RTEMS Contributor Guide

Release 5.0.0 (master)

CONTENTS

The RTEMS Project is hosted at http://www.rtems.org/. Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to the Community Project hosted at http://www.rtems.org/.

RTEMS Online Resources

Home	https://www.rtems.org/
Developers	https://devel.rtems.org/
Documentation	https://docs.rtems.org/
Bug Reporting	https://devel.rtems.org/query
Mailing Lists	https://lists.rtems.org/
Git Repositories	https://git.rtems.org/

CHAPTER

ONE

PREFACE

Bla blub.

CHAPTER

TWO

TEST FRAMEWORK

2.1 The T Test Framework

The *T Test Framework* helps to write test suites. A *test suite* is a collection of test cases. A *test case* consists of individual tests. A *test* checks if the actual value presented to the test meets its expectation. If the actual value is all right, then the test passes, otherwise the test fails. The test failures of a test case are summed up. A test case passes, if the failure count of this test case is zero, otherwise the test case fails. The test suite passes if all test cases pass, otherwise it fails.

2.1.1 Test Cases

You can write a test case with the *T_TEST_CASE()* macro followed by a function body:

```
T_TEST_CASE(name)
{
   /* Your test case code */
}
```

The test case *name* must be a valid C designator. The test case names must be unique within the test suite. Just link modules with test cases to the test runner to form a test suite. The test cases are automatically registered via static constructors.

Listing 1: Test Case Example

```
#include <t.h>

T_TEST_CASE(a_test_case)
{
    T_eq_int(0, 0);
    T_true(false, "not good");
}
```

Listing 2: Test Case Report

```
B:a_test_case
P:0:0:UI1:test-simple.c:5
F:1:0:UI1:test-simple.c:6:not good
Y:a_test_case:N:2:F:1:D:0.001657
```

The *B* line indicates the begin of test case *a_test_case*. The *P* line shows that the test in file *test-simple.c* at line 5 executed by task *UI1* on processor 0 as the test step 0 passed. The *F* lines shows that the test in file *test-simple.c* at line 6 executed by task *UI1* on processor 0 as the test step 1 failed with a message of *"not good"*. The *Y* line indicates the end of test case *a_test_case* resulting in a total of two test steps (*N*) and one test failure (*F*). The test case duration (*D*) was 0.001657 seconds. For test report details see: *Test Reporting*.

2.1.2 Test Fixture

You can write a test case with a test fixture with the *T_TEST_CASE_2()* macro followed by a function body:

```
T_TEST_CASE_2(name, fixture)
{
```

(continues on next page)

(continued from previous page)

```
3     /* Your test case code */
4  }
```

The test case *name* must be a valid C designator. The test case names must be unique within the test suite. The *fixture* must point to a statically initialized read-only object of type *T_test_fixture*. The test fixture provides methods to setup, stop and tear down a test case. A context is passed to the methods. The initial context is defined by the read-only fixture object. The context can be obtained by the *T_test_fixture_context()* function. It can be set within the scope of one test case by the *T_test_fixture_set_context()* function. This can be used for example to dynamically allocate a test environment in the setup method.

Listing 3: Test Fixture Example

```
1  #include <t.h>
2
3  static int initial_value = 3;
4
5  static int counter;
6
7  static void
8  setup(void *ctx)
9  {
10     int *c;
11
12     T_log(T_QUIET, "setup begin");
13     T_eq_ptr(ctx, &initial_value);
14     T_eq_ptr(ctx, T_test_fixture_context());
15     c = ctx;
16     counter = *c;
17     T_test_fixture_set_context(&counter);
18     T_eq_ptr(&counter, T_test_fixture_context());
19     T_log(T_QUIET, "setup end");
20 }
21
22 static void
23 stop(void *ctx)
24 {
25     int *c;
26
27     T_log(T_QUIET, "stop begin");
28     T_eq_ptr(ctx, &counter);
29     c = ctx;
30     ++(*c);
31     T_log(T_QUIET, "stop end");
32 }
33
34 static void
35 teardown(void *ctx)
36 {
37     int *c;
38
39     T_log(T_QUIET, "teardown begin");
40     T_eq_ptr(ctx, &counter);
41     c = ctx;
42     T_eq_int(*c, 4);
```

(continues on next page)

(continued from previous page)

```
    T_log(T_QUIET, "teardown end");
}

static const T_test_fixture fixture = {
    .setup = setup,
    .stop = stop,
    .teardown = teardown,
    .initial_context = &initial_value
};

T_TEST_CASE_2(fixture, &fixture)
{
    T_assert_true(true, "all right");
    T_assert_true(false, "test fails and we stop the test case");
    T_log(T_QUIET, "not reached");
}
```

Listing 4: Test Fixture Report

```
B:fixture
L:setup begin
P:0:0:UI1:test-fixture.c:13
P:1:0:UI1:test-fixture.c:14
P:2:0:UI1:test-fixture.c:18
L:setup end
P:3:0:UI1:test-fixture.c:55
F:4:0:UI1:test-fixture.c:56:test fails and we stop the test case
L:stop begin
P:5:0:UI1:test-fixture.c:28
L:stop end
L:teardown begin
P:6:0:UI1:test-fixture.c:40
P:7:0:UI1:test-fixture.c:42
L:teardown end
Y:fixture:N:8:F:1
```

2.1.3 Test Planning

Each non-quiet test fetches and increments the test step counter atomically. For each test case execution the planned steps can be specified with the *T_plan()* function.

```
void T_plan(unsigned int planned_steps);
```

This function must be invoked at most once in each test case execution. If the planned test steps are set with this function, then the final test steps after the test case execution must be equal to the planned steps, otherwise the test fails.

Use the *T_step_*(step, ...)* test variants to ensure that the test case execution follows exactly the planned steps.

Listing 5: Test Planning Example

```
#include <t.h>

T_TEST_CASE(wrong_step)
{
    T_plan(2);
    T_step_true(0, true, "all right");
    T_step_true(2, true, "wrong step");
}

T_TEST_CASE(plan_ok)
{
    T_plan(1);
    T_step_true(0, true, "all right");
}

T_TEST_CASE(plan_failed)
{
    T_plan(2);
    T_step_true(0, true, "not enough steps");
    T_quiet_true(true, "quiet test do not count");
}

T_TEST_CASE(double_plan)
{
    T_plan(99);
    T_plan(2);
}

T_TEST_CASE(steps)
{
    T_step(0, "a");
    T_plan(3);
    T_step(1, "b");
    T_step(2, "c");
}
```

Listing 6: Test Planning Report

```
B:wrong_step
P:0:0:UI1:test-plan.c:6
F:1:0:UI1:test-plan.c:7:planned step (2)
Y:wrong_step:N:2:F:1
B:plan_ok
P:0:0:UI1:test-plan.c:13
Y:plan_ok:N:1:F:0
B:plan_failed
P:0:0:UI1:test-plan.c:19
F:*:0:UI1:*:*:actual steps (1), planned steps (2)
Y:plan_failed:N:1:F:1
B:double_plan
F:*:0:UI1:*:*:planned steps (99) already set
Y:double_plan:N:0:F:1
B:steps
P:0:0:UI1:test-plan.c:31
```

(continues on next page)

(continued from previous page)

```
P:1:0:UI1:test-plan.c:33
P:2:0:UI1:test-plan.c:34
Y:steps:N:3:F:0
```

2.1.4 Resource Accounting

The framework can check if various resources are leaked during a test case execution. The resource checkers are specified by the test run configuration. On RTEMS, checks for the following resources are available

- workspace and heap memory,
- file descriptors,
- POSIX keys and key value pairs,
- RTEMS barriers,
- RTEMS user extensions,
- RTEMS message queues,
- RTEMS partitions,
- RTEMS periods,
- RTEMS regions,
- RTEMS semaphores,
- RTEMS tasks, and
- RTEMS timers.

