



Airoha IoT SDK GCC Build Environment Guide

Version: 1.4

Release date: 5 May 2017

© 2015 - 2018 Airoha Technology Corp.

This document contains information that is proprietary to Airoha Technology Corp. ("Airoha") and/or its licensor(s). Airoha cannot grant you permission for any material that is owned by third parties. You may only use or reproduce this document if you have agreed to and been bound by the applicable license agreement with Airoha ("License Agreement") and been granted explicit permission within the License Agreement ("Permitted User"). If you are not a Permitted User, please cease any access or use of this document immediately. Any unauthorized use, reproduction or disclosure of this document in whole or in part is strictly prohibited. THIS DOCUMENT IS PROVIDED ON AN "AS-IS" BASIS ONLY. AIROHA EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES OF ANY KIND AND SHALL IN NO EVENT BE LIABLE FOR ANY CLAIMS RELATING TO OR ARISING OUT OF THIS DOCUMENT OR ANY USE OR INABILITY TO USE THEREOF. Specifications contained herein are subject to change without notice.

Document Revision History

| Revision | Date | Description |
|----------|-----------------|---|
| 1.0 | 31 March 2016 | Initial release |
| 1.2 | 30 June 2016 | <ul style="list-style-type: none">Added support for build environment on Windows OS. |
| 1.3 | 13 January 2017 | Update providing: <ul style="list-style-type: none">A method to add a module.Support for C++ source files. |
| 1.4 | 5 May 2017 | <ul style="list-style-type: none">Added details on the pre-built folder.Added requirement to Install GNU Awk on LinkIt 2533 HDK. |

Table of contents

| | | |
|-----------|---|-----------|
| 1. | Overview | 1 |
| 2. | Environment..... | 2 |
| 2.1. | Installing the SDK build environment on Linux..... | 2 |
| 2.2. | Installing the SDK build environment on Microsoft Windows..... | 3 |
| 2.2.1. | Preparing the cross-compiler tool | 3 |
| 2.2.2. | Troubleshooting..... | 8 |
| 3. | Building the Project Using the SDK..... | 10 |
| 3.1. | Building the project from the SDK root directory..... | 10 |
| 3.1.1. | List all available boards and projects | 10 |
| 3.1.2. | Build the project | 11 |
| 3.1.3. | Build the project with the "b1" option | 11 |
| 3.2. | Building the project from the project GCC configuration directory | 12 |
| 4. | Folder Structure..... | 13 |
| 5. | Makefiles | 15 |
| 5.1. | Project Makefile | 15 |
| 5.2. | Configuration makefiles..... | 16 |
| 6. | Adding a Module to the Middleware | 18 |
| 6.1. | Files to add | 18 |
| 6.1.1. | Source and header files | 18 |
| 6.1.2. | Makefiles for the module | 19 |
| 6.2. | Adding a module to the build flow of the project | 21 |
| 7. | Create a Project | 22 |
| 7.1. | Using an existing project | 22 |
| 7.2. | Removing a module | 23 |
| 7.3. | User-defined source and header files..... | 23 |
| 7.4. | Test and verify | 25 |
| 7.4.1. | Troubleshooting..... | 25 |

Lists of tables and figures

| | |
|---|----|
| Table 1. Recommended build environment | 2 |
| Figure 1. MinGW Installation Manager Setup Tool..... | 3 |
| Figure 2. Keep default installation preferences | 4 |
| Figure 3. Download and set up MinGW Installation Manager..... | 5 |
| Figure 4. Basic setup on MinGW Installation Manager..... | 5 |
| Figure 5. A basic MinGW installation | 6 |
| Figure 6. Schedule of pending actions | 6 |
| Figure 7. Applying scheduled changes | 6 |
| Figure 8. MinGW folder structure..... | 7 |
| Figure 9. tools/gcc folder structure..... | 8 |
| Figure 10. MinGW Installation Manager..... | 9 |
| Figure 11. The SDK package folder structure..... | 13 |
| Figure 12. The folder structure of the project's output files | 14 |
| Figure 13. The generated files under example project | 14 |
| Figure 14. The location of the project's feature configuration file and makefile | 16 |
| Figure 15. The path to the chip.mk | 16 |
| Figure 16. The Path of board.mk..... | 17 |
| Figure 17. Place module source and header files under the module folder..... | 18 |
| Figure 18. Create a module.mk under module folder..... | 19 |
| Figure 19. Create a Makefile under module folder | 19 |
| Figure 20. Modify the Makefile under the GCC folder of my_project | 22 |
| Figure 21. Project source and header files under the project folder | 24 |
| Figure 22. Output image file, object files and dependency files after project is successfully built | 25 |

1. Overview

Airoha IoT SDK GNU Compiler Collection (GCC) build environment guide provides tools and utilities to install the supporting build environment and run your projects.

The document guides you through:

- Setting up the build environment
- Building a project using the SDK
- Adding a module to the middleware
- Creating your own project


The build environment guide is applied to the Airoha IoT Development Platform for RTOS including LinkIt 7687 HDK and LinkIt 2523 HDK. The examples in this document are based on the LinkIt 7687 HDK, but the content can also be applied to the LinkIt 2523 HDK.

2. Environment

This section provides detailed guideline on how to set up the SDK build environment with default GCC on Linux OS and on Microsoft Windows using [MinGW](#) cross-compilation tool.

2.1. Installing the SDK build environment on Linux

Default GCC compiler provided in the SDK is required to setup the build environment on Linux OS. The following description is based on the Ubuntu 14.04 LTS environment.

 Note, the Airoha IoT SDK can be used on any edition of Linux OS. The default GCC compiler provided in the SDK is based on the 32-bit architecture.

Before building the project, verify that you've installed the required toolchain for your build environment, as shown in Table 1.

Table 1. Recommended build environment


| Item | Description |
|----------|--|
| OS | Linux OS |
| make | GNU make 3.81 |
| Compiler | Linaro GCC Toolchain for ARM Embedded Processors 4.8.4 |

The following command downloads and installs the basic building tools on Ubuntu.

```
sudo apt-get install build-essential
```

Note, a compilation error occurs when building the Airoha IoT SDK with the default GCC cross compiler on a 64-bit system without installing the package to support the 32-bit executable binary, as shown below.

```
/bin/sh: 4: tools/gcc/gcc-arm-none-eabi/bin/arm-none-eabi-gcc: not found
```

 The commands to install the basic build tools and the package for supporting 32-bit binary executable on the Ubuntu 14.04 are shown below.

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6-i386
```

To generate external DSP image on LinkIt 2533 HDK, download and install GNU Awk on Ubuntu with the following command:

```
sudo apt-get install gawk
```

Wearable applications, such as a watch based on TouchGFX framework, requires Ruby to generate resource files. Find more information on how to install Ruby in `<sdk_root>\project\mt2523_watch\apps\watch_demo\readme.txt`

Install the SDK package according to the instructions at <sdk_root>/readme.txt. The default installation path of the GCC compiler is <sdk_root>/tools/gcc, and the compiler settings are in the <sdk_root>/config configuration file.

Setup the BINPATH in the .config file, as shown below.

