User Guide

Release v2.1.0-RC1

ARM CE-OSS

May 08, 2024

OVERVIEW

The Trusted Firmware-M(TF-M) Tests repo is meant to hold various tests for the Trusted Firmware-M. The TF-M tests mainly focus on functionalities of various TF-M componentes such as the TF-M SPM and various Secure Partitions.

CHAPTER

ONE

THE TF-M ERPC TEST FRAMEWORK

The TF-M eRPC Test Framework is an Remote Procedure Call (RPC) framework for testing purpose. It's based on the eRPC system. It is an additional framework to the existing one for running NS test suites. It enables you to run test codes on host machines as if they were running on the device. It has the following advantages.

- Off-load test codes from device to host

  Arm M-profile devices usually have limited flash storage which can only fit limited test suites. With test codes running on hosts you can run far more tests than on the devices.

- Less frequent image downloading for test code development.

  As the test codes run on the host, you don't need to update the image on device when you update test codes.

- Host can get test pass or failure directly from the return codes.

  This would be helpful for test automation because you don't need to parse the test logs.

1.1 How Does It Work

Originally, the NS tests are executed in the NSPE of the device. The NS image on the device contains the test codes, which calls into TF-M through the PSA client APIs.

With the eRPC test framework, the NS tests code are executed on the host machine. The NSPE of device side does not run the test codes anymore. When the tests call the PSA client APIs, they call into the eRPC framework. The eRPC framework communicates with the NSPE on the device, which calls into TF-M through the PSA client APIs.

The prototypes of the PSA client APIs are the same while the implementations are different. Refer to the following sections for more details.

1.1.1 The Stucture

The following diagram shows the software structure.

- eRPC Framework

  The eRPC framework system

- eRPC Client Shim

  The eRPC generated shim layer of remote APIs for clients. It serializes the identifier of the API and its parameters into a stream of bytes and transports to the server through a communication channel such as UART and TCP/IP. The codes are generated by the erpcgen tool.

- eRPC Server Shim

  The generated shim layer of the server. It registers a callback function to the eRPC framework. When the framework receives any requests from the client, it calls the callback function. The callback unserializes the bytes streams to determine what API to call and then invoke it with the corresponding parameters from the bytes streams. And then it returns the results to the client in the reverse routine.

- API Wrapper

  Part of the parameters of `psa_call` API is not supported by eRPC directly, thus an API wrapper is required to transform the `in_vec/out_vec` structures to the eRPC supported data types. The wrapper API is named as `erpc_psa_call`.

  On the client side, the wrapper implements the `psa_call` which calls the `erpc_psa_call` in the client shim layer. On the server side, the wrapper implements the `erpc_psa_call` which is called by the shim layer. The `erpc_psa_call` then calls the `psa_call`.

- Test Suites

  Can be the existing TF-M regression tests or any other tests that interact with TF-M using the PSA Client APIs.

- Host App

  Initializes the eRPC client and starts test suites.

- Target App

  Initializes the eRPC server and listens for requests from the client.

1.1.2 Supported APIs

The APIs supported for doing RPC are the PSA Client APIs because they are the lowest level APIs that interact with TF-M. You can build lots of test suites upon them. You can also add your own APIs in the `tfm.erpc` file. Please refer to IDL Reference for the syntax of the file.

API Grouping

PSA client APIs are categorised into common APIs and connection-based service APIs. Connection-based APIs are available when there are connection-based services in the TF-M. So in the eRPC integration, the APIs are also split into two groups so that the shim layer of the APIs can be separated into different files as well. Then build systems can decide which source files to build based on the existence of connection-based services.

Common APIs:

- psa_framework_version()
- psa_version()
- psa_call()

Connection-based specific:

- psa_connect()
- psa_close()

1.1.3 Transportation

On device side, only UART transportation is supported in NSPE. For the host side, both UART and TCP are supported. The TCP transportation support is basically for fast models where UART data are transferred between a TCP/IP socket on the host and a serial port on the target. See the fast model reference guide for more details.