Listing 7: Resource Accounting Example

```
#include <t.h>

#include <stdlib.h>

#include <rtems.h>

T_TEST_CASE(missing_sema_delete)
{
    rtems_status_code sc;
    rtems_id id;

    sc = rtems_semaphore_create(rtems_build_name('S', 'E', 'M', 'A'), 0,
        RTEMS_COUNTING_SEMAPHORE, 0, &id);
    T_rsc_success(sc);
}

T_TEST_CASE(missing_free)
{
    void *p;

    p = malloc(1);
```

(continues on next page)

(continued from previous page)

```
    T_not_null(p);
}
```

Listing 8: Resource Accounting Report

```
B:missing_sema_delete
P:0:0:UI1:test-leak.c:14
F:*:0:UI1:*:*:RTEMS semaphore leak (1)
Y:missing_sema_delete:N:1:F:1:D:0.004013
B:missing_free
P:0:0:UI1:test-leak.c:22
F:*:0:UI1:*:*:memory leak in workspace or heap
Y:missing_free:N:1:F:1:D:0.003944
```

2.1.5 Test Case Scoped Resources

You can allocate resources which are automatically destroyed after the test case execution.

```
void *T_malloc(size_t size);

void *T_calloc(size_t nelem, size_t elsize);

void *T_zalloc(size_t size);

void T_free(void *ptr);
```

Listing 9: Test Case Scoped Resource Example

```
#include <t.h>

T_TEST_CASE(malloc_free)
{
    void *p;

    p = T_malloc(1);
    T_not_null(p);
    T_free(p);
}

T_TEST_CASE(malloc_auto)
{
    void *p;

    p = T_malloc(1);
    T_not_null(p);
}
```

Listing 10: Test Case Scoped Resource Report

```
B:malloc_free
P:0:0:UI1:test-malloc.c:8
Y:malloc_free:N:1:F:0
B:malloc_auto
```

(continues on next page)

(continued from previous page)

```
P:0:0:UI1:test-malloc.c:17
Y:malloc_auto:N:1:F:0
```

2.1.6 Tests

A *test* checks if the actual value presented to the test meets its expectation. If the actual value is all right, then the test passes, otherwise the test fails. A failed test does not stop the test case execution immediately unless the *T_assert_*()* test variant is used. Each test increments a test step counter unless the *T_quiet_*()* test variant is used. The test step counter is initialized to zero before the test case begins to execute. The *T_step_*(step, ...)* test variants check that the test step counter is equal to the specified test step value, otherwise the test fails.

2.1.6.1 Test Parameter Conventions

The following names for test parameters are used throughout the tests:

step
The planned test step for this test.

a
The actual value to test against an expected value. It is usually the first parameter in all tests, except in the *T_step_*(step, ...)* test variants, here it is the second parameter.

e
The expected value of a test. This parameter is optional. Some tests have an implicit expected value. If present, then this parameter is directly after the actual value parameter of the test.

fmt
A printf()-like format string. Floating-point and exotic formats may be not supported.

2.1.6.2 Test Condition Conventions

The following names for test conditions are used:

eq
The actual value must equal the expected value.

ne
The actual value must not equal the value of the second parameter.

ge
The actual value must be greater than or equal to the expected value.

gt
The actual value must be greater than the expected value.

le
The actual value must be less than or equal to the expected value.

lt
The actual value must be less than the expected value.

If the actual value satisfies the test condition, then the test passes, otherwise it fails.

2.1.6.3 Test Variant Conventions

The *T_assert_*()* test variants stop the current test case execution if the test fails.

The *T_quite_*()* test variants do not increase the test step counter and only print a message if the test fails. This is helpful in case a test appears in a tight loop.

The *T_step_*(step, ...)* test variants check that the test step counter is equal to the specified test step value, otherwise the test fails.

The following names for test type variants are used:

ptr
The test value must be a pointer (*void **).

mem
The test value must be a memory area with a specified length.

str
The test value must be a null byte terminated string.

nstr
The length of the test value string is limited to a specified maximum.

char
The test value must be a character (*char*).

schar
The test value must be a signed character (*signed char*).

uchar
The test value must be an unsigned character (*unsigned char*).

short
The test value must be a short integer (*short*).

ushort
The test value must be an unsigned short integer (*unsigned short*).

int
The test value must be an integer (*int*).

uint
The test value must be an unsigned integer (*unsigned int*).

long
The test value must be a long integer (*long*).

ulong
The test value must be an unsigned long integer (*unsigned long*).

ll
The test value must be a long long integer (*long long*).

ull
The test value must be an unsigned long long integer (*unsigned long long*).

i8
The test value must be a signed 8-bit integer (*int8_t*).

u8
The test value must be an unsigned 8-bit integer (*uint8_t*).

i16
The test value must be a signed 16-bit integer (*int16_t*).

u16
The test value must be an unsigned 16-bit integer (*uint16_t*).

i32
The test value must be a signed 32-bit integer (*int32_t*).

u32
The test value must be an unsigned 32-bit integer (*uint32_t*).

i64
The test value must be a signed 64-bit integer (*int64_t*).

u64
The test value must be an unsigned 64-bit integer (*uint64_t*).

iptr
The test value must be of type *intptr_t*.

uptr
The test value must be of type *uintptr_t*.

ssz
The test value must be of type *ssize_t*.

sz
The test value must be of type *size_t*.

2.1.6.4 Boolean Expressions

The following tests for boolean expressions are available:

```
void T_true(bool a, const char *fmt, ...);
void T_assert_true(bool a, const char *fmt, ...);
void T_quiet_true(bool a, const char *fmt, ...);
void T_step_true(unsigned int step, bool a, const char *fmt, ...);

void T_false(bool a, const char *fmt, ...);
void T_assert_false(bool a, const char *fmt, ...);
void T_quiet_true(bool a, const char *fmt, ...);
void T_step_true(unsigned int step, bool a, const char *fmt, ...);
```

The message is only printed in case the test fails. The format parameter is mandatory.

