**Unit test**

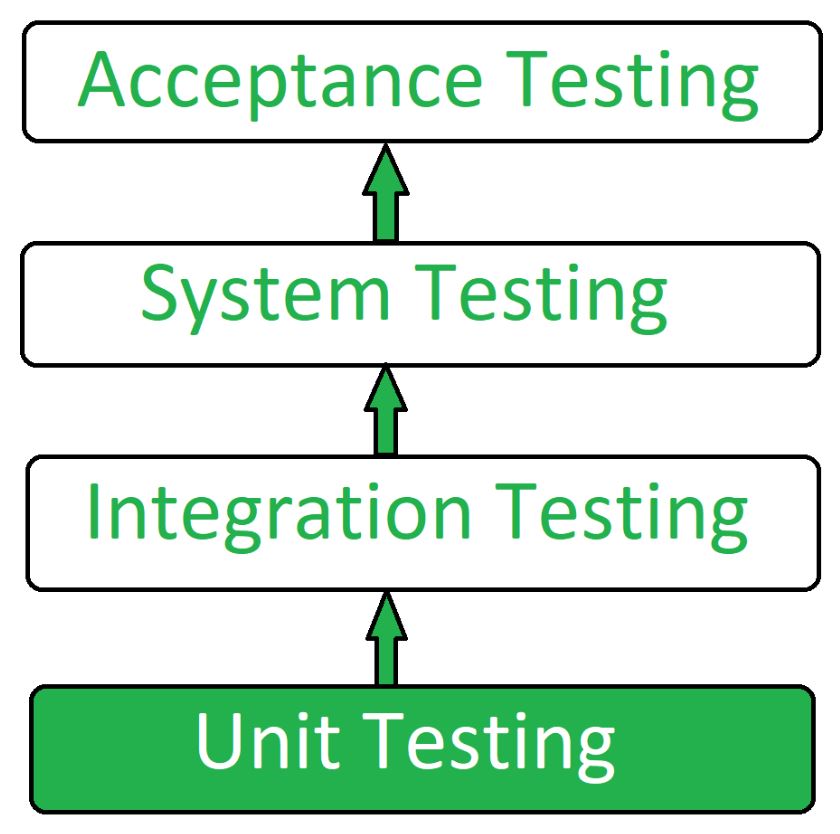
Introduction

**Why?**

* Make the code more reliable, can be reused
* Cost less when we recognize bug early
* Easier to fix whenever changes are made

