


Version: 1.2

Release date: 2022-08-16

Use of this document and any information contained therein is subject to the terms and conditions set forth in Exhibit 1. This document is subject to change without notice.

# Version History

| Version | Date | Description |
|---|---|---|
| 1.0 | 2021-05-05 | Official release |
| 1.1 | 2022-01-27 | Add blo option |
| 1.2 | 2022-08-16 | Add release/debug mode support |

# Table of Contents



## List of Figures

## List of Tables

# 1 Overview

This document provides information about the necessary tools and utilities to install the supporting build environment and run your projects.

The document provide a guide to help you:

- Set up the build environment
- Build a project in the SDK
- Add a module to the middleware
- Create your own project

The build environment guide is applied to the MediaTek IoT Development Platform.

# 2 Environment

This chapter provides detailed guidelines on how to set up the SDK build environment with default GCC on Linux OS and on Microsoft Windows using MSYS2 cross-compilation tool.

## 2.1 Installing the SDK Build Environment on Linux

### 2.1.1 Preparing Linux Build Environment

The toolchain provided in the SDK is required to set up the build environment on Linux OS (Ubuntu 18.10 64bit).

***Table 1. Recommended build environment***

| Item | Description |
|---|---|
| **OS** | Linux OS 18.10 |
| **Make** | GNU make 3.81 |

To install the SDK and build environment on Linux, please download the SDK from MediaTek git01 server. The SDK contains Arm GCC toolchain and MediaTek IoT SDK for demonstration.

```
Now you can start the first build, please change directory to
~/<sdk_root> and execute build command.
Example:
        cd ~/<sdk_root>
        ./build.sh mt7933_hdk qfn_sdk_demo
```

1) Build your project. Run the following command to build your project.

```
cd <sdk_root>
./build.sh mt7933_hdk qfn_sdk_demo
```

The screen output of build process should be as bellow.

```
$ ./build.sh mt7933_hdk qfn_sdk_demo
UE BUILD BOARD: mt7933_hdk
UE BUILD PROJECT: qfn_sdk_demo
platform=Linux
FEATURE = feature.mk
BL_FEATURE = bl_feature_XXXX.mk

Build bootloader...
make -C project/mt7933_hdk/apps/bootloader/GCC BUILD_DIR=...
...
```

# 3 Building the Project Using the SDK

## 3.1. Building Projects

Build projects with the script <sdk_root>/build.sh. To find more information about the script, navigate to the SDK root directory and execute the following command:

```
cd <sdk_root>
./build.sh
```

The outcome is:

```
==================================================================
Build Project
==================================================================
Usage: ./build.sh <board> <project> [bl|clean] <argument>
...
Argument:
    -f=<feature makefile> or --feature=<feature makefile>
       • Replace feature.mk with another makefile. For example, if
         there is a file, called feature_example.mk, under project
         folder, then -f=feature_example.mk will replace feature.mk
         with feature_example.mk.
    -o=<make option> or --option=<make option>
       • Assign additional make option. For example, to compile module
         sequentially, use -o=-j1.
       • To turn on specific feature in feature makefile, use -
         o=<feature_name>=y
       • To assign more than one options, use -o=<option_1> -
         o=<option_2>.
       • To specify the location of a Python virtual environment for
         bootloader wrapping generation, use -o=IMGTOOL_ENV=<path to
         activate script>, refer to the document 'MT793X IoT SDK for
         Build Environment Virtual_Env Installation' for more
         informaiton.
    -blo=<make option> or --bloption=<make option>
       • Assign additional make option for bootloader build when
         building a project with bl specified. For example, to compile
         module sequentially, use -blo=-j1.
       • To turn on specific feature in feature makefile, use -
         blo=<feature_name>=y
       • To assign more than one options, use -o=<option_1> -
         blo=<option_2>.
       • To specify the location of a Python virtual environment for
         bootloader wrapping generation, use -blo=IMGTOOL_ENV=<path
         to activate script>, refer to the document 'MT793X IoT SDK
         for Build Environment Virtual_Env Installation' for more
         informaiton.

==================================================================
List Available Example Projects
==================================================================
Usage: ./build.sh list
```

- List all available boards and projects.

