS_<filename>` variable. This variable should also be used for compiler definitions via the `-D` option. Default: empty list which provides no compiler options. Example:

```
set(APP_COMPILER_FLAGS_config0 -g -O2 -DMY_DEF=456)
```

APP_COMPILER_FLAGS_<filename>
List of options to the compiler for use when compiling the specified file. Only the filename is required, not a full path to the file; these compiler options will be used when compiling all files in the application directory which have that filename. This variable should also be used for compiler definitions via the `-D` option. Default: empty list which provides no compiler options. Example:

```
set(APP_COMPILER_FLAGS_feature0.c -Os -DMY_DEF=789)
```

APP_CXX_SRCS
List of C++ source files to compile. File paths are relative to the application directory. If not set, all `*.cpp` files in the `src` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any C++ sources. Examples:

```
set(APP_CXX_SRCS src/feature0/f0.cpp src/feature1/f1.cpp)
set(APP_CXX_SRCS "")
```

APP_DEPENDENT_ARCHIVES
List of static library archives to link into this application. The static library dependency must also be defined in the `APP_DEPENDENT_MODULES` variable. The items in this list can either be the name of the static library (if that static library only contains a single archive) or the name of an archive within a static library repository. Default: empty list, so the application does not attempt to link any static library archives. Examples:

```
set(APP_DEPENDENT_ARCHIVES lib_static0)
set(APP_DEPENDENT_ARCHIVES static0_archive1 static1_archive0)
```

APP_DEPENDENT_ARCHIVES_<config>

List of static library archives to link when building the specified application config. The static library dependency must also be defined in the APP_DEPENDENT_MODULES variable. The items in this list can either be the name of the static library (if that static library only contains a single archive) or the name of an archive within a static library repository. Default: empty list, so the application does not attempt to link any static library archives. Examples:

```
set(APP_DEPENDENT_ARCHIVES_config0 lib_static0)
```

APP_DEPENDENT_MODULES

List of this application's dependencies, which must be present when compiling. See the separate dependency management section about the dependency fetching process and the acceptable format for values in this list. Unlike other variables, the values to set for APP_DEPENDENT_MODULES should be quoted, as this is required when the string contains parentheses. Default: empty list, so the application has no dependencies. Example:

```
set(APP_DEPENDENT_MODULES "lib_i2c(6.1.1)"
                          "lib_i2s(5.0.0)")
```

APP_INCLUDES

List of directories to add to the compiler's include search path when compiling sources. Default: empty list, so no directories are added. Example:

```
set(APP_INCLUDES src src/feature0)
```

APP_PCA_ENABLE

Boolean option to enable Pre-Compilation Analysis for XC source files. Default: OFF. Example:

```
set(APP_PCA_ENABLE ON)
```

APP_XC_SRCS

List of XC source files to compile. File paths are relative to the application directory. If not set, all *.xc files in the src directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any XC sources. Examples:

```
set(APP_XC_SRCS src/feature0/f0.xc src/feature1/f1.xc)
set(APP_XC_SRCS "")
```

APP_XSCOPE_SRCS

List of xscope configuration files to use in the application. File paths are relative to the application directory. If not set, all *.xscope files in the src directory and its subdirectories will be used. An empty string can be set to avoid using any xscope configuration files. Examples:

```
set(APP_XSCOPE_SRCS src/config.xscope)
set(APP_XSCOPE_SRCS "")
```

SOURCE_FILES_<config>

List of source files to use only when building the specified application config. Each application config initially has the same source file list, which is created according to the behaviour of the language-specific source list variables. Then for each application config, sources are removed from their list if a different application config has specified that file in its SOURCE_FILES_<config> variable.

```
set(SOURCE_FILES_config0 src/config0.c)
```

XMOS_DEP_DIR_<module>

Directory containing the dependency <module> as an override to the default sandbox root directory in XMOS_SANDBOX_DIR. This is the path to the root of the module.

```
set(XMOS_DEP_DIR_lib_i2c /home/user/lib_i2c)
```