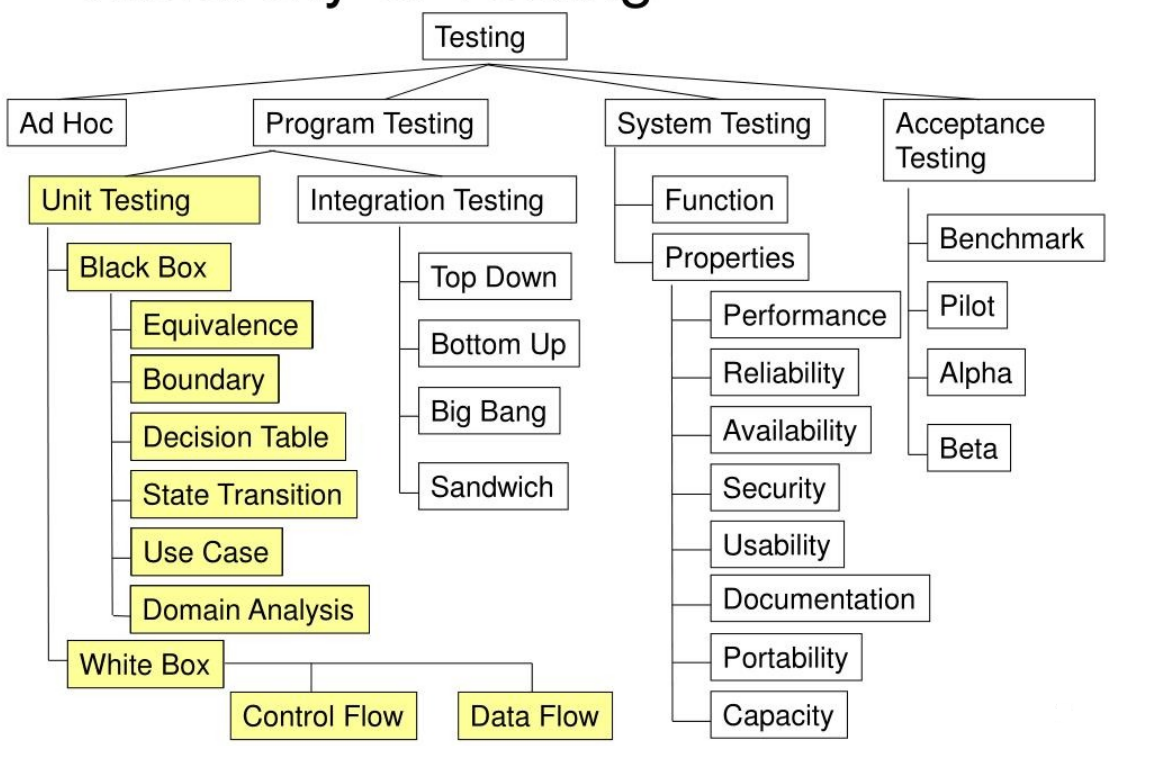
**Note!**

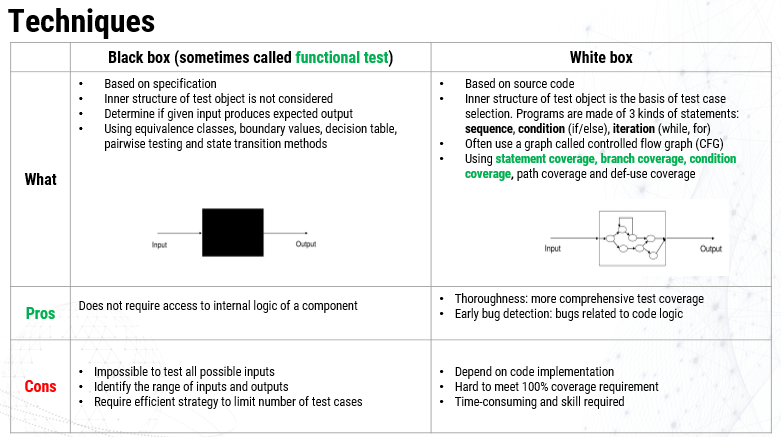
* Test case is independent. Do not call another test case when in a case
* Always checking every single module independently
* Make sure test case names are understandable, unique
* Whenever changes are made, make sure all above test case are tested again
* Bug need to be handle before going to next part
* Do not make ALL test case, only focus on things that can impact the system
* Also need test case for system performance



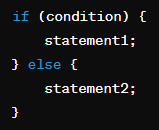
Unit Testing:

* Test focusing on the **smallest units of code**, such as (functions, procedures, subroutines, subprograms, methods, classes)
* Test component in **isolation from the rest** of the system and in a controlled environment (appropriately chosen input data, take guidance from component design)





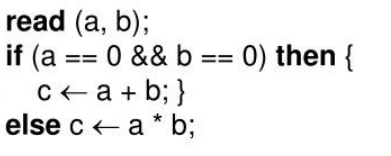
In this section, we focus on **Statement coverage (C0)** and **Branch coverage (C1)** in White box testing



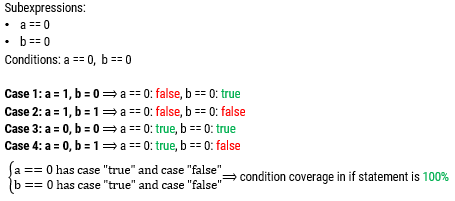
For example we have this function, **Statement coverage** means we make sure that statement1 and statement2 run at least 1 time (do not relevant to condition)

With **Branch coverage**, condition here have 2 branch is true and false, we need to make sure that 2 branchs are executed (do no relevant to statement)

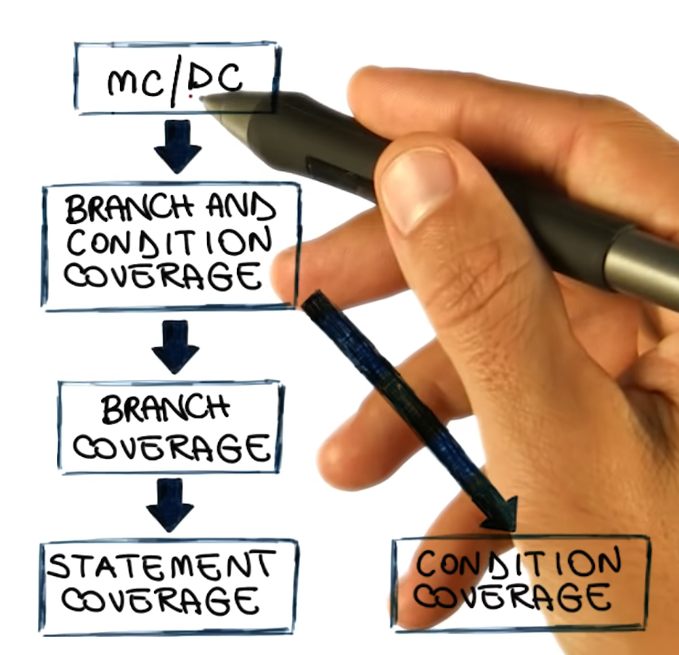
But what happen if ‘condition’ on above example have more than 1 condition:



* **Condition coverage (C2)**



**Modified Condition Decision Coverage (MCDC)** [**https://www.youtube.com/watch?v=DivaWCNohdw**](https://www.youtube.com/watch?v=DivaWCNohdw)