1.2 Platform Integration

First, the UART drivers of platforms shall support TX/RX control feature. The TF-M build system provides a `CONFIG_ENABLE_NS_UART_TX_RX_CONTROL` option to enable or disable the TX/RX control feature and it is disabled by default. When the eRPC test framework is enabled, the `CONFIG_ENABLE_NS_UART_TX_RX_CONTROL` will be enabled automatically.

Secondly, platforms need to create their folders under the `erpc/platform` and then create the `config_erpc_target.h` to specify the UART port drivers for eRPC transportation.

```
#define ERPC_UART           Driver_USART0
```

Note: The folder structure in `erpc/platform` must be the same as the `platform/ext/target` of TF-M repo.

It's recommended to use a different UART to the stdio UART. If the same UART is used for both, then the TF-M logs (both SPM and Secure Partitions) must be disabled. Otherwise, the eRPC transportation might fail.

1.3 Application Integration

The TF-M eRPC test framework provides two CMake libraries for integration. One is the `erpc_client`, the other is the `erpc_server`. Both include the eRPC framework, the shim layers, API wrappers and expose an initialization API for client and server respectively.

The initialization does not include the initialization of the transportation layer because it is use case specific which kind of transportation is used. So it is the client and server's responsibilities to initialize the transportation layers and pass them to the `erpc_client` and `erpc_server`.

TF-M provides a `app/erpc_app.c` as the default server application which initializes the UART transportation and starts the eRPC server.

A config option `CONFIG_TFM_ERPC_TEST_FRAMEWORK` is provided to enable the eRPC framework on device (server) side. The default server will be built and developers only need to focus on the client application developments.

In summary, on the server side, you only need to build with the `CONFIG_TFM_ERPC_TEST_FRAMEWORK` enabled. On the client side, you must

- Initializes the transportation layer using eRPC provided APIs.
- Call the initialization function provided by TF-M eRPC test framework with the transportation instance initialized above.
- Develop the application code
- Building with CMake
  - `add_subdirectory` with the `erpc/client`
  - link the `erpc_client` library

There is an example at `erpc/host_example` for reference.

CHAPTER

TWO

ADDING PARTITIONS FOR REGRESSION TESTS

2.1 Introduction

Some test group may need specific test services. These test services may support one or more groups thus developers shall determine the proper test scope. Currently, TF-M test services are located under `tf-m-tests/test/secure_fw`.

Folder name	Description
test/secure_fw/suites/<suite>/service	Test service divided into corresponding suite subdirectories.
test/secure_fw/common_test_services	Common test services.

2.2 Implementation

Adding a test partition to provide test services is same as adding a secure partition, generally the process can be referenced from the document Adding Secure Partition.

2.2.1 Test Partition Specific Manifest Attributes

Each test service must have resource requirements declared in a manifest file, the contents of test services are the same as secure partitions, but their locations are different. Test service manifests shall be set in `tf-m-tests/test/secure_fw/tfm_test_manifest_list.yaml`.

There are some test purpose attributes in Secure Partition manifests that are **NOT** compatible with FF-M. They should be used in Test Partitions only.

weak_dependencies

A TF-M regression test Partition calls other RoT services for test. But it can still run other tests if some of the RoT services are disabled. TF-M defines a `"weak_dependencies"` attribute in partition manifests of regression test partitions to describe test service access to other RoT services. It *shall* be only used for TF-M regression test services.

model

A TF-M regression test Partition may support both the SFN and IPC model. The actual model being used follows the SPM backend enabled.

The TF-M build system supports this by allowing Secure Partitions to set the `model` attribute to `dual`. The manifest tool will then change it to the corresponding value according to the current backend selected.

The Test Partitions use the following definitions to know what model is being built:

- `<<partition_name>>_MODEL_IPC`, 1 if IPC model is used.
- `<<partition_name>>_MODEL_SFN`, 1 if SFN model is used.

2.2.2 Test service implementation

Test service of individual test

An individual test dedicated test service should be put under the corresponding test folder `test/secure_fw/suites/<test_name>`.

`add_subdirectory(suites/<test_name>/<service_dir>)` shall be added into `tf-m-tests/test/secure_fw/secure_tests.cmake` to make sure that the test service is built with secure side configuration.

