Zephyr Testing @ Google

Yuval Peress
peress@google.com

Zephyr Developer Summit, 2022-06-08

yuval#5515 yperess

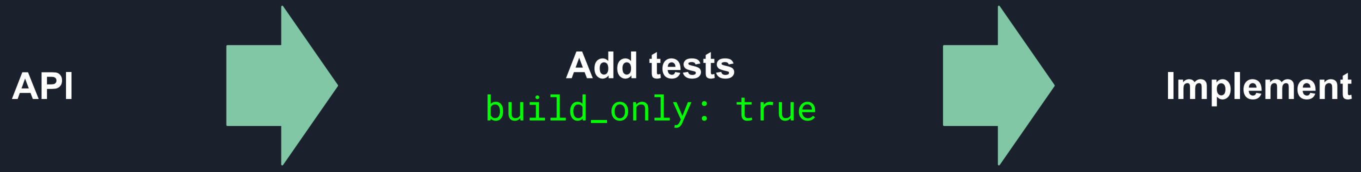
Save time with twister

- Naturally parallel builds
- Incremental build (`-n`)
- Rerun failed tests (`-f`)
- Generate coverage reports (`--coverage`)
- Use `-G` for integration mode (fast) and omit for nightly builds

- Run the same test for various configurations

```
tests:
  mathlib.float:
    extra_configs:
      - CONFIG_FPU=y
  mathlib.fixed:
    extra_configs:
      - CONFIG_FPU=n
```

