

Unit-Test

ESP32 中的单元测试

:link_to_translation: en:[English]

ESP-IDF

中附带了一个基于 **Unity** 的单元测试应用程序框架，且所有的单元测试用例分别保存在 **ESP-IDF** 仓库中每个组件的 **test** 子目录中。

添加常规测试用例

单元测试被添加在相应组件的 **test** 子目录中，测试用例写在 **C** 文件中，一个 **C** 文件可以包含多个测试用例。测试文件的名字要以 “**test**” 开头。

测试文件需要包含 **unity.h** 头文件，此外还需要包含待测试 **C** 模块需要的头文件。

测试用例需要通过 **C** 文件中特定的函数来添加，如下所示：

```
.. code:: c
TEST_CASE("test name", "[module name]")
{
// 在这里添加测试用例
}
```

- 第一个参数是字符串，用来描述当前测试。
- 第二个参数是字符串，用方括号中的标识符来表示，标识符用来对相关测试或具有特定属性的测试进行分组。

没有必要在每个测试用例中使用 **UNITY_BEGIN()** 和 **UNITY_END()** 来声明主函数的区域， **unity_platform.c** 会自动调用 **UNITY_BEGIN()**，然后运行测试用例，最后调用 **UNITY_END()**。

test 子目录需要包含 :ref: 组件 **CMakeLists.txt <component-directories>**，因为他们本身就是一种组件。**ESP-IDF** 使用了 **unity** 测试框架，需要将其指定为组件的依赖项。通常，组件 :ref: 需要手动指定待编译的源文件 **<cmake-file-globbing>**;但是，对于测试组件来说，这个要求被放宽了，仅仅是建议使用 “**COMPONENT_SRCDIRS**”。

总的来说，**test** 子目录下最简单的 **CMakeLists.txt** 文件可能如下所示：

.. code:: cmake

```
set(COMPONENT_SRCDIRS ".")
set(COMPONENT_ADD_INCLUDEDIRS ".")
set(COMPONENT_REQUIRES unity)

register_component()
```

更多关于如何在 **Unity** 下编写测试用例的信息，请查阅 <http://www.throwtheswitch.org/unity>。

添加多设备测试用例

常规测试用例会在一个 **DUT** (Device Under

Test, 在试设备) 上执行, 那些需要互相通信的组件 (比如 **GPIO**, **SPI**...) 不能使用常规测试用例进行测试。多设备测试用例支持使用多个 **DUT** 进行写入和运行测试。

以下是一个多设备测试用例 :

```
.. code:: c

void gpio_master_test()
{
    gpio_config_t slave_config = {
        .pin_bit_mask = 1 << MASTER_GPIO_PIN,
        .mode = GPIO_MODE_INPUT,
    };
    gpio_config(&slave_config);
    unity_wait_for_signal("output high level");
    TEST_ASSERT(gpio_get_level(MASTER_GPIO_PIN) == 1);
}

void gpio_slave_test()
{
    gpio_config_t master_config = {
        .pin_bit_mask = 1 << SLAVE_GPIO_PIN,
        .mode = GPIO_MODE_OUTPUT,
    };
    gpio_config(&master_config);
    gpio_set_level(SLAVE_GPIO_PIN, 1);
    unity_send_signal("output high level");
}

TEST_CASE_MULTIPLE_DEVICES("gpio multiple devices test example", "[driver]", gpio_master_test, gpio_slave_test);
```

宏 `TEST_CASE_MULTIPLE_DEVICES` 用来声明多设备测试用例,

- 第一个参数指定测试用例的名字。
- 第二个参数是测试用例的描述。
- 从第三个参数开始, 可以指定最多5个测试函数, 每个函数都是单独运行在一个 **DUT** 上的测试入口点。

在不同的 **DUT** 上运行的测试用例, 通常会要求它们之间进行同步。我们提供 `unity_wait_for_signal` 和 `unity_send_signal` 这两个函数来使用 **UART** 去支持同步操作。如上例中的场景, **slave** 应该在在 **master** 设置好 **GPIO** 电平后去读取 **GPIO** 电平, **DUT** 的 **UART** 终端会打印提示信息, 并要求用户进行交互。

DUT1 (master) 终端 :

```
.. code:: bash
```

```
Waiting for signal: [output high level]!
Please press "Enter" key once any board send this signal.
```

DUT2 (slave) 终端 :

```
.. code:: bash
```

```
Send signal: [output high level]!
```

一旦 **DUT2** 发送了该信号, 您需要在 **DUT2** 的终端输入回车, 然后 **DUT1** 会从 `unity_wait_for_signal` 函数中解除阻塞, 并开始更改 **GPIO** 的电平。