```
BINPATH = $(SOURCE_DIR)/tools/gcc/gcc-arm-none-eabi/bin
```

2.2. Installing the SDK build environment on Microsoft Windows

To build the project on Windows OS, install MinGW cross-compiler and integrate ARM GCC toolchain for Windows with the Airoha IoT SDK.

2.2.1. Preparing the cross-compiler tool

- 1) Download mingw-get-setup.exe from [here](#).
- 2) Launch the installer, and click **Install** (see Figure 1).

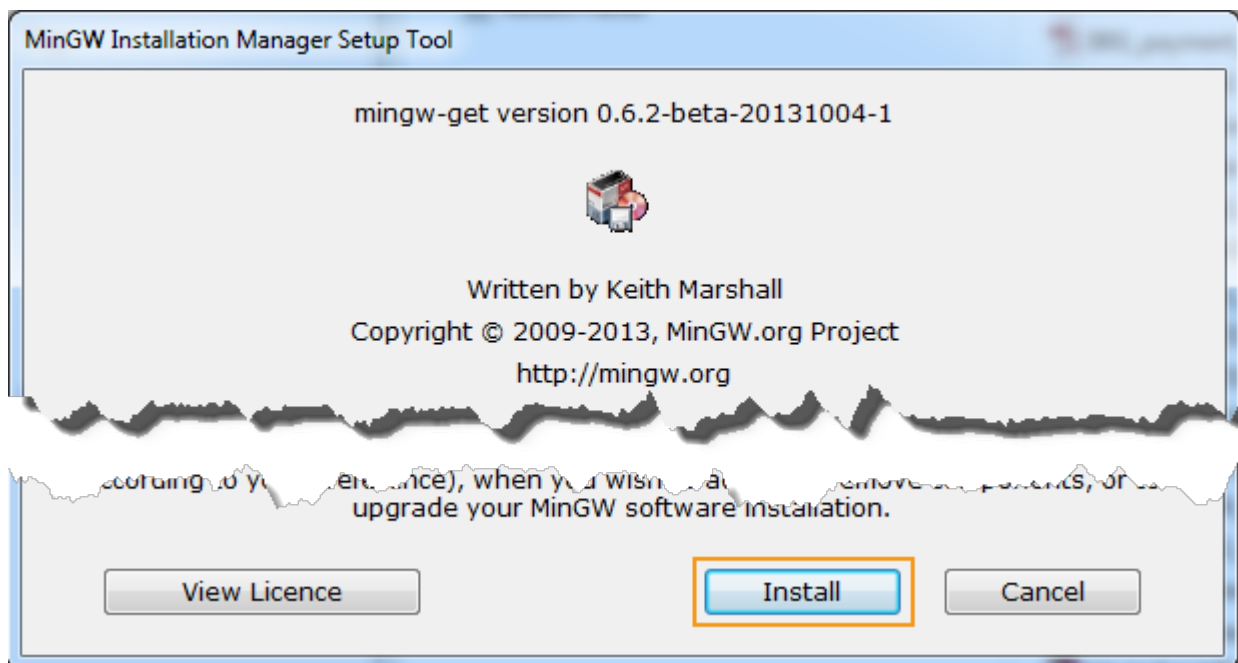


Figure 1. MinGW Installation Manager Setup Tool

- 3) Follow the on screen instructions and keep the default settings, then click **Continue** to download the tool to C:\MinGW installation directory (see Figure 2).

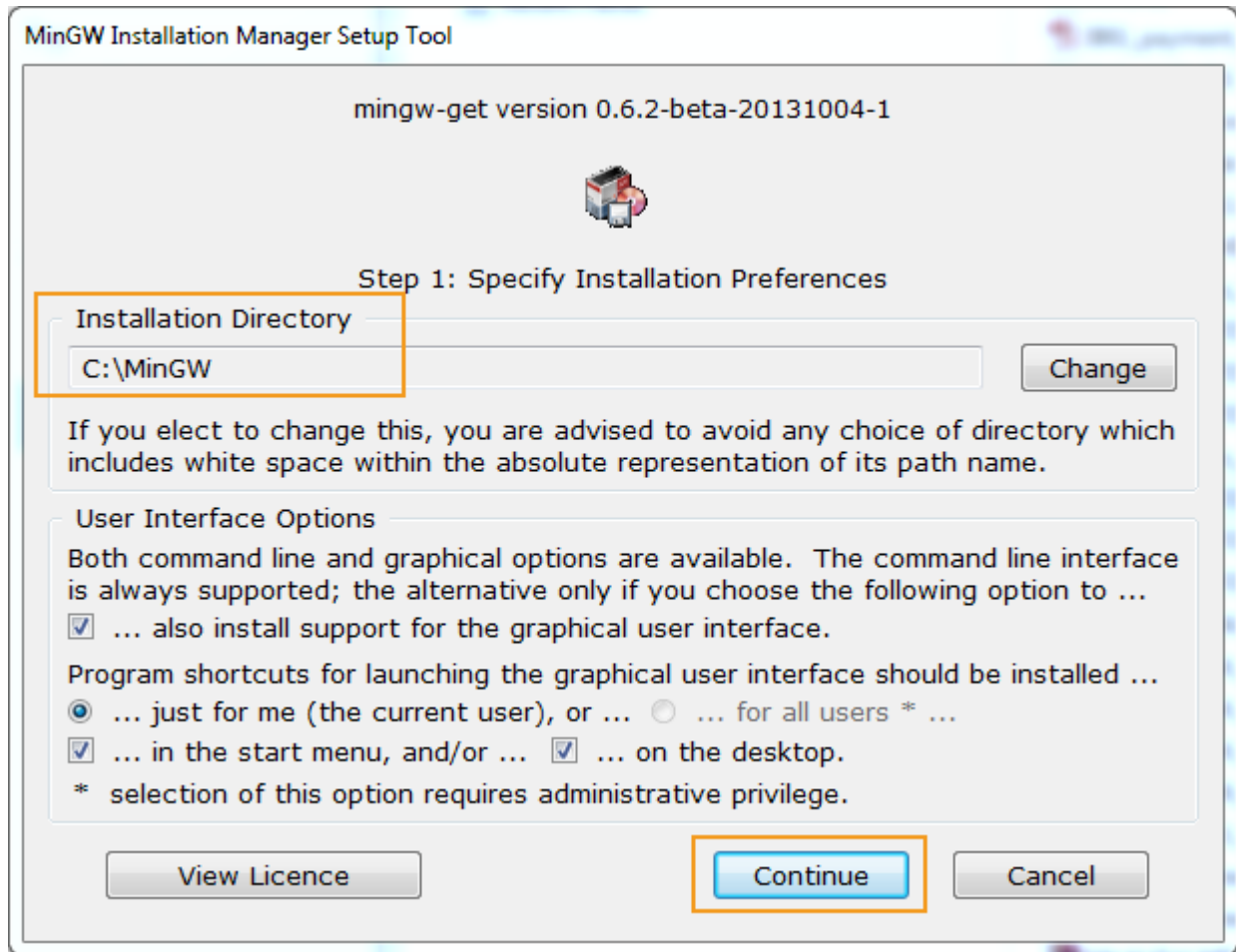


Figure 2. Keep default installation preferences

- 4) Click **Continue** on the MinGW Installation Manager Setup Tool after the download is complete (see Figure 3).

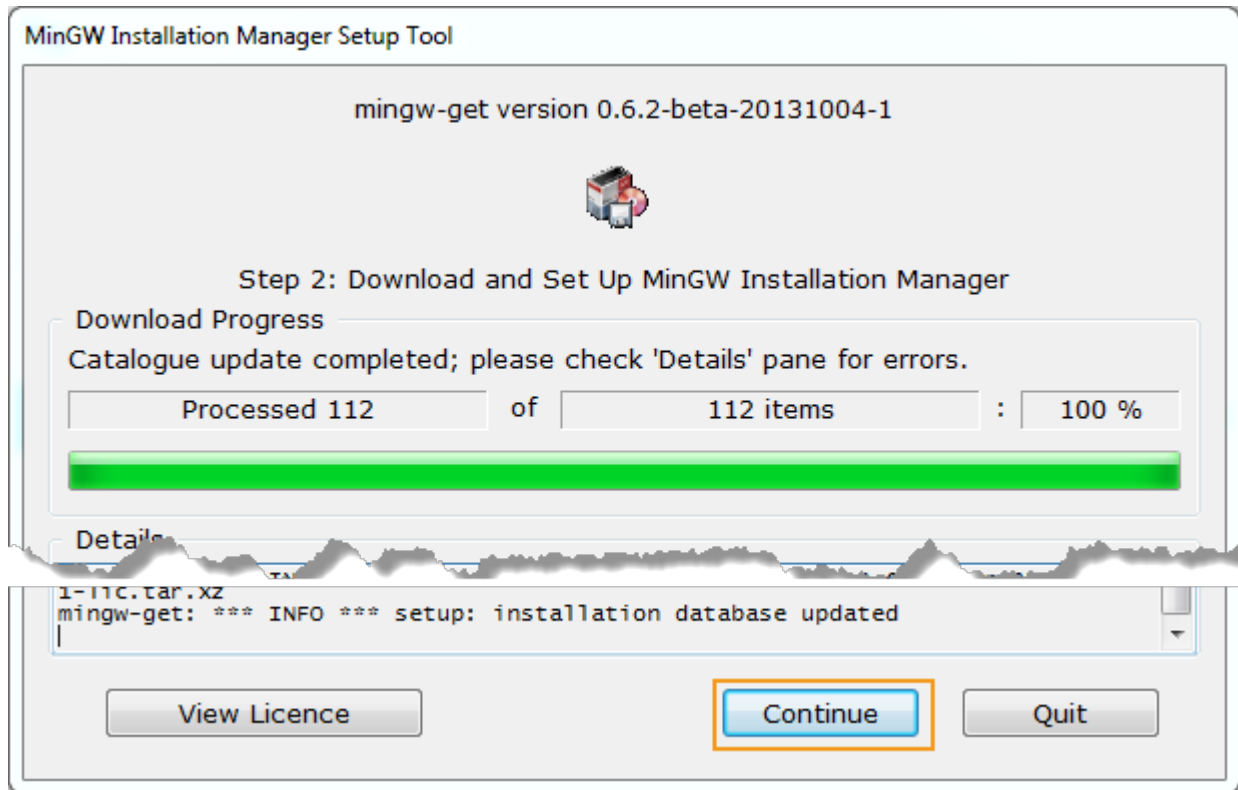


Figure 3. Download and set up MinGW Installation Manager

- 5) Select **msys-base** and **mingw32-base** from **Basic Setup** package list, and right click to bring up the menu options. Click **Mark for Installation** from the menu (see Figure 4).

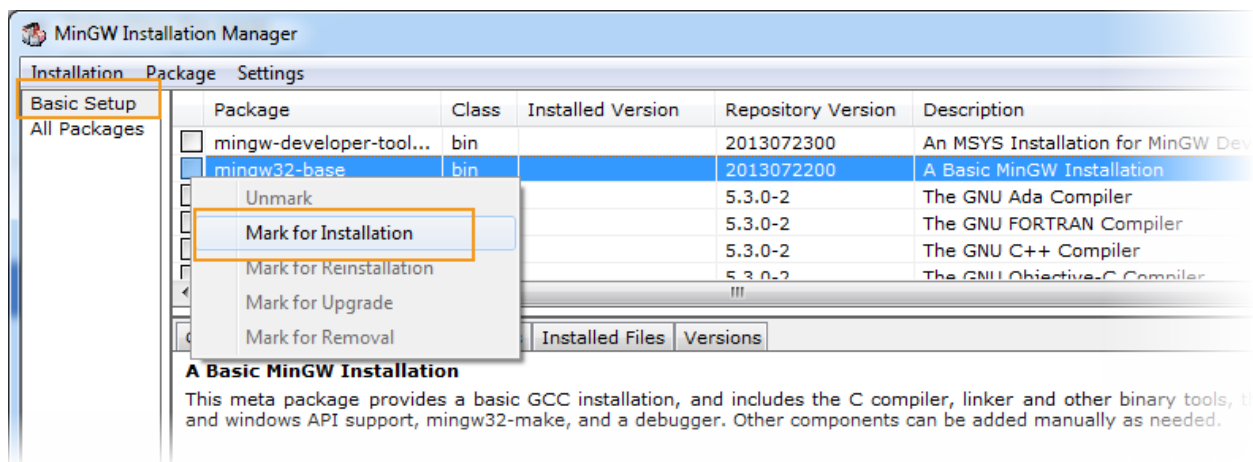


Figure 4. Basic setup on MinGW Installation Manager

- 6) Click **Apply Changes** from the **Installation** menu (see Figure 5).

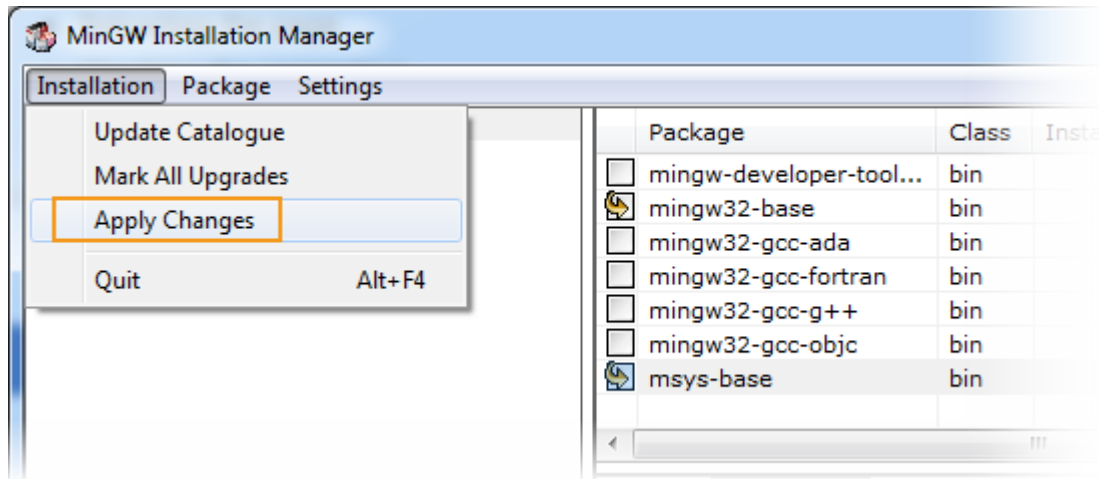


Figure 5. A basic MinGW installation

- 7) Click **Apply** on the pop-up dialog window (see Figure 6).