Listing 11: Boolean Tests Example

```
#include <t.h>

T_TEST_CASE(example)
{
    T_true(true, "test passes, no message output");
    T_true(false, "test fails");
```

(continues on next page)

(continued from previous page)

```
    T_quiet_true(true, "quiet test passes, no output at all");
    T_quiet_true(false, "quiet test fails");
    T_step_true(2, true, "step test passes, no message output");
    T_step_true(3, false, "step test fails");
    T_assert_false(true, "this is a format %s", "string");
}
```

Listing 12: Boolean Tests Report

```
B:example
P:0:0:UI1:test-example.c:5
F:1:0:UI1:test-example.c:6:test fails
F:*:0:UI1:test-example.c:8:quiet test fails
P:2:0:UI1:test-example.c:9
F:3:0:UI1:test-example.c:10:step test fails
F:4:0:UI1:test-example.c:11:this is a format string
Y:example:N:5:F:4
```

2.1.6.5 Generic Types

The following tests for data types with an equality (==) or inequality (!=) operator are available:

```
void T_eq(T a, T e, const char *fmt, ...);
void T_assert_eq(T a, T e, const char *fmt, ...);
void T_quiet_eq(T a, T e, const char *fmt, ...);
void T_step_eq(unsigned int step, T a, T e, const char *fmt, ...);

void T_ne(T a, T e, const char *fmt, ...);
void T_assert_ne(T a, T e, const char *fmt, ...);
void T_quiet_ne(T a, T e, const char *fmt, ...);
void T_step_ne(unsigned int step, T a, T e, const char *fmt, ...);
```

The type name *T* specifies an arbitrary type which must support the corresponding operator. The message is only printed in case the test fails. The format parameter is mandatory.

2.1.6.6 Pointers

The following tests for pointers are available:

```
void T_eq_ptr(const void *a, const void *e);
void T_assert_eq_ptr(const void *a, const void *e);
void T_quiet_eq_ptr(const void *a, const void *e);
void T_step_eq_ptr(unsigned int step, const void *a, const void *e);

void T_ne_ptr(const void *a, const void *e);
void T_assert_ne_ptr(const void *a, const void *e);
void T_quiet_ne_ptr(const void *a, const void *e);
void T_step_ne_ptr(unsigned int step, const void *a, const void *e);

void T_null(const void *a);
void T_assert_null(const void *a);
```

(continues on next page)

(continued from previous page)

```
void T_quiet_null(const void *a);
void T_step_null(unsigned int step, const void *a);

void T_not_null(const void *a);
void T_assert_not_null(const void *a);
void T_quiet_not_null(const void *a);
void T_step_not_null(unsigned int step, const void *a);
```

An automatically generated message is printed in case of a test failure.

2.1.6.7 Memory Areas

The following tests for memory areas are available:

```
void T_eq_mem(const void *a, const void *e, size_t n);
void T_assert_eq_mem(const void *a, const void *e, size_t n);
void T_quiet_eq_mem(const void *a, const void *e, size_t n);
void T_step_eq_mem(unsigned int step, const void *a, const void *e, size_t n);

void T_ne_mem(const void *a, const void *e, size_t n);
void T_assert_ne_mem(const void *a, const void *e, size_t n);
void T_quiet_ne_mem(const void *a, const void *e, size_t n);
void T_step_ne_mem(unsigned int step, const void *a, const void *e, size_t n);
```

The *memcmp()* function is used to compare the memory areas. An automatically generated message is printed in case of a test failure.

2.1.6.8 Strings

The following tests for strings are available:

```
void T_eq_str(const char *a, const char *e);
void T_assert_eq_str(const char *a, const char *e);
void T_quiet_eq_str(const char *a, const char *e);
void T_step_eq_str(unsigned int step, const char *a, const char *e);

void T_ne_str(const char *a, const char *e);
void T_assert_ne_str(const char *a, const char *e);
void T_quiet_ne_str(const char *a, const char *e);
void T_step_ne_str(unsigned int step, const char *a, const char *e);

void T_eq_nstr(const char *a, const char *e, size_t n);
void T_assert_eq_nstr(const char *a, const char *e, size_t n);
void T_quiet_eq_nstr(const char *a, const char *e, size_t n);
void T_step_eq_nstr(unsigned int step, const char *a, const char *e, size_t n);

void T_ne_nstr(const char *a, const char *e, size_t n);
void T_assert_ne_nstr(const char *a, const char *e, size_t n);
void T_quiet_ne_nstr(const char *a, const char *e, size_t n);
void T_step_ne_nstr(unsigned int step, const char *a, const char *e, size_t n);
```

The *strcmp()* and *strncmp()* functions are used to compare the strings. An automatically generated message is printed in case of a test failure.

2.1.6.9 Characters

The following tests for characters (*char*) are available:

```
void T_eq_char(char a, char e);
void T_assert_eq_char(char a, char e);
void T_quiet_eq_char(char a, char e);
void T_step_eq_char(unsigned int step, char a, char e);

void T_ne_char(char a, char e);
void T_assert_ne_char(char a, char e);
void T_quiet_ne_char(char a, char e);
void T_step_ne_char(unsigned int step, char a, char e);
```

An automatically generated message is printed in case of a test failure.

2.1.6.10 Integers

The following tests for integers are available:

```
void T_eq_xyz(I a, I e);
void T_assert_eq_xyz(I a, I e);
void T_quiet_eq_xyz(I a, I e);
void T_step_eq_xyz(unsigned int step, I a, I e);

void T_ne_xyz(I a, I e);
void T_assert_ne_xyz(I a, I e);
void T_quiet_ne_xyz(I a, I e);
void T_step_ne_xyz(unsigned int step, I a, I e);

void T_ge_xyz(I a, I e);
void T_assert_ge_xyz(I a, I e);
void T_quiet_ge_xyz(I a, I e);
void T_step_ge_xyz(unsigned int step, I a, I e);

void T_gt_xyz(I a, I e);
void T_assert_gt_xyz(I a, I e);
void T_quiet_gt_xyz(I a, I e);
void T_step_gt_xyz(unsigned int step, I a, I e);

void T_le_xyz(I a, I e);
void T_assert_le_xyz(I a, I e);
void T_quiet_le_xyz(I a, I e);
void T_step_le_xyz(unsigned int step, I a, I e);

void T_lt_xyz(I a, I e);
void T_assert_lt_xyz(I a, I e);
void T_quiet_lt_xyz(I a, I e);
void T_step_lt_xyz(unsigned int step, I a, I e);
```

The type variant *xyz* must be *schar*, *uchar*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *ll*, *ull*, *i8*, *u8*, *i16*, *u16*, *i32*, *u32*, *i64*, *u64*, *iptr*, *uptr*, *ssz*, or *sz*.