Common test service

If a new test service is required by multiple test suites, the code should be put under `test/secure_fw/common_test_services`. If the new test suite relies on a common test service, please make sure that the build implementation of the test service is linked correctly, including the header files and libraries.

CHAPTER

THREE

ADDING TF-M REGRESSION TEST SUITE

Table of Contents

3.1 Introduction of TF-M regression tests

TF-M regression tests test whether changes to TF-M code work as expected. A new regression test can consist of following 3 components:

1. `test suite`: A series of tests of a certain function.
2. `test case`: A specific test instance in test suites.
3. `test service or partition`: Optional secure services or partitions to support related test suites.

3.1.1 Source structure

Folder name	Description
test/bl1	TF-M bl1 test suites source code.
test/bl2	TF-M bL2 test suites source code.
test/config	The CMAKE test configurations files.
test/framework	Source code for test framework code, managing test suites.
test/secure_fw/suites	Test suites divided into subdirectories.
test/secure_fw/suites/<suite>/service	Test service divided into corresponding suite subdirectories.
test/secure_fw/common_test_services	Common test services.

3.2 Adding a new test suite

This section introduces how to add a new test suite and control its compilation with a test configuration in `tests_reg/test` repository.

3.2.1 Source code

The test suite example subdirectory named <test_name> is located under the path `tests_reg/test/secure_fw/suites`. If the new test suite includes both non-secure and secure parts, the source code shall be divided shared code and specific code. An example test suite folder can be organized as the figure below.

```
.
├── <test_name>_tests_common.c
├── non_secure
|   ├── CMakeLists.txt
|   ├── <test_name>_ns_interface_testsuite.c
|   └── <test_name>_ns_tests.h
└── secure
    ├── CMakeLists.txt
    ├── <test_name>_s_interface_testsuite.c
    └── <test_name>_s_tests.h
```

3.2.2 Creating test configurations

A test configuration controls whether one or multiple test suites are enabled. The doc TF-M Build Instructions. shows some test configurations which are already supported in current TF-M.

Developers shall assign corresponding test configurations to control the test suites.

Naming test configurations

The test configurations of example test suites are `TEST_NS_<TEST_NAME>` in non-secure and `TEST_S_<TEST_NAME>` in secure.

Note: The test configurations must be named with the prefixes `TEST_S_` and `TEST_NS_`, for secure regression tests and non-secure regression tests respectively. Otherwise, tests_reg/test build system may not recognize it.

Setting test configurations

When the test configurations have dependencies, the default value need to be set. The setting is performed in following four steps.

1. Command line input: The configuration can be enabled or disabled by the command `-DTEST_NS_<TEST_NAME>=ON/OFF -DTEST_S_<TEST_NAME>=ON/OFF`, and the value cannot be changed throughout the whole compilation of TF-M tests.
2. `tests_reg/test/config/config.cmake`: The test configurations shall be `OFF` if its dependencies are not supported. The dependencies are probably from:
    1. TF-M partitions configurations like `TFM_PARTITION_CRYPTO`, `TFM_PARTITION_INITIAL_ATTESTATION`, etc.
    2. TF-M build mode configuration like `CONFIG_TFM_SPM_BACKEND`.
    3. TF-M other configurations like `TFM_PARTITION_FIRMWARE_UPDATE`, etc.
3. `tests_reg/test/config/default_ns_test_config.cmake` or `tests_reg/test/config/default_s_test_config.cmake`: It is required to give the default value of the new test configuration in these two files when `TEST_NS` or `TEST_S` is ON. The recommended value is ON unless the single test's code or data size is very large.
4. `tests_reg/test/config/default_test_config.cmake`: It is required to give the default value of the new test configuration in the file when both `TEST_NS` and `TEST_S` are OFF. The default value must be OFF.

Note: The test configurations must be set as CACHE value in CMAKE files. The CACHE set cannot replace the value from command line, see Set Cache Entry.

Checking test configurations

The new test configurations must be checked by function `tfm_invalid_config()` if they have any dependence. The value comes from command line may be wrong when the dependencies are conflicting.