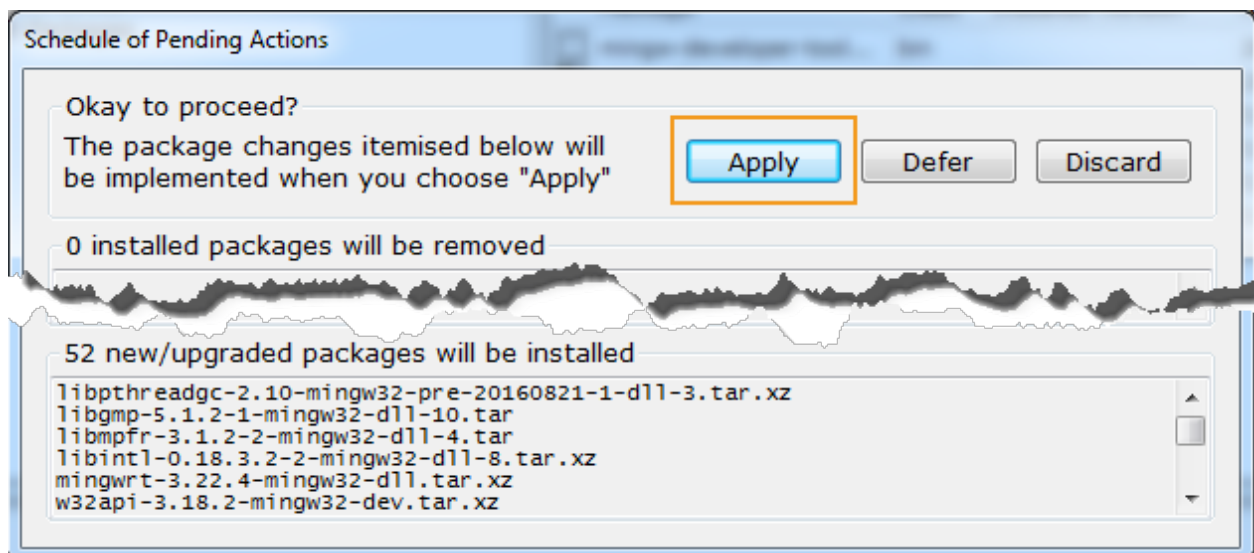


Figure 6. Schedule of pending actions

- 8) Click **Close** to close the dialog window once the operation is complete (see Figure 7).

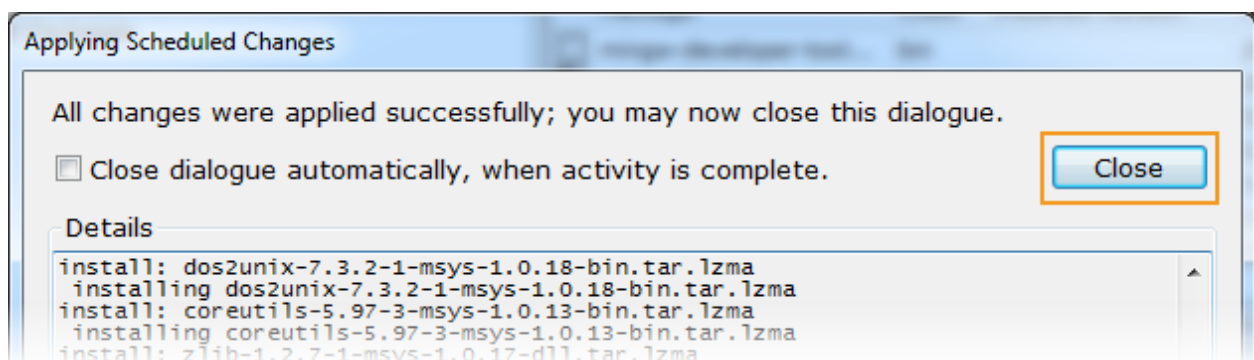


Figure 7. Applying scheduled changes

- 9) Navigate to C:\MinGW\msys\1.0 folder and launch the MinGW terminal by running `msys.bat` to create `home/<user_name>` folder.
- 10) Copy the SDK to MinGW `home/<user_name>` folder, as shown in Figure 8.

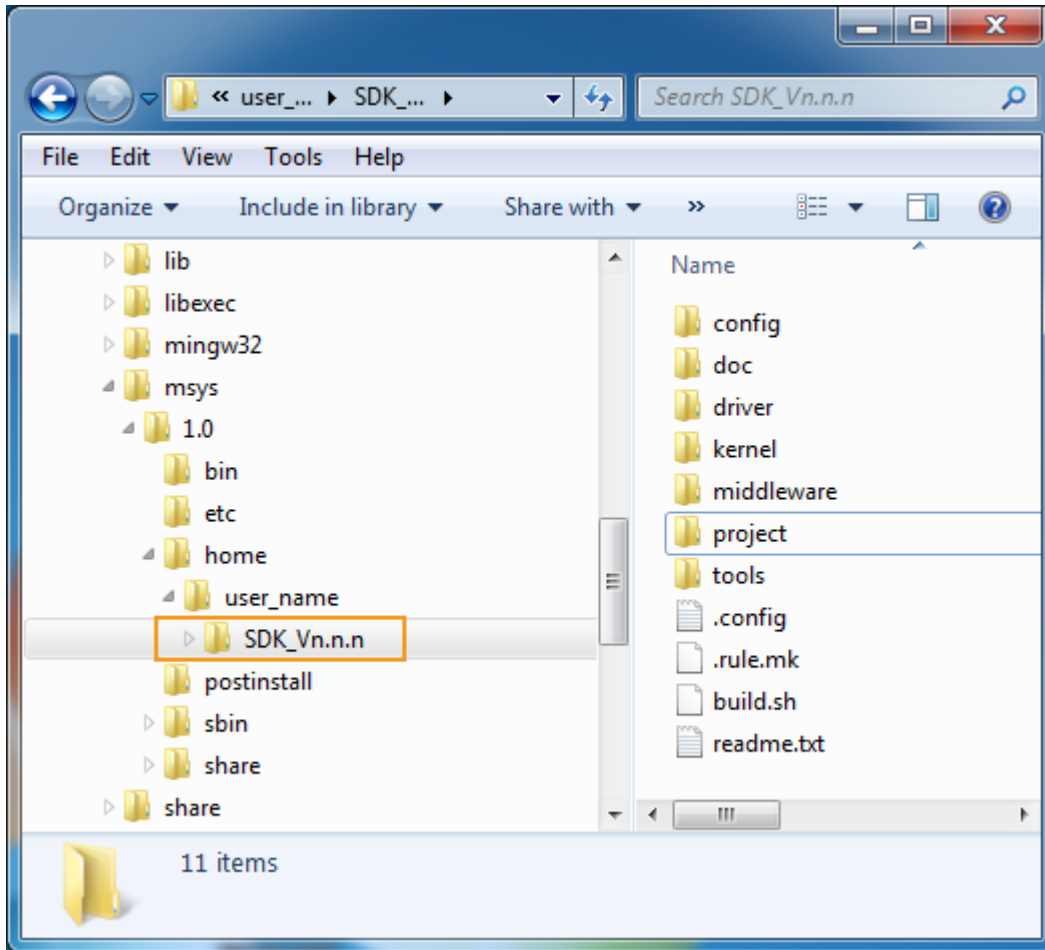


Figure 8. MinGW folder structure

- 11) Download ARM-GCC-win32 from [here](#).
 - a) Create a new folder named `win` under `<sdk_root>/tools/gcc/`.
 - b) Unzip the content of `gcc-arm-none-eabi-4_8-2014q3-20140805-win32.zip` to `<sdk_root>/tools/gcc/win/` folder.
 - c) Rename the unzipped `gcc-arm-none-eabi-4_8-2014q3-20140805-win32` folder to `gcc-arm-none-eabi`, as shown in Figure 9.

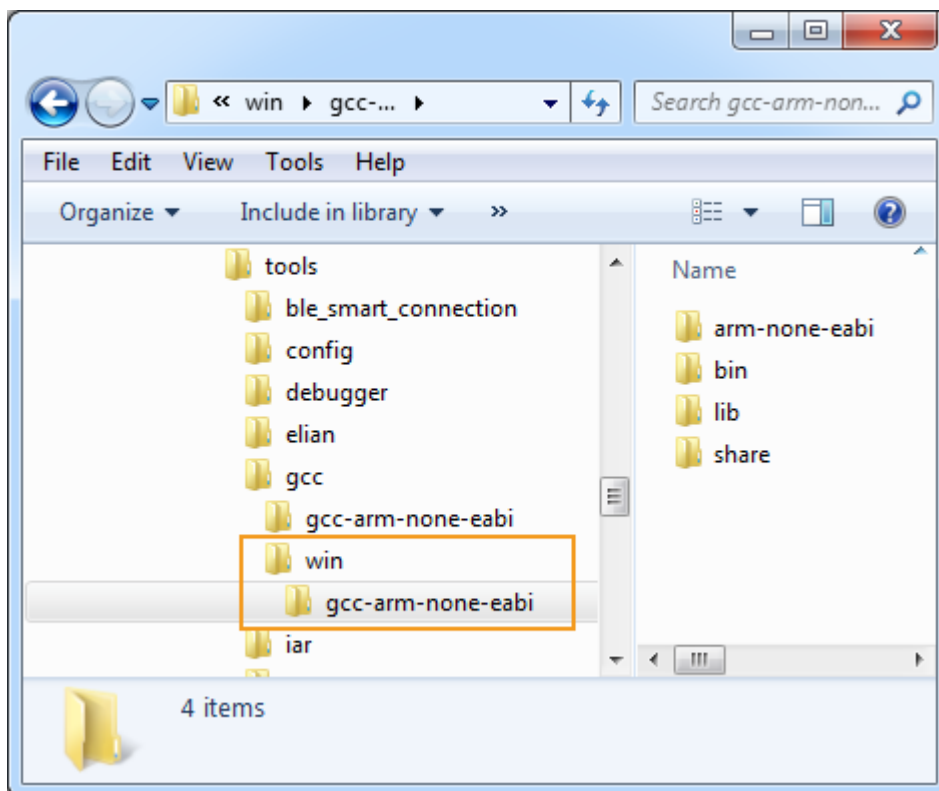


Figure 9. tools/gcc folder structure

- 12) Use `build.sh` to compile the project in MinGW terminal, as described in section 3, “Building the Project Using the SDK”.

2.2.2. Troubleshooting

Note the following caveats when building the project with MinGW on Windows OS.

- The folder name and file name in the Airoha IoT SDK should not contain ‘ ’, ‘[’ or ‘]’ characters.
- The project name should be less than 30 characters. Otherwise, a build error similar to the one shown below may occur due to the long path.