The type name *I* must be compatible to the type variant.

An automatically generated message is printed in case of a test failure.

2.1.6.11 RTEMS Status Codes

The following tests for RTEMS status codes are available:

```
void T_rsc(rtems_status_code a, rtems_status_code e);
void T_assert_rsc(rtems_status_code a, rtems_status_code e);
void T_quiet_rsc(rtems_status_code a, rtems_status_code e);
void T_step_rsc(unsigned int step, rtems_status_code a, rtems_status_code e);

void T_rsc_success(rtems_status_code a);
void T_assert_rsc_success(rtems_status_code a);
void T_quiet_rsc_success(rtems_status_code a);
void T_step_rsc_success(unsigned int step, rtems_status_code a);
```

An automatically generated message is printed in case of a test failure.

2.1.6.12 POSIX Error Numbers

The following tests for POSIX error numbers are available:

```
void T_eno(int a, int e);
void T_assert_eno(int a, int e);
void T_quiet_eno(int a, int e);
void T_step_eno(unsigned int step, int a, int e);

void T_eno_success(int a);
void T_assert_eno_success(int a);
void T_quiet_eno_success(int a);
void T_step_eno_success(unsigned int step, int a);
```

The actual and expected value must be a POSIX error number, e.g. EINVAL, ENOMEM, etc. An automatically generated message is printed in case of a test failure.

2.1.6.13 POSIX Status Codes

The following tests for POSIX status codes are available:

```
void T_psx_error(int a, int eno);
void T_assert_psx_error(int a, int eno);
void T_quiet_psx_error(int a, int eno);
void T_step_psx_error(unsigned int step, int a, int eno);

void T_psx_success(int a);
void T_assert_psx_success(int a);
void T_quiet_psx_success(int a);
void T_step_psx_success(unsigned int step, int a);
```

The *eno* value must be a POSIX error number, e.g. EINVAL, ENOMEM, etc. An actual value of zero indicates success. An actual value of minus one indicates an error. An automatically generated message is printed in case of a test failure.

Listing 13: POSIX Status Code Example

```
#include <t.h>

#include <sys/stat.h>
#include <errno.h>

T_TEST_CASE(stat)
{
    struct stat st;
    int status;

    errno = 0;
    status = stat("foobar", &st);
    T_psx_error(status, ENOENT);
}
```

Listing 14: POSIX Status Code Report

```
B:stat
P:0:0:UI1:test-psx.c:13
Y:stat:N:1:F:0
```

2.1.7 Custom Log Messages

You can print custom log messages with the *T_log()* function:

```
void T_log(T_verbosity verbosity, char const *fmt, ...);
```

A newline is automatically added to terminate the log message line.

Listing 15: Custom Log Message Example

```
#include <t.h>

T_TEST_CASE(log)
{
    T_log(T_NORMAL, "a custom message %i, %i, %i", 1, 2, 3);
    T_set_verbosity(T_QUIET);
    T_log(T_NORMAL, "not verbose enough");
}
```

Listing 16: Custom Log Message Report

```
B:log
L:a custom message 1, 2, 3
Y:log:N:0:F:0
```

2.1.8 Time Services

The test framework provides two unsigned integer types for time values. The *T_ticks* unsigned integer type is used by the *T_tick()* function which measures time using the highest frequency counter available on the platform. It should only be used to measure small time intervals.

The *T_time* unsigned integer type is used by the *T_now()* function which returns the current monotonic clock value of the platform, e.g. *CLOCK_MONOTONIC*.

```
T_ticks T_tick(void);

T_time T_now(void);
```

The reference time point for these two clocks is unspecified. You can obtain the test case begin time with the *T_test_case_begin_time()* function.

```
T_time T_test_case_begin_time(void);
```

You can convert time into ticks with the *T_time_to_ticks()* function and vice versa with the *T_ticks_to_time()* function.

```
T_time T_ticks_to_time(T_ticks ticks);

T_ticks T_time_to_ticks(T_time time);
```

You can convert seconds and nanoseconds values into a combined time value with the *T_seconds_and_nanoseconds_to_time()* function. You can convert a time value into separate seconds and nanoseconds values with the *T_time_to_seconds_and_nanoseconds()* function.

```
T_time T_seconds_and_nanoseconds_to_time(uint32_t s, uint32_t ns);

void T_time_to_seconds_and_nanoseconds(T_time time, uint32_t *s, uint32_t *ns);
```

You can convert a time value into a string represention. The time unit of the string representation is seconds. The precision of the string represention may be nanoseconds, microseconds, milliseconds, or seconds. You have to provide a buffer for the string (*T_time_string*).

```
const char *T_time_to_string_ns(T_time time, T_time_string buffer);

const char *T_time_to_string_us(T_time time, T_time_string buffer);

const char *T_time_to_string_ms(T_time time, T_time_string buffer);

const char *T_time_to_string_s(T_time time, T_time_string buffer);
```

Listing 17: Time String Example

```
#include <t.h>

T_TEST_CASE(time_to_string)
{
    T_time_string ts;
    T_time t;
    uint32_t s;
    uint32_t ns;

    t = T_seconds_and_nanoseconds_to_time(0, 123456789);
    T_eq_str(T_time_to_string_ns(t, ts), "0.123456789");
    T_eq_str(T_time_to_string_us(t, ts), "0.123456");
    T_eq_str(T_time_to_string_ms(t, ts), "0.123");
    T_eq_str(T_time_to_string_s(t, ts), "0");
```

(continues on next page)

(continued from previous page)

```
15
16      T_time_to_seconds_and_nanoseconds(t, &s, &ns);
17      T_eq_u32(s, 0);
18      T_eq_u32(ns, 123456789);
19  }
```

Listing 18: Time String Report

```
1  B:time_to_string
2  P:0:0:UI1:test-time.c:11
3  P:1:0:UI1:test-time.c:12
4  P:2:0:UI1:test-time.c:13
5  P:3:0:UI1:test-time.c:14
6  P:4:0:UI1:test-time.c:17
7  P:5:0:UI1:test-time.c:18
8  Y:time_to_string:N:6:F:0:D:0.005250
```

You can convert a tick value into a string represention. The time unit of the string representation is seconds. The precision of the string represention may be nanoseconds, microseconds, milliseconds, or seconds. You have to provide a buffer for the string (*T_time_string*).

```
1  const char *T_ticks_to_string_ns(T_ticks ticks, T_time_string buffer);
2
3  const char *T_ticks_to_string_us(T_ticks ticks, T_time_string buffer);
4
5  const char *T_ticks_to_string_ms(T_ticks ticks, T_time_string buffer);
6
7  const char *T_ticks_to_string_s(T_ticks ticks, T_time_string buffer);
```

