arm

Trusted Firmware A

Unit Testing in TF-A

Lauren Wehrmeister

6/18/2020

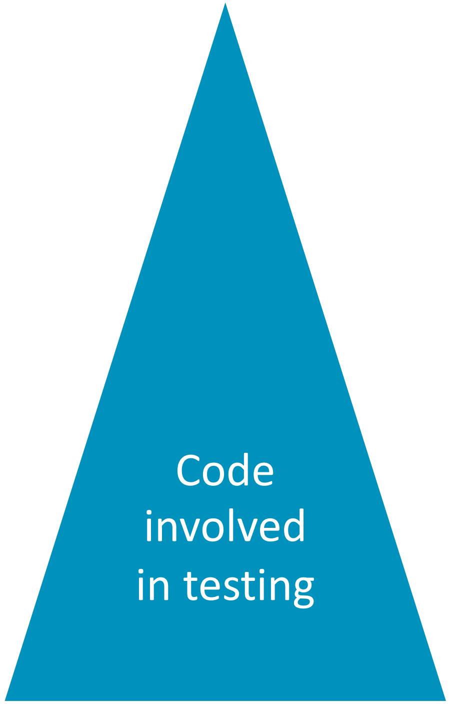
Agenda

- The Concept of Unit Testing
- Current Framework & Components
- Future Work

The Concept of Unit Testing

Levels of Software Testing

- Unit testing
  - Testing each unit separately
- Integration testing
  - Checking multiple interacting units together
- System testing
  - Testing the whole system against the specification
- Acceptance testing
  - Checking business requirements for delivery



Unit Testing

- Testing of small, isolated software units
  - Object-oriented unit - class
  - C unit - set of functions around a feature
- C/C++ unit tests and the code under test are compiled into an executable
- Advantages
  - Less errors caused by lower abstraction level functions on higher levels
  - Validates the existing behavior in every run helps refactoring
  - Makes the programmer able to test rare events
  - Encourages modular software structure, because it’s hard to test spaghetti code
  - Helps documenting as it works as an example code
  - Another advantage in embedded environment is that is helps development without hardware
- Disadvantages
  - It requires more work from the software developer (but it pays off later)
  - It doesn’t test all the interactions between units so higher-level testing is still required

arm

Current Framework & Components

Current Framework

- The TF-A Unit Testing framework allows testing parts of C/C++ code.
- Currently only works internally to Arm as the c-picker tool is not available outside Arm.

- At its current stage, the framework:
  - Can define and build unit test cases, there are currently a few that exist
  - Runs with lcov based code-coverage when compiled with GCC, which is the same as used for FVP based TF-A code-coverage.
  - Documentation exists for getting started, building, running and debugging tests
  - Some basic mocks have been implemented for TF-A.
  - c-picker tool created with can split code fragments from original code and map coverage info back to original code location

Components

- CMake – Build environment
- CppUTest – Unit test framework
  - Includes CppUMock – Mocking framework
- c-picker – Python based code-cut tool for isolating functions
- trusted-firmware-a – Code under test
- tf-a-unit-tests – Unit test repository
  - Unit tests
  - Mocks
  - Build system and utilities

CMake

- CMake is a tool to describe and generate buildsystems chosen as the build environment for the TF-A unit test framework
- We are currently integrating into TF-A -> Refer to Javier's presentation on CMake from past Tech Forum

- Motivation for Unit Test framework is that ctest is included
  - ctest is an executable of CMake
  - CMake-generated build trees created for projects that use the **`enable_testing()`** and **`add_test()`** commands have testing support.
  - ctest will run the tests and report results.