```
arm-none-eabi-gcc.exe: error:
../../../../../../../../out/mt2523_hdk/i2c_communication_with_EEPROM_dma/obj/project/mt2523_hdk/hal_examples/i2c_communication_with_EEPROM_dma/src/system_mt2523.o: No such file or directory
```

- The makefile in your project should not use any platform dependent commands or files, such as `stat` or `/proc/cpuinfo`.
- MinGW installation directory should be `C:\MinGW`. Otherwise, build errors may occur if MinGW installation path is very deep.
- To build an `httpd` project, export MinGW/bin path for `gcc` command, then install **msys-vim** package for `xxd` command by launching the **MinGW Installation Manager** (`mingw-get-setup.exe`) and choosing the **reinstall**, as shown in Figure 10.

```
export PATH=$PATH:/c/MinGW/bin:
```

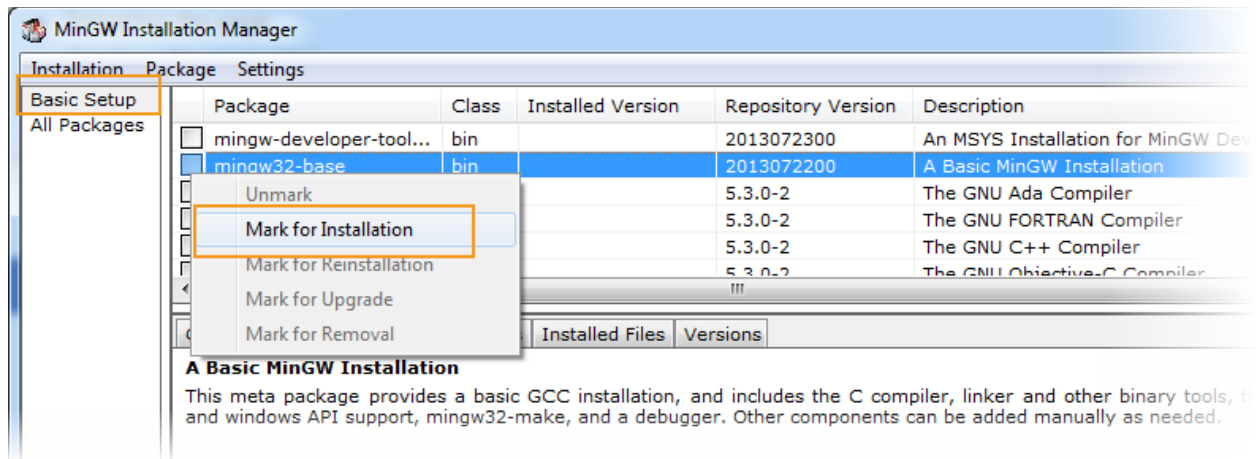


Figure 10. MinGW Installation Manager

- By default, the parallel build feature is enabled in `build.sh` to speed up the compilation. Disable the parallel build feature, if any unreasonable build error or system exception occurs.
 - To disable the parallel build feature for all modules, change the value “-j” of `EXTRA_VAR` to “-j1” in `build.sh`, as shown below.