2.1.9 Code Runtime Measurements

You can measure the runtime of code fragments in several execution environment variants with the *T_measure_runtime()* function. This function needs a context which must be created with the *T_measure_runtime_create()* function. The context is automatically destroyed after the test case execution. The runtime measurement is performed for the *body* request handler of the measurement request (*T_measure_runtime_request*). The *setup* request handler is called before each invocation of the *body* request handler. The *teardown* request handler is called after each invocation of the *body* request handler.

```
1  typedef struct {
2      size_t sample_count;
3  } T_measure_runtime_config;
4
5  typedef struct {
6      const char *name;
7      int flags;
8      void (*setup)(void *);
9      void (*body)(void *);
10     void (*teardown)(void *);
11     void *arg;
12 } T_measure_runtime_request;
13
```

(continues on next page)

(continued from previous page)

```
T_measure_runtime_context *T_measure_runtime_create(
    const T_measure_runtime_config *config);

void T_measure_runtime(T_measure_runtime_context *ctx,
    const T_measure_runtime_request *request);
```

The execution environment variants (*M:V*) are:

ValidCache
Before the *body* request handler of a request is called a memory area with a size of the outermost data cache is completely read. This fills the data cache with valid cache lines which are unrelated to the *body* request handler.

HotCache
Before the *body* request handler of a request is called the *body* request handler is called without measuring the runtime. The aim is to load all data used by the *body* request handler to the cache.

DirtyCache
Before the *body* request handler of a request is called a memory area with a size of the outermost data cache is completely written with new data. This should produce a data cache with dirty cache lines which are unrelated to the *body* request handler. In addition, the entire instruction cache is invalidated.

Load
The *body* request handler is executed while other processors try to fully load the memory system. The load is produced through writes to a memory area with a size of the outer-most data cache. The load variant is performed multiple times with a different set of active load worker threads (*M:L*). The active workers range from one up to the processor count.

Each execution in an environment variant produces a sample set of *body* request handler runtime measurements. The minimum (*M:Min*), first quartile (*M:Q1*), median (*M:Q2*), third quartile (*M:Q3*), maximum (*M:Max*), median absolute deviation (*M:MAD*), and the sum of the sample values (*M:D*) is reported.

Listing 19: Code Runtime Measurement Example

```
#include <t.h>

static void
empty(void *arg)
{
    (void)arg;
}

T_TEST_CASE(measure_empty)
{
    static const T_measure_runtime_config config = {
        .sample_count = 1024
    };
    T_measure_runtime_context *ctx;
    T_measure_runtime_request req;

    ctx = T_measure_runtime_create(&config);
    T_assert_not_null(ctx);
```

(continues on next page)

(continued from previous page)

```
19
20      memset(&req, 0, sizeof(req));
21      req.name = "Empty";
22      req.body = empty;
23      T_measure_runtime(ctx, &req);
24  }
```

Listing 20: Code Runtime Measurement Report

```
1  B:measure_empty
2  P:0:0:UI1:test-rtems-measure.c:18
3  M:B:Empty
4  M:V:ValidCache
5  M:N:1024
6  M:MI:0.000000000
7  M:Q1:0.000000000
8  M:Q2:0.000000000
9  M:Q3:0.000000000
10 M:MX:0.000000009
11 M:MAD:0.000000000
12 M:D:0.000000573
13 M:E:Empty:D:0.936464309
14 M:B:Empty
15 M:V:HotCache
16 M:N:1024
17 M:MI:0.000000002
18 M:Q1:0.000000003
19 M:Q2:0.000000003
20 M:Q3:0.000000003
21 M:MX:0.000000006
22 M:MAD:0.000000000
23 M:D:0.000002517
24 M:E:Empty:D:0.000014027
25 M:B:Empty
26 M:V:DirtyCache
27 M:N:1024
28 M:MI:0.000000007
29 M:Q1:0.000000007
30 M:Q2:0.000000007
31 M:Q3:0.000000009
32 M:MX:0.000000320
33 M:MAD:0.000000000
34 M:D:0.000016085
35 M:E:Empty:D:0.260411428
36 M:B:Empty
37 M:V:Load
38 M:L:1
39 M:N:1024
40 M:MI:0.000000002
41 M:Q1:0.000000003
42 M:Q2:0.000000003
43 M:Q3:0.000000003
44 M:MX:0.000000009
45 M:MAD:0.000000000
46 M:D:0.000002608
```

(continues on next page)

(continued from previous page)

```
M:E:Empty:D:0.000008656
[... 22 more load variants ...]
M:B:Empty
M:V:Load
M:L:24
M:N:1024
M:MI:0.000000002
M:Q1:0.000000003
M:Q2:0.000000003
M:Q3:0.000000003
M:MX:0.000002214
M:MAD:0.000000000
M:D:0.000009994
M:E:Empty:D:0.000015991
Y:measure_empty:N:1:F:0:D:1.784504
```

2.1.10 Test Runner

You can call the *T_main()* function to run all registered test cases.

```
int T_main(const T_config *config);
```

The *T_main()* function returns 0 if all test cases passed, otherwise it returns 1. Concurrent execution of the *T_main()* function is undefined behaviour.

You can ask if you execute within the context of the test runner with the *T_is_runner()* function:

```
bool T_is_runner(void);
```

It returns *true* if you execute within the context of the test runner (the context which executes for example *T_main()*). Otherwise it returns *false*, for example if you execute in another task, in interrupt context, nobody executes *T_main()*, or during system initialization on another processor.

On RTEMS, you have to register the test cases with the *T_register()* function before you call *T_main()*. This makes it possible to run low level tests, for example without the operating system directly in *boot_card()* or during device driver initialization. On other platforms, the *T_register()* is a no operation.

```
void T_register(void);
```

You can run test cases also individually. Use *T_run_initialize()* to initialize the test runner. Call *T_run_all()* to run all or *T_run_by_name()* to run specific registered test cases. Call *T_test_case_begin()* to begin a freestanding test case and call *T_test_case_end()* to finish it. Finally, call *T_run_finalize()*.

```
void T_run_initialize(const T_config *config);

void T_run_all(void);

void T_run_by_name(const char *name);

void T_test_case_begin(const char *name, const T_test_fixture *fixture);
```

(continues on next page)

(continued from previous page)

```
8
9  void T_test_case_end(void);
10
11 bool T_run_finalize(void);
```

The *T_run_finalize()* function returns *true* if all test cases passed, otherwise it returns *false*. Concurrent execution of the runner functions (including *T_main()*) is undefined behaviour. The configuration must be persistent throughout the test run.

2.1.11 Test Verbosity

Three test verbosity levels are defined:

T_QUIET
Only the test suite begin, system, test case end, and test suite end lines are printed.

T_NORMAL
Prints everything except passed test lines.

T_VERBOSE
Prints everything.

The test verbosity level can be set within the scope of one test case with the *T_set_verbosity()* function:

```
1 T_verbosity T_set_verbosity(T_verbosity new_verbosity);
```

The function returns the previous verbosity. After the test case, the configured verbosity is automatically restored.