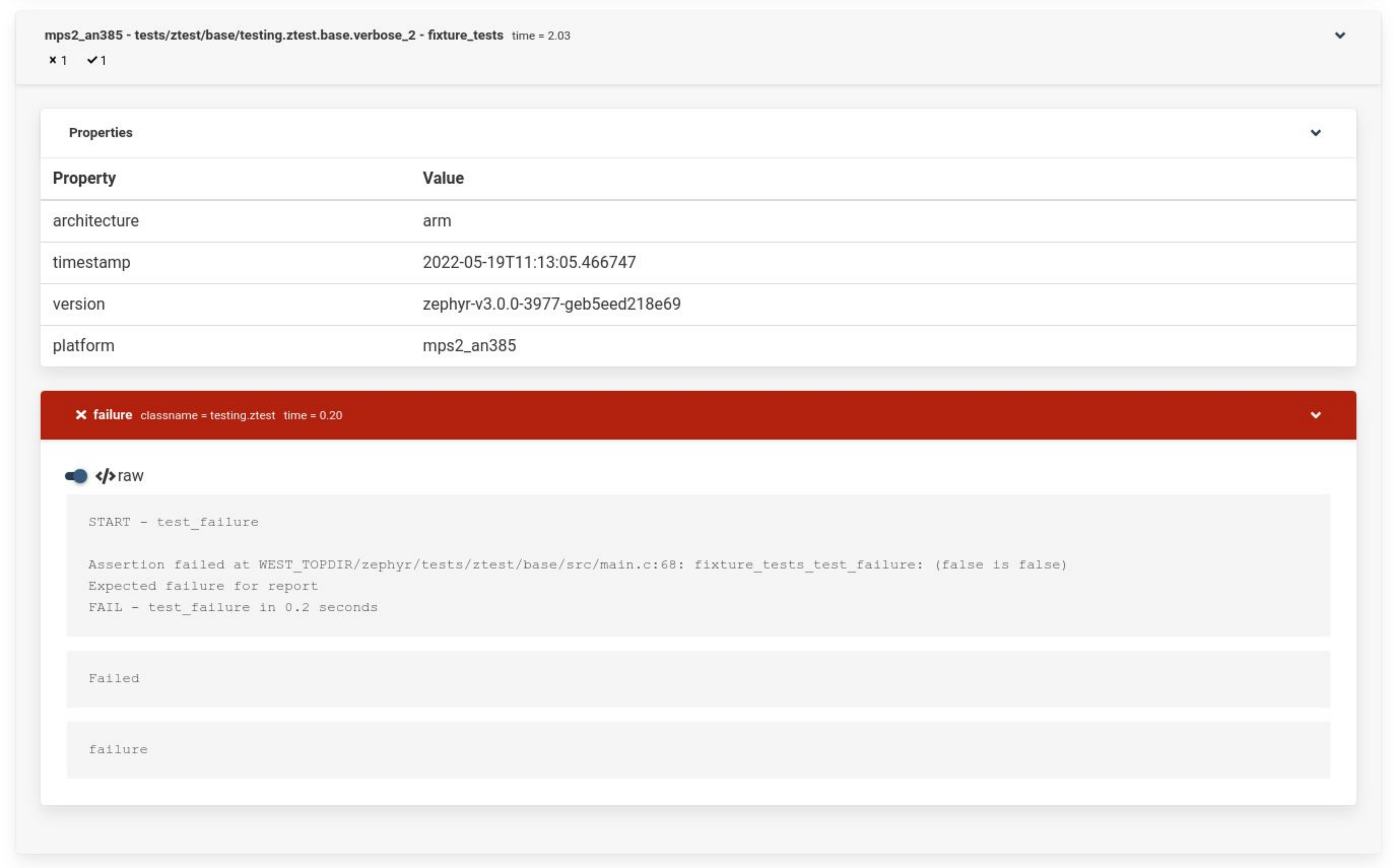
Test Driven Development



API

Add tests
`build_only: true`

Implement



mps2_an385 - tests/ztest/base/testing.ztest.base.verbose_2 - fixture_tests time = 2.03

✗ 1 ✓ 1

Properties

Property	Value
architecture	arm
timestamp	2022-05-19T11:13:05.466747
version	zephyr-v3.0.0-3977-geb5eed218e69
platform	mps2_an385