添加多阶段测试用例

常规的测试用例无需重启就会结束（或者仅需要检查是否发生了重启），可有些时候我们想在某些特定类型的重启事件后运行指定的测试代码，例如，我们想在深度睡眠唤醒后检查复位的原因是否正确。首先我们需要出发深度睡眠复位事件，然后检查复位的原因。为了实现这一点，我们可以定义多阶段测试用例来将这些测试函数组合在一起。

```
.. code:: c

static void trigger_deepsleep(void)
{
    esp_sleep_enable_timer_wakeup(2000);
    esp_deep_sleep_start();
}

void check_deepsleep_reset_reason()
{
    RESET_REASON reason = rtc_get_reset_reason(0);
    TEST_ASSERT(reason == DEEPSLEEP_RESET);
}

TEST_CASE_MULTIPLE_STAGES("reset reason check for deepsleep", "[esp32]", trigger_deepsleep,
check_deepsleep_reset_reason);
```

多阶段测试用例向用户呈现了一组测试函数，它需要用户进行交互（选择用例并选择不同的阶段）来运行。

编译单元测试程序

按照 **esp-idf** 顶层目录的 **README** 文件中的说明进行操作，请确保 **IDF_PATH** 环境变量已经被设置指向了 **esp-idf** 的顶层目录。

切换到 **tools/unit-test-app** 目录下进行配置和编译：

- **idf.py menuconfig** - 配置单元测试程序。
- **idf.py -T all build** - 编译单元测试程序，测试每个组件 **test** 子目录下的用例。
- **idf.py -T xxx build** - 编译单元测试程序，测试指定的组件。
- **idf.py -T all -E xxx build** - 编译单元测试程序，测试所有（除开指定）的组件。例如
idf.py -T all -E ulp mbedtls build -
编译所有的单元测试，不包括 **ulp** 和 **mbedtls** 组件。

当编译完成时，它会打印出烧写芯片的指令。您只需要运行 **idf.py flash** 即可烧写所有编译输出的文件。

您还可以运行 **idf.py -T all flash** 或者
idf.py -T xxx flash 来编译并烧写，所有需要的文件都会在烧写之前自动重新编译。

使用 **menuconfig** 可以设置烧写测试程序所使用的串口。

运行单元测试

烧写完成后重启 **ESP32**，它将启动单元测试程序。

当单元测试应用程序空闲时，输入回车键，它会打印出测试菜单，其中包含所有的测试项目。

```
.. code:: bash
```

```
Here's the test menu, pick your combo:
(1) "esp_ota_begin() verifies arguments" [ota]
(2) "esp_ota_get_next_update_partition logic" [ota]
(3) "Verify bootloader image in flash" [bootloader_support]
```

- (4) "Verify unit test app image" [bootloader_support]
- (5) "can use new and delete" [cxx]
- (6) "can call virtual functions" [cxx]
- (7) "can use static initializers for non-POD types" [cxx]
- (8) "can use std::vector" [cxx]
- (9) "static initialization guards work as expected" [cxx]
- (10) "global initializers run in the correct order" [cxx]
- (11) "before scheduler has started, static initializers work correctly" [cxx]
- (12) "adc2 work with wifi" [adc]
- (13) "gpio master/slave test example" [ignore][misc][test_env=UT_T2_1][multi_device]
 - (1) "gpio_master_test"
 - (2) "gpio_slave_test"
- (14) "SPI Master clockdiv calculation routines" [spi]
- (15) "SPI Master test" [spi][ignore]
- (16) "SPI Master test, interaction of multiple devs" [spi][ignore]
- (17) "SPI Master no response when switch from host1 (HSPI) to host2 (VSPI)" [spi]
- (18) "SPI Master DMA test, TX and RX in different regions" [spi]
- (19) "SPI Master DMA test: length, start, not aligned" [spi]
- (20) "reset reason check for deepsleep" [esp32][test_env=UT_T2_1][multi_stage]
 - (1) "trigger_deepsleep"
 - (2) "check_deepsleep_reset_reason"

常规测试用例会打印用例名字和描述，主从测试用例还会打印子菜单（已注册的测试函数的名字）。

可以输入以下任意一项来运行测试用例：

- 引号中的测试用例的名字，运行单个测试用例。
- 测试用例的序号，运行单个测试用例。
- 方括号中的模块名字，运行指定模块所有的测试用例。
- 星号，运行所有测试用例。

[multi_device] 和 [multi_stage]

标签告诉测试运行者该用例是多设备测试还是多阶段测试。这些标签由 `TEST_CASE_MULTIPLE_STAGES` 和 `TEST_CASE_MULTIPLE_DEVICES` 宏自动生成。

一旦选择了多设备测试用例，它会打印一个子菜单：

```
.. code:: bash
```

```
Running gpio master/slave test example...
gpio master/slave test example
(1) "gpio_master_test"
(2) "gpio_slave_test"
```

您需要输入数字以选择在 **DUT** 上运行的测试。

与多设备测试用例相似，多阶段测试用例也会打印子菜单：

```
.. code:: bash
```

```
Running reset reason check for deepsleep...
reset reason check for deepsleep
(1) "trigger_deepsleep"
(2) "check_deepsleep_reset_reason"
```

第一次执行此用例时，输入 **1** 来运行第一阶段（触发深度睡眠）。在重启

DUT 并再次选择运行此用例后，输入 **2**

来运行第二阶段。只有在最后一个阶段通过并且之前所有的阶段都成功触发了复位的情况下，该测试才算通过。