**GCC C-MAKE FRAMEWORK**

Author : Long Dao | <https://louisvn.com>

***USER MANUAL***

[01 Use special inputs 1](#_Toc170077372)

[02 Debug 2](#_Toc170077373)

[03 Use the Code Testing Tool (utest) 2](#_Toc170077374)

[04 Use the Code Coverage Measurement (CCOV) tool 5](#_Toc170077375)

**\*\*\***

# Use special inputs

You can pass data into your program through special inputs such as definitions (**USER\_DEFS**), arguments (**VAR\_ARGS**), or through environment variables (**USER\_ENVS**).

* + **CCDEFS** | **ASDEFS** : A definition of "**-D...**" will be created on **CCOPTS** | **ASOPTS** and is affiliated with **VSCode**. Definition variables need to be declared in **makefile** with syntax : **DEF\_<VAR\_NAME>**.
    - Ex : DEF\_HELLO := “Hello world” > CCDEFS:= HELLO
  + **VAR\_ARGS** : The arguments are separated by spaces. In case the inside of an argument contains spaces, it needs to enclose **quotation marks**.
    - Ex : VAR\_ARGS := Hello\_world “Hello world” Hello” “world
  + **USER\_ENVS** : Environment variables need to be defined inside **makefile**. Then add the variable name to the **USER\_ENVS**.
    - Ex : VAR\_ENV := Hello world > USER\_ENVS := VAR\_ENV
  + In addition, special inputs may contain special ASCII characters such as the following:

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | | |
| **CCDEFS** | **ASDEFS** | **VAR\_ARGS** | **USER\_ENVS** |
| \\ | \ | \ | \\ |
| \” | “ | “ | \” |
| \n | [ENTER] | \n | \n |
| \t | [TAB] | \t | \t |
| “a b” | a b | a b | “a b” |

# Debug

There are 2 solutions to debug a project after compilation has been completed (.exe file is available).

**Using the command line (CLI)**

* + Run the "**make debug**" command and perform debugging using **GDB**.

**Using the VSCode interface (GUI)**

* + Run the "**make vsinit**" command to ensure you have updated all necessary environment variables, definitions, paths to the VSCode interface.
  + On the **VSCode** interface, select the "**Run and Debug** (**Ctrl + Shift + D**)" tab. Then select the "**CMFramework Debug**" task.

A screen shot of a computer

Description automatically generated

* + Finally, click on the "**Play** (**F5**)" button in the image above to perform debugging.
  + **Note**:You need to understand the commands used for debugging GDB, and/or how to debug on VSCode before using this feature.

# Use the Code Testing Tool (utest)

* + To use this tool, you first need to add the path of this library to your makefile.
  + Open the "**user\_cfg.mk**" file. In the variable "**SRC\_DIRS**" used to scan and retrieve source code files (.c) and the variable "**INC\_DIRS**" used to scan and retrieve header files (.h), you need to add the following path to both variables: "**$(TOOL\_DIRS)/lib**".

A screenshot of a computer program

Description automatically generated

* + Specifically, **$(TOOL\_DIRS)** is the path to the "**tools**" directory of the Framework. Then"**/lib**" is for accessing the "**lib**" directory inside. The **"\**" sign to join the line below. You can write values that are the same line and separated by "**spaces**".
  + After adding the path, you run the "**make vsinit**" command to update the path to VSCode, thereby being able to prompt you better.
  + Go back to your source file, just include "**utest.h**" and experience.

A screen shot of a computer

Description automatically generated

* + To run tests, you need to create test functions in the given format.
    - Syntax: **FuncTest( name\_test ) { /\* body \*/ }**
    - In particular, **"name\_test**" is the name of the function specified by yourself. The upper jaw is also equivalent to :  **void name\_test ( void ) { /\* body \*/ }** . However, writing the correct syntax will make it easier for readers to understand.

A screenshot of a computer

Description automatically generated

* + In the above example, we have a test function called "**Test\_001**".
  + The purpose of testing is created to check the correctness of the source code. So, to know if your testing is right or wrong, you need to have the test conditions.
  + For example, you want to prove that "1+1=2" and "2+2=4" ... Those are the test conditions. Then, "**UT\_Assert**" is the command that helps you check the conditions.

A screenshot of a computer code

Description automatically generated

* + In the example above, I have 3 conditions to test. As long as one of the conditions is wrong, the whole test function will result in failure. So, we can say that "**Test\_001**" failed because the 3rd test condition was wrong.
  + After creating the test functions, you need to add these functions to a container to execute it.

A black background with white lines

Description automatically generated

* + The above 2 macros will create a container, you just need to add the test functions in the syntax: **UT\_AddTest( name\_test, "Brief for name\_test" )**
  + Where "**name\_test**" is the name of the test function that was created earlier."**Brief for name\_test**" is a title you set yourself.

A computer screen shot of a message

Description automatically generated

* + If you have multiple test functions, you can simply add more **"UT\_AddTest**" commands. Note that there is **no** "**;**" after these commands. At the same time, this container is not written in any one function. Tests that don't currently want to execute, just discard or note the inside of the container.
  + After adding to the container, you need to make sure that the test functions are declared before using it.

A black background with white lines

Description automatically generated

* + To execute the tests inside the container one by one, you need to use the following command : **UT\_RunTests()**

A screenshot of a computer

Description automatically generated

* + Finally, you run the "**make**" command to compile and run the executable. Then, run **"make report**" to export the result to an html file and view it in a web browser. This html file is saved in the "**doc**" folder of the project.

A screenshot of a computer

Description automatically generated

A screenshot of a test

Description automatically generated

A screenshot of a computer code

Description automatically generated

* + As you can see, Test Condition 3 was wrong at line 55 in the source code file.

# Use the Code Coverage Measurement (CCOV) tool

* + Code coverage is an indispensable tool for source code testing, it shows how much coverage of the testing process is on the source code.
  + To run the coverage code, go to "**user\_cfg.mk**", add the variable **"$(DEV\_DIR)**" to the paths of "**SRC\_DIRS**". That allows makefile to scan source code files in the "**dev**" folder.

A screenshot of a computer screen

Description automatically generated

* + The "**dev**" directory is the directory used to measure CCOV. All functions inside all files in this directory are measured.
  + Next, fix the value of the variable **"RUN\_CCOV**" to "**on**" to allow the use of CCOV.

A close up of a report

Description automatically generated

* + After that, you need to create a source code file inside **"dev**" to write the functions to measure CCOV.

A screenshot of a computer program

Description automatically generated

* + The above example created a "**checkpos**" function to check for positive integers (n > 0) in the "**dev.c**" file located inside the "**dev**" directory.

A computer code with numbers and symbols

Description automatically generated

* + Then, I created 2 conditions for it to scan both the "**if**" and "**else**"branches inside the **"checkpos**" function. From there, my CCOV index will be completely covered.
  + Finally, you run "**make**" to compile and execute the source code. Then run **"make** **report**" to generate reports including test results and CCOV. They are all stored in the project's "**doc**" folder.

A screen shot of a computer

Description automatically generated

A screenshot of a test

Description automatically generated

A table with numbers and text

Description automatically generated A close-up of a computer code

Description automatically generated

**\*\*\***

**THE END**