✗ **failure** classname = testing.ztest time = 0.20

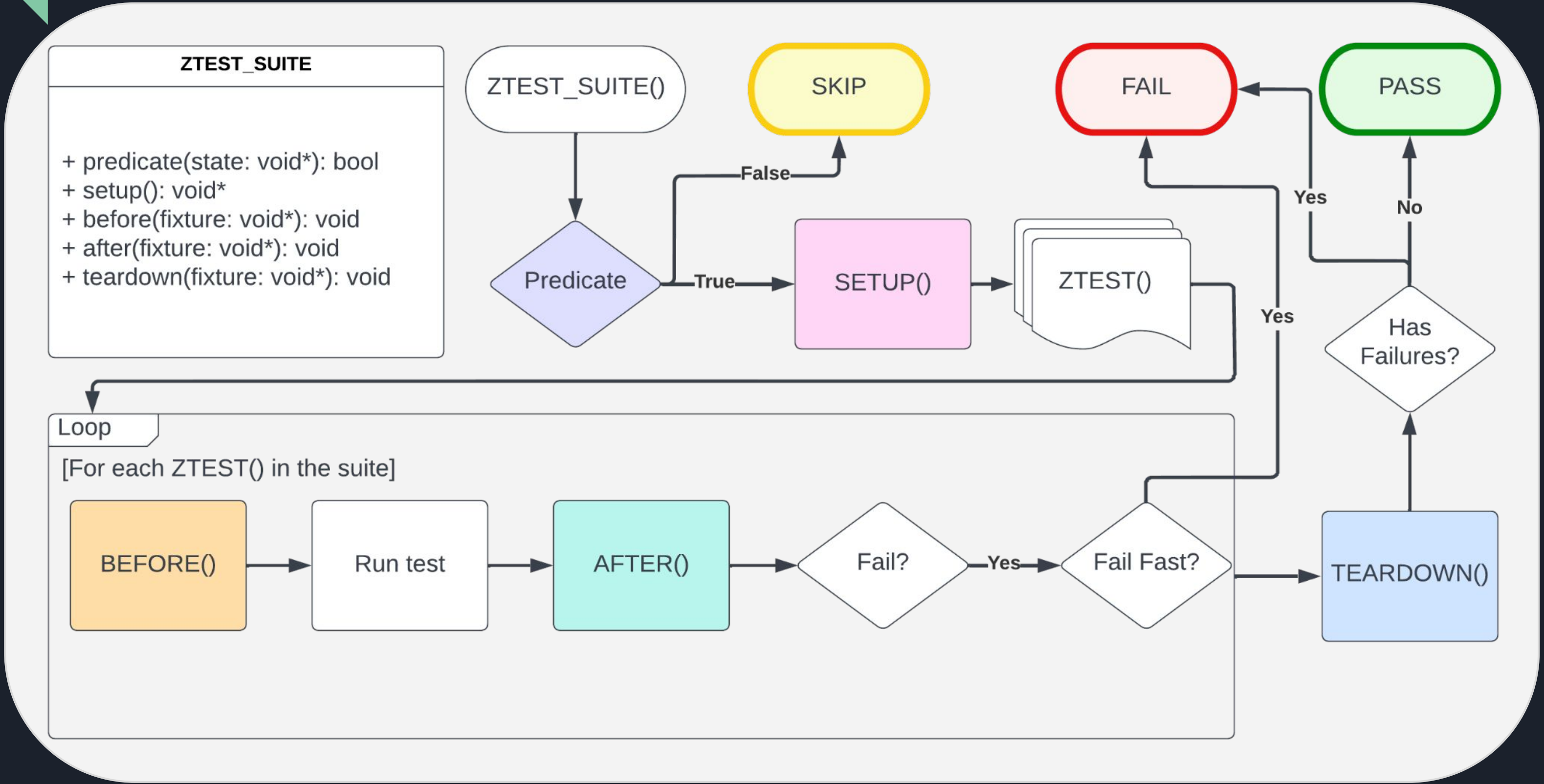
```
START - test_failure

Assertion failed at WEST_TOPDIR/zephyr/tests/ztest/base/src/main.c:68: fixture_tests_test_failure: (false is false)
Expected failure for report
FAIL - test_failure in 0.2 seconds
```

```
Failed
```

```
failure
```

ZTEST_SUITE(...)


ZTEST_SUITE
+ predicate(state: void*): bool
+ setup(): void*
+ before(fixture: void*): void
+ after(fixture: void*): void
+ teardown(fixture: void*): void
ZTEST_SUITE()
SKIP
FAIL
PASS
False
Predicate
True
SETUP()
ZTEST()
Yes
Yes
No
Has Failures?
Loop
[For each ZTEST() in the suite]
BEFORE()
Run test
AFTER()
Fail?
Yes
Fail Fast?
TEARDOWN()

When to use predicates?

```c
struct test_state {
  int count;
};

static bool my_suite_predicate(const void *state) {
  return ((const struct test_state *)state)->count == 2;
}

ZTEST_SUITE(my_suite, my_suite_predicate, NULL, NULL, NULL, NULL);

void test_main(void) {
  struct test_state state = { .count = 0, };
  for (; state.count < 10; state.count++) {
    ztest_run_test_suites(&state);
  }
  ztest_verify_all_test_suites_ran();
}
```

** Read more about predicates in the documentation*

Modular design: tests

```
ZTEST(my_suite, test_0) {
 // Normal test, runs when my_suite runs
}

ZTEST_USER(my_suite, test_1) {
 // Runs like test_0 but the thread is in
 // userspace if enabled.
}
```

```
ZTEST_F(my_suite, test_2) {
  // Normal test but also get ‘this’ which
  // has the type:
  // ‘struct my_suite_fixture *’
}

ZTEST_USER_F(my_suite, test_3) {
  // Same as ‘test_1’ but includes ‘this’
  // which has the type
  // ‘struct my_suite_fixture *’
}
```