An example with *T_QUIET* verbosity:

```
1 A:xyz
2 S:Platform:RTEMS
3 [...]
4 Y:a:N:2:F:1
5 Y:b:N:0:F:1
6 Y:c:N:1:F:1
7 Y:d:N:6:F:0
8 Z:xyz:C:4:N:9:F:3
```

The same example with *T_NORMAL* verbosity:

```
1  A:xyz
2  S:Platform:RTEMS
3  [...]
4  B:a
5  F:1:0:UI1:test-verbosity.c:6:test fails
6  Y:a:N:2:F:1
7  B:b
8  F:*:0:UI1:test-verbosity.c:12:quiet test fails
9  Y:b:N:0:F:1
10 B:c
11 F:0:0:UI1:test-verbosity.c:17:this is a format string
12 Y:c:N:1:F:1
```

(continues on next page)

(continued from previous page)

```
B:d
Y:d:N:6:F:0
Z:xyz:C:4:N:9:F:3
```

The same example with *T_VERBOSE* verbosity:

```
A:xyz
S:Platform:RTEMS
[...]
B:a
P:0:0:UI1:test-verbosity.c:5
F:1:0:UI1:test-verbosity.c:6:test fails
Y:a:N:2:F:1
B:b
F:*:0:UI1:test-verbosity.c:12:quiet test fails
Y:b:N:0:F:1
B:c
F:0:0:UI1:test-verbosity.c:17:this is a format string
Y:c:N:1:F:1
B:d
P:0:0:UI1:test-verbosity.c:22
P:1:0:UI1:test-verbosity.c:23
P:2:0:UI1:test-verbosity.c:24
P:3:0:UI1:test-verbosity.c:25
P:4:0:UI1:test-verbosity.c:26
P:5:0:UI1:test-verbosity.c:27
Y:d:N:6:F:0
Z:xyz:C:4:N:9:F:3
```

2.1.12 Test Reporting

The test reporting is line based which should be easy to parse with a simple state machine. Each line consists of a set of fields separated by colon characters (:). The first character of the line determines the line format:

A
A test suite begin line. It has the format:

A:<TestSuite>

A description of the field follows:

<TestSuite>
The test suite name. Must not contain colon characters (:).

S
A test suite system line. It has the format:

S:<Key>:<Value>

A description of the fields follows:

<Key>
A key string. Must not contain colon characters (:).

<Value>
An arbitrary key value string. May contain colon characters (:).

B

A test case begin line. It has the format:

B:<TestCase>

A description of the field follows:

<TestCase>
A test case name. Must not contain colon characters (*:*).

P

A test pass line. It has the format:

P:<Step>:<Processor>:<Task>:<File>:<Line>

A description of the fields follows:

<Step>
Each non-quiet test has a unique test step counter value in each test case execution. The test step counter is set to zero before the test case executes. For quiet tests, there is no associated test step and the character * instead of an integer is used to indicate this.

<Processor>
The processor index of the processor which executed at least one instruction of the corresponding test.

<Task>
The name of the task which executed the corresponding test if the test executed in task context. The name *ISR* indicates that the test executed in interrupt context. The name *?* indicates that the test executed in an arbitrary context with no valid executing task.

<File>
The name of the source file which contains the corresponding test. A source file of * indicates that no test source file is associated with the test, e.g. it was produced by the test framework itself.

<Line>
The line of the test statement in the source file which contains the corresponding test. A line number of * indicates that no test source file is associated with the test, e.g. it was produced by the test framework itself.

F

A test failure line. It has the format:

F:<Step>:<Processor>:<Task>:<File>:<Line>:<Message>

A description of the fields follows:

<Step> <Processor> <Task> <File> <Line>
See above **P** line.

<Message>
An arbitrary message string. May contain colon characters (*:*).

L

A log message line. It has the format:

L:<Message>

A description of the field follows:

<Message>
An arbitrary message string. May contain colon characters (:).

Y
A test case end line. It has the format:

Y:<TestCase>:N:<Steps>:F:<Failures>:D:<Duration>

A description of the fields follows:

<TestCase>
A test case name. Must not contain colon characters (:).

<Steps>
The final test step counter of a test case. Quiet tests produce no test steps.

<Failures>
The count of failed tests of a test case.

<Duration>
The test case duration in seconds.

Z
A test suite end line. It has the format:

Z:<TestSuite>:C:<TestCases>:N:<OverallSteps>:F:<OverallFailures>:D:<Duration>

A description of the fields follows:

<TestSuite>
The test suite name. Must not contain colon characters (:).

<TestCases>
The count of test cases in the test suite.

<OverallSteps>
The overall count of test steps in the test suite.

<OverallFailures>
The overall count of failed tests in the test suite.

<Duration>
The test suite duration in seconds.

M
A code runtime measurement line. It has the formats:

M:B:<Name>

M:V:<Variant>

M:L:<Load>

M:N:<SampleCount>

M:Min:<Minimum>

M:Q1:<FirstQuartile>

M:Q2:<Median>

M:Q3:<ThirdQuartile>

M:Max:<Maximum>

M:MAD:<MedianAbsoluteDeviation>

M:D:<SumOfSampleValues>

M:E:<Name>:D:<Duration>

A description of the fields follows:

<Name>
A code runtime measurement name. Must not contain colon characters (:).

<Variant>
The execution variant which is one of **ValidCache**, **HotCache**, **DirtyCache**, or **Load**.

<Load>
The active load workers count which ranges from one to the processor count.

<SampleCount>
The sample count as defined by the runtime measurement configuration.

<Min>
The minimum of the sample set in seconds.

<Q1>
The first quartile of the sample set in seconds.

<Q2>
The median of the sample set in seconds.

<Q3>
The third quartile of the sample set in seconds.

<Max>
The maximum of the sample set in seconds.

<MedianAbsoluteDeviation>
The median absolute deviation of the sample set in seconds.

<SumOfSampleValues>
The sum of all sample values of the sample set in seconds.

<Duration>
The runtime measurement duration in seconds. It includes time to set up the execution environment variant.