7.2.2 Modules

7.2.2.1 Required module variables

LIB_DEPENDENT_MODULES
List of this module's dependencies, which must be present when compiling. See the separate dependency management section about the dependency fetching process and the acceptable format for values in this list. If this module has no dependencies, this variable must be set as an empty string. Unlike other variables, the values to set for `LIB_DEPENDENT_MODULES` should be quoted, as this is required when the string contains parentheses. Examples:

```
set(LIB_DEPENDENT_MODULES "lib_logging(3.1.1)"
                          "lib_xassert(4.1.0)")
set(LIB_DEPENDENT_MODULES "")
```

LIB_INCLUDES
List of directories to add to the compiler's include search path when compiling sources. Example:

```
set(LIB_INCLUDES api src/feature0)
```

LIB_NAME
String of the name for this module. This string will be the name used by the dependent modules list variables for any applications/modules that require this module. Example:

```
set(LIB_NAME lib_logging)
```

LIB_VERSION
String of the three-part version number for this module. Example:

```
set(LIB_VERSION 3.1.1)
```

7.2.2.2 Optional module variables

LIB_ASM_SRCS
List of assembly source files to compile. File paths are relative to the module directory. If not set, all `*.S` files in the `src` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any assembly sources. Examples:

```
set(LIB_ASM_SRCS src/feature0/f0.S src/feature1/f1.S)
set(LIB_ASM_SRCS "")
```

LIB_C_SRCS
List of C source files to compile. File paths are relative to the module directory. If not set, all `*.c` files in the `src` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any C sources. Examples:

```
set(LIB_C_SRCS src/feature0/f0.c src/feature1/f1.c)
set(LIB_C_SRCS "")
```

LIB_COMPILER_FLAGS
List of options to the compiler for use when compiling all source files, except those which have their own options via the `LIB_COMPILER_FLAGS_<filename>` variable. This variable should also be used for compiler definitions via the `-D` option. Default: empty list which provides no compiler options. Example:

```
set(LIB_COMPILER_FLAGS -g -O3 -Wall -DMY_DEF=123)
```

LIB_COMPILER_FLAGS_<filename>

List of options to the compiler for use when compiling the specified file. Only the filename is required, not a full path to the file; these compiler options will be used when compiling all files in the module directory which have that filename. This variable should also be used for compiler definitions via the `-D` option. Default: empty list which provides no compiler options. Example:

```
set(APP_COMPILER_FLAGS_feature0.c -Os -DMY_DEF=456)
```

LIB_CXX_SRCS

List of C++ source files to compile. File paths are relative to the module directory. If not set, all `*.cpp` files in the `src` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any C++ sources. Examples:

```
set(LIB_CXX_SRCS src/feature0/f0.cpp src/feature1/f1.cpp)
set(LIB_CXX_SRCS "")
```

LIB_OPTIONAL_HEADERS

List of header files that can optionally be present in an application or module which requires this module. These files are not present in this module. If they are present in an application or module, the preprocessor macro `__<name>_h_exists__` will be set. Files within this module can then contain code which is conditionally compiled based on the presence of these optional headers in other applications. Every module or static library has an automatic optional header; for a library named `lib_foo`, the optional header `foo_conf.h` will automatically be configured, so it doesn't need to be set in this variable. Default: empty list which provides no optional headers. Example:

```
set(LIB_OPTIONAL_HEADERS abc_conf.h)
```

LIB_XC_SRCS

List of XC source files to compile. File paths are relative to the module directory. If not set, all `*.xc` files in the `src` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any XC sources. Examples:

```
set(LIB_XC_SRCS src/feature0/f0.xc src/feature1/f1.xc)
set(LIB_XC_SRCS "")
```

LIB_XSCOPE_SRCS

List of xscope configuration files to use for this module. File paths are relative to the module directory. If not set, all `*.xscope` files in the `src` directory and its subdirectories will be used. An empty string can be set to avoid using any xscope configuration files for this module. Examples:

```
set(LIB_XSCOPE_SRCS src/config.xscope)
set(LIB_XSCOPE_SRCS "")
```

7.2.3 Static Libraries

Static library repositories have two possible modes of use: building the static library archive from source, and linking the static library (with any additional sources) into an application. Most static library variables in the XCommon CMake API are used in just one of these two modes of use.