```
platform=$(uname)
if [[ "$platform" =~ "MINGW" ]]; then
    export EXTRA_VAR=-j1
else
    export EXTRA_VAR=-j`cat /proc/cpuinfo |grep ^processor|wc -l`
fi...
```

- If build errors are due to a particular module’s parallel build, set the value `<module>_EXTRA` as “-j1” in `.rule.mk` to disable the parallel build for that particular module, as shown below.

```
OS_VERSION := $(shell uname)
ifneq ($(filter MINGW%, $(OS_VERSION)),)
    $(DRV_CHIP_PATH)_EXTRA := -j1
    $(MID_MBEDTLS_PATH)_EXTRA := -j1
    $(<module>)_EXTRA := -j1
endif...
```

Then, rebuild your project.

```
./build.sh mt7687_hdk <project_name> clean
./build.sh mt7687_hdk <project_name>
```

3. Building the Project Using the SDK

This section describes two methods on how to build a project:

- Build a project from the SDK root directory.
- Build a project from the project GCC configuration directory.

3.1. Building the project from the SDK root directory

Build the project using the script at <sdk_root>/build.sh. To find out more about the usage of the script, navigate to the SDK's 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>

Example:
./build.sh mt7687_hdk iot_sdk
./build.sh mt7687_hdk iot_sdk bl      (build with bootloader)
./build.sh clean                      (clean folder: out)
./build.sh mt7687_hdk clean          (clean folder: out/mt7687_hdk)
./build.sh mt7687_hdk iot_sdk clean (clean folder:
out/mt7687_hdk/iot_sdk)

Argument:
-f=<feature makefile> or --feature=<feature makefile>
  Replace feature.mk with another makefile. For example,
  the feature_example.mk is under project folder, -
f=feature_example.mk
  will replace feature.mk with feature_example.mk.

-o=<make option> or --option=<make option>
  Assign additional make options. 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>.

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

3.1.1. 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 based on the related configuration files under `<sdk_root>/config/project/<board>/<project>` folder. The console output is shown below.

```
=====
Available Build Projects:
=====
    mt7687_hdk
    iot_sdk
    ...
```

3.1.2. Build the project

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

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

The output files will be placed under `<sdk_root>/out/<board>/<project>/` folder.

For example, to build a project on the LinkIt 7687 HDK, run the following build command.

```
./build.sh mt7687_hdk iot_sdk
```

The output files will be placed under `<sdk_root>/out/mt7687_hdk/iot_sdk/` folder.

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 for 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
```



Note, the bootloader download mechanism is slightly different on LinkIt 2523 HDK from that on the LinkIt 76x7 HDK. On LinkIt 2523 HDK the bootloader image is combined with the project image file under a new image file named as `flash.bin`.

