How to Structure Your Software Using CMake Application Note

DesignWare® ARC® MetaWare EV Development Toolkit

Contents

1 Introduction

This application note is based on the 2019.03 version of DesignWare® ARC® MetaWare EV Development Toolkit. Filenames and module may change. See the DesignWare® ARC® MetaWare EV Development Toolkit documentation for up-to-date information.

The DesignWare® ARC® MetaWare EV Development Toolkit uses CMake in its build system to enable modularity and easily resolve dependencies. This application note explains how this system works and how you can structure your own projects or modules in the same way.

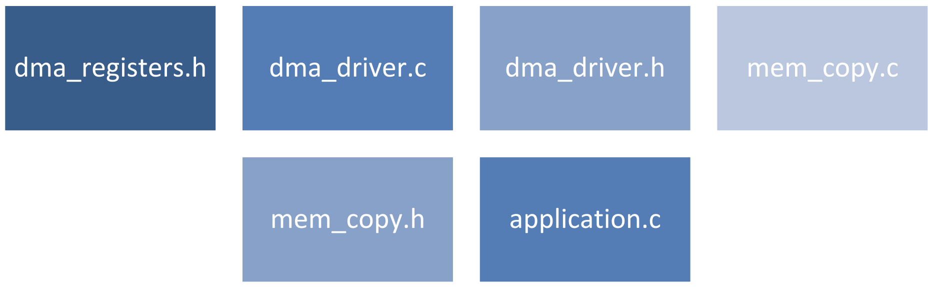
1.1 Problem Statement

When software projects grow larger, managing complexity becomes a challenge. This section explains some of these challenges by using an example.

For example, you might have an application that uses a small library for efficiently copying buffers in memory called **mem_copy**. For efficient operation, the library uses a DMA engine with a driver called **dma_driver**.

1.1.1 Files in the Example Application

The example application the following files, all in the same directory:



- **dma_registers.h**: contains definitions of registers of the DMA engine
- **dma_driver.h**: contains prototypes of DMA transfer functions
- **dma_driver.c**: contains the implementation of DMA transfer functions
- **mem_copy.h**: contains prototypes of memory copy functions
- **mem_copy.c**: contains the implementation of memory copy functions
- **application.c**: contains the application that uses the memory copy functions

1.1.2 Building the Application Using GNU Make

Without CMake, you normally write regular makefiles, to be used by the GNU Make utility.

To build this example application, the C files need to be compiled and linked together. GNU Make has built-in rules to facilitate this process; for example, `name.o` is made automatically from `name.c` by calling:

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c name.c
```

GNU Make also has a built-in rule for linking:

```
$(CC) $(LDFLAGS) $(LDLIBS) object1.o object2.o ... -o name
```

So for building the application, you can add the following to your Makefile:

```
application: application.o mem_copy.o
```

This method, however, will cause a problem: the **mem_copy** library internally uses the **dma_driver** component. Linking just the **mem_copy** library results in linker errors as it the linker is unable to find the **dma_driver** functions used inside **mem_copy**.

Note	Regular Makefiles typically require you to manually spell out dependencies

1.1.3 Information hiding

Because the example application has all header files in the same directory, it is quite easy for the application to include `dma_registers.h` directly and do some DMA transfers on its own by directly programming the DMA registers using the definitions from `dma_registers.h`.

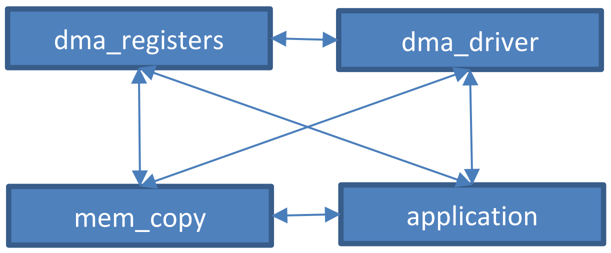
If you access the DMA engine directly from the application, chances are high that you are interfering with the DMA driver.

To prevent this from happening, you can make this header file private to the **dma_driver** component; no other component can then access it. This concept is also known as *encapsulation*.

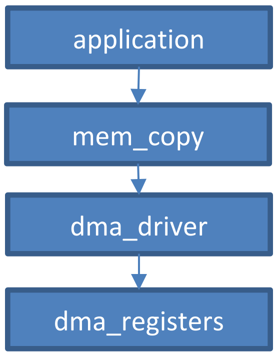
Note	Regular Makefiles typically have one set of compiler flags used to compile all files. In this set of compiler flags, usually the same include header path is used for all files.

1.1.4 Module Dependency Graph

By building the application as outlined so far, you are effectively building it using the following dependency diagram:



However, the following dependency diagram depicts what you actually need:



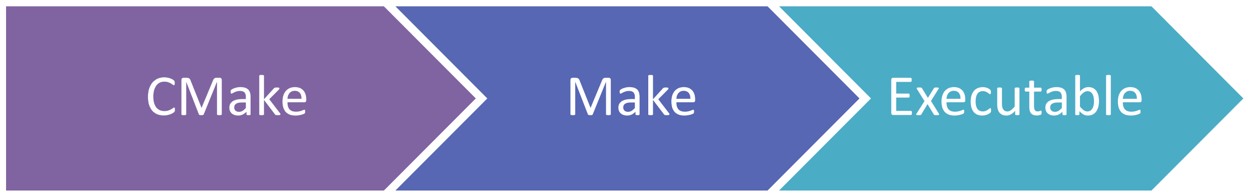
The application should not need to know that the memory copy library it is using requires **dma_driver**. In fact, when the application is built for a different architecture, a DMA engine might not be present, and the implementation of the **mem_copy** component needs to be different.

While `make` has a dependency mechanism, it does not keep track of the header files and include paths required by each component. The `make` dependency mechanism only ensures the correct build order. If you want to hide the dependencies of **dma_driver** for example (so that no one is tempted to include the definitions of the DMA registers and interfere with **dma_driver**), you must manually change the build command for each component relying on **mem_copy** so that they do not include the **dma_driver** headers explicitly.

2 Introducing CMake

You can handle dependencies and hide information using regular makefiles, but doing so requires substantial effort. The MetaWare EV Development Toolkit must also support different environments, such as Windows and Linux, and different compilers.

CMake is a *build system generator* that makes these steps easier. As a build system generator, CMake can generate makefiles to be used by `make`. With the same input files, it is also capable of generating projects for Eclipse CDT, or for Ninja (to name a few examples).



In traditional projects, the include header paths of all modules were usually concatenated, such that each source file could theoretically access all headers, even the ones that contain references to internal functions and data structures.

CMake can help manage this, by clearly distinguishing between *interface*, *public* and *private*. This classification applies not only to header search paths, but also compiler flags; for example, it might be desirable to pass a certain optimization flag to one single module. On the other hand, if you are using C++ 11 code in your module (and headers), you probably want to propagate this requirement to all users of your module.

2.1 Defining Targets in CMake

A target is something that you can build. CMake has three types of targets:

- Executables
- Libraries
- Custom targets

Executables and libraries are defined by using *add_executable* or *add_library*. You need to add these commands to a file called **CMakeLists.txt**, which you can place at the same location where you normally have your makefile.

The syntax is:

```
add_executable (<name> [<source 1>] [<source 2> …])
add_library (<name> [STATIC | SHARED] [<source 1>] [<source 2> …])
```

For example, the sources are the C and C++ files that form the executable or library. For libraries, you can specify whether the library needs to be statically or dynamically linked (`STATIC` vs `SHARED`).

Example:

```
add_executable (hello_openvx main.cc)
```

2.2 Specifying Include Header Paths and Compiler Options

2.2.1 Include header paths

To specify the directories that need to be searched for include header files, use *target_include_directories*. The syntax is as follows:

```
target_include_directories (<target> <INTERFACE | PUBLIC | PRIVATE>
   [items...])
```

Example:

```
target_include_directories(evdev PUBLIC include)
```

This statement includes the `include` directory when compiling the `evdev` component. Since this header path is PUBLIC, any component using the `evdev` component also gets `include` in its include header path. However, some implementation details might exist inside some of the headers in `include`. It is better to hide these details to prevent users from using internal shortcuts and bypassing the intended API. For example, you can put the user-facing API in separate header files, in a separate directory:

```
target_include_directories(evdev
   PUBLIC  include/interface
   PRIVATE include/internal
)
```

This command adds both paths to the compiler command line while building `evdev`, but any upstream component using `evdev` only has access to `include/interface`.

Note	CMake does also have a command *include_directories*, so without `target_` this command is not very useful for modular building your project, as it adds directories unconditionally to all targets inside the same **CMakeLists.txt** file. This command also does not export these header file paths, meaning that every application using your component needs to add the header file paths again, manually.

2.2.1.1 Use ${CMAKE_CURRENT_SOURCE_DIR}

If you would like to organize your source code in a directory hierarchy, you can place **CMakeLists.txt** files inside each subdirectory and include them from a **CMakeLists.txt** file higher up in the directory hierarchy using *add_subdirectory*.

When you do this, the header file paths need some prefixing, as for example `include/interface` becomes `evdev/include/interface` when the compiler is executed from a different location.

The solution is to prefix the header paths with the `${CMAKE_CURRENT_SOURCE_DIR}` variable:

```
target_include_directories(evdev
   PUBLIC  ${CMAKE_CURRENT_SOURCE_DIR}/include/interface
   PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include/internal
)
```

2.2.1.2 Specifying Header Paths at Build Time or After Installing

When you want to redistribute the components that you are creating so that they can be reused by other projects, you can *install* them. The location of the installed header files is different than the source directory, so you likely want to provide different header paths after building the component.

A convenient way to achieve this is through *generator expressions*. In this case, the `INSTALL_INTERFACE` and `BUILD_INTERFACE` generator expressions are useful. The `INSTALL_INTERFACE` generator expression resolves to nothing while building, and the `BUILD_INTERFACE` generator expression disappears when the component is used by a higher level component.

Syntax:

```
$<BUILD_INTERFACE:…>
```

This command expands to … when you build the target, or to the empty string otherwise. The syntax for `INSTALL_INTERFACE` behaves in the same way.

The following snippet applies these commands to the example from section 2.2.1.1.

```
target_include_directories(evdev
   PUBLIC
      $<INSTALL_INTERFACE:include/interface>
      $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include/interface>
   PRIVATE
      $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include/internal>
)
```

2.2.2 Compiler Options

If a library or executable requires specific compiler options, you can pass them using *target_compile_options*. The syntax is:

```
target_compile_options (<target> [INTERFACE | PUBLIC | PRIVATE]
   [<option 1>] [<option 2> …])
```

Example:

```
target_compile_options (openvx PRIVATE “-Werror”)
```

This command enables `-Werror` while building the OpenVX library, but because the option specification is declared as `PRIVATE`, applications using OpenVX are not built with `-Werror` when they use the makefile. So `-Werror` only applies to the library.

On other occasions, you might want to propagate the compiler option to upstream components. For example, if your component uses C++ 11 in its header files:

```
target_compile_options (openvx PUBLIC "-std=c++11")
```

2.2.3 Compiler Features

One problem with using public and private options settings is that some compilers require different options for certain compiler features. For some compiler features, such as the C++ standard, CMake has look-up tables internally that convert a certain requested feature to the correct compiler option:

```
target_compile_features(openvx PUBLIC cxx_std_11)
```

Examples of compiler features are: `c_std_90`, `c_std_99`, `c_std_11`, `c_variadic_macros` for C, and `cxx_std_11`, `cxx_std_14`, `cxx_long_long_type` for C++. A full list can be found on the CMake website.

2.3 Linking to Libraries

To link to libraries, CMAKE provides the *target_link_libraries* command. Its syntax is very similar to the other commands starting with `target_`:

```
target_link_libraries (<target> [INTERFACE | PUBLIC | PRIVATE]
   <item 1> […])
```

The `INTERFACE | PUBLIC | PRIVATE` keyword is not required, and by default `PUBLIC` is used.

Example:

```
target_link_libraries (hello_threads pthread)
```

This command ensures that `-lpthread` is passed to the linker.

target_link_libraries is more powerful than this, however: instead of linking to library files directly, you can also link to *library targets*. This is very useful, because it not only adds the correct linker flag to link to your library, but also adds the correct (PUBLIC) compiler flags and definitions, and adds the include header paths of your library. Plus, it does this *transitively*. For example, to be able to use the OpenVX API in the MetaWare EV Development Toolkit, all that is required is the following:

```
find_package (openvx 1.2 REQUIRED)
target_link_libraries (hello_openvx openvx::openvx)
```

2.3.1 Double-Colon Notation

To use the OpenVX API in the MetaWare EV Development Toolkit, linking is done against `openvx::openvx`, using the *double-colon notation*. This notation is used inside the MetaWare EV Development Toolkit to help distinguish library targets from actual library file names. Because of the double-colon, CMake knows that it cannot pass this command as-is to the linker, so it substitutes the correct target definitions or gives an error message if it does not know about the target. For convenience, the MetaWare EV Development Toolkit always uses the same name before and after the double-colon, but this is not a strict CMake requirement.

2.3.2 find_package

Another useful command is *find_package*. This command tries to find the package and import the target definitions. The syntax is as follows:

```
find_package (<package> [version] [REQUIRED])
```

It is good practice to always supply a version number and the `REQUIRED` keyword. If the `REQUIRED` keyword is omitted, CMake does not generate an error right away if the package is not found, and this may lead to a more obscure error message later on.

2.4 Installing and Exporting Targets

To be able to find your own module using `find_package`, you need to install it and export the targets. CMake can help do most of this for you, but it requires some boilerplate code.

Here are the steps to install and export.

1. Install the library itself (the `library.a` or `library.so` file):

   ```
   install (TARGETS ${project_name}
       EXPORT ${project_name}Targets
       DESTINATION lib
   )
   ```

   The `EXPORT` command registers the library binary file to the exported definitions used in the next step.

2. Generate and install code to enable importing definitions into another project:

   ```
   install (EXPORT ${project_name}Targets
       FILE
           ${project_name}Targets.cmake
       NAMESPACE
           ${project_name}::
       DESTINATION
           lib/cmake/${project_name}
   )
   ```

   This step creates a file called **${project_name}Targets.cmake**, which contains all linker and compiler flags required to use this component in another project and install this file.. It is used by *find_package()*.

3. Generate and install code to check the required version:

   ```
   include(CMakePackageConfigHelpers)
   write_basic_package_version_file(
       ${CMAKE_CURRENT_BINARY_DIR}/${project_name}ConfigVersion.cmake
       VERSION ${project_version}
       COMPATIBILITY AnyNewerVersion
   )
   ```

   This step creates a file called **${project_name}ConfigVersion.cmake**, which contains the logic to compare the requested version with the actual version of a package. In this case, the compatibility policy `AnyNewerVersion` is applied. The file is **not** installed yet. This file is also used by *find_package()*.

4. Install **${project_name}ConfigVersion.cmake" and "cmake/${project_name}Config.cmake**.

   ```
   install(FILES
       ${CMAKE_CURRENT_LIST_DIR}/cmake/${project_name}Config.cmake
       ${CMAKE_CURRENT_BINARY_DIR}/${project_name}ConfigVersion.cmake
   ```

```
            DESTINATION
                lib/cmake/${project_name}
        )
```

This step installs the generated file from the previous step, and installs the **cmake/${project_name}Config.cmake** file. (see 2.5: Create a CMake Configuration File for Your Component).

5. Generate an alias to use double-colon notation

```
add_library(${project_name}::${project_name} ALIAS ${project_name})
```

The library is now exported inside the `${project_name}::` namespace, which means that it is available using the double-colon notation *after* importing it using *find_package()*. This alias is added so that you can use the component inside the same project.

These steps are essentially the same for each library or component that you want to export. The MetaWare EV Development Toolkit contains a file called **evrt.cmake**. This file contains a macro called *export_library()* that can do all of these steps automatically for you.

2.4.1 Installing Header Files

While all definitions and libraries are installed automatically, one item is not done automatically: the header files. Since you only specify include header **paths**, CMake does not know which files need to be installed.

Syntax:

```
install([FILES | DIRECTORY] <items to install> DESTINATION <dir>)
```

Example:

```
install(FILES include/header.h DESTINATION include)
```

Note that install automatically adds the prefix set in `${CMAKE_INSTALL_PREFIX}` to the destination directory.

2.5 Create a CMake Configuration File for Your Component

If you use *target_include_directories*, *target_link_libraries*, etc., CMake knows the correct dependencies of your component. However, it is not smart enough to automatically import all these dependencies. When *find_package()* is called, it searches for a file named `<package>Config.cmake`, and executes it. In this file, you should:

- Call *find_package()* for all dependencies
- Include the target definitions

find_package() can have many parameters that might need to be passed when calling *find_package()* for the dependencies. MetaWare EV provides a macro that does this automatically for you, and it is called *find_dependency()*. This macro needs to be included using *include(CMakeFindDependencyMacro).*

Example:

Here are the full contents of the *evdev/cmake/evdevConfig.cmake* file.

```
include(CMakeFindDependencyMacro)
find_dependency(evthreads 1.0)
include("${CMAKE_CURRENT_LIST_DIR}/evdevTargets.cmake")
```

In this example, the macro definition is included, and *find_dependency()* calls *find_package()* for `evthreads`. This call results in other calls to *find_dependency()* and *find_package()* to load all dependencies for `evthreads`. Finally, the **evdevTargets.cmake** file is loaded from the same directory as the **evdevConfig.cmake** file.

2.6 Combining Multiple Sub-Projects into a Single Top-Level Lists File

It is possible to go through the cycle of calling CMake and then make for each of the components in your project, but you still need to ensure that you follow the correct build order, so that you do not try to build components or applications that have dependencies on a component that is not built yet.

Also, to benefit from a concurrent build (for example, by using `ninja` or `make -j32`), it is beneficial to run CMake only once and have it generate a single build system that contains your complete project with all components inside it.

To achieve this goal, you can have **CMakeLists.txt** files in each component subdirectory. At the top-level you can create another **CMakeLists.txt** file, that adds all these subdirectories recursively, using *add_subdirectory()*.

Syntax:

```
add_subdirectory(source_dir)
```

Caution: You might have applications or components using *find_package()* to find other components that are now inside the same overall project. This situation generates error messages related to duplicate definitions.

To work around situation, it is possible to override the *find_package()* command as follows:

```
macro (find_package pkgname)
    if(NOT "${pkgname}" IN_LIST as_subproject)
        _find_package(${pkgname} CONFIG ${ARGN})
    endif()
endmacro()
```

This override ensures that the real *find_package()* (now available as *_find_package()*, after overriding), is only called when the requested package is not inside the `as_subproject` list. You can also find this macro inside **evrt.cmake**, which is part of the MetaWare EV Development Toolkit.

Aside from including **evrt.cmake**, you need to update the `as_subproject` list each time you add a subdirectory. So, in the end, your top-level **CMakeLists.txt** file might look like this:

```
cmake_minimum_required(VERSION 3.8 FATAL_ERROR)
include(evrt)
list(APPEND as_subproject evthreads evdev openvx ocl_kernels)
add_subdirectory(evthreads)
add_subdirectory(evdev)
add_subdirectory(ocl_kernels)
add_subdirectory(ovx)
add_subdirectory(examples/OpenVX/hello_openvx)
```

Note that in this case, the `hello_openvx` directory does not need to be added to the `as_subproject` list, because this directory only contains an application, and not a library or package that is exported.

3 Summary of Steps

3.1 Creating an Application Using CMake

- Create a CMakeLists.txt file
    1. Add a target (executable) using *add_executable* and add the source files you want to be compiled
    2. Add compiler options using *target_compile_options* (optional)
    3. Add include header paths using *target_include_directories* (optional)
- If your application uses other CMake components:
    1. Add *find_package* for each component required
    2. Add *target_link_libraries* for each component required

3.2 Creating a Library Using CMake

1. Create a CMakeLists.txt file
    a. Add a target (library) using *add_library* and add the source files you want to be compiled
    b. Add compiler options using *target_compile_options* (optional)
    c. Add include header paths using *target_include_directories* (optional)
2. If your library uses other CMake components:
    a. Add *find_package* for each component required
    b. Add *target_link_libraries* for each component required
3. Include **evrt.cmake**
4. Call the *export_library* macro from **evrt.cmake**
5. Make sure necessary header files are installed by using the *install* command
6. Create a CMake configuration file for your component, listing all dependencies using *find_dependency*