Listing 21: Example Test Report

```
A:xyz
S:Platform:RTEMS
S:Compiler:7.3.0 20180125 (RTEMS 5, RSB 30da0c720b78eba16a3f5272206c07415368617b, Newlib␣
↪2ab57ad59bc35dafffa69cd4da5e228971de069f)
S:Version:5.0.0.af5840fbb3d5e5aae24af3da84c75a011098f086
S:BSP:erc32
S:RTEMS_DEBUG:0
S:RTEMS_MULTIPROCESSING:0
S:RTEMS_POSIX_API:1
S:RTEMS_PROFILING:0
S:RTEMS_SMP:0
B:rsc
P:0:0:UI1:test-rtems.c:33
```

(continues on next page)

(continued from previous page)

```
F:1:0:UI1:test-rtems.c:34:RTEMS_INVALID_NUMBER == RTEMS_INVALID_ID
F:*:0:UI1:test-rtems.c:36:RTEMS_INVALID_NUMBER == RTEMS_INVALID_ID
P:2:0:UI1:test-rtems.c:37
F:3:0:UI1:test-rtems.c:38:RTEMS_INVALID_NUMBER == RTEMS_INVALID_ID
Y:rsc:N:4:F:3
B:rsc_success
P:0:0:UI1:test-rtems.c:44
F:1:0:UI1:test-rtems.c:45:RTEMS_INVALID_NUMBER == RTEMS_SUCCESSFUL
F:*:0:UI1:test-rtems.c:47:RTEMS_INVALID_NUMBER == RTEMS_SUCCESSFUL
P:2:0:UI1:test-rtems.c:48
F:3:0:UI1:test-rtems.c:49:RTEMS_INVALID_NUMBER == RTEMS_SUCCESSFUL
Y:rsc_success:N:4:F:3
B:timer
P:0:0:UI1:test-rtems.c:74
P:1:0:UI1:test-rtems.c:78
P:2:0:ISR:test-rtems.c:61
P:3:0:UI1:test-rtems.c:83
P:4:0:UI1:test-rtems.c:84
P:5:0:UI1:test-rtems.c:87
Y:timer:N:6:F:0
Z:xyz:C:3:N:14:F:6
```

2.1.13 Supported Platforms

The framework runs on FreeBSD, MSYS2, Linux and RTEMS.

2.2 Test Framework Requirements for RTEMS

The requirements on a test framework suitable for RTEMS are:

2.2.1 License Requirements

TF.License.Permissive
The test framework must have a permissive open source license such as BSD-2-Clause.

2.2.2 Portability Requirements

TF.Portability
The test framework must be portable.

TF.Portability.RTEMS
The test framework must run on RTEMS.

TF.Portability.POSIX
The test framework must be portable to POSIX compatible operating systems. This allows to run test cases of standard C/POSIX/etc. APIs on multiple platforms.

TF.Portability.POSIX.Linux
The test framework must run on Linux.

TF.Portability.POSIX.FreeBSD
The test framework must run on FreeBSD.

TF.Portability.C11
The test framework must be written in C11.

TF.Portability.Static
Test framework must not use dynamic memory for basic services.

TF.Portability.Small
The test framework must be small enough to support low-end platforms (e.g. 64KiB of RAM/ROM should be sufficient to test the architecture port, e.g. no complex stuff such as file systems, etc.).

TF.Portability.Small.LinkTimeConfiguration
The test framework must be configured at link-time.

TF.Portability.Small.Modular
The test framework must be modular so that only necessary parts end up in the final executable.

TF.Portability.Small.Memory
The test framework must not aggregate data during test case executions.

2.2.3 Reporting Requirements

TF.Reporting
Test results must be reported.

TF.Reporting.Verbosity
The test report verbosity must be configurable. This allows different test run scenarios, e.g. regression test runs, full test runs with test report verification against the planned test output.

TF.Reporting.Verification
It must be possible to use regular expressions to verify test reports line by line.

TF.Reporting.Compact
Test output must be compact to avoid long test runs on platforms with a slow output device, e.g. 9600 Baud UART.

TF.Reporting.PutChar
A simple output one character function provided by the platform must be sufficient to report the test results.

TF.Reporting.NonBlocking
The ouptut functions must be non-blocking.

TF.Reporting.Printf
The test framework must provide printf()-like output functions.

TF.Reporting.Printf.WithFP
There must be a printf()-like output function with floating point support.

TF.Reporting.Printf.WithoutFP
There must be a printf()-like output function without floating point support on RTEMS.

TF.Reporting.Platform
The test platform must be reported.

TF.Reporting.Platform.RTEMS.Git
The RTEMS source Git commit must be reported.

TF.Reporting.Platform.RTEMS.Arch
The RTEMS architecture name must be reported.

TF.Reporting.Platform.RTEMS.BSP
The RTEMS BSP name must be reported.

TF.Reporting.Platform.RTEMS.Tools
The RTEMS tool chain version must be reported.

TF.Reporting.Platform.RTEMS.Config.Debug
The must be reported if RTEMS_DEBUG is defined.

TF.Reporting.Platform.RTEMS.Config.Multiprocessing
The must be reported if RTEMS_MULTIPROCESSING is defined.

TF.Reporting.Platform.RTEMS.Config.POSIX
The must be reported if RTEMS_POSIX_API is defined.

TF.Reporting.Platform.RTEMS.Config.Profiling
The must be reported if RTEMS_PROFILING is defined.

TF.Reporting.Platform.RTEMS.Config.SMP
The must be reported if RTEMS_SMP is defined.

TF.Reporting.TestCase
The test cases must be reported.

TF.Reporting.TestCase.Begin
The test case begin must be reported.

TF.Reporting.TestCase.End
The test case end must be reported.

TF.Reporting.TestCase.Tests
The count of tests of the test case must be reported.

TF.Reporting.TestCase.Failures
The count of failed tests of the test case must be reported.

TF.Reporting.TestCase.Timing
Test case timing must be reported.

TF.Reporting.TestCase.Tracing
Automatic tracing and reporting of thread context switches and interrupt service routines shall be optionally performed.

2.2.4 Environment Requirements

TF.Environment
The test framework must support all environment conditions of the platform.

TF.Environment.SystemStart
The test framework must run during early stages of the system start, e.g. valid stack pointer, initialized data and cleared BSS, nothing more.

TF.Environment.BeforeDeviceDrivers
The test framework must run before device drivers are initialized.

TF.Environment.InterruptContext
The test framework must support test case code in interrupt context.

2.2.5 Usability Requirements

TF.Usability
The test framework must be easy to use.

TF.Usability.TestCase
It must be possible to write test cases.

TF.Usability.TestCase.Independence
It must be possible to write test cases in modules independent of the test runner.

TF.Usability.TestCase.AutomaticRegistration
Test cases must be registered automatically, e.g. via constructors or linker sets.

TF.Usability.TestCase.Order
It must be possible to sort the registered test cases (e.g. random, by name) before they are executed.

TF.Usability.TestCase.Resources
It must be possible to use resources with a life time restricted to the test case.

TF.Usability.TestCase.Resources.Memory
It must be possible to dynamically allocate memory which is automatically freed once the test case completed.

TF.Usability.TestCase.Resources.File
It must be possible to create a file which is automatically unlinked once the test case completed.

TF.Usability.TestCase.Resources.Directory
It must be possible to create a directory which is automatically removed once the test case completed.

TF.Usability.TestCase.Resources.FileDescriptor
It must be possible to open a file descriptor which is automatically closed once the test case completed.

TF.Usability.TestCase.Fixture
It must be possible to use a text fixture for test cases.