Run the command to show all available boards and projects:

```
./build.sh list
```

The available boards and projects are listed below.

```
==================================================================
Available CM4 Build Projects:

  Example:      Using feature.mk as default for every project.
...
==================================================================
  mt7933_hdk
    qfn_sdk_demo
        |-feature_XXXX.mk
...
```

### 3.1.1 Build the Project.

To build a specific project, simply run the following command.

```
./build.sh <board> <project>
```

The output files are then put in the `<sdk_root>/out/<board>/<project>` folder.

For example, to build a project in the MT7933 HDK, run the following build command:

```
./build.sh mt7933_hdk qfn_sdk_demo
```

The standard output in the terminal window is as follows:

```
$./build.sh mt7933_hdk fqn_sdk_demo
UE BUILD BOARD: mt7933_hdk
UE BUILD PROJECT: qfn_sdk_demo
platform= Linux
FEATURE = feature.mk
BL_FEATURE = bl_feature_XXXX.mk

Build bootloader...
...
```

The output files are then put in the `<sdk_root>/out/mt7933_hdk/qfn_sdk_demo/` folder.

### 3.1.2 Clean the out folder.

The build script `<sdk_root>/build.sh` provides options for removing the generated output files, as shown below.

Clean the `<sdk_root>/out` folder.

```
./build.sh clean
```

Clean the `<sdk_root>/out/<board>` folder.

```
./build.sh <board> clean
```

Clean the `<sdk_root>/out/<board>/<projecct>` folder.

```
./build.sh <board> <project> clean
```

The output folder is defined under variable BUILD_DIR in the `Makefile` in `<sdk_root>/project/mt7933_hdk/apps/qfn_sdk_demo/GCC`:

```
BUILD_DIR = $(PWD)/Build
PROJ_NAME = $(shell basename $(dir $(PWD)))
```

A project image earbuds_ref_design.bin is generated under `<sdk_root>/project/mt7933_hdk/apps/qfn_sdk_demo/GCC/Build`.

### 3.1.3 Build the Project with the "bl" Option

By default, the pre-built bootloader image file is copied to the `<sdk_root>/out/<board>/<project>/` folder after the project is built. The main purpose of the bootloader image is to download the Flash Tool. Apply the "`bl`" option to rebuild the bootloader and use the generated bootloader image file instead of the pre-built one, as shown below.

```
./build.sh <board> <project> bl
```

To build the project on the MT7933 HDK:

```
cd                                                    <sdk_root>
./build.sh mt7933_hdk qfn_sdk_demo bl
```

The output image file of the project and the bootloader, along with the merged image file `flash.bin`, are placed under `<sdk_root>/out/mt7933_hdk/qfn_sdk_demo` folder.

- Clean the out folder

The build script `<sdk_root>/build.sh` provides options to remove the generated output files, as follows.

1) Clean the `<sdk_root>/out` folder.

```
./build.sh clean
```

2) Clean the `<sdk_root>/out/<board>` folder.

```
./build.sh <board> clean
```

3) Clean the `<sdk_root>/out/<board>/<projecct>` folder.

```
./build.sh <board> <project> clean
```

### 3.1.4 Build the Project with the "release/debug" Option

For different release purpose, it could build with different release mode option -
Release mode, formal release for product. It is expected to remove debug code and optimized for image size.
Debug mode, debug purpose for product. It is expected to include more debug information and debug code/command for issue debug. The image file is copied to the `<sdk_root>/out/<board>/<project>/<release_mode>` folder after the project is built.
Apply the "`release/debug`" option to rebuild the image and use the generated image file instead of the pre-built one, as shown below.

```
./build.sh <board> <project> release/debug
```

To build the project on the MT7933 HDK:

```
cd                                                          <sdk_root>
./build.sh mt7933_hdk qfn_tfm_4m bl release
./build.sh mt7933_hdk qfn_tfm_4m bl debug
```

The output image file of the project and the bootloader, along with the merged image file `mt7931an_xip_qfn_bw.bin`, are placed under `<sdk_root>/out/mt7933_hdk/qfn_tfm_4m/release` or `debug` folder.