7.2.3.1 Variables for archive build

LIB_ARCHIVES

List of archives to build. If not set, a single archive (per supported architecture) is built and is named by the `LIB_NAME` variable. Example:

```
set(LIB_ARCHIVES archive0 archive1)
```

LIB_ARCHIVE_ARCHS

List of architectures for which to build archives. If not set, and `LIB_ARCHIVE_ARCHS_<archive>` is also unset, then the default archive build architecture is `xs3a`. Example:

```
set(LIB_ARCHIVE_ARCHS xs2a xs3a)
```

LIB_ARCHIVE_ARCHS_<archive>

List of architectures for which to build the named archive. If this is not set, the named archive will be built for the architectures in `LIB_ARCHIVE_ARCHS`, and if that is unset, the default is `xs3a`. Example:

```
set(LIB_ARCHIVE_ARCHS_archive0 xs2a xs3a)
```

LIB_ARCHIVE_ASM_SRCS

List of assembly source files to compile to create the archive. File paths are relative to the static library directory. If not set, all `*.S` files in the `libsrc` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any assembly sources. Examples:

```
set(LIB_ARCHIVE_ASM_SRCS libsrc/feature0/f0.S libsrc/feature1/f1.S)
set(LIB_ARCHIVE_ASM_SRCS "")
```

LIB_ARCHIVE_COMPILER_FLAGS

List of options to the compiler for use when compiling all source files. This variable should also be used for compiler definitions via the `-D` option. Default: empty list which provides no compiler options. Example:

```
set(LIB_ARCHIVE_COMPILER_FLAGS -O3 -Wall -DMY_DEF=123)
```

LIB_ARCHIVE_COMPILER_FLAGS_<archive_name>

List of options to the compiler for use when compiling all source files when building the named archive. This variable should also be used for compiler definitions via the `-D` option. Default: empty list which provides no compiler options. Example:

```
set(LIB_ARCHIVE_COMPILER_FLAGS_archive0 -O3 -Wall -DMY_DEF=123)
```

LIB_ARCHIVE_CXX_SRCS

List of C++ source files to compile to create the archive. File paths are relative to the static library directory. If not set, all `*.cpp` files in the `libsrc` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any C++ sources. Examples:

```
set(LIB_ARCHIVE_CXX_SRCS libsrc/feature0/f0.cpp libsrc/feature1/f1.cpp)
set(LIB_ARCHIVE_CXX_SRCS "")
```

LIB_ARCHIVE_C_SRCS

List of C source files to compile to create the archive. File paths are relative to the static library directory.

If not set, all `*.c` files in the `libsrc` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any C sources. Examples:

```
set(LIB_ARCHIVE_C_SRCS libsrc/feature0/f0.c libsrc/feature1/f1.c)
set(LIB_ARCHIVE_C_SRCS "")
```

LIB_ARCHIVE_DEPENDENT_MODULES
List of this static library's dependencies, which must be present when compiling. See the separate dependency management section about the dependency fetching process and the acceptable format for values in this list. If this static library has no dependencies, this variable must be set as an empty string. Unlike other variables, the values to set for `LIB_ARCHIVE_DEPENDENT_MODULES` should be quoted, as this is required when the string contains parentheses. Examples:

```
set(LIB_ARCHIVE_DEPENDENT_MODULES "lib_logging(3.1.1)"
                                  "lib_xassert(4.1.0)")
set(LIB_ARCHIVE_DEPENDENT_MODULES "")
```

LIB_ARCHIVE_INCLUDES
List of directories to add to the compiler's include search path when compiling sources. Example:

```
set(LIB_INCLUDES api src/feature0)
```

LIB_ARCHIVE_XC_SRCS
List of XC source files to compile to create the archive. File paths are relative to the static library directory. If not set, all `*.xc` files in the `libsrc` directory and its subdirectories will be compiled. An empty string can be set to avoid compiling any XC sources. Examples:

```
set(LIB_ARCHIVE_XC_SRCS libsrc/feature0/f0.xc libsrc/feature1/f1.xc)
set(LIB_ARCHIVE_XC_SRCS "")
```

XMOS_SANDBOX_DIR
The path to the root of the sandbox directory. This is only required if `LIB_ARCHIVE_DEPENDENT_MODULES` is non-empty. This must be set in the static library's `lib_build_info.cmake` before the call to `XMOS_REGISTER_STATIC_LIB`. See *Sandbox Structure*.

```
set(XMOS_SANDBOX_DIR ${CMAKE_CURRENT_LIST_DIR}/../..)
```

7.2.3.2 Variables for application build with archive

Variables for this mode of use are the same as the *Required module variables* and *Optional module variables*. Variables that affect source files (eg. compiler flags, selection of source files) are only applied to any additional source files that may be present in the static library repository; the static library archive is not rebuilt.