Assert, Expect, & Assume

```c
ZTEST(my_suite, test_fn) {
  // Assume that configure_component() will work.
  // If not, mark the test as skipped.
  zassume_ok(configure_component(), NULL);

  // Expect both of these to be true.
  // If one fails, keep going but the test will be considered
  // as ‘failed’.
  zexpect_equal(5, get_component_value0(), NULL);
  zexpect_equal(7, get_component_value1(), NULL);

  // Assert that this is true, ‘fail’ the test immediately if not.
  zassert_ok(component_shutdown(), NULL);
}
```

Modularity

ZTEST() and ZTEST_SUITE() can be in different .c files

```
### CMakeLists.txt

# Add the test suite
zephyr_library_sources(my_test_suite.c)

# Add tests based on a Kconfig
zephyr_library_sources_ifdef(
    CONFIG_OPTION1_NAME feature_tests_for_option1.c)

# Get the path for i2c0 nodelabel
dt_nodelabel(i2c0_path NODELABEL “i2c0”)

# Add tests if path exists
if(i2c0_path)
  zephyr_library_sources(tests_for_i2c0.c)
endif()
```

Modular design: test rules

- Modeled after junit test rules
- Have access to both the current test (via `const struct ztest_unit_test *`) and the fixture (via `void *`).
- Provide global before/after functions for every test in every suite.

```
static void my_rule_before(
    const struct ztest_unit_test *test,
    void *fixture
) {...}

static void my_rule_after(
    const struct ztest_unit_test *test,
    void *fixture
) {...}

ZTEST_RULE(my_rule_name, my_rule_before, my_rule_after);
```

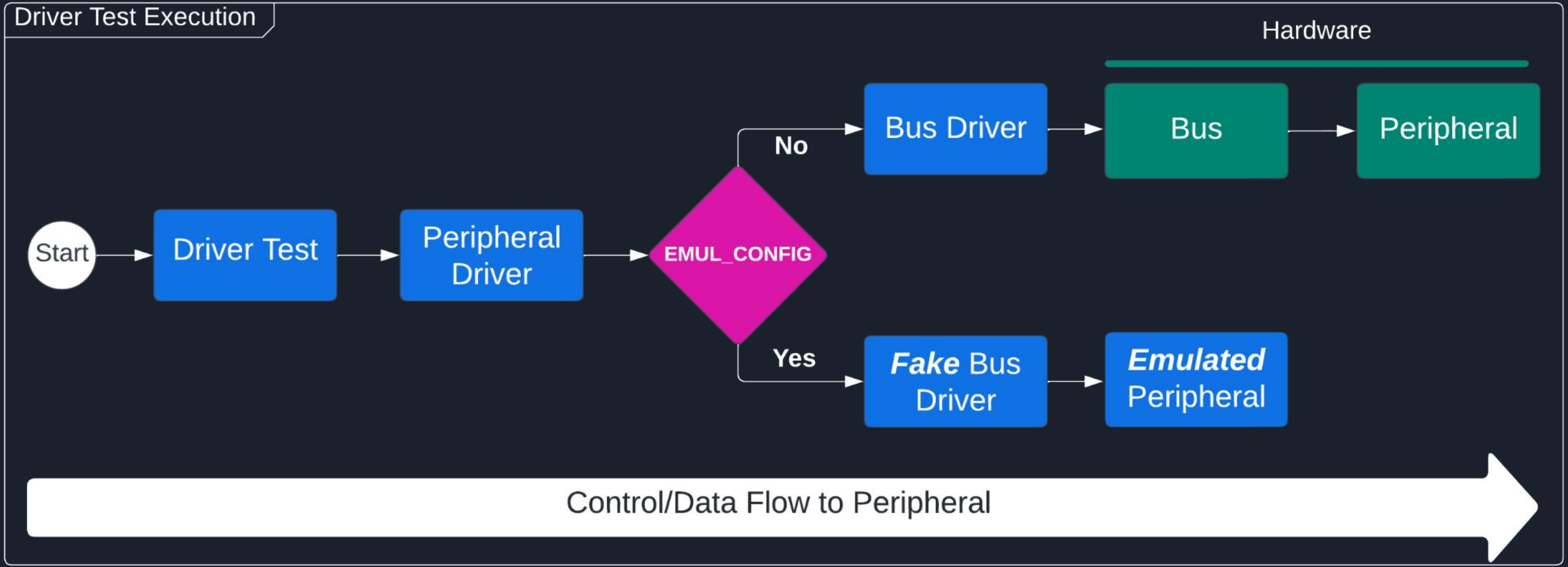
Mocking (FFF)