Implement necessary checks in `tests_reg/test/config/check_config.cmake`.

3.2.3 Export TF-M configurations

If the new test depends on some TF-M configurations, export their value via `tests_reg/test/config/config_ns_tests.cmake.in`. TF-M secure build will install `config_ns_tests.cmake` and export configuration values. tf-m-tests non-secure build will include `config_ns_tests.cmake` and receive TF-M configuration values.

3.2.4 Applying test configurations

The mission of test configurations is to control the build. They are applied in `test/secure_fw/suites/<test_name>/CMakeLists.txt` like the example below.

```
if (NOT TEST_NS_<TEST_NAME> AND NOT TEST_S_<TEST_NAME>)
    return()
endif()

######################## Non Secure ###############################################

if (TEST_NS_<TEST_NAME>)
    add_library(tfm_test_suite_<test_name>_ns STATIC EXCLUDE_FROM_ALL)
    # target_sources()
    # target_include_directories()
    target_compile_definitions(tfm_test_suite_<test_name>_ns
        INTERFACE
            TEST_NS_<TEST_NAME>
    )
    # target_link_libraries()
endif()

######################## Secure ###################################################

if (TEST_S_<TEST_NAME>)
    add_library(tfm_test_suite_<test_name>_s STATIC EXCLUDE_FROM_ALL)
    # target_sources()
    # target_include_directories()
    target_compile_definitions(tfm_test_suite_<test_name>_s
        INTERFACE
            TEST_S_<TEST_NAME>
    )
    # target_link_libraries()
endif()
```

The function `target_compile_definitions` will export the macros `TEST_NS_<TEST_NAME>` or `TEST_S_<TEST_NAME>` into source code. and in the file `tests_reg/test/framework/non_secure_suites.c` or `tests/framework/secure_suites.c`, the definitions of these macros will be checked, and then the head file will be included and test cases will be registered if the macro is defined.

```
#ifdef TEST_NS_<TEST_NAME>
#include "<test_name>_ns_tests.h"
#endif

static struct test_suite_t test_suites[] = {
/* Non-secure example test cases */
    // ......
#ifdef TEST_NS_<TEST_NAME>
    {&register_testsuite_ns_<test_name>_interface, 0, 0, 0},
#endif
};
```

```
#ifdef TEST_S_<TEST_NAME>
#include "<test_name>_s_tests.h"
#endif

static struct test_suite_t test_suites[] = {
/* Secure example test cases */
    // ......
#ifdef TEST_S_<TEST_NAME>
    {&register_testsuite_s_<test_name>_interface, 0, 0, 0},
#endif
};
```

Note: On most platforms non-secure tests and secure tests run on the same CPU core, but dual-core platform is an exception. So secure test library and secure services shall be linked together in the file `tests_reg/test/test/secure_fw/secure_tests.cmake`. Thus they can be built on secure CPU core and non-secure tests library and RTOS are built on non-secure CPU core.

```
# ...
if (TEST_S_<TEST_NAME>)
    add_library(tfm_test_suite_<test_name>_s STATIC EXCLUDE_FROM_ALL)
endif()
```

3.3 Adding a new test case in test suite

The test cases usually express as a function in source code. They will be added into an array with structure type called `test_t` defined in `tests_reg/test/test/framework/test_framework.h`.

```
struct test_t {
    TEST_FUN * const test;          /*!< Test function to call */
    const char *name;               /*!< Test name */
    const char *desc;               /*!< Test description */
};
```