7.2.4 Output Variables

Experienced CMake users are able to add custom CMake code around the XCommon CMake build system. To support this, some variables are exposed from the `XMOS_REGISTER_APP` function.

APP_BUILD_ARCH
String of the architecture of the application being built. This variable allows the CMake code for a module to be conditionally configured based on the target architecture.

APP_BUILD_TARGETS
List of the target names for the applications which have been configured. This allows relationships to be defined with custom CMake targets that a user may create.

XCOMMON_CMAKE_VER
String containing the version number of XCommon CMake. This is printed as part of a version string

message when run with `--log-level=VERBOSE` at the beginning of the CMake configuration stage. This can be used to write CMake code using knowledge of which versions of XCommon CMake include the required features. Version number comparisons must be performed with the `VERSION_` binary tests in the `if` function for the correct interpretation of the version number sub-components. For example, if a feature is added in v1.1.0:

```
if(XCOMMON_CMAKE_VER VERSION_GREATER_EQUAL 1.1.0)
    # Use the supported feature
else()
    # Do something else as feature is not supported
endif()
```

7.3 Command-line Options

Extra functionality can be activated using command-line options which are implemented in XCommon CMake. These are passed to the CMake command via its `-D` option, and multiple options can be provided to a single CMake command if required:

```
cmake -G "Unix Makefiles" -B build -D <option0>=<value0> -D <option1>=<value1>
```

7.3.1 Supported options

BUILD_NATIVE

Boolean option to configure the build for the native host CPU rather than an xcore target. See *Native CPU Builds* for more details about this feature. Example:

```
cmake -G "Unix Makefiles" -B build -D BUILD_NATIVE=ON
```

DEPS_CLONE_SHALLOW

Boolean option to perform a shallow clone of all missing dependencies. The git repository for each dependency will be cloned as a single commit, rather than the complete history. This can reduce the disk usage, but if the full git history is later required, it will need to be fetched manually. Example:

```
cmake -G "Unix Makefiles" -B build -D DEPS_CLONE_SHALLOW=TRUE
```

7.4 Dependency Format

The `APP_DEPENDENT_MODULES` and `LIB_DEPENDENT_MODULES` variables hold the list of module dependencies for an application, module or static library. The format is flexible to support releases to external users and also internal development practices.

External releases use this format:

1. **lib_abc(1.2.3)**

   get lib_abc tag v1.2.3 from github.com/xmos; a check is made to see if SSH key access is available, otherwise HTTPS is used. The 'v' character is prepended so the released tag must be v1.2.3

For internal development, there are ways to specify the location and version.

Firstly the location can be specified at the beginning of the string:

2. **lib_abc**

   uses `github.com/xmos` as the location from which to clone lib_abc; same behaviour as format 1 for SSH/HTTPS access

3. **`myuser/lib_abc`**
   clones lib_abc from `github.com/myuser`; same behaviour as format 1 for SSH/HTTPS access

4. **`othergitserver.com:myuser/lib_abc`**
   SSH access to clone `git@othergitserver.com:myuser/lib_abc`

5. **`https://othergitserver.com/myuser/lib_abc`**
   clones this URL via HTTPS, without checking for SSH access

Then the version can be specified at the end of the string:

6. **`lib_abc`**
   no version specified, gets the head of whichever branch has been configured as the default

7. **`lib_abc(v1.2.3)`**
   acceptable alternative to format 1, gets tag v1.2.3

8. **`lib_abc(develop)`**
   gets the head of the develop branch

9. **`lib_abc(4fa35fe)`**
   gets commit 4fa35fe

Any of formats 2-5 can be combined with any of 6-9 to specify both the location and the version. For example:

```
set(APP_DEPENDENT_MODULES "myuser/lib_abc(develop)"
                          "othergitserver.com:myuser/lib_foo(1.0.0)")
```

- check out the head of the develop branch of lib_abc from `github.com/myuser`, using an SSH key if available, otherwise via HTTPS.
- check out tag v1.0.0 of lib_foo, cloned from `othergitserver.com/myuser` via SSH access.