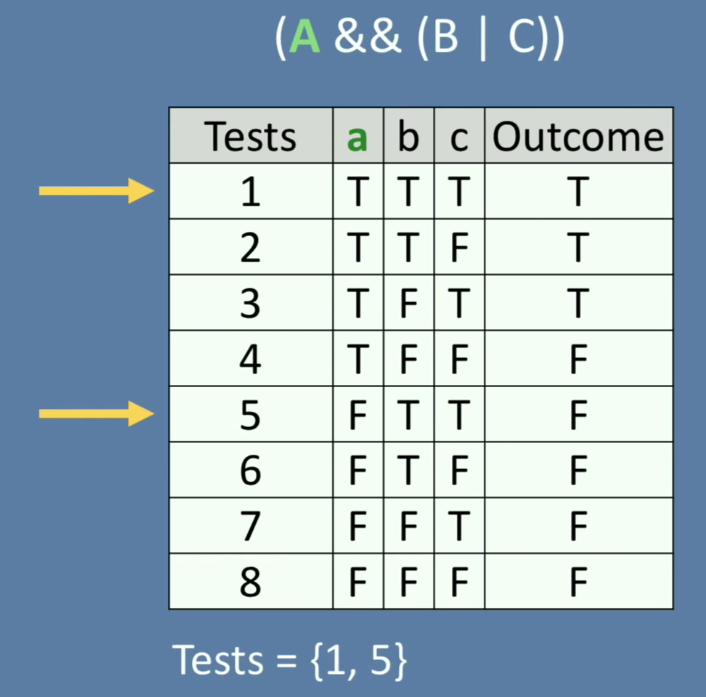
Why?

This includes:

* Branch coverage
* Statement coverage
* Conditions coverage
* Every condition independently affect the outcome (other changes doesn’t affect)

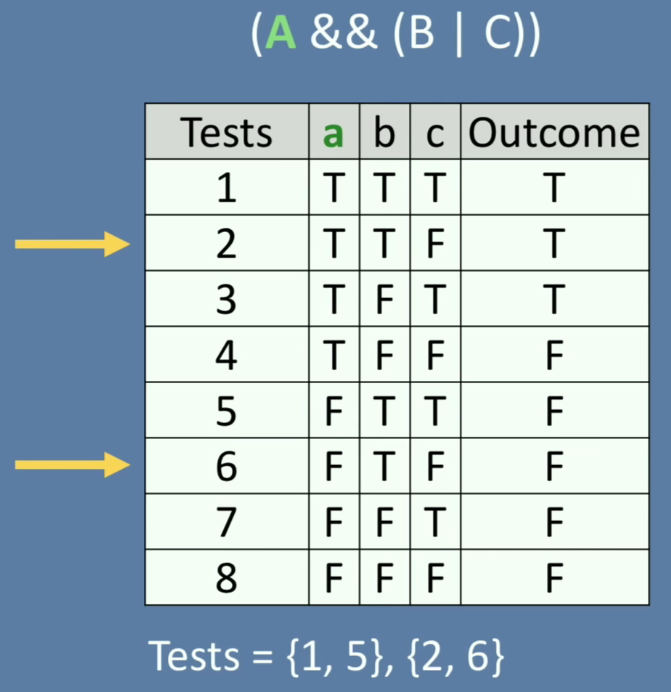
But can decrease the amount of test from 2n to n+1 with n is the number of conditions

**For example:**



In this example we need to check the independency of A by searching on truth table. Let start with test 1, and we change A from T to F to see if the outcome changes also. And luckily, test 5 show that if we change A -> Outcome change to F. This means B and C changes is not neccesary (these changes are reduntdant -> make our test becomes more complex)

Doing the same way we get test 2 and test 6; test 3 and test 7



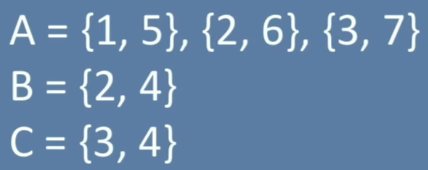
But test 4 and test 8 do not match the rule (outcome is the same)

* With A we got test {1,5,2,6,3,7} that can change the system behavior

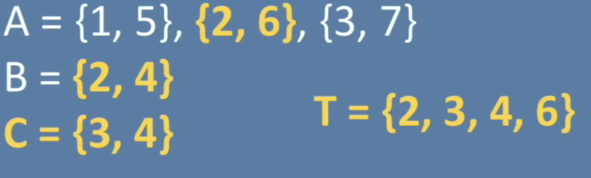
Coming to B, we start with fair test 1 and test 3 -> not match

+ Fair 2 and 4 -> match. The rest cases do not match

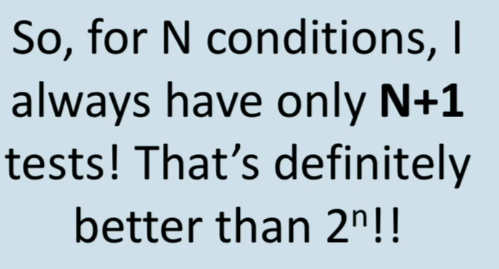
Coming to C, we only get fair 3 and 4



We just need AT LEAST 1 test for each condition -> we got :