- Overriding plain `__attribute__((weak))` functions has limitations when multiple code paths are involved.
- FFF provides a means for a single mock function that can be augmented per test.
- When combined with C++, this becomes powerful:

```
FAKE_VOID_FUNC(func_to_mock);

ZTEST_F(my_suite, test_with_side_effect) {
  // Use a lambda function to encapsulate the custom fake and also
  // capture the fixture ‘this’.
  func_to_mock_fake.custom_fake = [this]() {
    // Custom logic with side-effects.
  };
}
```

Emulating peripherals



Enabling Zephyr emulators

Application Kconfig

```
# Enable the I2C bus
CONFIG_I2C=y

# Enable my device
CONFIG_MY_DEV=y
```

Test Kconfig

```
# Device-Emulators
CONFIG_EMUL=y

# Enable I2C emulation
CONFIG_I2C_EMUL=y

# My device emulator
CONFIG_EMUL_MY_DEV=y
```

Board specific Device Tree

```
/ {
  i2c0 {
    compatible = “vndr,i2c”;
  };
};
#include “i2c0_peripherals.dtsi”
```

Test Specific Device Tree

```
/ {
  i2c0 {
    compatible = “zephyr,i2c-emul-controller”;
  };
};
#include “i2c0_peripherals.dtsi”
```

Defining an emulator

```c
// This function handles all the communication with the I2C bus.
// Read data can be written to msgs[n].buff
static int my_emul_transfer_i2c(const struct emul *emulator, struct i2c_msg *msgs, int num_msgs,
                                int addr) {
  LOG_INF("received %d I2C messages @0x%p", num_msgs, (void*)addr);
  for (int i = 0; i < num_msgs; ++i) {
    LOG_INF("msg[%d](len=%u, flags=0x%02x)", i, msgs[i].len, msgs[i].flags);
  }
}

static struct i2c_emul_api my_emul_i2c_api = {
   .transfer = my_emul_transfer_i2c,
};

#define MY_EMUL_I2C(n)                                                                        \
   MY_EMUL_DATA(n);                                                                           \
   MY_EMUL_CONFIG(n);                                                                         \
   EMUL_DT_INST_DEFINE(n, my_emul_init, &my_emul_cfg_##n, &my_emul_data_##n, &my_emul_i2c_api)

#define MY_EMUL_DEF(n)                                                                        \
   COND_CODE_1(DT_INST_ON_BUS(n, spi),                                                        \
   (MY_EMUL_SPI(n)),                                                                          \
   (MY_EMUL_I2C(n)))

DT_INST_FOREACH_STATUS_OKAY(MY_EMUL_DEF)
```

Reliability of tests

- Shuffle (test-order-independency)
    - Enable KConfig option `ZTEST_SHUFFLE=y` to randomize order tests are executed.
    - Twister reports the seed value used on failing test cases*.
    - Helpful in identifying tests that don't have proper setup or teardown.
- Repeatability
    - Twister accepts `--seed` argument to reproduce test sequence used in Shuffling*.
    - KConfig options to repeat suites and tests

      ```
      ZTEST_SHUFFLE_SUITE_REPEAT_COUNT=5
      ZTEST_SHUFFLE_TEST_REPEAT_COUNT=3
      ```
- Test Selection*
    - The executable accepts `-test=suite_a::test_1,suite_a::test_2,suite_b::*` argument to run selected tests.
    - Helpful for debugging test cases under development.

* POSIX only