The configure files for release/debug mode include – **feature.mk, hal_feature.mk, memory.ld, mt7933_flash.ld**. For release mode, the folder path should be at `<sdk_root>/mt7933_hdk/apps/<project>/GCC/` and the debug mode, the path should be at `<sdk_root>/mt7933_hdk/apps/<project>/GCC/debug/` .

It can use cli command – "ver" to confirm the release mode of the current image.

```
$ver
SDK Ver: SDK_2.0.0 (Debug)
Build Time     : Apr 21 2022 10:39:49
Official Build  Time : 2022_03_30_07_09_51
```

### 3.1.5 Building the Project from the Configuration Directory

To build the project:

1) Change the current directory to project source directory where the SDK is located.
2) There are makefiles provided for the project build configuration. For example, the project qfn_sdk_demo is built by the project makefile under `<sdk_root>/project/mt7933_hdk/apps/qfn_sdk_demo/GCC`.
3) Navigate to the example project's path.

For project qfn_sdk_demo:

```
cd <sdk_root>/project/mt7933_hdk/apps/qfn_sdk_demo/GCC
```

4) Run the make command.

```
Make
```

The project output folder is defined under variable BUILD_DIR in the `Makefile` located at `<sdk_root>/project/mt7933_hdk/apps/qfn_sdk_demo/GCC`:

```
BUILD_DIR = $(PWD)/build
PROJ_NAME = qfn_sdk_demo
```

A project image qfn_sdk_demo.bin is generated under `<sdk_root>/project/mt7933_hdk/apps/qfn_sdk_demo/GCC/build`.

# 4 Folder Structure

This chapter shows the structure of the SDK and introduces the content of each folder. The SDK package is organized in a folder structure, as shown in Figure 1.

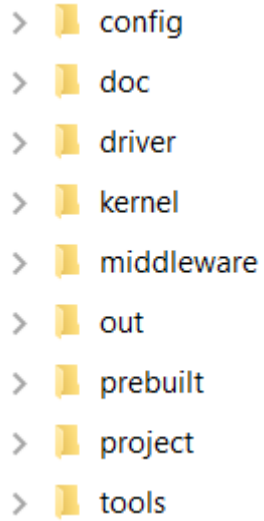


***Figure 1. The SDK package folder structure***

This package contains the source and library files of the main features, the build configuration, related tools and documentation. A brief description of the layout of these files is provided below:

- `config` — includes make and compile configuration files for compiling a binary project.
- `doc` — includes SDK related documentation, such as developer and SDK API reference guides.
- `driver` — includes common driver files, such as board drivers, peripheral and CMSIS-CORE interface drivers.
- `kernel` — includes the underlying RTOS and system services for exception handling and error logging.
- `middleware` — includes software features for HAL and OS, such as network and advanced features.
- `out` — contains binary files, libraries, objects and build logs.
- `prebuilt` — contains binary files, libraries, header files, makefiles and other pre-built files.
- `project` — includes pre-configured example and demo projects using Wi-Fi, HTTP, HAL, and more.
- `tools` — includes tools to compile, download and debug projects using the SDK.

To enable building several different projects simultaneously, each project's output files are placed under the corresponding `<sdk_root>/out/<board>/<project>/` folder.
A brief description of the files is provided below:

- Binary files
  - `project image` — the naming rule of the image file is `PROJ_NAME.bin`, which in this case is `qfn_sdk_demo.bin`. The variable `PROJ_NAME` is defined in `Makefile`, please refer to Section 5, "Makefiles".
  - `bootloader image` — the naming rule of the image file is `<board>_bootloader.bin`, which in this case is defined as `mt7933_bootloader.bin`.
- `elf file` — contains information about the executable, object code, shared libraries and core dumps.
- `map file` — contains the link information of the project libraries.
- `lib folder` — contains module libraries.
- `log folder` — contains build log including build information, timestamp and error messages.
- `obj folder` — contains object and dependency files.

# 5 Makefiles

The SDK package contains several makefiles. The usage and the relation between each makefile are described below.
This chapter uses qfn_sdk_demo project as a reference.