CppUTest

- CppUTest is a C/C++ based unit xUnit test framework

- Why CppUTest?
  - C/C++ support
  - Small footprint (compared to the popular Google Test)
  - Easy portability for embedded systems
  - Built-in mocking system (CppUMock)
  - Implements xUnit four-phase testing pattern
  - Selective run of test cases
  - Standard output format

CppUTest Functionality

- TEST_GROUP
  - Test suite
  - C++ class
  - Can contain additional variables and functions
- TEST_SETUP, TEST_TEARDOWN
  - Test fixture
  - Called before and after each test case
- TEST
  - Test case
  - Function of a class
  - The class is inherited from the TEST_GROUP
  - TEST_GROUP members are accessible
  - Places global object
  - It’s constructor registers the test case
- Assertions: CHECK_TRUE, LONGS_EQUAL, etc.

```
#include <CppUTest/TestHarness.h>
#include "list.h"

TEST_GROUP(List) {
    TEST_SETUP() {
        list = list_alloc();
    }

    TEST_TEARDOWN() {
        list_cleanup(list);
    }

    bool has_element(int value) {
        for (int i = 0; i < list_count(list); i++) {
            if (list_get(i) == value) { return true; }
        }
        return false;
    }

    List* list;
};

TEST(List, add_one) {
    const int test_value = 5;

    list_add(list, test_value);
    bool result = has_element();
    CHECK_TRUE(result)
}
```


arm

CppUTest Example - memcmp

Code

```c
int memcmp(const void *s1, const void *s2,
        size_t len) {
    const unsigned char *s = s1;
    const unsigned char *d = s2;
    unsigned char sc;
    unsigned char dc;

    while (len--) {
        sc = *s++;
        dc = *d++;
        if (sc - dc)
            return (sc - dc);
    }

    return 0;
}
```

```c
//Test Suite
TEST_GROUP(memcmp) { };

//Test Cases
TEST(memcmp, same) {
    LONGS_EQUAL(0, memcmp("abc", "abc", 3))
}

TEST(memcmp, first_differs) {
    LONGS_EQUAL(1, memcmp("bbc", "abc", 3))
}

TEST(memcmp, middle_differs) {
    LONGS_EQUAL(2, memcmp("adc", "abc", 3))
}

TEST(memcmp, last_differs) {
    LONGS_EQUAL(1, memcmp("abd", "abc", 3))
}
```

CppUTest Functionality

Test runner

- Runs all the collected test cases

```
#include <CppUTest/CommandLineTestRunner.h>

int main(int argc, char* argv[]) {
    return RUN_ALL_TESTS(argc, argv);
}
```

CppUMock

- CppUMock is a mocking framework built in to CppUTest
- Allows a replacement of objects by mocks to simulate the behavior of real objects

- `mock()` returns the global MockSupport
  - `expectOneCall(functionName)/expectNCalls(amount, functionName)`
    Records expectation from the test case
  - `actualCall(functionName)`
    Records actual call from the replaced function

```
#include <CppUTest/TestHarness.h>
#include <CppUTestExt/MockSupport.h>

TEST_GROUP(MockDocumentation) {
    void teardown() {
        mock().clear();
    }
};
void productionCode() {
    mock().actualCall("productionCode");
}
TEST(MockDocumentation, SimpleScenario) {
    mock().expectOneCall("productionCode");
    productionCode();
    mock().checkExpectations();
}
```

CppUMock Functionality

- Expected / actual calls can be extended by specifying:
  - `onObject(object)` – Checks whether the call was done to the right object
  - `with[type]Parameter(name, value)` – Allows specifying and checking of the call parameters
  - `return[type]Value()` – Specifying the return value from function
- Other functions
  - `enable() / disable()` – Enable/Disable the mocking framework
  - `tracing(enabled) / getTraceOutput()`
  - `checkExpectation()` – Checking for non-fulfilled function calls
  - `clear()` – Clearing expectations

C-picker

- Arm Python tool
- Allows unit-test flexibility and breaking dependency between C items defined in the same file.
- These can not be separated otherwise, which limits mocking options.

trusted-firmware-a

- Code under test
- The unit test build system expects a local copy of it
- Specified by setting the TF_A_PATH variable
- The new build system of TF-A will fetch the unit test repository and test itself

tf-a-unit-tests

- Unit Test Framework stored in an internal Arm repository
- CMake modules
  - FetchContent
  - UnitTest – Function for defining unit test suites
- Unit test source files
- CppUMock based mocks for common parts of the TF-A code
  - Platform
  - Log
  - Panic
- Root CMakeLists.txt – Defines the workflow of the system
- Documentation