For example, a new test case called `TFM_NS_<TEST_NAME>_TEST_1001` is created and the function `tfm_<test_name>_test_1001` needs to be defined in file `<test_name>_ns_interface_testsuite.c`. Then the function shall be appended into the array which will be quoted in function `register_testsuite_ns_<test_name>_interface`. See the reference code below.

```
/* List of test cases */
static void tfm_<test_name>_test_1001(struct test_result_t *ret);

/* Append test cases */
static struct test_t <test_name>_tests[] = {
    {&tfm_<test_name>_test_1001, "TFM_NS_<TEST_NAME>_TEST_1001",
    "Example test case"},
};

/* Register test case into test suites */
void register_testsuite_ns_<test_name>_interface(struct test_suite_t *p_test_suite)
{
    uint32_t list_size;

    list_size = (sizeof(<test_name>_tests) / sizeof(<test_name>_tests[0]));

    set_testsuite("<TEST_NAME> non-secure interface test (TFM_NS_<TEST_NAME>_TEST_1XXX)",
                  <test_name>_tests, list_size, p_test_suite);
}

static void tfm_<test_name>_test_1001(struct test_result_t *ret)
{
   /* test case code */
}
```

3.4 Adding test services

Some test group may need specific test services. These test services may support one or more groups thus developers shall determine the proper test scope. Refer to *Adding partitions for regression tests* to get more information.

3.5 Out-of-tree regression test suites

TF-M supports out-of-tree regression test suites build, whose source code folder is outside tf-m-tests repo. There are two configurations for developers to include the source code.

- `EXTRA_NS_TEST_SUITE_PATH`

  An absolute directory of the out-of-tree non-secure test suite source code folder. TF-M build system searches `CMakeLists.txt` of non-secure test suite in the source code folder.

- `EXTRA_S_TEST_SUITE_PATH`

  An absolute directory of the out-of-tree secure test suite source code folder.

3.5.1 Example usage

Take non-secure test as an example in tf-m-extras. A single out-of-tree test suite folder can be organized as the figure below:

```
extra_ns
├── CMakeLists.txt
├── ns_test.c
└── ns_test_config.cmake
```

In the example above, `EXTRA_NS_TEST_SUITE_PATH` in the build command can be specified as listed below.

```
-DEXTRA_NS_TEST_SUITE_PATH=<Absolute-path-extra-test-folder>
```

3.5.2 Coding instructions

This is a demo of source code so the structure has been simplified. Files like `.c` and `.h` can be expanded to `src` and `include` folders respectively. The `CMakeLists.txt` is required in the root path and `ns_test_config.cmake` is optional.

Header Files

The header file `extra_ns_tests.h` must be included by out-of-tree source code. This file contains the declaration of `void register_testsuite_extra_ns_interface(struct test_suite_t *p_test_suite)`.

Source code

To connect the out-of-tree source code and regression test framework, the test case function/functions must be defined first. An example format is:

```
void ns_test(struct test_result_t *ret)
{
    /* Add platform specific non-secure test suites code here. */

    ret->val = TEST_PASSED;
}
```

This function follows the standard TF-M test case function prototype.

Note: Extra tests can have one or more test cases. This is simplified example so only one test case is added.

After `ns_test()` is defined, a structure variable need to be created like:

```
static struct test_t plat_ns_t[] = {
    {&ns_test, "TFM_EXTRA_TEST_1001",
     "Extra Non-Secure test"},
};
```

It will be used by function `register_testsuite_extra_ns_interface()` to register the test by calling the `set_testsuite()` function:

```
void register_testsuite_extra_ns_interface(struct test_suite_t *p_test_suite)
{
    uint32_t list_size;

    list_size = (sizeof(plat_ns_t) /
                 sizeof(plat_ns_t[0]));

    set_testsuite("Extra Non-Secure interface tests"
                  "(TFM_NS_EXTRA_TEST_1XXX)",
                  plat_ns_t, list_size, p_test_suite);
}
```

The platform initialization code can be added in this function because it runs before `ns_test()`.

Note: Function `register_testsuite_extra_ns_interface()` is declared in tf-m-tests repository without definition. It is supplied to out-of-tree source code and need to be defined with no change of its format, like returns error code and parameter name.

CMakeLists.txt

After extra test suite file were created they must be linked to `tfm_test_suite_extra_ns` CMAKE target:

```
target_sources(tfm_test_suite_extra_ns
    PRIVATE
        ${CMAKE_CURRENT_SOURCE_DIR}/ns_test.c
)
```

ns_test_config.cmake

The CMAKE configuration file is optional. If out-of-tree source already exists another configuration file, a new one can be ignored.