## 5.1 Project Makefile

The project makefile is mainly used to generate the project image. It is placed under the <sdk_root>/project/<board>/apps/<project>/GCC/ folder, named as Makefile. The purpose of the project makefile is summarized below:

- Configures project settings, including the root directory, project name, project path, and more.
- Includes other makefiles for the configuration, such as feature.mk, chip.mk and board.mk.
- Sets the file path of the project's source code.
- Sets the include path of the project's header files.
- Sets the dependency rules for the build flow of the project.
- Sets the module libraries to link when creating the image file.
- Triggers a make command for each module to create a module library.

## 5.2 Configuration Makefiles

This section provides more details on the configuration makefiles; feature.mk, chip.mk and board.mk.

1) Feature configuration file (feature.mk) is placed at <sdk_root>/project/<board>/apps/<project>/GCC/feature.mk. You can turn on or turn off a specific feature by simply changing the value of the feature option defined in the feature.mk.

The IC_CONFIG and BOARD_CONFIG are also defined in the feature.mk.

For example, the feature.mk under project/mt7933_hdk/apps/qfn_sdk_demo/GCC/ is shown below:

```
IC_CONFIG                                = <board>
BOARD_CONFIG                             = <board>
...
# debug level: none, error, warning, info and debug
MTK_DEBUG_LEVEL                          = info
...
```

2) `chip.mk` is located at `<sdk_root>/config/chip/<board>/chip.mk` and defines the common settings, compiler configuration, include path and middleware module path of the chip. The major functions of the `chip.mk` are described below:

- Configures the common settings of the chip.
- Defines the CFLAGS macro.
- Sets the include path of the kernel and the driver header file.
- Sets the module folder path that contains the Makefile.

For example, a partial list of the `chip.mk` under config/chip/<board>/ is shown below.

```
PRODUCT_VERSION                          = <board>
...

AR      = $(BINPATH)/arm-none-eabi-ar
CC      = $(BINPATH)/arm-none-eabi-gcc
CXX     = $(BINPATH)/arm-none-eabi-g++
OBJCOPY = $(BINPATH)/arm-none-eabi-objcopy
SIZE    = $(BINPATH)/arm-none-eabi-size
OBJDUMP = $(BINPATH)/arm-none-eabi-objdump

...

COM_CFLAGS  +=  $(ALLFLAGS)  $(FPUFLAGS)  -ffunction-sections  -fdata-
sections -fno-builtin -Wimplicit-function-declaration
COM_CFLAGS += -gdwarf-2 -Os -Wall -fno-strict-aliasing -fno-common
COM_CFLAGS     +=     -Wall     -Wimplicit-function-declaration     -
Werror=uninitialized  -Wno-error=maybe-uninitialized  -Werror=return-
type
COM_CFLAGS += -DPCFG_OS=2 -D_REENT_SMALL -Wno-error -Wno-switch
COM_CFLAGS += -DPRODUCT_VERSION=$(PRODUCT_VERSION)
...

#Incldue Path
COM_CFLAGS += -I$(SOURCE_DIR)/...
...
```

# 6 Adding a Module to the Middleware

This chapter provides details on adding a module or a custom defined feature into an existing project. The added module is compiled, archived and linked with other libraries to create the final image file during the project build. The following example shows how to add a module named `mymodule` into `qfn_sdk_demo` project on the MT7933 HDK development board.

## 6.1 Files to Add

### 6.1.1 Source and Header Files

Create a module folder with module name under `<sdk_root>/middleware/third_party/` folder to place the module files. Module source and header files should be placed under the `"src"` and the `"inc"` folders, respectively, as shown in Figure 2.