To build the project on the LinkIt 2523 HDK:

```
./build.sh mt2523_hdk iot_sdk_demo bl
```

The output image file of the project and the bootloader, along with the merged image file `flash.bin`, will be placed under `<sdk_root>/out/mt2523_hdk/iot_sdk_demo` folder.

To build the project on the LinkIt 76x7 HDK:

```
./build.sh mt7687_hdk iot_sdk bl
```

The output image file of the project and the bootloader will be placed under `<sdk_root>/out/mt7687_hdk/iot_sdk` 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>/<project>` folder.

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

3.2. Building the project from the project GCC 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 `iot_sdk` is built by the project makefile under `<sdk_root>/project/mt7687_hdk/apps/iot_sdk/GCC`.
- 3) Navigate to the example project's location.

```
cd <sdk_root>/project/mt7687_hdk/apps/iot_sdk/GCC
```

- 4) Run the make command.

```
make
```

The output folder is defined under variable `BUILD_DIR` in the Makefile located at `<sdk_root>/project/mt7687_hdk/apps/iot_sdk/GCC`:

```
BUILD_DIR = $(PWD)/Build  
PROJ_NAME = mt7687_iot_sdk
```

A project image `mt7687_iot_sdk.bin` is generated under `<sdk_root>/project/mt7687_hdk/apps/iot_sdk_demo/GCC/Build`.

4. Folder Structure

This section 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 11.

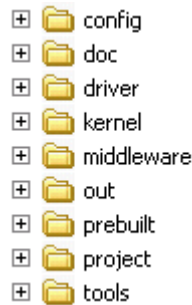


Figure 11. The SDK package folder structure

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** — contains makefiles and compiler configuration files to compile a project image.
- **doc** — contains the SDK related documentation, such as developer guides and the SDK API references.
- **driver** — contains common driver files, such as board drivers, peripheral and CMSIS-CORE interface drivers.
- **kernel** — contains the FreeRTOS and system services for exception handling and error logging.
- **middleware** — includes software features for HAL and FreeRTOS, such as Bluetooth/Bluetooth Low Energy 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** — contains pre-configured examples and demo projects using GNSS, FOTA, HAL, and more.
- **tools** — includes tools to compile projects. The contents of this folder are not in the main package.

To enable building several different projects simultaneously, each project's output files are placed under the corresponding `<sdk_root>/out/<board>/<project>/` folder. Figure 12 shows the folder structure under the out folder after project `iot_sdk` is built on the LinkIt 7687 HDK.

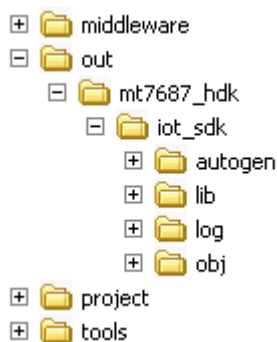


Figure 12. The folder structure of the project's output files

In this case, the target folder path is `<sdk_root>/out/mt7687_hdk/iot_sdk/`. The content of this folder is shown in Figure 13.

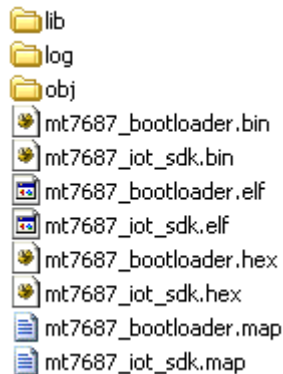


Figure 13. The generated files under example project

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 `mt7687_iot_sdk.bin`. The variable `PROJ_NAME` is defined in `Makefile`, see section 5, “Makefiles”.
 - `bootloader image` — the naming rule of the image file is `<board>_bootloader.bin`, which in this case is defined as `mt7687_bootloader.bin`.
 - `flash image` — this file is only generated for LinkIt 2523 development board, see section 3.1.3, “Build the project with the “b1” ”. This file is the merged image file from the bootloader image and the project image. It’s used with the flash tool to update the firmware of the board.
- `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.

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/iot_sdk_demo/GCC/` folder, named as `Makefile` (see Figure 14). 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.

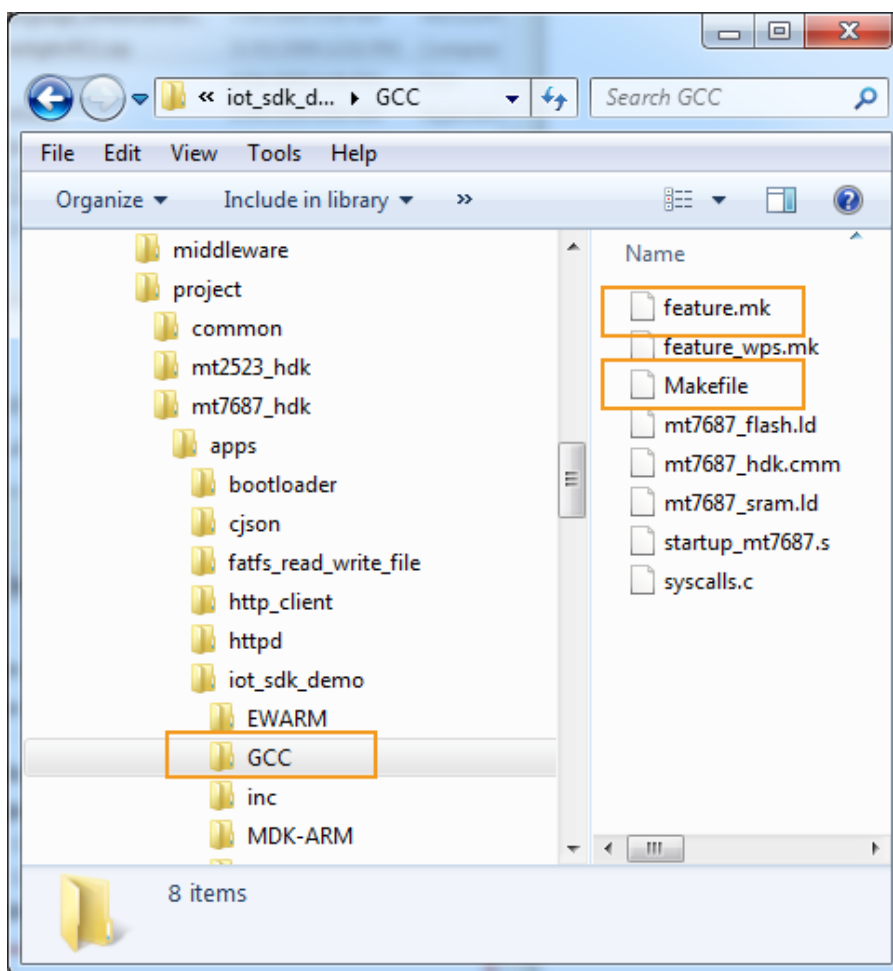


Figure 14. The location of the project's feature configuration file and makefile

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/iot_sdk/GCC/feature.mk` (see Figure 14). Users 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`.

```
IC_CONFIG          = mt7687
BOARD_CONFIG       = mt7687_hdk
MTK_FW_VERSION     = mt7687_fw
MTK_HAL_LOWPOWER_ENABLE = y
MTK_BSPEXT_ENABLE = y
MTK_IPERF_ENABLE   = y
MTK_PING_OUT_ENABLE = y
MTK_WIFI_TGN_VERIFY_ENABLE = y
MTK_MINICLI_ENABLE = y
MTK_SMTCN_ENABLE   = y
MTK_WIFI_WPS_ENABLE = n
MTK_WIFI_REPEATER_ENABLE = n
```

- 2) `chip.mk` is located at `<sdk_root>/config/board/<board>/chip.mk` (see Figure 15) 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:
 - 5) Configures the common settings of the chip.
 - 6) Defines the `CFLAGS` macro.
 - 7) Sets the include path of the kernel and the driver header file.
 - 8) Sets the module folder path that contains the Makefile.