(Why don’t choose {1,5} or {3,7}, here we can pick {3,7} because 3 is in C test case already -> less case. Choose {1,5} is still ok but it increases 1 more case in total)

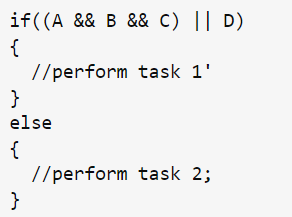


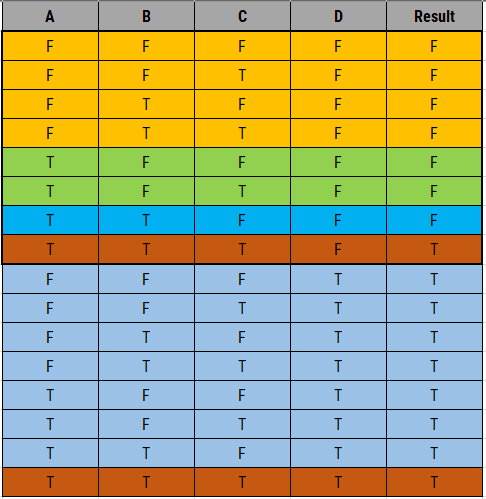
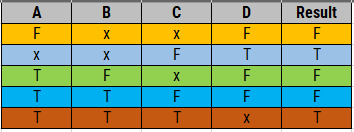
<https://www.youtube.com/watch?v=bwtALQVx86w>

<https://www.youtube.com/watch?v=DivaWCNohdw&list=PLAwxTw4SYaPkoQFThzsc9e7Fe3QV_KJCs&index=69>

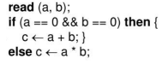
If there is no independent condition -> We need to comeback to C0,C1 and C2

Use to remove redundancy in Condition coverage (C2) -> only test the important combination condition

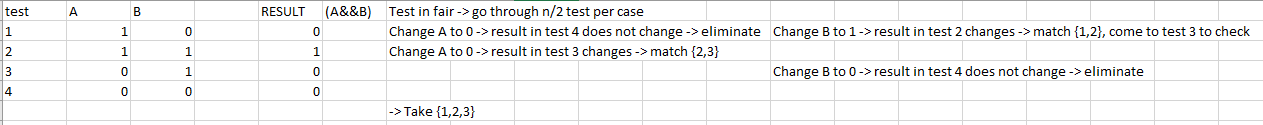


 -> 

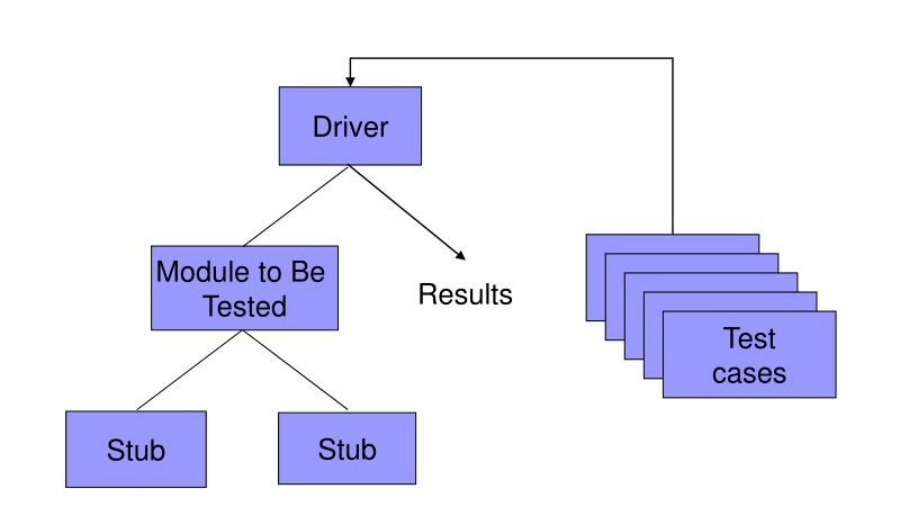
**Example:**



This is how we brainstorm



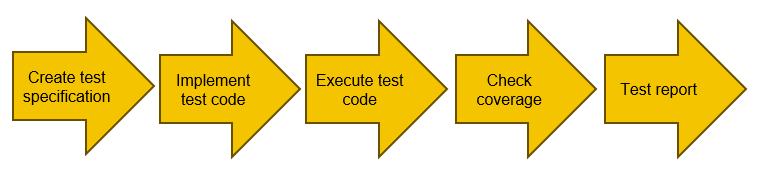
**Unit test procedures**



Start with test cases above, after filter necessary test cases(is the core of Unit Test, written by tester), we take it to **Driver**: that can link module to be tested, stub and test cases

* What is stub (stub function/classes,…): Imagine our program need to be tested but some part of it does not complete/already tested, so we need to stimulate the behavior of that part (return an output with specific input). For example the module multiple\_two\_number(int a, int b) does not complete, we can use stub to take input with a = 2, b=3 and output is 6 without going through calculation inside that function.