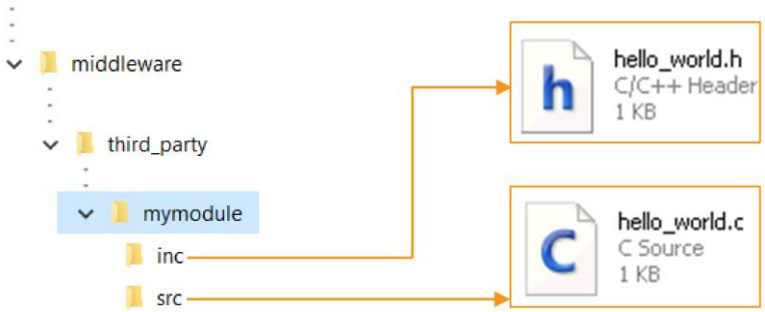


***Figure 2. Place module source and header files under the module folder***

The sample source code `hello_world.c` and header file `hello_world.h` with their corresponding path are shown below:

`<sdk_root>/middleware/third_party/mymodule/src/hello_world.c`

```
#include "hello_world.h"

void myFunc(void)
{
    printf("%s", "hello world\n");
}
```

`<sdk_root>/middleware/third_party/mymodule/inc/hello_world.h`

```
#ifndef __HELLO_WORLD__
#define __HELLO_WORLD__

#include <stdio.h>

void myFunc(void);

#endif
```

### 6.1.2 Makefiles for the Module

Create a makefile under the module folder (see Figure 3) named `module.mk`. It defines the module path, module sources that need to be compiled and include path for the compiler to search for the header files during compilation.

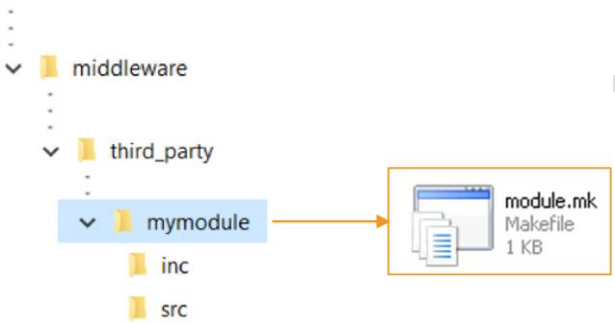


*Figure 3. Create a module.mk under module folder*

In this example, the `module.mk` is at `<sdk_root>/middleware/third_party/mymodule/module.mk`. `C_FILES` and `CFLAGS` are built-in variables that store the module's .c source files and include paths, respectively. The corresponding built-in variables to support compiling source files of a module (`.cpp`) are `CXX_FILES` and `CXXFLAGS`.

```
#module path
MYMODULE_SRC = middleware/third_party/mymodule

#source file
C_FILES  +=  $(MYMODULE_SRC)/src/hello_world.c
CXX_FILES+=

#include path
CFLAGS   += -I$(SOURCE_DIR)/middleware/third_party/mymodule/inc
CXXFLAGS +=
```

Besides **`module.mk`**, another makefile under `mymodule` folder named **`Makefile`** (`<sdk_root>/middleware/third_party/mymodule/Makefile`) is required to generate a module library, as shown in Figure 4.

Most of the dependency rules and definitions in the file are written for common usage. Simply copy the code below and modify the value of the variable `PROJ_PATH` and `TARGET_LIB`. The variable `PROJ_PATH` is the path to the project folder that contains the **`Makefile`** and the variable `TARGET_LIB` is the library for the added module.