Figure 15. The path to the chip.mk

The `chip.mk` includes the `CFLAGS`, include paths and module folder paths, as shown below.

```
PRODUCT_VERSION=7687

# Default common setting
MTK_SYS_TRNG_ENABLE    ?= y
MTK_MINISUPP_ENABLE    ?= y
MTK_BSP_INBAND_NEW_ENABLE ?= y
MTK_USING_NVRAM        ?= y
...

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

...

#Include Path
COM_CFLAGS += -I$(SOURCE_DIR)/kernel/rtos/FreeRTOS/inc
COM_CFLAGS += -I$(SOURCE_DIR)/kernel/rtos/FreeRTOS/Source/include
COM_CFLAGS += -I$(SOURCE_DIR)/driver/CMSIS/Device/MTK/mt7687/Include
COM_CFLAGS += -I$(SOURCE_DIR)/driver/CMSIS/Include
COM_CFLAGS += -I$(SOURCE_DIR)/driver/chip/mt7687/include
COM_CFLAGS += -I$(SOURCE_DIR)/driver/chip/mt7687/inc
COM_CFLAGS += -I$(SOURCE_DIR)/driver/chip/inc

...

#Middleware Module Path
MID_TFTP_PATH    = $(SOURCE_DIR)/middleware/MTK/tftp
MID_FOTA_PATH    = $(SOURCE_DIR)/middleware/MTK/fota
MID_LWIP_PATH    = $(SOURCE_DIR)/middleware/third_party/lwip
MID_CJSON_PATH   = $(SOURCE_DIR)/middleware/third_party/cjson
MID_DHCPD_PATH   = $(SOURCE_DIR)/middleware/third_party/dhcpd
MID_HTTPCLIENT_PATH = $(SOURCE_DIR)/middleware/third_party/httpclient
MID_MBEDTLS_PATH = $(SOURCE_DIR)/middleware/third_party/mbedtls
MID_MINICLI_PATH = $(SOURCE_DIR)/middleware/MTK/minicli
MID_MINISUPP_PATH = $(SOURCE_DIR)/middleware/protected/minisupp

...
```

- 3) `board.mk` is located at `<sdk_root>/config/board/<board>/board.mk` (see Figure 16) and defines the extra include paths to the header files related to the development board.



Figure 16. The Path of board.mk

```
CFLAGS += -I$(SOURCE_DIR)/middleware/MTK/nvram/inc
CFLAGS += -I$(SOURCE_DIR)/driver/board/mt76x7_hdk/inc
```

6. Adding a Module to the Middleware

This section provides details on adding a module or a custom defined feature into an existing project. Added module will be 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 `iot_sdk` project on the LinkIt 7687 development board. This section can also be applied to the LinkIt 2523 HDK.

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 17.

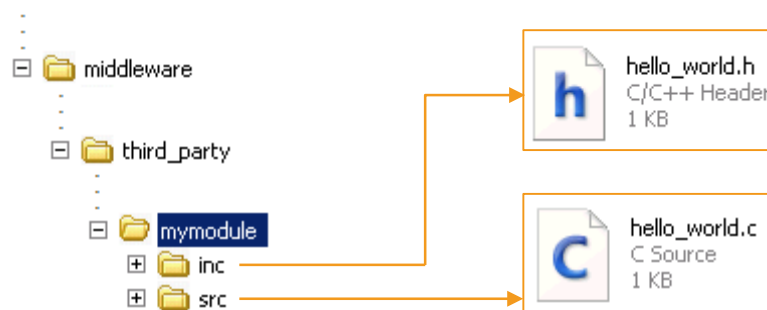


Figure 17. Place module source and header files under the module folder

The sample source code `hello_world.c` and header file `hello_world.h` with their corresponding locations 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 18) 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.

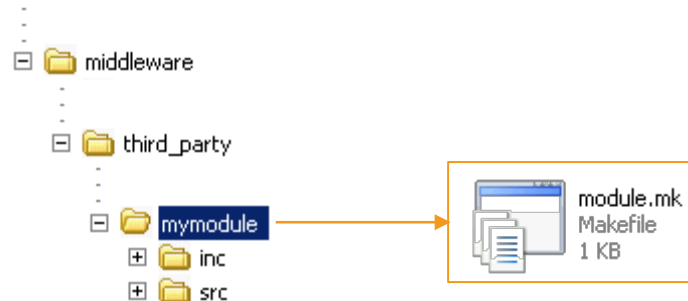


Figure 18. Create a module.mk under module folder

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 19.

Most of the dependency rules and definitions in the file are written for a common use. 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.

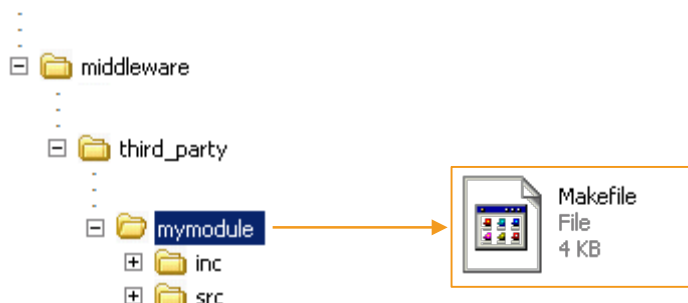


Figure 19. Create a Makefile under module folder

```

SOURCE_DIR = ../../..
BINPATH    = ~/gcc-arm-none-eabi/bin
PROJ_PATH  = ../../../project/mt7687_hdk/apps/iot_sdk/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/mt7687_hdk/apps/iot_sdk/Makefile`.

In the Makefile, there's a section with rules defined as `include XXXX/module.mk`. This section defines the modules required by the project when 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`.

You've successfully added a module to an existing project. The next section describes how to create a project.

Note, starting from Airoha IoT SDK v4.1.0, a new method to add a module to the build flow of the project is introduced 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 HAL module, however, in SDK v4.1.0 modify the include path, such as `"include $(SOURCE_DIR)/driver/chip/mt2523/module.mk"`.
- Add required libraries in `LIBS`.



7. Create a Project

This section provides details on creating your own project named `my_project` on LinkIt 7687 development board using `iot_sdk` project as a reference.

7.1. Using an existing project

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

- 1) Copy the folder `<sdk_root>/project/mt7687_hdk/apps/iot_sdk/` to a new directory `<sdk_root>/project/mt7687_hdk/apps/my_project/`.

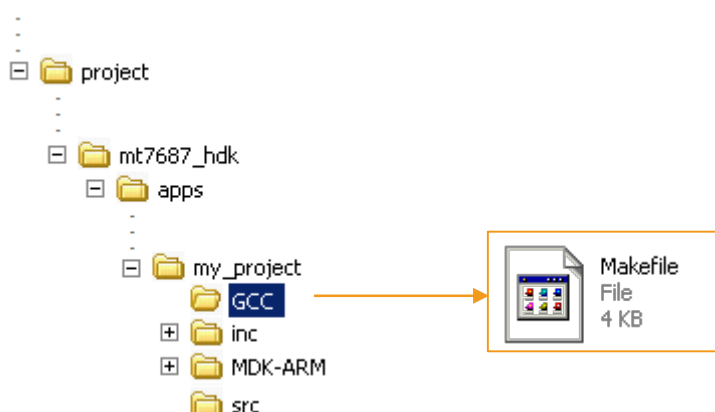


Figure 20. Modify the Makefile under the GCC folder of my_project

- 2) Modify the following settings defined in the `<sdk_root>/project/mt7687_hdk/apps/my_project/GCC/Makefile` under the GCC folder (see Figure 20) `my_project`, as shown below:

- `SOURCE_DIR`: the path to `<sdk_root>`.
- `PROJ_NAME`: project name.
- `APP_PATH`: project path.