**How to perform UT?**



Investigate makefile first:

Dựa trên cấu trúc thư mục và Makefile bạn đã cung cấp, chương trình thực thi UT.exe được tạo ra từ việc biên dịch và liên kết các tệp nguồn C sau:

1. ut\_main.c trong thư mục gốc.
2. Các tệp nguồn C trong thư mục framework và các thư mục con của nó (include và util). Framework là nơi các mã nguồn về test được chứa
3. Các tệp nguồn C trong thư mục ut và các thư mục con của nó (config, DMA, include, stub).: Trong thư mục ut chứa các file và các patterns sắp được test

Các tệp nguồn này được biên dịch thành các tệp đối tượng và sau đó được liên kết lại để tạo ra UT.exe. Quá trình này được tự động hóa bởi Makefile.

Vui lòng lưu ý rằng, nếu có thêm các tệp nguồn khác được thêm vào Makefile sau này, chúng cũng sẽ được biên dịch và liên kết vào UT.exe. Hy vọng điều này giúp bạn hiểu rõ hơn! Nếu bạn có thêm câu hỏi, đừng ngần ngại hỏi.

Phần 2 của makefile định nghĩa cấu hình cho việc kiểm tra độ phủ mã nguồn (code coverage) trong quá trình kiểm thử phần mềm. Dưới đây là một số điểm chính:

* **Thư mục Đối tượng và Phụ thuộc**: OBJDIR\_COV và DEPDIR\_COV được đặt để lưu trữ các tệp đối tượng và phụ thuộc cho việc kiểm tra độ phủ.
* **Thư mục Bao gồm**: INCDIR\_COV chứa các đường dẫn đến thư mục bao gồm cần thiết cho việc biên dịch.
* **Nguồn và Đối tượng**: C\_DRV\_SRC\_COV và DRV\_OBJS\_COV xác định các tệp nguồn và tệp đối tượng tương ứng cho việc kiểm tra độ phủ.
* **Cờ Độ Phủ**: COVERAGE\_FLAG được thêm vào để kích hoạt các tùy chọn liên quan đến việc kiểm tra độ phủ trong quá trình biên dịch.
* **Tệp Đầu ra**: OUTPUT\_COV là tên của tệp thực thi được tạo ra sau khi liên kết các tệp đối tượng.
* **Quy trình Kiểm tra Độ Phủ**: Các quy tắc coverage và $(OUTPUT\_COV) định nghĩa cách tạo ra tệp thực thi kiểm tra độ phủ.
* **Dọn dẹp**: Phần clean định nghĩa cách xóa các tệp và thư mục tạm thời được tạo ra trong quá trình biên dịch và liên kết.

Đây là những thành phần cơ bản của một makefile được sử dụng để tự động hóa quá trình kiểm tra độ phủ mã nguồn trong phát triển phần mềm.

**In ut\_main.c**

**Variable for c0c1 checking process**

/\* C0C1 \*/

bool DMA\_API\_C0C1\_line\_activate\_list[SIZE\_OF\_LIST];

bool data\_format\_convert\_C0C1\_line\_activate\_list[SIZE\_OF\_LIST];

bool pre\_check\_config\_C0C1\_line\_activate\_list[SIZE\_OF\_LIST];

bool check\_image\_and\_square\_C0C1\_line\_activate\_list[SIZE\_OF\_LIST];

bool check\_noise\_C0C1\_line\_activate\_list[SIZE\_OF\_LIST];

bool crop\_image\_and\_divide\_C0C1\_line\_activate\_list[SIZE\_OF\_LIST];

uint8\_t DMA\_API\_C0C1\_list[SIZE\_OF\_LIST] = {0};

uint8\_t data\_format\_convert\_C0C1\_list[SIZE\_OF\_LIST] = {0};

uint8\_t pre\_check\_config\_C0C1\_list[SIZE\_OF\_LIST] = {0};

uint8\_t check\_image\_and\_square\_C0C1\_list[SIZE\_OF\_LIST] = {0};

uint8\_t check\_noise\_C0C1\_list[SIZE\_OF\_LIST] = {0};

uint8\_t crop\_image\_and\_divide\_C0C1\_list[SIZE\_OF\_LIST] = {0};

uint32\_t DMA\_API\_C0C1\_count = 0;

uint32\_t data\_format\_convert\_C0C1\_count = 0;

uint32\_t pre\_check\_config\_C0C1\_count = 0;

uint32\_t check\_image\_and\_square\_C0C1\_count = 0;

uint32\_t check\_noise\_C0C1\_count = 0;

uint32\_t crop\_image\_and\_divide\_C0C1\_count = 0;

Các biến và mảng được liệt kê ở trên được sử dụng để kiểm tra và ghi nhận việc thực thi các dòng mã (line) trong các hàm hoặc phương thức cụ thể trong chương trình. Dưới đây là mô tả của từng biến và mảng:

1. `DMA\_API\_C0C1\_line\_activate\_list`, `data\_format\_convert\_C0C1\_line\_activate\_list`, `pre\_check\_config\_C0C1\_line\_activate\_list`, `check\_image\_and\_square\_C0C1\_line\_activate\_list`, `check\_noise\_C0C1\_line\_activate\_list`, `crop\_image\_and\_divide\_C0C1\_line\_activate\_list`: Đây là các mảng boolean dùng để đánh dấu việc thực thi của từng dòng mã trong các hàm tương ứng. Mỗi phần tử của mảng đại diện cho việc thực thi của một dòng mã trong một hàm cụ thể.