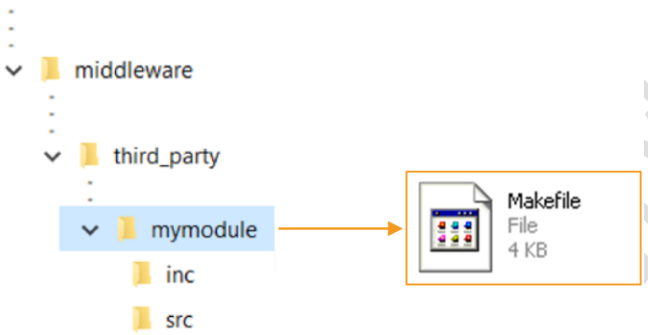


***Figure 4. Create a `Makefile` under module folder***

```
SOURCE_DIR  = ../../..
BINPATH     = ~/gcc-arm-none-eabi/bin
PROJ_PATH   = ../../../project/mt7933_hdk/apps/qfn_sdk_demo/GCC
CONFIG_PATH ?= .

CFLAGS += -I$(PROJ_PATH)/../inc
CFLAGS += -I$(SOURCE_DIR)/$(CONFIG_PATH)


FEATURE    ?= feature.mk
include $(PROJ_PATH)/$(FEATURE)

# Global Config
-include $(SOURCE_DIR)/.config
# IC Config
-include $(SOURCE_DIR)/config/chip/$(IC_CONFIG)/chip.mk
# Board Config
-include $(SOURCE_DIR)/config/board/$(BOARD_CONFIG)/board.mk

# Project name
TARGET_LIB=libmymodule

BUILD_DIR = Build
OUTPATH   = Build

# Sources
include module.mk

C_OBJS   = $(C_FILES:%.c=$(BUILD_DIR)/%.o)
  CXX_OBJS = $(CXX_FILES:%.cpp=$(BUILD_DIR)/%.o)

.PHONY: $(TARGET_LIB).a

all: $(TARGET_LIB).a
    @echo Build $< Done

include $(SOURCE_DIR)/.rule.mk

clean:
    rm -rf $(OUTPATH)/$(TARGET_LIB).a
    rm -rf $(BUILD_DIR)
```

## 6.2 Adding a Module to the Build Flow of the Project

The rules to compile module sources to a single library are now complete and the module is ready to build. To add the module into the project's build flow, modify the `Makefile` under the project folder. In this example, it is at `<sdk_root>/project/mt7933_hdk/apps/qfn_sdk_demo/GCC/Makefile`.

The `Makefile` contains a section with rules defined as `include XXXX/module.mk`. This section defines the modules required by the project for creating an image file. Add a new line into the section to include the module in the project.

Include your module's `module.mk` path in the `Makefile`, as shown below.

```
…
#######################################################################
#########
#
# SDK source files
#
#######################################################################
#########


#include cjson
include $(SOURCE_DIR)/middleware/third_party/cjson/module.mk

#include xml
include $(SOURCE_DIR)/middleware/third_party/xml/module.mk
#include mymodule
include $(SOURCE_DIR)/middleware/third_party/mymodule/module.mk

…
```

After the module is successfully built, the object and the dependency files of the added module can be found under `<sdk_root>/out/<board>/<project>/obj/middleware/third_party/mymodule` folder. In this example the file path is `<sdk_root>/out/<board>/<project>/obj/middleware/third_party/mymodule/hello_world.o` and `<sdk_root>/out/<board>/<project>/obj/middleware/third_party/mymodule/hello_world.d`.

This chapter shows how to add a module to an existing project. The next chapter describes how to create a project.

Note, starting from Mediatek IoT SDK v4.1.0, a new method to add a module to the build flow of the project is introduced, that is, by including the module's `module.mk`. Please merge your project configuration in SDK v4.0.0 with corresponding project's Makefile in SDK v4.1.0 according to the following rules:

- Add .c and `.cpp` source files to C_FILES and CXX_FILES.
- Add your C flags and include headers for C and CPP to CFLAGS and CXXFLAGS accordingly.
- Include modules into your project through `module.mk` located in SDK's `module` folder. Note that some modules in SDK v4.0.0 use makefile to generate a library. For example, in SDK v4.0.0, use the command "`LIBS += $(OUTPATH)/libhal.a`" to include the HAL module; however, in SDK v4.1.0, modify the include path, such as "`include $(SOURCE_DIR)/driver/chip/mt7933/module.mk`".
- Add required libraries in LIBS.

# 7 Creating a Project

This chapter provides details of how to use an existing project and create your own project named `my_project` on MT7933 HDK using qfn_sdk_demo project as a reference.

## 7.1 Using an Existing Project

Apply an existing project as a reference design for your own project development.

Copy the folder `<sdk_root>/project/mt7933_hdk/apps/qfn_sdk_demo` to a new directory `<sdk_root>/project/mt7933_hdk/apps/` and rename `qfn_sdk_demo` as the new project name `my_project`.

## 7.2 Removing a Module

The copied project has modules that could be removed in order to have a clean start for your project development. After the previous steps, a project with the same features has been created. It can be built to generate image file as the original project.