8 Advanced Usage

8.1 Native CPU Builds

XCommon CMake supports building libraries and applications for the native host CPU instead of for an xcore device. An example use-case for this feature would be to compile and run a unit test to check the logic of a signal-processing algorithm, where the higher clock speed of a non-embedded CPU allows greater test coverage for the available amount of time.

This is advanced usage of XCommon CMake, and not all modules are guaranteed to have native build support.

8.1.1 CMake Generation

The `BUILD_NATIVE` option must be enabled for the CMake command that generates the build environment, and then the build can continue as normal to produce libraries and applications for the host CPU. The build environment will be configured with the default compiler toolchain for the system, so a suitable toolchain must be installed as a prerequisite.

```
cmake -G "Unix Makefiles" -B build -D BUILD_NATIVE=ON
cd build
xmake
```

8.1.2 Conditional Configuration

If an application or library supports building for an xcore device as well as for the native host CPU, it is possible that the values of XCommon CMake variables need to be set to different values based on which build is being performed. For example, compilation for the native build may occur with a toolchain that requires different compiler options. These variables can be set inside conditional blocks to achieve the desired behaviour.

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(app)

set(APP_HW_TARGET XCORE-AI-EXPLORER)

if(NOT BUILD_NATIVE)
    # Compiler options for xcore build
    set(APP_COMPILER_OPTIONS -Os -mno-dual-issue)
else()
    # Compiler options for native build
    set(APP_COMPILER_OPTIONS -O2)
endif()
```

8.2 Advanced Dependency Management

8.2.1 Dependency Location

By default, the location of dependent modules is the root of the sandbox as defined by the XMOS_SANDBOX_DIR variable. This can be overridden on a per-dependency basis by setting a variable for each non-default dependency location.

Note: The recommended behaviour is to use the sandbox as defined by XMOS_SANDBOX_DIR and overriding the dependency location should only be done in exceptional circumstances.

A variable named XMOS_DEP_DIR_<module> can be used to override the location of dependency <module>. For example, XMOS_DEP_DIR_lib_i2c could be set to the path of the root of a copy of the lib_i2c module in a location other than the root of the sandbox. Then the build system will search for source code for lib_i2c in this location, instead of in a lib_i2c directory in the root of the sandbox.

Any sub-dependencies of this module will be found in their default location in XMOS_SANDBOX_DIR unless they too have an override named variable with their module name.

8.2.2 CMake file contents

Returning to the previous example sandbox structure we will now examine the contents of the CMakeLists.txt and lib_build_info.cmake files when using XMOS_DEP_DIR_lib_mod1 to specify an non-standard location for lib_mod1.

```
sandbox/
       |-- lib_mod0/
       |           |-- lib_mod0/
       |                       |-- api/
       |                       |-- src/
       |                       |-- lib_build_info.cmake
       |
       |-- sw_app0/
       |          |-- app_app0_xcoreai/
       |                              |-- src/
       |                              |-- CMakeLists.txt
other_srcs/
          |-- lib_mod1/
                      |-- lib_mod1/
                                  |-- api/
                                  |-- src/
                                  |-- lib_build_info.cmake
```

sandbox/sw_app0/app_app0_xcoreai/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.21)
include($ENV{XMOS_CMAKE_PATH}/xcommon.cmake)
project(app0_xcoreai)

set(APP_HW_TARGET XCORE-AI-EXPLORER)
set(APP_DEPENDENT_MODULES "lib_mod0")

set(XMOS_SANDBOX_DIR ${CMAKE_CURRENT_LIST_DIR}/../..)
set(XMOS_DEP_DIR_lib_mod1 ${CMAKE_CURRENT_LIST_DIR}/../../../other_srcs/lib_mod1)

XMOS_REGISTER_APP()
```

sandbox/lib_mod0/lib_mod0/lib_build_info.cmake

```
set(LIB_NAME lib_mod0)
set(LIB_VERSION 1.0.0)
set(LIB_INCLUDES api)
set(LIB_DEPENDENT_MODULES "lib_mod1")

XMOS_REGISTER_MODULE()
```

other_srcs/lib_mod1/lib_mod1/lib_build_info.cmake

```
set(LIB_NAME lib_mod1)
set(LIB_VERSION 1.0.0)
set(LIB_INCLUDES api)
set(LIB_DEPENDENT_MODULES "")

XMOS_REGISTER_MODULE()
```