2. `DMA\_API\_C0C1\_list`, `data\_format\_convert\_C0C1\_list`, `pre\_check\_config\_C0C1\_list`, `check\_image\_and\_square\_C0C1\_list`, `check\_noise\_C0C1\_list`, `crop\_image\_and\_divide\_C0C1\_list`: Đây là các mảng uint8\_t dùng để ghi nhận kết quả của việc thực thi dòng mã tương ứng. Mỗi phần tử của mảng sẽ được gán giá trị 0 nếu dòng mã tương ứng không được thực thi và sẽ được gán giá trị khác 0 nếu dòng mã được thực thi.

3. `DMA\_API\_C0C1\_count`, `data\_format\_convert\_C0C1\_count`, `pre\_check\_config\_C0C1\_count`, `check\_image\_and\_square\_C0C1\_count`, `check\_noise\_C0C1\_count`, `crop\_image\_and\_divide\_C0C1\_count`: Đây là các biến uint32\_t dùng để đếm số lượng dòng mã được thực thi trong các hàm tương ứng. Mỗi biến sẽ ghi nhận số lượng dòng mã được thực thi trong hàm tương ứng.

Các biến và mảng này được sử dụng để kiểm tra bao phủ mã (code coverage) và đánh giá chất lượng của mã nguồn trong quá trình kiểm thử.

* Dưới đây là một ví dụ về cách các biến được sử dụng trong quá trình kiểm thử:

Giả sử chúng ta có một hàm `check\_image\_and\_square`, và chúng ta muốn kiểm tra xem bao nhiêu dòng mã trong hàm này được thực thi và ghi nhận kết quả thực thi của từng dòng mã.

```c

// Hàm check\_image\_and\_square để kiểm tra và xử lý hình ảnh

void check\_image\_and\_square(int image\_id) {

// Đoạn mã kiểm tra và xử lý hình ảnh

// ...

if (image\_id % 2 == 0) {

// Task 1

// ...

} else {

// Task 2

// ...

}

}

```

1. Đầu tiên, khi chạy hàm kiểm thử, các biến `check\_image\_and\_square\_C0C1\_line\_activate\_list`, `check\_image\_and\_square\_C0C1\_list` và `check\_image\_and\_square\_C0C1\_count` sẽ được sử dụng.

2. Trong quá trình thực thi hàm `check\_image\_and\_square`, mỗi dòng mã sẽ được kiểm tra xem nó có được thực thi hay không. Nếu dòng mã được thực thi, thì phần tử tương ứng trong mảng `check\_image\_and\_square\_C0C1\_line\_activate\_list` sẽ được đánh dấu là true, và giá trị tương ứng trong mảng `check\_image\_and\_square\_C0C1\_list` sẽ được ghi nhận.

3. Sau khi kết thúc thực thi hàm `check\_image\_and\_square`, biến `check\_image\_and\_square\_C0C1\_count` sẽ chứa tổng số lượng dòng mã được thực thi trong hàm.

4. Dựa trên kết quả này, bạn có thể đánh giá bao phủ mã (code coverage) của hàm `check\_image\_and\_square` và xác định xem các dòng mã nào đã được thực thi và dòng mã nào chưa được thực thi trong quá trình kiểm thử.

**Log\_C0C1 function in ut\_main.c**

void log\_C0C1(char\* function\_name, uint32\_t C0C1\_count, uint8\_t\* C0C1\_list, uint32\_t total) {

    for (uint32\_t i = 0; i < SIZE\_OF\_LIST; i++) {

        C0C1\_count += C0C1\_list[i];

    }

    if(C0C1\_count < total) {

        printf(ANSI\_COLOR\_RED "[V][RESULT][%3d%%] C0C1 coverage of %-23s, statements covered: [ACTUAL] %d <==> [EXPECTED] %d\n" ANSI\_COLOR\_RESET, (C0C1\_count\*100/total), function\_name, C0C1\_count, total);

    }

    else {

        printf(ANSI\_COLOR\_GREEN "[V][RESULT][%3d%%] C0C1 coverage of %-23s, statements covered: [ACTUAL] %d <==> [EXPECTED] %d\n" ANSI\_COLOR\_RESET, (C0C1\_count\*100/total), function\_name, C0C1\_count, total);

    }

}

This part show up the final result in UT\_coverage

* **Ut\_main.c is for UT\_coverage.exe**

UT.exe : Environment configuration

Make for this part:

#--------------------------------------------------------------

# For Environment Configurations

#--------------------------------------------------------------

TEST\_CONFIGS  += -DTEST\_ENV\_UT

TEST\_CONFIGS  += -DWINDOWS\_OS

TEST\_CONFIGS  += -DNDEBUG

EXTRA\_CFLAGS += -std=c99 -g -O0

CC := gcc

LD := g++

OUTPUT  = UT.exe

# CANTPP\_CMD = cppccd "--comp:x86-Win32-gpp8.2-bundled" --no\_link

# CANTPP\_LD\_CMD = cppccd "--comp:x86-Win32-gpp8.2-bundled"

# CANTPP\_TARGET\_OPTS = "--parse:--line\_directives" "--parse:-W2" "--sm:--call\_seq\_code" "--ci:--instr:func;call;stmt;decn;log;" "--analyse"

TEST\_CONFIGS += -DWIN32

all: $(OUTPUT)

$(OUTPUT) : $(OBJS) $(DEPLIB)

    $(CANTPP\_LD\_CMD) $(CANTPP\_TARGET\_OPTS) $(LD) -o $@ $(OBJS)

$(OBJDIR)/%.c.o: ./%.c

    $(eval ODIR=$(dir $@))

    $(eval DDIR=$(subst $(OBJDIR)/,$(DEPDIR)/,$(ODIR)))

    @if NOT EXIST $(subst /,\\,$(ODIR)) (mkdir $(subst /,\\,$(ODIR)))

    @if NOT EXIST $(subst /,\\,$(DDIR)) (mkdir $(subst /,\\,$(DDIR)))

    $(CC) $(INCDIR) $(EXTRA\_CFLAGS) $(TEST\_CONFIGS) -MD -MF $(addprefix $(DEPDIR)/,$(subst .c,.c.d,$<)) -c -o $(subst ./$(OBJDIR)/,./,$@)  $<

./obj/src/DMA.c.o: $(abspath ../src/DMA.c)

    $(eval ODIR=$(dir $@))

    $(eval DDIR=$(subst ./$(OBJDIR)/src/obj,./$(OBJDIR)/src/dep,$(ODIR)))

    @if NOT EXIST $(subst /,\\,$(ODIR)) (mkdir $(subst /,\\,$(ODIR)))

    @if NOT EXIST $(subst /,\\,$(DDIR)) (mkdir $(subst /,\\,$(DDIR)))

    $(CANTPP\_CMD) $(CANTPP\_TARGET\_OPTS) $(CC) $(INCDIR) $(EXTRA\_CFLAGS) $(TEST\_CONFIGS) -MD -c -o $@ $<

-include $(DEPS)

* Information printed out from **UT\_coverage.exe** and **UT.exe** is both from testfw.c

In ut\_main.c we can see that there is a line call **“TEST\_MAIN(argc, argv)”**

This line lead to **TEST\_UT\_main** that is defined in **ut.c** (extern from **testfw.c**) and execute TEST\_main

**In TEST\_main:**

TEST\_main(argc, argv, "UT", "MODULE\_TEST", TEST\_UT\_Info);

We tranfer an argument TEST\_UT\_Info here, so there is a place to input more Unit Test for this framework

* It will run TEST\_ParseArges to check option input (autorun or not ?)
* In this test we do not input anything (autorun???) so this function will run TEST\_RunAllTests
* Only ILOG print out message, FLOG doesn’t print anything (???why)

The **TEST\_main** function is the primary function used to initialize and run tests. Here's what each argument in this function means:

1. **argc**: This is the number of command-line arguments passed into the program. In C/C++ applications, **argc** is often used to count the number of arguments passed in from the command line.
2. **argv**: This is the array containing pointers to command-line arguments. Each element of this array is a pointer pointing to a string of characters (command-line arguments). Typically, **argv[0]** is the name of the program.
3. **env**: This argument represents the environment of the test. In the context of this function, can be used to define specific test environments or certain environment configurations.
4. **target**: This argument specifies the goal or scope of the test. This can be the name of the module, function, or other test scope.
5. **testInfo**: This argument is a pointer to a **TestInfo** structure, which contains information about the tests defined. The **TestInfo** structure can contain descriptions of test functions, including names, types, test subfunctions, and other information required for test execution.

The specific example **TEST\_main(argc, argv, "UT", "MODULE\_TEST", TEST\_UT\_Info)** means:

1. The program is launched with the number of command-line arguments, and the arguments are passed through **the argv array**.
2. The test environment is set to **"UT"** (assuming "UT" means "Unit Test").
3. The test objective is set as **"MODULE\_TEST".**
4. Information about the tests is taken from the **TEST\_UT\_Info** variable.