TF.Usability.TestCase.Fixture.SetUp
It must be possible to provide a set up handler for each test case.

TF.Usability.TestCase.Fixture.TearDown
It must be possible to provide a tear down handler for each test case.

TF.Usability.TestCase.Context
The test case context must be verified a certain points.

TF.Usability.TestCase.Context.VerifyAtEnd
After a test case exection it must be verified that the context is equal to the context at the test case begin. This helps to ensure that test cases are independent of each other.

TF.Usability.TestCase.Context.VerifyThread
The test framework must provide a function to ensure that the test case code executes in normal thread context. This helps to ensure that operating system service calls return to a sane context.

TF.Usability.TestCase.Context.Configurable
The context verified in test case must be configurable at link-time.

TF.Usability.TestCase.Context.ThreadDispatchDisableLevel
It must be possible to verify the thread dispatch disable level.

TF.Usability.TestCase.Context.ISRNestLevel
It must be possible to verify the ISR nest level.

TF.Usability.TestCase.Context.InterruptLevel
It must be possible to verify the interrupt level (interrupts enabled/disabled).

TF.Usability.TestCase.Context.Workspace
It must be possible to verify the workspace.

TF.Usability.TestCase.Context.Heap
It must be possible to verify the heap.

TF.Usability.TestCase.Context.OpenFileDescriptors
It must be possible to verify the open file descriptors.

TF.Usability.TestCase.Context.Classic
It must be possible to verify Classic API objects.

TF.Usability.TestCase.Context.Classic.Barrier
It must be possible to verify Classic API Barrier objects.

TF.Usability.TestCase.Context.Classic.Extensions
It must be possible to verify Classic API User Extensions objects.

TF.Usability.TestCase.Context.Classic.MessageQueues
It must be possible to verify Classic API Message Queue objects.

TF.Usability.TestCase.Context.Classic.Partitions
It must be possible to verify Classic API Partition objects.

TF.Usability.TestCase.Context.Classic.Periods
It must be possible to verify Classic API Rate Monotonic Period objects.

TF.Usability.TestCase.Context.Classic.Regions
It must be possible to verify Classic API Region objects.

TF.Usability.TestCase.Context.Classic.Semaphores
It must be possible to verify Classic API Semaphore objects.

TF.Usability.TestCase.Context.Classic.Tasks
It must be possible to verify Classic API Task objects.

TF.Usability.TestCase.Context.Classic.Timers
It must be possible to verify Classic API Timer objects.

TF.Usability.TestCase.Context.POSIX
It must be possible to verify POSIX API objects.

TF.Usability.TestCase.Context.POSIX.Keys
It must be possible to verify POSIX API Key objects.

TF.Usability.TestCase.Context.POSIX.KeyValuePairs
It must be possible to verify POSIX API Key Value Pair objects.

TF.Usability.TestCase.Context.POSIX.MessageQueues
It must be possible to verify POSIX API Message Queue objects.

TF.Usability.TestCase.Context.POSIX.Semaphores
It must be possible to verify POSIX API Named Semaphores objects.

TF.Usability.TestCase.Context.POSIX.Shms
It must be possible to verify POSIX API Shared Memory objects.

TF.Usability.TestCase.Context.POSIX.Threads
It must be possible to verify POSIX API Thread objects.

TF.Usability.TestCase.Context.POSIX.Timers
It must be possible to verify POSIX API Timer objects.

TF.Usability.Assert
There must be functions to assert test objectives.

TF.Usability.Assert.Safe
Test assert functions must be safe to use, e.g. assert(a == b) vs. assert(a = b) vs. assert_eq(a, b).

TF.Usability.Assert.Continue
There must be assert functions which allow the test case to continue in case of an assertion failure.

TF.Usability.Assert.Abort
There must be assert functions which abourt the test case in case of an assertion failure.

TF.Usability.EasyToWrite
It must be easy to write test code, e.g. avoid long namespace prefix rtems_test_*.

TF.Usability.Threads
The test framework must support multi-threading.

TF.Usability.Pattern
The test framework must support test patterns.

TF.Usability.Pattern.Interrupts
The test framework must support test cases which use interrupts, e.g. spintrcritical*.

TF.Usability.Pattern.Parallel
The test framework must support test cases which want to run code in parallel on SMP machines.

TF.Usability.Pattern.Timing
The test framework must support test cases which want to measure the timing of code sections under various platform conditions, e.g. dirty cache, empty cache, hot cache, with load from other processors, etc..

TF.Usability.Configuration
The test framework must be configurable.

TF.Usability.Configuration.Time
The timestamp function must be configurable, e.g. to allow test runs without a clock driver.

2.2.6 Performance Requirements

TF.Performance.RTEMS.No64BitDivision
The test framework shall not use 64-bit divisions on RTEMS.

2.3 Off-the-shelf Test Frameworks

There are several off-the-shelf test frameworks for C/C++. The first obstacle for test frameworks is the license requirement (*TF.License.Permissive*).

2.3.1 bdd-for-c

In the bdd-for-c framework the complete test suite must be contained in one file and the main function is generated. This violates *TF.Usability.TestCase.Independence*.

2.3.2 CBDD

The CBDD framework uses the C blocks extension from clang. This violates *TF.Portability.C11*.

2.3.3 Google Test

Google Test 1.8.1 is supported by RTEMS. Unfortunately, it is written in C++ and is to heavy weight for low-end platforms. Otherwise it is a nice framework.

2.3.4 Unity

The Unity Test API does not meet our requirements. There was a discussion on the mailing list in 2013.

2.4 Standard Test Report Formats

2.4.1 JUnit XML

A common test report format is JUnit XML.

```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuites id="xyz" name="abc" tests="225" failures="1262" time="0.001">
  <testsuite id="def" name="ghi" tests="45" failures="17" time="0.001">
    <testcase id="jkl" name="mno" time="0.001">
      <failure message="pqr" type="stu"></failure>
      <system-out>stdout</system-out>
      <system-err>stderr</system-err>
    </testcase>
  </testsuite>
</testsuites>
```

The major problem with this format is that you have to output the failure count of all test suites and the individual test suite before the test case output. You know the failure count only after a complete test run. This runs contrary to requirement *TF.Portability.Small.Memory*. It is also a bit verbose (*TF.Reporting.Compact*).

It is easy to convert a full test report generated by *The T Test Framework* to the JUnit XML format.

2.4.2 Test Anything Protocol

The Test Anything Protocol (TAP) is easy to consume and produce.

```
1..4
ok 1 - Input file opened
not ok 2 - First line of the input valid
ok 3 - Read the rest of the file
not ok 4 - Summarized correctly # TODO Not written yet
```

You have to know in advance how many test statements you want to execute in a test case. The problem with this format is that there is no standard way to provide auxiliary data such as test timing or a tracing report.

It is easy to convert a full test report generated by *The T Test Framework* to the TAP format.