To remove a module:

1) Open the project `Makefile` from `<sdk_root>/project/mt7933_hdk/apps/my_project/GCC/Makefile.`
2) Locate the module include list of the project and remove any unwanted module by removing or commenting out the corresponding include statement.

```
#####################################################
...
include $(SOURCE_DIR)/middleware/MTK/.../module.mk
...
```

## 7.3 User-defined Source and Header Files

User defined project source and header files should be put under the `src` and the `inc` folder respectively. See Figure 5.

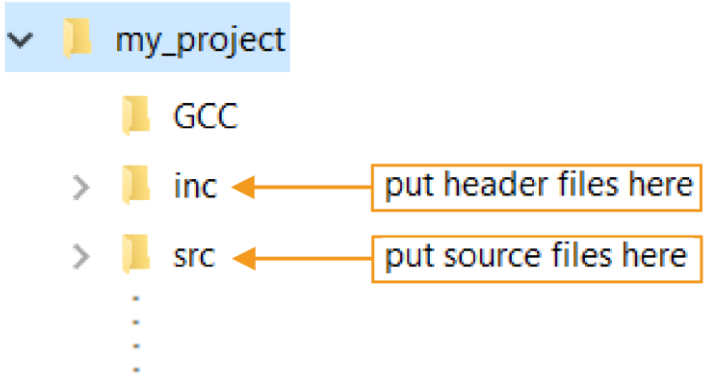


*Figure 5. Project source and header files under the project folder*

To compile the added source code, simply add the .c source files to variable "C_FILES" and the header search path to variable "CFLAGS" in the project Makefile, as shown below. The corresponding variables to support compiling the source files (.cpp) of the module are CXX_FILES and CXXFLAGS).

In current Makefile, there are two intermediate define "APP_FILES" and "SYS_FILES". Both of them are added in C_FILES. This line "include $(SOURCE_DIR)/$(APP_PATH_SRC)/apps/module.mk" which is in Makefile includes the C files in folder <my_projet>/src/apps

**<sdk_root>/project/mt7933_hdk/apps/my_project/GCC/Makefile**

```
...
APP_FILES       += $(APP_PATH_SRC)/main.c
APP_FILES       += $(APP_PATH)/GCC/XXX.c
...
SYS_FILES       += $(APP_PATH_SRC)/system_mt7933.c

...
CXX_FILES       += ...
...
C_FILES         += $(APP_FILES) $(SYS_FILES)
...
```

## 7.4 Test and Verify

After the project is successfully built, the final image file can be found under `<sdk_root>/out/<board>/<project>/` folder. In this example, it's `<sdk_root>/out/mt7933_hdk/my_project/`.

The object and the dependency files of your project can be found under `<sdk_root>/out/<board>/<project>/obj/project/<board>/apps/<project>/src/` folder. In this example, it is `<sdk_root>/out/mt7933_hdk/my_project/obj/project/mt7933_hdk/apps/my_project/src/`.

The path of the image file, object and dependency files after the example project is built is as shown in Figure 6.