**TEST\_UT\_Info in /ut/include/test\_case.h**

By using

DECLARE\_TESTCASE\_TABLE(DMA\_API);

Which is equivalent to:

extern struct TestCase UT\_DMA\_API\_All\_Tests[];

so that we can investigate that struct inside **DMA\_API\_table.c**

TEST\_CASE\_F(UT, DMA\_API, RPI3, DMA\_API\_0001) { EXPECT\_EQ( true, TEST\_DMA\_API("PCL", 1)); }

struct TestCase UT\_DMA\_API\_All\_Tests[] = {

    TEST\_CASE\_T(UT, DMA\_API, RPI3, DMA\_API\_0001, DMA\_API\_0001),

    TEST\_CASE\_END

};

So after going into this file, we can see 2 macro TEST\_CASE\_F (test case function) and TEST\_CASE\_T (test case type) and can be explored in testenv.h

#define TEST\_CASE\_F(type, func, target, id)\

    void type##\_##func##\_##id##\_##target##\_Test()

#define TEST\_CASE\_T(type, func, target, funcid, id)\

    { #type, #func, #target, #id, TEST\_FUNC\_NAME(type##\_##func, target, funcid) }

And so many macros apply on this, finally we got this:

void UT\_DMA\_API\_DMA\_API\_0001\_RPI3\_Test()

{

    bool result = TEST\_DMA\_API("PCL", 1);

    if (result == true)

    {

        printf("[          OK ] %s (from testenv.h)\n\n\n\n", g\_test.runningTestName);

        g\_test.passed++;

    }

    else

    {

        printf("%s:%d: Failure\n", \_\_FILE\_\_, \_\_LINE\_\_);

        printf("\tExpected: %s\n", (true) ? "true" : "false");

        printf("\tWhich is: %s\n", (result) ? "true" : "false");

        printf("[          NG ] %s\n", g\_test.runningTestName);

        g\_test.failed++;

    }

}

bool TEST\_DMA\_API(const char \*category, int32\_t no)

{

    bool b = true;

    struct TestParams \*params = TEST\_CreateParam("DMA\_API", category, no);

    struct TEST\_DMA\_API\_Pattern \*pattern = &DMA\_API\_PCL[no - 1];

    struct DMA\_API\_input \*inputs = &pattern->input;

    struct DMA\_API\_params func\_params = {0};

    struct DMA\_API\_expect outputs = {0};

    ut\_init\_config(false);

    memcpy(&func\_params.x, &inputs->x, sizeof(func\_params.x));

    memcpy(&func\_params.y, &inputs->y, sizeof(func\_params.y));

    if (inputs->\_p\_dma\_config == TEST\_ADDR\_NULL)

    {

        func\_params.p\_dma\_config = NULL;

    }

    else

    {

        func\_params.p\_dma\_config = &inputs->p\_dma\_config;

    }

    outputs.ReturnValue = DMA\_API(func\_params.x, func\_params.y, func\_params.p\_dma\_config);

    if (pattern->expected.\_p\_dma\_config == TEST\_ADDR\_NULL || pattern->expected.\_p\_dma\_config == TEST\_ADDR\_NOT\_NULL)

    {

        outputs.\_p\_dma\_config = (func\_params.p\_dma\_config == NULL) ? TEST\_ADDR\_NULL : TEST\_ADDR\_NOT\_NULL;

    }

    else

    {

        outputs.\_p\_dma\_config = (uint32\_t)func\_params.p\_dma\_config;

    }

    if (func\_params.p\_dma\_config != NULL)

    {

        memcpy(&outputs.p\_dma\_config, func\_params.p\_dma\_config, sizeof(outputs.p\_dma\_config));

    }

    b &= TEST\_GetAddr\_DMA\_API(&pattern->expected.\_p\_dma\_config, &func\_params);

    b &= TEST\_Validate\_DMA\_API(&outputs, &pattern->expected, pattern->validator);

    TEST\_ValidateResult(b, params);

    ut\_deinit\_config();

    TEST\_DestroyParam(params);

    return b;

}

**Operation process**:

1. When you call **UT\_DMA\_API\_DMA\_API\_0001\_RPI3\_Test** function in your program, it performs steps in its body.
2. First, it will call the **TEST\_DMA\_API** function to perform the test and assign the result to the **result variable**.
3. It will then check this result using the **EXPECT\_EQ macro**.
4. The results of the test will be printed to the screen and the variables **g\_test.passed** or **g\_test.failed** will be increased accordingly to count the number of tests that have succeeded or failed.

OK, so in **DMA\_API\_table.c,** we can find out the function **UT\_DMA\_API\_DMA\_API\_0001\_RPI3\_Test()** that include the pattern for Unit Test: (which stored in **DMA\_API\_PCL.h**)

    struct TEST\_DMA\_API\_Pattern \*pattern = &DMA\_API\_PCL[no - 1];

In which stores all pattern that we prepared in **test\_spec\_ref.xlxs** . So at this point, we can barely understand that file start will **\*\_table.c** and **\*\_PCL.h** can affect the result in **C0C1 and C2 coverage check**

A black background with red and blue text

Description automatically generated

A black background with red and blue text

Description automatically generated

That makes sense to the changes in % coverage of that function, but what happens if we want to test another module/function ? Let’s look back at **TEST\_UT\_Info in /ut/include/test\_case.h** to add another TESTCASE\_TABLE into that.