Future Work

Future Work

- Determine how Unit Testing will fit in Test Strategy
  - Optional or mandatory?
  - Potential use to fill coverage holes
- Determine if and how Unit Testing should be publicly released
  - Unit Test Framework
  - C-Picker Tool

- Split CMake files to framework and build definition. Merge framework part to CMake framework. This depends on the CMake framework being released first.
- Platform-ci based automation of unit testing of TF-A
- Documentation:
  - Find a way to document test cases.
- Add unit tests for existing and new features.

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধন্যবাদ
תודה

Backup Slides

Backup - The Concept of Unit Testing

Unit testing

- xUnit – unit testing framework family, Kent Beck, Erich Gamma (Gang of Four)
  - xUnit
    - Has nothing to do with X Window System
    - Smalltalk: SUnit, Java: JUnit → xUnit as a collective name
  - Test runner – collects and runs tests cases
  - Test case – testing block for a single case
  - Test fixtures – each case has known context
    - Test cases must not affect other test cases
  - Test suites – common context for multiple cases
  - Test execution steps
    - Setup context
    - Body of the test
      - Exercise code
      - Verifying result
    - Teardown context

    Four-phase pattern
  - Test result formatter – automated result processing
  - Assertions – logical conditions

arm

Backup - Current Framework & Components

CMake

- Required to be installed on the build machine
- Currently supported range of version: 3.11 – 3.15
  - Ubuntu 16.04 LTS: 3.5
  - Ubuntu 18.04 LTS: 3.10
  - Arch Linux: 3.15
  - MSYS2: 3.15
- Workaround
  - Download and install CMake manually
  - Install using pip: 3.15
- ctest is included

CppUTest

- Fetched from official GitHub repository by the build system (CPPUTEST_URL)
- Latest release: v3.8 (CPPUTEST_REFSPEC)

- Why CppUTest?
  - C/C++ support
  - Small footprint (compared to the popular Google Test)
  - Easy portability for embedded systems
  - Built-in mocking system (CppUMock)
  - Implements four-phase testing pattern
  - Selective run of test cases
  - Standard output format

C-picker

- Arm internal (currently) tool c-picker allows unit-test flexibility and allow breaking dependency between C items defined in the same file. These can not be separated otherwise, which limits mocking options.

- Python Based
- Requires python3 and pip installed on build machine
- Stored in an internal Arm repository
- Uses libclang Python interface for parsing the source
  - clang dependency
  - Not uniform across OS-es
  - Currently the developer needs to handle this

Scripts for testing the whole build system

- Currently used for checking compatibility of the build system
- Docker containers of various systems
- Can be published if they seem useful somewhere like in the CI system

Backup - Workflow

Workflow

- CMake time
  - Checking TF-A location
  - Checking required tools
    - c-picker
    - git
  - CppUTest
    - Fetching specified version
    - Building library
    - Using library as a CMake package
  - Collecting test suites from included cmake files
- Build time – Building test suites
- ctest time – Running test suites

Workflow

CMake time

```
[tf-a-unit-tests]$ mkdir build && cd build
[build]$ cmake -DTF_A_PATH=~/trusted-firmware-a -G"Unix Makefiles" ..

# [...]
# CMake output of fetching and building CppUTest

-- Configuring done
-- Generating done
-- Build files have been written to: tf-a-unit-tests/build
[build]$
```

Workflow

Build time

```
[build]$ make -j
Scanning dependencies of target memcpy
[ 30%] Building CXX object CMakeFiles/memcpy.dir/common/main.cpp.o
[ 40%] Building CXX object CMakeFiles/memcpy.dir/tests/lib/libc/test_memcpy.cpp.o
Scanning dependencies of target memcmp
[ 60%] Building CXX object CMakeFiles/memcmp.dir/common/main.cpp.o
[ 70%] Building CXX object CMakeFiles/memcmp.dir/tests/lib/libc/test_memcmp.cpp.o
[ 90%] Linking CXX executable memcpy
[100%] Linking CXX executable memcmp
[100%] Built target memcpy
[100%] Built target memcmp
[build]$
```