```
...
SOURCE_DIR = ../../../../..
BINPATH    = ~/gcc-arm-none-eabi/bin
PWD        = $(shell pwd)
DATETIME   = $(shell date --iso=seconds)
V          = 0
...
# Project name
PROJ_NAME   = mt7687_my_project
PROJ_PATH   = $(PWD)
OUTPATH     = $(PWD)/Build

APP_PATH    = project/mt7687_hdk/apps/my_project
APP_PATH_SRC = $(APP_PATH)/src
```

...

7.2. Removing a module

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

To delete a module:

- 1) Open the project Makefile from
`<sdk_root>/project/mt7687_hdk/apps/my_project/GCC/Makefile.`
- 2) Locate the module include list of the project and remove any unwanted module by deleting or commenting out the corresponding include statement.

```
...
#####
####
#
# 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
...
```

7.3. User-defined source and header files

User defined project source and header files should be placed under the `src` and the `inc` folder respectively, as shown in Figure 21.

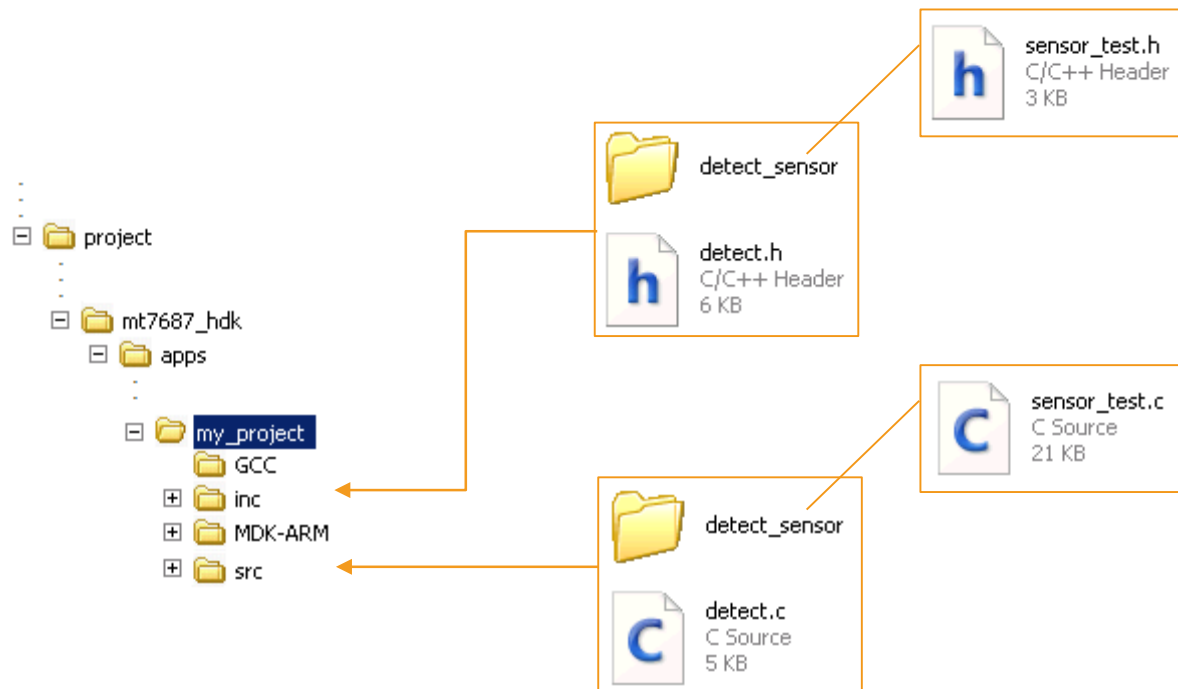


Figure 21. Project source and header files under the project folder

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).

<sdk_root>/project/mt7687_hdk/apps/my_project/GCC/Makefile

```

...
C_FILES += $(APP_PATH_SRC)/main.c \
            $(APP_PATH)/GCC/syscalls.c \
            $(APP_PATH_SRC)/detect.c \
            $(APP_PATH_SRC)/detect_sensor/sensor_test.c

CXX_FILES +=
...
CFLAGS += -I$(SOURCE_DIR)/kernel/service/inc
CFLAGS += -I$(SOURCE_DIR)/$(APP_PATH)/inc/detect_sensor

CXXFLAGS +=
...

```

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/mt7687_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's `<sdk_root>/out/mt7687_hdk/my_project/obj/project/mt7687_hdk/apps/my_project/src/`.

The location of the image file, object and dependency files after the example project is built, as shown in Figure 22.

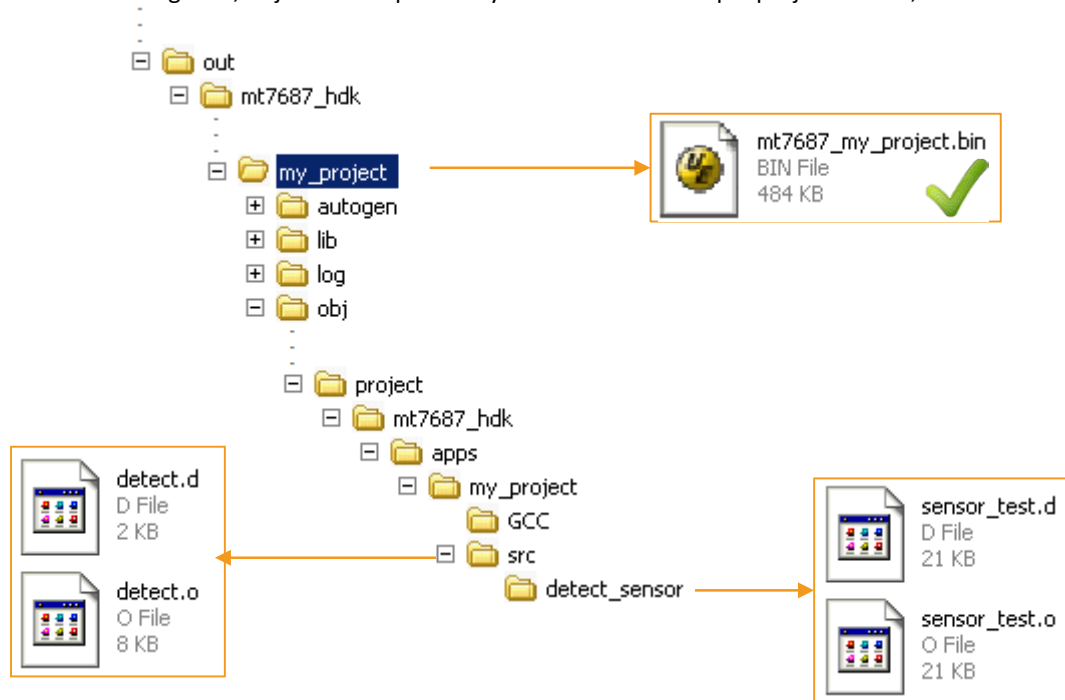


Figure 22. Output image file, object files and dependency files after project is successfully built

7.4.1. Troubleshooting

When a build process is failed 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.

```
...
TOTAL BUILD: FAIL
```