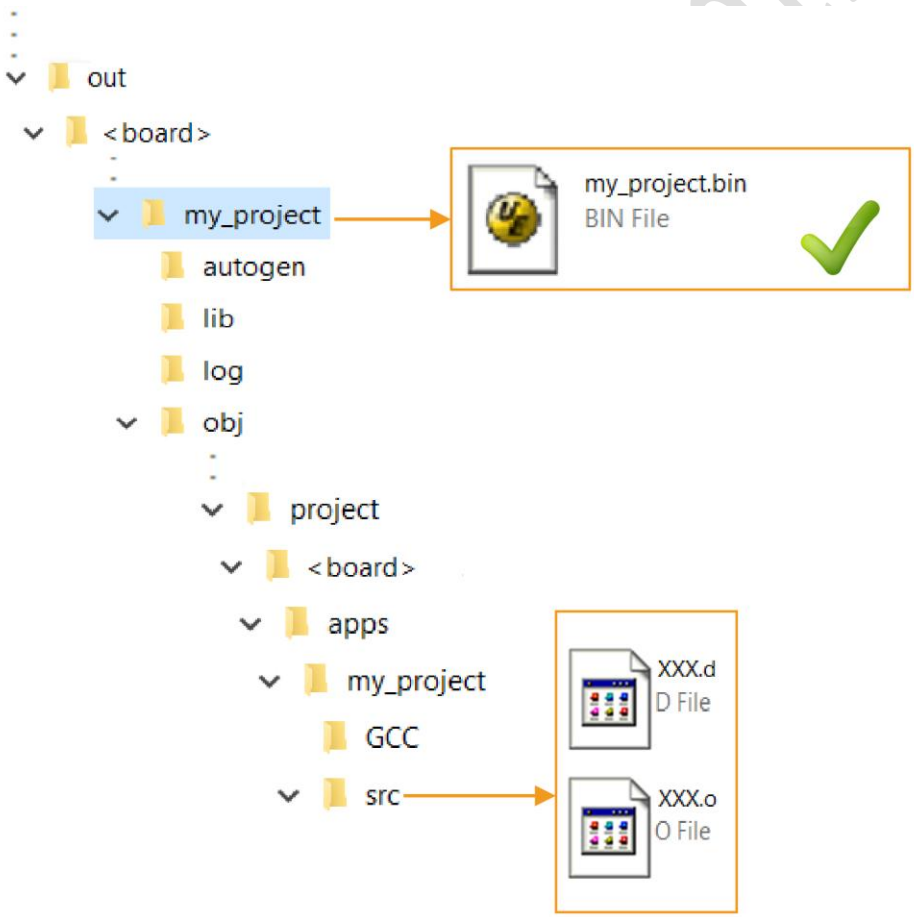


***Figure 6. Output image file, object files and dependency files after project is successfully built***

### 7.4.1 Troubleshooting

When a build process fails as shown below, the error messages are written to the `err.log` file under `<sdk_root>/out/<board>/<project>/log/` folder. Please see the `err.log` file for more information.

# Exhibit 1 Terms and Conditions

Your access to and use of this document and the information contained herein (collectively this "Document") is subject to your (including the corporation or other legal entity you represent, collectively "You") acceptance of the terms and conditions set forth below ("T&C"). By using, accessing or downloading this Document, You are accepting the T&C and agree to be bound by the T&C. If You don't agree to the T&C, You may not use this Document and shall immediately destroy any copy thereof.

This Document contains information that is confidential and proprietary to MediaTek Inc. and/or its affiliates (collectively "MediaTek") or its licensors and is provided solely for Your internal use with MediaTek's chipset(s) described in this Document and shall not be used for any other purposes (including but not limited to identifying or providing evidence to support any potential patent infringement claim against MediaTek or any of MediaTek's suppliers and/or direct or indirect customers). Unauthorized use or disclosure of the information contained herein is prohibited. You agree to indemnify MediaTek for any loss or damages suffered by MediaTek for Your unauthorized use or disclosure of this Document, in whole or in part.

MediaTek and its licensors retain titles and all ownership rights in and to this Document and no license (express or implied, by estoppels or otherwise) to any intellectual propriety rights is granted hereunder. This Document is subject to change without further notification. MediaTek does not assume any responsibility arising out of or in connection with any use of, or reliance on, this Document, and specifically disclaims any and all liability, including, without limitation, consequential or incidental damages.

THIS DOCUMENT AND ANY OTHER MATERIALS OR TECHNICAL SUPPORT PROVIDED BY MEDIATEK IN CONNECTION WITH THIS DOCUMENT, IF ANY, ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE. MEDIATEK SPECIFICALLY DISCLAIMS ALL WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, COMPLETENESS OR ACCURACY AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MEDIATEK SHALL NOT BE RESPONSIBLE FOR ANY MEDIATEK DELIVERABLES MADE TO MEET YOUR SPECIFICATIONS OR TO CONFORM TO A PARTICULAR STANDARD OR OPEN FORUM.

Without limiting the generality of the foregoing, MediaTek makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does MediaTek assume any liability arising out of the application or use of any product, circuit or software. You agree that You are solely responsible for the designing, validating and testing Your product incorporating MediaTek's product and ensure such product meets applicable standards and any safety, security or other requirements.

The above T&C and all acts in connection with the T&C or this Document shall be governed, construed and interpreted in accordance with the laws of Taiwan, without giving effect to the principles of conflicts of law.