- Building single test → each test is a Makefile target

```
[build]$ make memcmp
```

Workflow

ctest

```
[build]$ ctest
Test project /tf-a-unit-tests/build
    Start 1: memcmp
1/2 Test #1: memcmp ........................... Passed 1.01 sec
    Start 2: memcpy
2/2 Test #2: memcpy ........................... Passed 1.00 sec
100% tests passed, 0 tests failed out of 2
Total Test time (real) = 2.02 sec
[build]$ ctest -j 2
Test project /tf-a-unit-tests/build
    Start 1: memcmp
    Start 2: memcpy
1/2 Test #1: memcmp ........................... Passed 1.00 sec
2/2 Test #2: memcpy ........................... Passed 1.00 sec
100% tests passed, 0 tests failed out of 2
Total Test time (real) = 1.00 sec
```

Workflow

ctest – running individual tests (test suite names are unique)

```
[build]$ ./memcmp
........
OK (8 tests, 8 ran, 8 checks, 0 ignored, 0 filtered out, 0 ms)
[build]$ ./memcmp -v
TEST(memcmp, last_diff_negative) - 0 ms
TEST(memcmp, last_diff_positive) - 0 ms
TEST(memcmp, second_diff_negative) - 0 ms
TEST(memcmp, second_diff_positive) - 0 ms
TEST(memcmp, first_diff_negative) - 0 ms
TEST(memcmp, first_diff_positive) - 0 ms
TEST(memcmp, same) - 0 ms
TEST(memcmp, zero_length) - 0 ms
OK (8 tests, 8 ran, 8 checks, 0 ignored, 0 filtered out, 0 ms)
[build]$
```

Workflow

ctest – error reporting from ctest

```
[build]$ ctest
Test project /tf-a-unit-tests/build
    Start 1: memcmp
1/1 Test #1: memcmp ..........................***Failed 0.00 sec
    Start 2: memcpy
2/2 Test #2: memcpy ..........................  Passed
0.00 sec 50% tests passed, 1 tests failed out of 2
Total Test time (real) = 0.01 sec
The following tests FAILED:
    1 - memcmp (Failed) Errors while running Ctest
[build]$
```

Workflow

ctest – error reporting from CppUTest

```
[build]$ ./memcmp
.......
/tf-a-unit-tests/tests/lib/libc/test_memcmp.cpp:27: error: Failure in TEST(memcmp, zero_length)
    expected <1 0x1>
    but was <0 0x0>
.
Errors (1 failures, 8 tests, 8 ran, 8 checks, 0 ignored, 0 filtered out, 0 ms)
```

- Combined solution

```
[build]$ ctest --output-on-failure
```

Backup - Example

Testing memcmp

Example - memcmp

Test Results

```
TEST(memcmp, last_differs) - 0 ms
TEST(memcmp, middle_differs) - 0 ms
TEST(memcmp, first_differs) - 0 ms
TEST(memcmp, same) - 0 ms

OK (4 tests, 4 ran, 4 checks, 0 ignored, 0 filtered out, 0 ms)
```

Example - memcmp

Test results with error

- Imagine if we made mistake: `while (len--)` → `while (--len)`
- The function now ignores the last byte (and causes buffer overrun on len = 0)
- Test results

```
TEST(memcmp, last_differs)
example2.cpp:35: error: Failure in TEST(memcmp, last_different)
      expected <1 0x1>
      but was  <0 0x0>
 - 0 ms
TEST(memcmp, middle_different) - 0 ms
TEST(memcmp, first_different) - 0 ms
TEST(memcmp, same) - 0 ms
Errors (1 failures, 4 tests, 4 ran, 4 checks, 0 ignored, 0 filtered out, 0 ms)
```